Author
Simon Primetzhofer

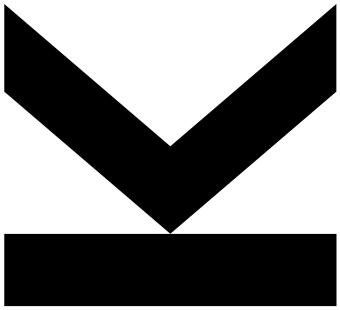
Submission
Institute for System Software

Thesis Supervisor
DI Dr. Markus Weninger BSc

External Thesis Supervisor
DI Thomas Reichenberger

March 2023

Typinator: Windows Application for Automatic Text Expansion



Bachelor's Thesis

to confer the academic degree of

Bachelor of Science

in the Bachelor's Program

Informatik

JOHANNES KEPLER UNIVERSITY LINZ
Altenberger Str. 69
4040 Linz, Austria
www.jku.at
DVR 0093696

JOHANNES KEPLER
UNIVERSITY LINZ

Bachelor's Thesis

Typinator: Windows Application for Automatic Text Expansion

Dipl.-Ing. Dr.
Markus Weninger, BSc
Institute for System Software

P +43-732-2468-4361
F +43-732-2468-4345
markus.weninger@jku.at

Student: Simon Primetzhofer
Advisor: Dipl.-Ing. Dr. Markus Weninger, BSc
Ergonis Supervisor: Dipl.-Ing. Thomas Reichenberger, BSc
Start date: March 2022

The company *ergonis* distributes the Mac application “Typinator", a tool that enables users to define abbreviations together with a corresponding text expansion. These text expansions are then automatically applied system-wide while typing. Due to platform restrictions, the tool is not available on Windows or Linux.

The goal of this bachelor thesis is the technical evaluation and development of a prototype, porting Typinator to Windows. Following questions must be investigated as part of this thesis, and suitable solutions must be implemented:

- Evaluation of technologies and methods for making Typinator available as a Windows application
  - Investigation of technological limits and possibilities in Windows
  - Evaluate which of Typinators current features are possible in Windows and if there are any limitations within different groups of applications (Browsers, Systray Applications, Native Windows Applications, …)
- Develop and use a shared code-base with other platform-specific versions of Typinator for commonly used tasks and features
- Implementation of text expansion in the Windows environment based on a set of pre-defined abbreviations.
- (nice to have): Prototype of a management view that offers the functionality of defining and managing abbreviations

Modalities:
The progress of the project should be discussed at least every two weeks with the ergonis supervisor and at least once per month with the advisor. A time schedule and a milestone plan must be set up within the first 3 weeks and discussed with the advisor and the supervisors. It should be continuously refined and monitored to make sure that the thesis will be completed in time. The final version of the thesis must be submitted not later than 28.08.2022.

Abstract

While working with keyboards on computers, one might come across recurring patterns of text or common mistakes which take a considerable amount of time to either type or fix. Due to the fact that many modern jobs shift towards being office-related, these problems become more relevant to a vast amount of people that try to be as efficient as possible. Not only do they want to save their precious time, they also like to conveniently handle their tasks.

Thus, *Typinator for Windows* provides a low-threshold entrance into office workflow optimization by analyzing keyboard inputs, checking whether they correspond to predefined abbreviations in a database and, in case, inserting their corresponding expansions, i.e., long versions, inside the active text area.

One way of using this mechanism is setting up a dictionary of repeatedly misspelled words and having them corrected automatically while typing. Other ways to use auto-expanded abbreviations are to expand them to predefined text templates, Unicode characters such as emojis or even images.

Typinator for Windows is an easy-to-handle background application which does not interfere with the user's actions by operating at high performance. Through displaying a system tray icon only, the application ensures being a lightweight extension to every Windows computer.

Kurzfassung

Bei der Tastaturarbeit am Computer begegnet man oftmals wiederkehrenden Textmustern oder häufig auftretenden Tippfehlern, was zu einem nicht zu vernachlässigenden zeitlichen Aufwand führt, um diese zu tippen oder zu beheben. Da sich viele moderne Arbeitsplätze in Richtung Büroarbeit verlagern, werden diese Probleme für viele Menschen, die so effizient wie möglich sein wollen, immer relevanter. Diese wollen nicht nur ihre kostbare Zeit sparen, sondern auch ihre Aufgaben auf praktische Weise erledigen.

Deshalb bietet *Typinator für Windows* einen niederschwelligen Einstieg in die Optimierung von Arbeitsabläufen im Büro, indem es Tastatureingaben analysiert, prüft, ob sie mit vordefinierten Abkürzungen in einer Datenbank übereinstimmen und, falls dies der Fall ist, die entsprechenden Erweiterungen, d.h. die Langversionen, in den aktiven Textbereich einfügt.

Ein Weg, diesen Mechanismus zu nutzen, besteht darin, ein Wörterbuch mit wiederholt falsch geschriebenen Wörtern anzulegen und diese während der Eingabe automatisch korrigieren zu lassen. Andere Möglichkeiten, um automatisch erweiterte Abkürzungen zu verwenden, bestehen darin, diese zu vordefinierten Textvorlagen, Unicode-Zeichen wie Emojis oder sogar Bildern zu erweitern.

Typinator für Windows ist eine einfach zu handhabende Hintergrundanwendung, die den Benutzer durch ihre hohe Performanz nicht bei seinen Aktionen blockiert. Da die Anwendung ausschließlich als System-Tray Icon angezeigt wird, stellt sie eine leichtgewichtige Erweiterung für jeden Windows-Computer dar.

Table of Content

Contents

1 Introduction

Productivity tools are helping to increase working efficiency and making one's life easier. For instance, time planners, to-do lists, various communication tools and many more applications fall under this category and accomplish said goal. But most important, a vast amount of office workers experience typing repetitive text or using recurring text templates to some extent which in the end leads to a mentionable time consumption.

For overcoming these problems, various tools exist and one of them is *Typinator*. It supports people using keyboards by managing a set of abbreviations that map to user-specified text templates, images and code snippets which can be inserted at the cursors' current position after the given abbreviation has been typed. Unfortunately, *Typinator* is only available for MacOS until now and thus excludes the majority of computer operators from using it at all. As shown in Figure 1, most desktop computers and laptops nowadays run Microsoft Windows as their operating system and as a result most likely could profit from *Typinator* as well.

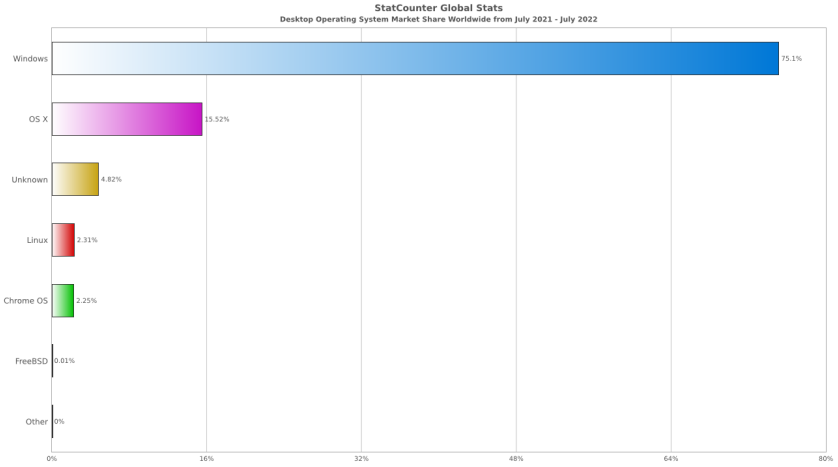


Figure 1: Current desktop PC market share[1].

The main contribution of this thesis is a working implementation of *Typinator* for Windows providing some base functionalities of the original product. Additionally, a system-agnostic library named *Typinator Core* is being developed (not part of this thesis) and integrated into *Typinator for Windows*. It is responsible for executing background tasks that are not dependent on the operating system. Due to the fact that *Typinator Core* is being developed at *Ergonis*, this thesis only considers operating-system-specific operations such as listening to keystrokes and inserting texts and images at cursor's current position.
All of this results in the opportunity for the product's original developers at *Ergonis* to open *Typinator* to the Windows market and thus a broader spectrum of new potential users and customers.

[1] `https://gs.statcounter.com/os-market-share/desktop/worldwide/#monthly-202107-202207-bar`

2 Background

This section describes the most important knowledge and history about the base product *Typinator*, how it manages abbreviations and text expansions and reasons for developing a Windows-specific *Typinator* version.

2.1 History of *Typinator*

Ergonis is a company which is specialized in providing productivity tools for their customers. Moreover, due to their technical know-how in combination with research knowledge at first hand, they were able to create easy-to-use but still powerful tools such as:

- KeyCue
  - Exclusively for MacOS.
  - Create custom shortcuts to open frequently used applications, files, folders and more.
- PopChar
  - For Windows and MacOS.
  - Provides a window for finding and typing single special characters.
  - Search for characters in thousands of different fonts.
  - View font details.
  - Check out how text fragments look like in certain fonts.
- Typinator
  - Exclusively for MacOS.
  - Inserts texts and images (called *expansions*) at the cursor's position when certain abbreviations have been typed.
  - Complex features such as executing scripts and calculations are supported.

The company *Ergonis* publicly released the first version of *Typinator* in June 2005. Back then, only a small set of functions was available which were continuously extended over time. Some major milestones with their most important features [2] are are illustrated in the following list:

- June 2005 - Release of Typinator 1.0
- June 2007 - Typinator 2.0
  - Manage abbreviations in sets.
    - Group abbreviations and expansions by usage type (e.g. Work and Private).
    - Identify related abbreviations and expansions and administrate them as a whole.
  - Import and export sets from and to files.
  - Inserting current clipboard content within expansions.

- April 2008 - Typinator 3.0
  - Dockless application only visible by a status bar menu item.
- May 2010 - Typinator 4.0
  - Globally enable and disable given sets.
- January 2012 - Typinator 5.0
  - Support for script execution (e.g., PHP, Perl, Python, Ruby, AppleScript, JavaScript and many more) within expansions.
  - Inclusion of text file contents within expansions.
  - Date and time calculations.
- June 2014 - Typinator 6.0
  - Support of regular expression to define abbreviations for flexible patterns.
- November 2016 - Typinator 7.0
  - Subscriptions: Add sets from remote sources with automated updates.
  - Publish sets for other *Typinator* users to subscribe to.
- June 2019 - Typinator 8.0
  - Statistics mode showing frequently used abbreviations.
  - Use special keys (e.g., CTRL and ALT) in abbreviations.

2.2 Abbreviations and Text Expansions

Managing and recognizing **abbreviations** is a key part of *Typinator*. In general, abbreviations are of textual form and are solely recognized by typed keystrokes. In addition, they may also contain special keys which can trigger different kinds of expansions which must not be solely text. Abbreviations are replaced by **text expansions** that can be one of the following (summarized from [2]):

- Text
- Images
- Results from script executions (e.g., JavaScript or Python code)

Abbreviations and expansions are logically grouped in *sets*. A set has a unique name, may contain arbitrary many abbreviation-expansion pairs with a prefix and/or suffix and options such as a rank or an enabled-flag. A practical example of how abbreviations and expansions are managed in sets is visualized in Table 1. Further, basic expansions of type text and image, as they are supported by *Typinator for Windows*, are shown in Table 2. For completeness, more complex expansions like execution of PHP, Perl, Python, Ruby, AppleScript or JavaScript [3] scripts are shown in Table 2 (line 5) but are only featured in the MacOS version and thus not described in more detail.

Rank	Set name	Prefix	Abbreviation	Expansion	Enabled
1.	private	private-	tel	07262 98769	true
2.	work	jku-	tel	0664 12345	true

Table 1: A person might use *Typinator* for private and work purposes and thus manage a separate set for each. In this example, the work set and private set both contain an abbreviation *tel* with a corresponding phone number as expansion. By typing *tel* on the keyboard without any further options, *Typinator* runs into a conflict. First, *Typinator* checks if the set is enabled or disabled. Second, sets are searched through by in ascending order of their rank. Hereby, the set private immediately delivers a result and the private phone number gets expanded. To explicitly expand the work phone number, the user would have to type *jku-tel* which indicates that the set with the prefix *jku-* shall be used.

	Abbreviation	Expansion
1.	name␣	Max Mustermann
2.	tel␣	06641234567890
3.	emoji␣	◡̈
4.	logo␣	JKU JOHANNES KEPLER UNIVERSITÄT LINZ
5.	{ /my/script.js }␣	21.5 °C

Table 2: Textual abbreviations can have different expansion types. On the left hand side we see abbreviations (only textual) and on the right hand side their corresponding expansion (three of type text, whereby the third one is a single Unicode character, an image and the execution result of a script file).

2.3 Reasons for Developing a Windows-specific *Typinator* Version

Even though it would, in general, be possible to run Objective-C code (the language the MacOS version is written in) on Windows, *Typinator* heavily relies on native MacOS system calls. Thus, the existing code base is not suitable for Windows due to the OS-specific hooks which cannot be reused as-is. Nevertheless, storage and handling of the data does not depend on the operating

system and therefore has been extracted into an operating-system-independent library (*Typinator Core*) in parallel with this thesis.

As a consequence of the necessity to reimplement *Typinator* for Windows, the usage of **User32.dll** from the **Win32 API** comes at hand. Basically, this dynamic link library (or short DLL) provides, besides many other features, useful functions for listening to the keyboard and simulating keystrokes. But most important, it is globally available in the operating system and thus, no external dependency is required. To sum up, *Typinator for Windows* would not be feasible without Win32 and it makes *Typinator for Windows* usable on all current Windows computers.

The programming language of choice for implementing *Typinator for Windows* is C# since it fits well into the Windows ecosystem and makes it easy to work with Win32 through built in features (i.e., P/Invoke).

3 Approach

This section discusses and illustrates conceptual considerations of *Typinator for Windows*. First, the composition of the whole *Typinator* environment is explained. Following, a brief overview is given of how *Typinator for Windows* in combination with *Typinator Core* stores data and handles synchronisation between multiple devices. Last, a typical flow from recognizing abbreviations to applying expansions is shown in detail.

3.1 Architecture

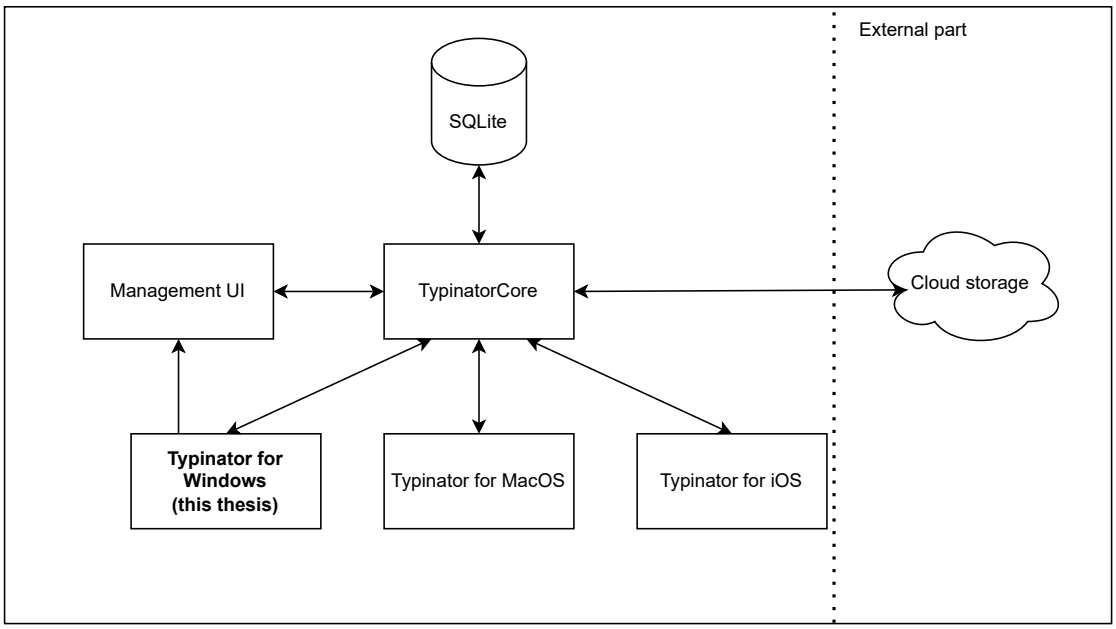


Figure 2: This figure shows the overall architecture of internal and external software parts in the *Typinator* environment. *Typinator Core* is the main interface between all other components and is directly used by each platform-specific *Typinator* implementation (i.e., *Typinator for Windows*, *Typinator for MacOS* and *Typinator for iOS*. The cloud storage is a third-party tool, e.g., Dropbox, OneDrive or iCloud, and is used via the internet.

Typinator Core is a completely newly developed library to ensure code sharing between all supported platforms in order to bundle system-agnostic features into a central unit. Such features are for instance:

- Data synchronisation
- Set storage and retrieval
- Efficient search algorithm for abbreviations in sets

Evidently, *Typinator Core* is the heart of the whole environment as can be seen in Figure 2. First of all, it is responsible for communicating with a local database that synchronises all sets

over a cloud storage provider of the end user's choice. Furthermore, *Typinator Core* is crucial for the different frontends such as the platform-specific *Typinator* implementations and the Management UI. Thus, *Typinator Core* is provided as DLL and can be integrated in most common programming languages with ease which achieves the target of being platform-independent.

3.2 Data Storage and Synchronisation

Since *Typinator* is built for working with different data types, e.g., texts and images, and possibly large amounts of data, sophisticated storage mechanisms are put in place. All of *Typinator*'s set data is stored inside an SQLite database which is managed by *Typinator Core*. But *Typinator Core* has to distinguish between two cases: (1) local sets and (2) synchronised sets. Local sets are only kept on the current machine and are excluded from synchronisation to other devices. Contrary to that, synchronised sets are shared via a cloud storage provider as can be seen in Figure 3.

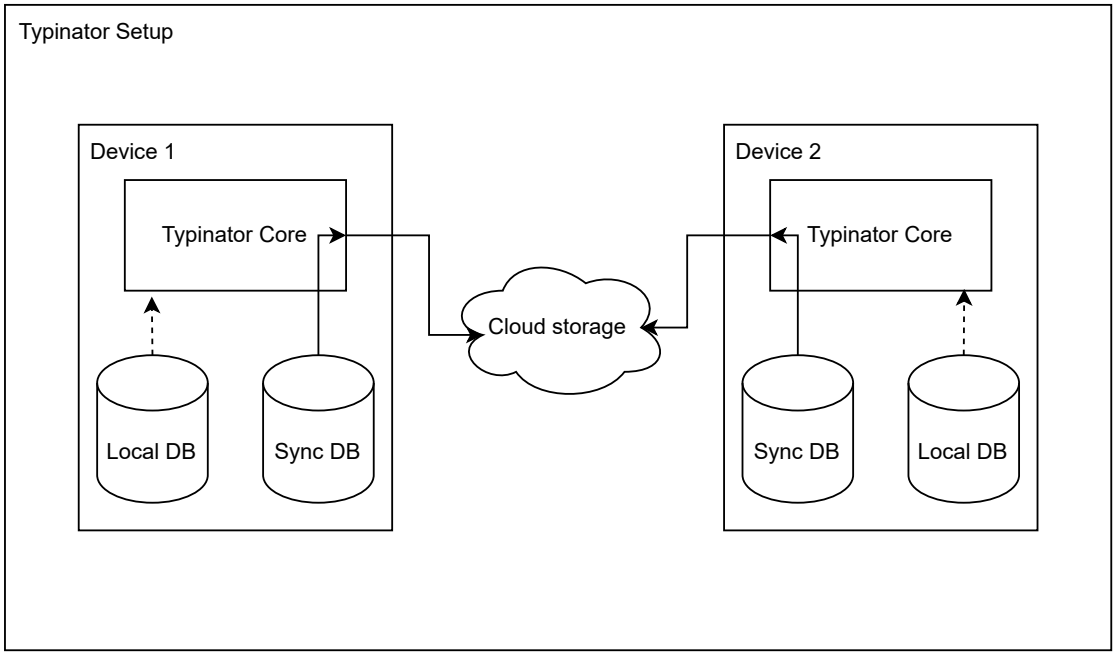


Figure 3: This figure shows a *Typinator* setup with two distinct *Typinator for Windows* applications that are only connected via the owner's personal cloud storage space. Since every *Typinator for Windows* installation comes with *Typinator Core* as central unit, *Typinator Core* manages a *Local DB* and a *Sync DB* per device. All sets which are inside the *Sync DB* are synchronised via a cloud storage provider and are therefore available on Device 1 and Device 2 at the same time. Local sets are solely stored in the *Local DB* which is not part of synchronisation.

3.3 Inner Workflow

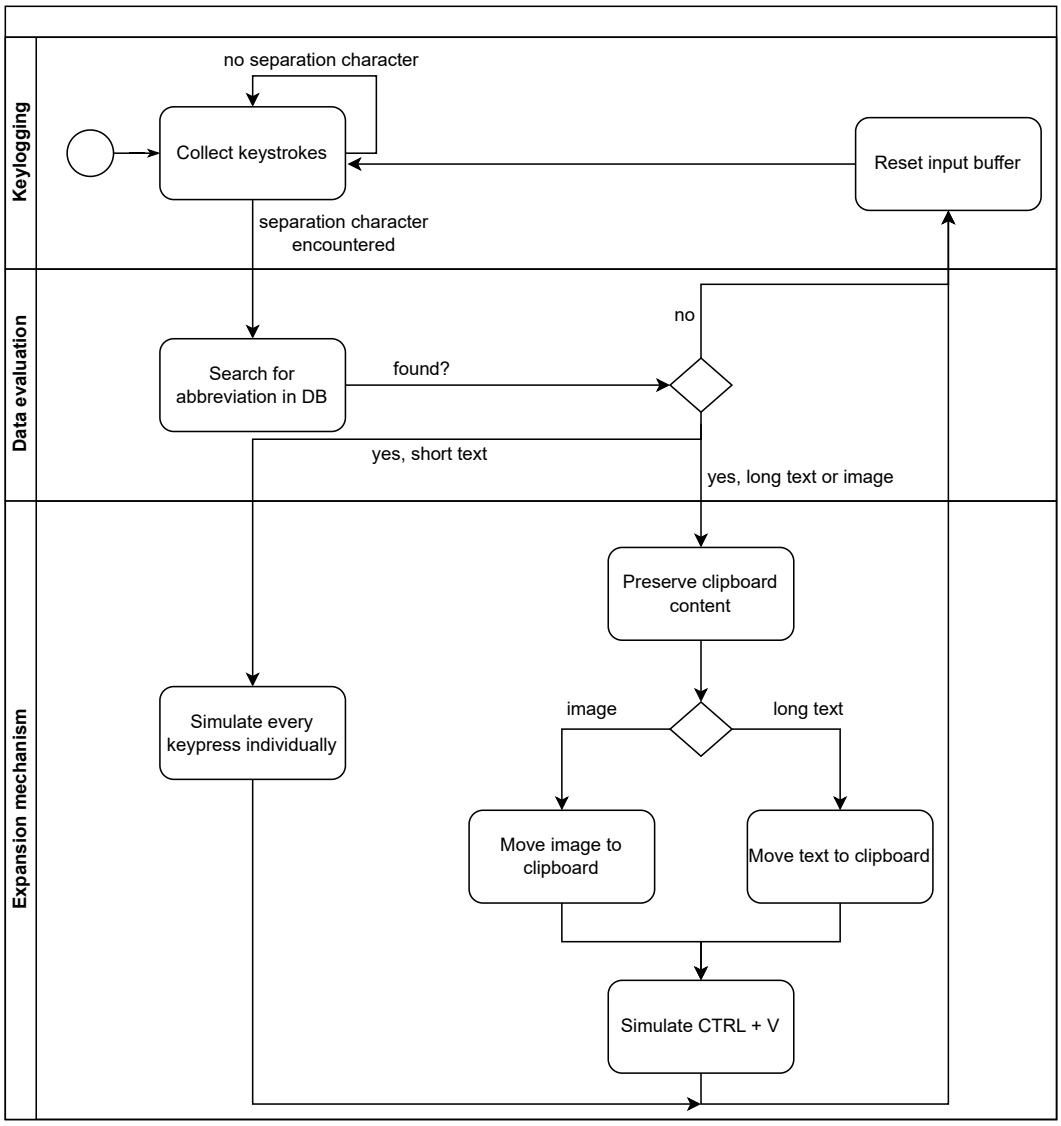


Figure 4: Inner workflow from keystroke recognition over searching in the database to expanding different kinds of results.

Collecting keystrokes is the entry point to a text expansion, as can be seen in Figure 4. As long as no separation character, i.e., whitespace, tab or return, is encountered, all alphanumerical characters are appended to an input buffer. In contrast, special characters such as CTRL and ALT are ignored intentionally since they do not contribute to an abbreviation.

Once a separation character is recognized, a **search for the entered abbreviation** in the database is performed by passing the input buffer content to *Typinator Core*. In case the ab-

breviation is found in a set, the corresponding expansion gets passed back to the caller. There, *Typinator for Windows* has to distinguish between three different cases: (1) short texts with less than or equal 50 characters, (2) long texts with more than 50 characters and (3) images. The different expansion types determine if either the clipboard must be used or the keystrokes are simulated separately due to performance reasons.

By simulating every keypress individually, we can save some time with regards to performance. Through empirical evaluation, *Ergonis* found out that especially for short texts with less than 50 characters, typing the characters sequentially is faster than using the clipboard.

Preserving the clipboard content is only relevant if the expansion is a long text or an image. Since the user might already have content inside the clipboard when *Typinator* wants to use it, its contents must be temporarily kept in-memory during this short period of time. Afterwards, it must be restored such that the user does not even notice that the clipboard was used internally.

Moving an image or text to the clipboard is conceptually the same but requires some different parameters when calling Win32 API functions (e.g., for data format). After moving the desired content to the clipboard, **simulating CTRL + V** is performed to tell the operation system to paste the newly added text or image to the cursor's current position. This logic is very much similar to physically pressing CTRL and V simultaneously.

Last but not least, **resetting the input buffer** must be performed in order to restart the whole process and being able to listen to new inputs again.

4 Implementation

While Section 3 discussed the composition of *Typinator for Windows* and how it functions in principle, this section describes the actual realization and which technologies were used for said purpose.

First, we have a look at the used frameworks and necessary technologies and why they are suitable for implementing *Typinator for Windows*.

Second, we dive into P/Invoke which provides classes and attributes for calling DLL functions. This is especially useful for communicating with the operating system.

Last, the core features of *Typinator for Windows* are discussed: collecting keystrokes, evaluating keyboard inputs and the expansion mechanism.

4.1 .NET as Software Platform

When it comes to working with the Windows operating system, .NET comes hand in hand. C# provides an easy-to-handle interface for using DLLs and is thus ideal for working with unmanaged code. Furthermore, .NET is preinstalled on Windows and updates do not have to be managed by the user manually which makes maintenance easier than with other frameworks.

Although *Typinator for Windows* does not have a user interface, it is built with **Windows Presentation Foundation (WPF)**. *Typinator for Windows* relies on another important WPF feature instead: the system tray. Additionally, WPF's lifecycle hooks and classes are quite useful for creating applications such as *Typinator for Windows*.

Yet, using the **system tray** from a WPF context requires imports of the the `System.Windows.Forms` namespace. Nevertheless, the classes `ContextMenuStrip` and `NotifyIcon` can be nicely integrated without any further ado. Only bundling the Windows Forms DLL must be ensured when building *Typinator for Windows*. We could have also created *Typinator for Windows* as a Windows Forms application but WPF is definitely the state of the art nowadays.

4.2 P/Invoke - Calling DLL Functions

Platform invoke, or short P/Invoke, is a technology designed to enable working with unmanaged libraries from within managed code. All required C# attributes and classes are inside the namespaces `System` and `System.Runtime.InteropServices`[4].

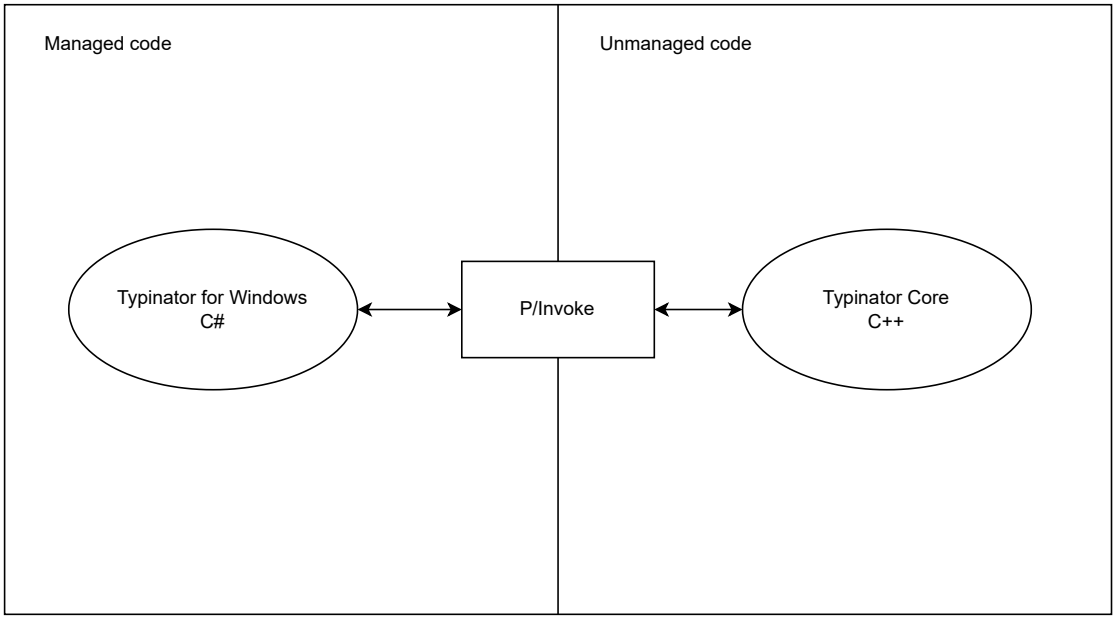


Figure 5: Simplified overview of the functional principle of P/Invoke in the context of *Typinator for Windows*

As can be seen in Figure 5, P/Invoke is a bridge between managed code of a .NET language (e.g., C#, Visual Basic or F#) and some unmanaged libraries. With regards to this thesis, P/Invoke is used for utilizing User32.dll of the Win32 API and *Typinator Core*.

4.2.1 Utilizing User32.dll

```
1 [DllImport("user32.dll", SetLastError = true)]
2 internal static extern uint SendInput(uint nInputs, Input[] pInputs, int cbSize);
3
4 [DllImport("user32.dll", SetLastError = true)]
5 internal static extern int GetAsyncKeyState(Int32 i);
6
7 [DllImport("user32.dll")]
8 internal static extern IntPtr GetForegroundWindow();
```

Listing 1: Used functions of the `Win32 API` for writing to the keyboard, collecting keystrokes and checking the active window.

The function `SendInput` is for simulating keystrokes one by one on the keyboard. Since it it is part of an unmanaged library, we have to supply the number of elements in the `pInputs` array as first function parameter. Second, the actual `pInputs` array is passed in and last, the size of

the `Input` struct as can be seen in Listing 1 (line 2). The `Input` struct itself consists of an input type (which always is set to `Keyboard` for *Typinator for Windows*) together with the key to be typed and flags that indicate whether the key should be pressed or released.

Next, by calling `GetAsyncKeyState` with a certain keycode we receive the state of the key on the keyboard (line 5). The method's return value signalizes that the key is either pressed (by a return value of 32769) or that the key is not pressed (by a return value of 0).

Last but not least, `GetForegroundWindow` is used for fetching a `Pointer` to the current active window. It does not need any function parameters since the operating system is aware of the currently running application (line 8). The returned `IntPtr` can be used for determining whether the user has changed the active window which is relevant for resetting the input buffer. After every successful expansion or recognition of a whitespace character or in case the user changes the active window, the input buffer gets cleared as we do not want to carry inputs longer than we have to. A way how to detect window changes is given in Listing 2.

```
1 // check if window changed and reset input
2 IntPtr currentWindow = DLLFunctions.GetForegroundWindow();
3 if (currentWindow != lastActiveWindow) {
4     lastActiveWindow = currentWindow;
5     INPUT_TEXT = "";
6     continue;
7 }
```

Listing 2: The Win32 API provides a simple feature for getting a pointer to the active window. By comparing this window reference to the previous one after every keystroke we can reset the input buffer accordingly in case of a detected change.

4.2.2 Utilizing *Typinator Core*

```
1 [DllImport(@"\lib\TypinatorCore.dll", SetLastError = true, CharSet = CharSet.
      Unicode)]
2 internal static extern int set_typinator_base_dirs(string local_dir, string
      synchronised_dir, string host_uuid);
3
4 [DllImport(@"\lib\TypinatorCore.dll", SetLastError = true)]
5 internal static extern int sync_typinator_model(string input_json, StringBuilder
      output_json);
```

Listing 3: Currently available functions of *Typinator Core* for setting the synchronised and local set directories, as well as to modify and fetch available sets.

Typinator Core manages shared and private sets. For this purpose, `set_typinator_base_dirs` is called during the initialization phase of *Typinator for Windows* and receives (1) the absolute file path of the local/private set directory and (2) the absolute file path of the shared set directory. Additionally, *Typinator for Windows* has to provide (3) the host UUID in order to track modifications for version management of sets as shown in Listing 3 (line 2). `sync_typinator_model` is capable of updating sets via the parameter `input_json` and of obtaining all available set data via `output_json` (line 5).

4.3 Collecting Keystrokes

Recognizing keystrokes is the first step during the expansion lifecycle of *Typinator for Windows*. Thus, it is necessary to use the existing operating system function `GetAsyncKeyState` from Listing 4 (line 4) in such a way that we can recreate a meaningful input string out of the clicked keys. This leads us to the need to filter out some special keys and also restricting the range of keys down to only those which really make sense.

4.3.1 Keycodes and Processing

The range of all possible keycodes is located between $0x01_{16}$ and $0xFE_{16}$. Since there are less than 254 keys on a keyboard, not all of the available keycodes are of use which requires filtering of unused keycodes as shown in Listing 4 (line 2). Filtering is done by iterating over all relevant keycodes which are situated in the numeric ranges:

- Special keys - $0x01_{16}$ to $0x2F_{16}$
  Only $0x11_{16}$ (CTRL) and $0x12_{16}$ (ALT) are relevant for *Typinator for Windows* as it can be seen in Listing 4 (line 9 to line 14).
- Numbers - $0x30_{16}$ to $0x39_{16}$
- Letters - $0x41_{16}$ to $0x5A_{16}$
- Numpad numbers - $0x60_{16}$ to $0x69_{16}$

All keycodes above $0x69_{16}$ have no use for *Typinator for Windows* and are therefore not iterated.

```
1  // Scan over all possible input keys
2  for (int keyCode = 1; keyCode <= MAX_KEY_CODE; keyCode++) {
3      // Get state of key
4      int keyState = DLLFunctions.GetAsyncKeyState(keyCode);
5      if (keyState == PRESSED_STATE) {
6          char c = (char)keyCode;
7
8          // check if a modifier (CTRL or ALT) was encountered -> ignore next char
9          if (ModifierPreceeding) {
10             ModifierPreceeding = false;
11             continue;
12         }
13         // check if current character is a modifier
14         ModifierPreceeding = c == STRG_KEY_CODE || c == ALT_KEY_CODE;
15
16         // append text while user is typing
17         if (char.IsLetterOrDigit(c)) {
18             INPUT_TEXT += c;
19         }
20     }
21 }
```

Listing 4: The range of all relevant keycodes is polled sequentially for obtaining the written text without special keys. These special keys require separate handling since they are not part of an abbreviation in *Typinator for Windows*.

4.3.2 Get Key State Asynchronously

When looping over the relevant keycodes we have to evaluate their state in order to determine the written text. As shown in Listing 4, the loop variable is passed to `GetAsyncKeyState` which returns an `int` describing the state of the key (line 4). We can check whether the key was pressed by comparing the state to the value 32769 being held in the static variable `PRESSED_STATE` (line 5).

An edge case of the keystroke procedure is **handling modifier keys** (CTRL and ALT) because they are certainly not of textual nature and therefore should not end up inside the input buffer. Thus, we have to check if the last pressed key was such a modifier and skip to the next one in this case (line 9 to line 11). If no such key is preceding, we finally have to check if the current key is CTRL or ALT for the next loop iteration (line 14). The last step is to only **add letters or digits** to the input buffer by using the function `char.IsLetterOrDigit(char c)` (line 17 to line 19).

4.4 Evaluating Keyboard Inputs

The second step of the expansion lifecycle is initiated after filling the input buffer. *Typinator for Windows* must decide whether to expand a possible abbreviation at the cursor's current position or not. This is performed according to one of two different techniques.

The user can either choose to evaluate whether an existing abbreviation has been typed in after every whitespace or after every character. Additionally, performance has to be considered here as well since querying the database after every character can result in a loss of performance, especially for large sets (e.g., a set with 500 000 abbreviations and expansions which is used by a real-world client).

Evaluating after whitespaces is thus the default case and initiates the search mechanism solely after a whitespace character has been encountered. We can check for a whitespace by calling `char.isWhitespace(char c)`. Nevertheless can the user switch to **evaluation after every character**. This is convenient since the user does not have to type a whitespace. Yet, this can also lead to conflicts. Table 3 shows an exemplary set definition.

	Abbreviation	Expansion
1.	tel	07262 56789
2.	tel2	06641234567890

Table 3: Both abbreviations start with the same three characters and will be expanded to a phone number.

When evaluating the input after every whitespace we do not have any difficulties deciding which abbreviation to choose from the two available ones. Since the whitespace functions as separator we can clearly decide whether the user typed *tel* or *tel2* and expand accordingly. Yet, if we perform character by character evaluation, we cannot trivially decide the abbreviation exactly. For now, the database is queried after every keystroke and thus, the input *tel* is recognized and the corresponding phone number gets expanded. Even if the user wanted to type *tel2* it would not be possible as *tel* would always be recognized first.

4.5 Expansion Mechanism

The next step after evaluating the input and checking whether the user typed an abbreviation stored in one of the sets is to actually expand the underlying text or image. First, removing the user's input (the abbreviation) is performed. Second, in case of expanding via the clipboard, the clipboard content is preserved. Last, the expansion is done via the clipboard or by simulating every keypress.

4.5.1 Removing Input

First, the typed abbreviations must be removed from the user's current editor. Therefore, we utilize the Win32 API function `SendInput` and create the required `Input` objects as shown in Listing 5. For signalizing keyboard input, the parameter `type` must be set to `InputType.Keyboard` but could in principle also be `InputType.Mouse` or `InputType.Hardware` (line 6 and line 14). As it can be seen in Listing 5, we have to add two entries to the `Input` array for every keypress (line 5 and line 13). `wVk` stands for the virtual key code of the desired key which is $0x08_{16}$ for DELETE (line 9 and line 17). For the first keypress, we just specify the keycode, for the second one we also add a flag for releasing the key in `dwFlags` (line 18). Finally, `SendInput` takes the `Input` array, its amount of elements and the size of the `Input` struct (line 23).

```
1  private static void RemoveCharacters(int amount) {
2      // Remove as many characters as stated by amount
3      Input[] removeInput = new Input[amount * 2];
4      for (int i = 0; i < removeInput.Length; i += 2) {
5          removeInput[i] = new Input {
6              type = (int)InputType.Keyboard,
7              u = new InputUnion {
8                  ki = new KeyboardInput {
9                      wVk = DELETE_KEY_CODE
10                 }
11             }
12         };
13         removeInput[i + 1] = new Input {
14             type = (int)InputType.Keyboard,
15             u = new InputUnion {
16                 ki = new KeyboardInput {
17                     wVk = DELETE_KEY_CODE,
18                     dwFlags = (uint)(KeyEventF.KeyUp)
19                 }
20             }
21         };
22     }
23     _ = DLLFunctions.SendInput((uint)removeInput.Length, removeInput, Marshal.
       SizeOf(typeof(Input)));
24 }
```

Listing 5: We create an array of `Input` which contains as many `DELETE` keystrokes as the input has characters. It is crucial that for every keypress we add two elements to the `Input` array: One for pressing the key and one for releasing it again. Last, we call the Win32 API function `SendInput` which executes the keystrokes.

4.5.2 Preserving Clipboard Data

For long texts and images we also have to preserve the current clipboard content. Users expect the clipboard to still contain the original content they added themselves and not that *Typinator for Windows* interferes and creates an inconsistent state.

```
1  private static void WriteByClipboard(string text) {
2      Thread staThread = new(delegate () {
3          // retrieve whatever is inside the clipboard currently
4          IDataObject originalClipboardContent = Clipboard.GetDataObject();
5
6          // wrap text expansion as dataobject
7          IDataObject data = new DataObject();
8          data.SetData(text);
9
10         // move expansion to the clipboard and paste it
11         Clipboard.SetDataObject(text, true);
12         PasteFromClipboard();
13
14         // timeout in order to avoid race condition when pasting
15         Thread.Sleep(CLIPBOARD_TIMEOUT);
16
17         // move previous content to clipboard
18         Clipboard.SetDataObject(originalClipboardContent, true);
19     });
20     // clipboard must be accessed via STA-thread
21     staThread.SetApartmentState(ApartmentState.STA);
22     staThread.Start();
23     staThread.Join();
24 }
```

Listing 6: The listing shows the whole expansion process using the clipboard with the intermediate step of preserving the original clipboard content. It starts a new Thread which temporarily saves the clipboard content, moves the expansion to the clipboard, triggers a PASTE operation and finally restores the original clipboard content.

The reason for executing the preserving step (line 4) and expansion step (line 11 and line 12) in a separate `Thread`, as shown in Listing 6, is due to restrictions of WPF. It enforces that the `Clipboard` class can only be accessed from a Single-Threaded Apartment Thread (STA Thread) which is normally the main thread of a WPF application. Since collecting keystrokes and the expansion mechanism themselves run in a separate asynchronous `Task`, we thus start an additional `Thread` using the `ApartmentState.STA` state (line 2, line 21 and line 22).

4.5.3 Expanding by Clipboard

As shown in Listing 6, using the clipboard for expansions is performed in between the preserving logic. For **long texts** (> 50 characters), we only have to wrap the input `string` as `IDataObject`. Through empirical evaluation by *Ergonis*, a threshold of 50 characters has proven suitable. For **images** we have to define some additional options as shown in Listing 7. We wrap the image as `IDataObject` to bring it into a universal format for the clipboard and also specify the data format `Bitmap`. (line 3) The last parameter with value `false` stands for `autoConvert` and prohibits the clipboard from converting the content to another format on retrieval [5].

```
1 private static void MoveImageToClipboard(string imageUrl) {
2     IDataObject data = new DataObject();
3     data.SetData(DataFormats.Bitmap, Image.FromFile(imageUrl), false);
4     ...
5 }
```

Listing 7: Similar to textual expansions, images have to be put into a universal wrapper but further parameters have to be supplied. The preservation mechanism and expansion mechanism are the same as shown in Listing 6, except for the data object containing an image.

For pasting the clipboard content we simulate a CTRL + V with the `SendInput` function, as shown in Listing 8. First, we press CTRL (line 8) followed by a V (line 16) by setting `wVk` to their respective keycodes and then release the keys in reverted order (line 24 and line 33). This is the way to simulate simultaneous pressing of two keys.

```
1  public static void PasteFromClipboard() {
2      // Write expansion text
3      Input[] input = {
4          new Input {
5              type = (int)InputType.Keyboard,
6              u = new InputUnion {
7                  ki = new KeyboardInput {
8                      wVk = CTRL_KEY_CODE
9                  }
10             }
11         },
12         new Input {
13             type = (int)InputType.Keyboard,
14             u = new InputUnion {
15                 ki = new KeyboardInput {
16                     wVk = V_KEY_CODE
17                 }
18             }
19         },
20         new Input {
21             type = (int)InputType.Keyboard,
22             u = new InputUnion {
23                 ki = new KeyboardInput {
24                     wVk = V_KEY_CODE,
25                     dwFlags = (uint)(KeyEventF.KeyUp)
26                 }
27             }
```

```
28          },
29          new Input {
30              type = (int)InputType.Keyboard,
31              u = new InputUnion {
32                  ki = new KeyboardInput {
33                      wVk = CTRL_KEY_CODE,
34                      dwFlags = (uint)(KeyEventF.KeyUp)
35                  }
36              }
37          }
38      };
39
40      _ = DLLFunctions.SendInput((uint)input.Length, input, Marshal.SizeOf(typeof(
        Input)));
41  }
```

Listing 8: The PasteFromClipboard method performs a paste operation by simulating sequentially pressing the CTRL and V keys. The keystrokes are then executes by the Win32 API function `SendInput`.

4.5.4 Expanding by Simulating Keypresses

For shorter texts (≤ 50 characters), we virtually type each character in order to save some time by not using the clipboard. This can again be done with `SendInput` from the Win32 API. The big difference to our other methods that use `SendInput` are the options which are supplied in the `KeyboardInput` object as shown in Listing 9.
For writing arbitrary unicode characters, we first have to set the parameter `wVk`, the virtual key code[2], to 0 in order to signalize that the key is not defined in a hardcoded way by a specific virtual key code (line 13). To be concise, we take `char` by `char` of the `string text` (line 1) and pass it to the `wScan` parameter (line 14). By doing so, the `SendInput` function takes care of typing the character of `wScan`. But this only works in combination with adding `KeyEventF.Unicode` to the `dwFlags` parameter (line 28).

```
1   private static void WriteCharacters(string text) {
2       // Write expansion text
3       List<Input> input = new();
4       foreach (char c in text.ToCharArray()) {
5           // we have to send KeyDown and KeyUp in order to print multiple characters
6           input.Add(new Input
7           {
8               type = (int)InputType.Keyboard,
9               u = new InputUnion
10              {
11                  ki = new KeyboardInput
12                  {
13                      wVk = 0,
14                      wScan = c,
15                      dwFlags = (uint)KeyEventF.Unicode
16                  }
17              }
```

[2] https://learn.microsoft.com/en-us/windows/win32/inputdev/virtual-key-codes

```
18          });
19          input.Add(new Input
20          {
21              type = (int)InputType.Keyboard,
22              u = new InputUnion
23              {
24                  ki = new KeyboardInput
25                  {
26                      wVk = 0,
27                      wScan = c,
28                      dwFlags = (uint)(KeyEventF.KeyUp | KeyEventF.Unicode)
29                  }
30              }
31          });
32      }
33
34      _ = DLLFunctions.SendInput((uint)input.Count, input.ToArray(), Marshal.SizeOf(
        typeof(Input)));
35  }
```

Listing 9: We iterate over all characters of the expansion and set the following parameters: `wVk` is 0 since we specify the character to type by the attribute `wScan`. For `wScan` we can directly pass the `char` and do not have to know its virtual key code. Last, `dwFlags` contains a flag for allowing any Unicode character.

5 Evaluation / Usage

This section covers the practical application of *Typinator for Windows*. First, a brief overview of the application's embedding into the Windows system tray is shown. Following, two exemplary workflows (of a software developer and a sales person) from abbreviation to expansion are worked through.

5.1 System Tray Icon and Context Menu

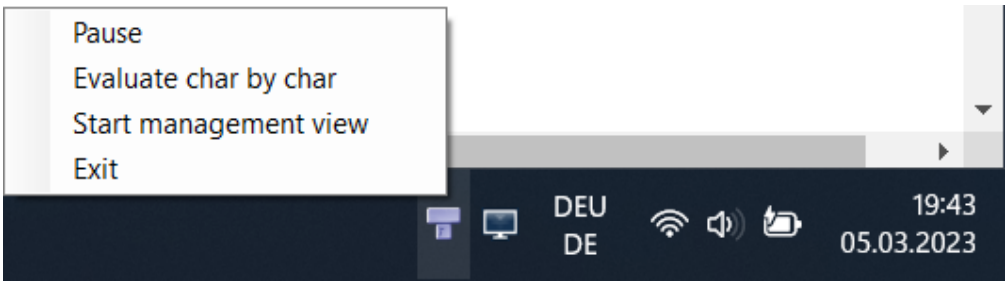


Figure 6: The icon of *Typinator for Windows* is displayed in the system tray. A right click opens the context menu.

The icon itself only shows the running state of *Typinator for Windows* but has no effect when left-clicking on it. Other than that, a right-click opens the menu which allows interaction with the application, as shown in Figure 6.

1. Pause: Stops the app from listening to keystrokes
   (a) Context menu option *Pause* is changed to *Resume*.
   (b) Toggling between Paused and Running state possible.
2. Evaluate char by char: Check DB for current input after every typed character
   (a) Context menu option *Evaluate char by char* changes to *Eval after whitespace*.
   (b) Toggling between the two evaluation modes possible.
3. Start management view: Starts a local webserver and opens the Management UI
4. Exit: Shutdown the application

5.2 Software Development Workflow

	Abbreviation	Expansion
1.	newhtml	<!DOCTYPE html><html><head><title>New HTML page</title><script></script></head><body></body></html>
2.	centered	TODO: enter html

Table 4: A software developer sometimes comes across the problem to create a new HTML document and to center a div. For not having to research these things in the internet, a *Typinator* set has been created which automatically inserts the required code.

First, the software developer opens a code editor, e.g., Visual Studio Code, and enters the predefined abbreviation *newhtml* from Table 4 as shown in Figure 7.

```
<!DOCTYPE html>
<html>
  <head>
    <title>New HTML page</title>
    <script></script>
  </head>
  <body>centered</body>
</html>
```

Figure 7: This figure shows a HTML page skeleton with the unexpanded abbreviation *centered* inside the body tag.

Now, the software developer wants to add a `div` element which is horizontally centered. Therefore, the abbreviation *centered* gets expanded by *Typinator for Windows* as shown in Figure 8. Instead of typing 179 characters manually, only 15 characters have to be entered via the keyboard which results in a decrease of nearly 92%.

```
<!DOCTYPE html>
<html>
  <head>
    <title>New HTML page</title>
    <script></script>
  </head>
  <body>
    <div style="margin: auto; width: 50%">TODO: enter html</div>
  </body>
</html>
```

Figure 8: After expanding *centered*, the HTML page contains a centered div element with a placeholder text.

5.3 Sales Workflow

	Abbreviation	Expansion
1.	dc	Dear Customer,
2.	off1	Thank you for your email. I'm happy to make you a special offer for a 10% discount which is valid for 15 days.
3.	ct	Please do not hesitate to contact me if there are any questions left.
4.	wbr	With best regards,
5.	name	Max Mustermann
6.	job	Ergonis Head of Sales

Table 5: The sales person in this scenario works for *Ergonis* and frequently gets in touch with potential new customers for the product *Typinator for Windows*. For that, a new *Typinator* set with commonly used phrases has been created.

For replying to the customer, the sales person opens an email client such as Outlook. *Typinator for Windows* is already running and has a predefined set according to Table 5. The sales person types the abbreviations as shown in Figure 9 and by entering a whitespace after each, the abbreviation gets replaced by the corresponding expansion from the predefined set.

Figure 9: This figure shows the abbreviations which the sales person has to enter inside the mail client. Simultaneously, *Typinator for Windows* inserts the underlying expansions when recognizing a whitespace character while the sales person is actively typing.

The resulting email is shown in Figure 10. Instead of typing 243 characters manually, the sales person only has to enter 18 characters which decreases the typing effort by approximately 93%.

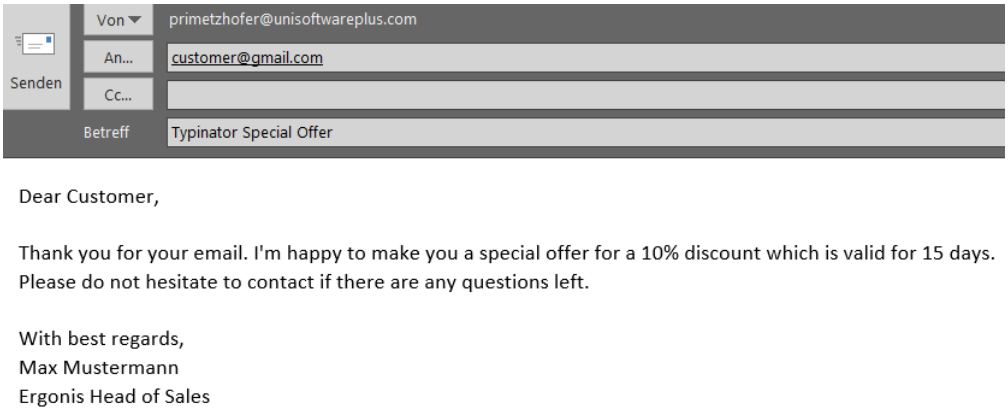


Figure 10: All abbreviations have been expanded by *Typinator for Windows* according to the predefined set and thus the sales person is able to send the mail to the customer.

6 Related Work

This section covers an overview of competing products on the market and their differences as well as similarities in comparison to *Typinator for Windows*.

6.1 Competing Products

Typinator for Windows' biggest competitor is *TextExpander* [7]. *TextExpander* for instance offers expanding texts and images or sharing sets via a *TextExpander* user account. Additionally, users can also invoke scripts written in JavaScript or AppleScript from within expansions [7]. Furthermore, *TextExpander* is also available for iOS and as Google Chrome extension. An economic limitation of *TextExpander* are the monthly costs, which depend on the amount of concurrent users, due to its subscription-based payment model.

Next, *FastKeys* [1] is a Windows-only text expansion tool with focus on automating many different aspects of working with Windows. Users can, besides some other features, (1) expand abbreviations just as *Typinator for Windows*, (2) add custom keyboard shortcuts for opening certain applications, (3) record mouse gestures and execute an action when the movement pattern is recognized and (4) a clipboard manager which keeps track of the clipboard's content history. *FastKeys* offers three different license types for personal, professional or enterprise customers. Only lifetime licenses are sold and therefore no recurring payments are required.

Following, *aText* [6] is a text expansion tool with a simplistic user-interface but underbids the other competitors by its rather low price of $4.99 for a lifetime license. Feature-wise *aText* is alike the other products but for enterprise customers, it might not be an ideal solution since it is only maintained by a single person.

All in all, it must be said that most text expansion tools meanwhile have a very similar set of features and thus the main selling point is interoperability among multiple platforms.

7 Conclusion

This thesis has shown how a basic version of *Typinator for Windows* can be implemented with modern technologies. Nevertheless, we have only considered a core of all available features already available in MacOS *Typinator* as otherwise this would have resulted in a scope too big for a Bachelor's thesis. At first, introduced underlying concepts such as abbreviations, expansions and sets and why we have to redevelop *Typinator* for Windows. Second, we had a closer look at the *Typinator* architecture and which components *Typinator for Windows* interacts with. We discussed the flow from recognizing an abbreviation by collecting keystrokes to expanding texts or images.

Next, we saw crucial parts of the implementation for a better understanding of how to utilize C# and the WPF framework to create *Typinator for Windows*. Not only does C# perform well with unmanaged code such as the *Typinator Core* library, it also makes developing Windows-specific applications much easier due to its surrounding .NET ecosystem. Additionally, we gained some insights to working with the Windows operating system for obtaining the currently active window, recognizing keystrokes, writing to the keyboard and using the global clipboard.

Finally, *Typinator for Windows* is a big chance for *Ergonis* to bring an already successful and profitable MacOS product to many more potential users. It provides an opportunity to open up to bigger markets and make the *Typinator* toolset thrive even more.

List of Tables

List of Figures

Listings

We iterate over all characters of the expansion and set the following parameters: `wVk` is 0 since we specify the character to type by the attribute `wScan`. For `wScan` we can directly pass the `char` and do not have to know its virtual key code. Last, `dwFlags` contains a flag for allowing any Unicode character. 20

References

[1] F. Automation. Fastkeys features. https://www.fastkeysautomation.com/. [Online; accessed 08-March-2023].

[2] Ergonis. Typinator 8.14. https://www.ergonis.com/products/typinator/history.html, 2023. [Online; accessed 03-March-2023].

[3] Ergonis. Typinator learning center. https://ergonis.com/typinator/learn, 2023. [Online; accessed 08-March-2023].

[4] Microsoft. Platform invoke (p/invoke). https://docs.microsoft.com/en-us/dotnet/standard/native-interop/pinvoke, 2022. [Online; accessed 05-March-2023].

[5] Microsoft. Setdata method. https://learn.microsoft.com/en-us/dotnet/api/system.windows.idataobject.setdata?view=windowsdesktop-7.0#system-windows-idataobject-setdata(system-string-system-object-system-boolean), 2023. [Online; accessed 05-March-2023].

[6] T. K. Nam. atext text automation. https://www.trankynam.com/atext/. [Online; accessed 08-March-2023].

[7] I. TextExpander. Textexpander features. https://textexpander.com/features. [Online; accessed 08-March-2023].