FATEC – FACULDADE DE TECNOLOGIA DE MOGI DAS CRUZES

Multikeys

Mogi das Cruzes - SP

2017

FATEC – FACULDADE DE TECNOLOGIA DE MOGI DAS CRUZES

Multikeys

This essay was presented to FATEC Mogi das Cruzes as a requirement for course graduation.
Under guidance of Prof. Dr. Bruno Marques Panccioni.

Mogi das Cruzes - SP

2017

TAKAHASHI, Rafael Kenji. ***Multikeys***: Application for Key Remapping in Windows. Mogi das Cruzes, 2017. 62 pag. Conclusion Work Project, Faculdade de Tecnologia de Mogi das Cruzes.

ABSTRACT

There are many situations where a desktop user needs to insert special characters, which is especially true for users who frequently deal with less common symbols, such as mathematicians or linguists. Furthermore, keys are frequently used to trigger shortcuts and sequences of keystrokes in games and apps. A possible solution for these cases would be to use an additional keyboard to extend the user's options according to the user's needs, even allowing for inputting multiple languages at once. However, even though there is software, such as text editors, capable of inserting a wide array of Unicode symbols, there does not exist a simple solution for changing the behavior of keys in specific keyboards connected to the computer, making it impossible for the user to implement this proposed solution. This work is a qualitative research that details the development of a piece of software using the Raw Input API and a keyboard hook to manipulate keyboard input in order to remap actions, including Unicode input, to keys in specific keyboards, and the development of a graphical user interface for easily configuring custom layouts. An iterative design was chosen for development, employing the principles of agile software development methodology, as well as some UML diagrams. Because of the frequent interactions with the Windows API, the module responsible for intercepting and changing user input was implemented in the C++ language, while C# and WPF were used for coding the user interface. XML was used for communication and data persistence. The software developed in this work has many potential applications, such as typing in multiple languages, and using a second keyboard for shortcuts.

Keywords: keyboard, remapping, Windows API, Raw Input, C++, C#, WPF, XML.

LIST OF FIGURES

LIST OF ABBREVIATIONS AND ACRONYMS

ABNT – Associação Brasileira de Normas Técnicas, Brazilian Association of Technical Norms;

API – Application Programming Interface;

ASCII – American Standard Code for Information Exchange;

BSD – Berkeley License Distribution, an open-source software license;

DLL – Dynamic-Link Library;

GoF – Gang of Four, the authors Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides.

IBM – International Business Machines;

IBM PC – IBM Personal Computer;

IBM XT – IBM Extended Technology;

IDE – Integrated Development Environment;

ISO – International Organization for Standardization;

MS-DOS – Microsoft Disk Operating System;

MSDN – Microsoft Developer Network, a website where documentation for official Microsoft technologies can be found, including .NET and the Windows API;

.NET – Software framework developed by Microsoft primarily for the Windows OS;

DBMS – Database Management System;

SGML – Standard Generalized Markup Language;

UML – Unified Modeling Language;

UTF – Unicode Transformation Format;

WinAPI – Shortened form of Windows API;

WndProc – Shortened form of Window Procedure;

WPF – Windows Presentation Foundation, a graphic subsystem made by Microsoft for a window-based UI;

XML – Extensible Markup Language, a subset of SGML;

INDEX

INTRODUCTION

1.1 PROBLEM INTRODUCTION

Computer users frequently need to insert special characters, depending on the language or situation. For most users, the many layouts offered by the OS are sufficient to fulfill those needs.

However, there exist more specialized characters, such as mathematical symbols or phonetic alphabet letters that do not have an easy way of being inserted. Moreover, we should also consider the need for shortcut keys for games and applications, and language-specific letters. Some software allows for remapping specific keys on the keyboard with custom behavior, but there does not exist a solution for customizing behavior depending on which keyboard a signal originated from.

Though it’s possible to combine more than one piece of software to achieve this goal, the process requires knowledge of scripting languages, and the creation of custom layouts is impractical for very large layouts.

This means that users who make frequent use of special characters don’t have an easy way to customize a keyboard layout according to their needs, and there isn’t an easy way of remapping the keys of multiple keyboards to expand functionality either.

1.2 OBJECTIVES

1.2.1 General Objective

Develop and publish a software tool capable of assigning custom behavior to specific keys, distinguishing between input from multiple keyboard devices connected to the computer.

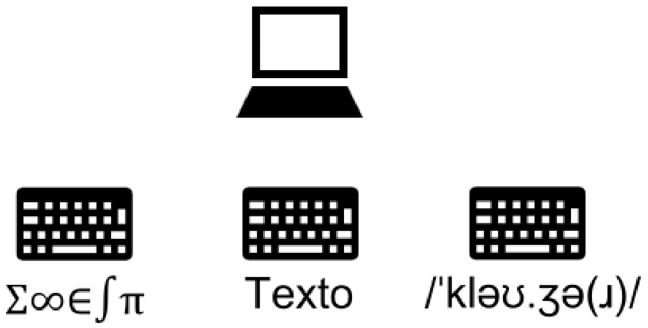
1.2.2 Specific Objectives

- Investigate and explain the details of keyboard input interception in the Windows OS;
- Develop the application’s functionalities;
- Provide a user interface for editing layouts;

1.3 JUSTIFICATION

Remapping keys is a common solution for those who want to expand the functionality of their keyboards; for convenience and productivity, a second input device would be most useful for certain users, such as those who regularly need to insert mathematical characters, or those who would like to automate long and repetitive tasks (like opening a website or press sequences of characters and shortcuts) with a single keypress. Using multiple keyboards would also allow the user to type in more than one language at once.

Figure 1: Using Multiple Keyboards



Source: Own authorship (2017)

According to Wells (2007), special character input is currently not an entirely resolved problem. There already exists software to aid the input of Unicode characters, but each has its own limitations.

One of the many limitations to consider is that the OSes of the Windows NT family do not distinguish input from multiple keyboards connected to the computer, and thus, do not allow for software to respond differently to input from different keyboards. Currently available tools for remapping keys (like KeyTweak and AutoHotKey) do not offer the feature to distinguish between keyboards.

Furthermore, apps with advanced features (like AutoHotKey and LuaMacros) require the user to use scripting languages even for simple tasks like remapping a key, limiting their use and the target audience.

This software aims to respond to those necessities, not only multiplying the amount of actions the user can execute from their input devices, but also giving them an easy-to-use tool for remapping keyboards.

1.4 CHAPTER STRUCTURE

The “Introduction” chapter lays out the objectives and justifications of this essay, as well as the problem to resolve.

The second topic “Methodology” lays out the structuring of the project development.

The third topic “Theoretical Background” focuses on the topics that were researched for the project’s development, as well as for writing this essay. It contains the following subtopics:

- Architecture of Windows Applications: Describes the architecture of the Windows NT family operating systems and their interaction with the programs executing in them.
- Messages: On messages and message queues, a major topic in the Windows architecture.
- Keyboard Input: On keyboard input in Windows; contains the theoretical background for intercepting keyboard messages;
- Input Simulation: On simulating messages to execute input on behalf of the user;
- Existing software: A short discussion on existing software available with similar features, and their limitations in the specific scenario presented in this essay;

The fourth topic “Development” documents the problems encountered throughout software development, and the solutions to those problems. In this topic, we also lay out the project’s architectural choices. It contains the following subtopics:

- Architecture: On the project’s architecture, and justifications for architectural choices.

- Employed Technologies: On the technologies, patterns and programming languages used for this project’s development, and the reasons for choosing them.

- Message manipulation: On the program’s approaches for manipulating messages;

- Obstacles: On the relevant problems encountered during development, as well as the resolutions for each.

- Input Simulation: On the implementation of user input simulation.

- Layout Editing: On the user interface’s components, problems and solutions.

- Limitations: On the remaining issues not resolved by Multikeys, either by technical unfeasibility or by falling outside the project’s scope.

- Publishing: A brief explanation on how this software was published.

The topic “Final Remarks” wraps up this essay with a retrospective of all the relevant points, as well as what has been learned and whether the objectives were achieved, among others.

2 METHODOLOGY

This essay is a technical report and has resulted in the development of a desktop application. The development was broken up in the following steps:

- A bibliographic research on detecting and intercepting keyboard input, as well as simulating user input using the Windows API. This stage also included research on Unicode text encoding and manipulation, and also an in-depth investigation on the behavior of keyboard messages and mechanisms for message interception in Windows.

- Coding the program's features as a Windows C++ application (named Multikeys Core), using the theoretical background acquired during research.

- Developing a C# graphical application (named Multikeys Editor) for allowing the user to easily interact with the main program. Multikeys' architecture is such that the Core does not depend on the Editor to work.

Regarding the project's development, the system will be separated in modules and separated libraries and executables that may have their own requirements. As such, an agile and iterative approach was considered adequate to the development of this software. The following stages were identified:

1. Requirement analysis: At the start of every iteration, the system requirements were revised and updated. The requirements were based on the observed behavior of a standard Windows keyboard layout, including modifier keys and dead keys, which are discussed further in this essay.

2. Modeling: Consists in the general architecture of the program (the modules in C++ and C# and their communication, persisting data in XML and WPF interface), modeling with UML diagrams using StarUML and Microsoft Visio, XML validation with XSD instead of databases, and communication using messages.

3. Coding: Writing the code in accordance to its previously defined description, mainly in C++ and C# languages and the Visual Studio Community IDE.

4. Tests: The implemented components go through tests to determine their conformity to requirements, in particular the behavior of simulated keyboard layouts including modifier keys and many odd situations. The standard testing tools available in Visual Studio were used to automatically test parts of Multikeys Editor; however,

Multikeys Core still needed manual testing due to difficulties in implementing the tests for it.

These steps were repeated continuously and iteratively. The main identified iterations are:

- Iteration 1: Intercepting and interpreting data from keystroke messages; results in a system capable of intercepting and blocking user keypresses, extracting enough information to deduce which keyboard generated the message.

- Iteration 2: An internal model of keyboards and layouts; results in a system capable of blocking keystrokes from specific keyboards and responding with various actions, such as sending simulated Unicode input, or simulating a sequence of keystrokes.

- Iteration 3: Custom layout persistence; results in a system capable of reading all the necessary configuration from a file.

- Iteration 4: User interface; results in a graphical user interface for creating and editing custom layouts.

3 THEORETICAL BACKGROUND

3.1 - ARCHITECTURE OF WINDOWS APPLICATIONS:

According to the MSDN (2017), “Unlike MS-DOS-based applications, Windows-based applications are event-driven”. Instead of calling functions to obtain input, applications wait for messages addressed to them.

An event-driven system carries out actions according to external activities, instead of receiving calls and returning an answer. These events are broadcast asynchronously (without awaiting an answer) to whatever interested party wants to receive it, without a set order. Other features of event orientation are the absence of a stack call in favor of an event channel, and a more distant interaction between components, which for example, do not know each other’s method names (HOHPE, 2006).

Events correspond to state changes that influence the final result and are represented by event objects. Levina and Stantchev (2009) define the following terms in event-oriented architecture:

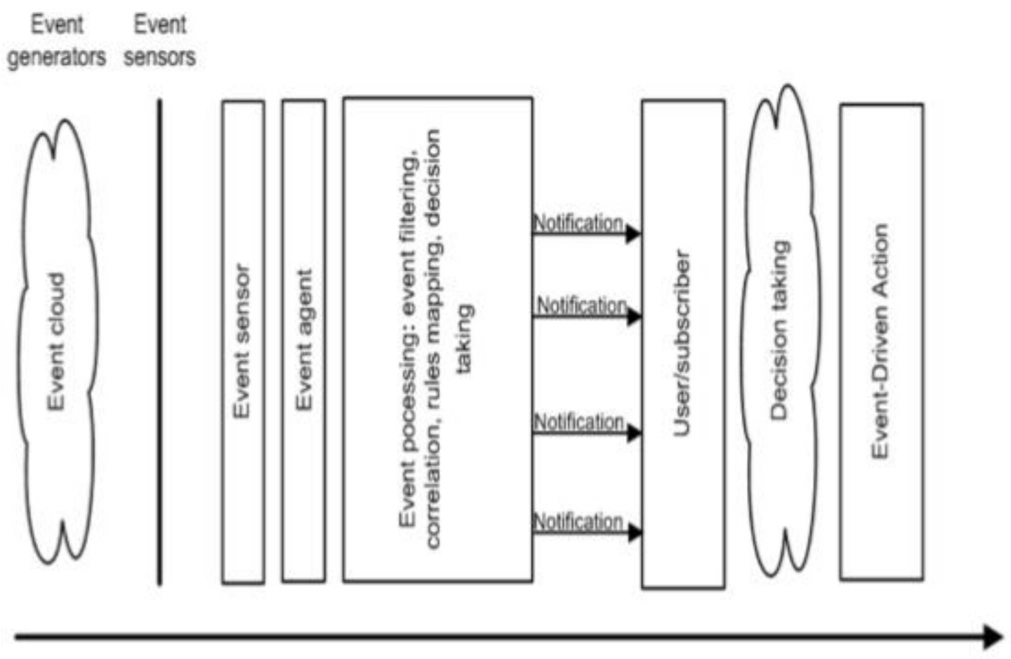
Event source: An abstract state change, internal or external to the application, that causes the creation of an event object.

Event sink: An object or element that’s interested in receiving messages to carry out some processing or for redirecting events.

Event agent: The component responsible for capturing and sending events.

These terms are summarized in **Error! Reference source not found.**.

Figure 2: Event-driven architecture



Source: LEVINA and STANTCHEV (2009)

The term “event” may be used to mean more than one concept in computer science, which may cause confusion (FOWLER, 2017). In this context, events represent the notion of event notification, which happens when messages are sent to notify parts of the system of the domain’s relevant occurrences.

3.1.1 Reactive Systems

According to Aceto et al. (2007), in contrast to the classical notion of a computer program as a black box receiving input and producing output, a reactive system responds to environment stimuli, without ever necessarily reaching an end point.

It must be noted that there’s a difference between reactive programming, which is event-oriented and focuses on data flow, and reactive systems, which is message oriented and focuses on resilience. Messages differ from events in that they specify a destination, while events merely represent facts that can be reacted to; because of that, message-oriented architectures place more emphasis in the components themselves and the communication between them rather than on facts and responses (BONÉR e KLANG, 2016).

Fowler (2017) uses “event message” to describe the communication between components to notify of a domain state change. Fowler (2006a) further states that though similarities exist between events and commands, events notify of state changes without concern to what the receives does with that information, while commands encapsulate a task to be executed.

A desirable trait of message-oriented reactive systems is resilience, the capacity of a system to stay responsive even after a failure and to avoid propagating errors to other components. Employing messages allows for weaker coupling between components and removes fault recovery responsibilities from the client (BONÉR and KLANG, 2016).

However, a negative consequence of this architecture’s weaker coupling is the difficulty in tracking event flows because of implicit collaborations; a particularly problematic question is event cascading, in which an event tree, with each event causing the next few, becomes extremely problematic to debug as the code increases in complexity (FOWLER, 2006b).

3.1.2 Real-Time Systems

According to Lipari and Palopoli (2015), “real-time systems are computational systems whose correctness depends not only on the correctness of the produced results, but also on the time at which they are produced”.

Real-time systems are a type of reactive system where, in addition to occurring in response to environment input stimuli, processing is subject to timely deadlines (FARINES, FRAGA and OLIVEIRA, 2000).

Stankovic (1992) specifies the difference between hard real-time systems, where the result is no longer useful after the deadline, and soft real-time systems, where the quality of the result is degraded after the deadline, but may be still worth executing; this is called the *strictness* of the deadline. According to Juvka (1998), hard real-time tasks are often critical and must be predictable and carefully planned. Several scheduling algorithms exist for ensuring that tasks have enough resources, including processing time, in order to be completed within the deadline.

3.2 MESSAGES

Because each process in Windows has its own memory space, pointers are limited to referencing memory addresses inside the same process; this contributes to the system’s robustness and security but requires additional mechanisms to permit inter-process communication (RICHTER and NASARRE, 2008).

Such communication occurs by means of messaging, a central concept in the architecture of the Windows NT family operating systems. Structurally, the operating systems use object orientation to organize windows (including window controls such as buttons), but inter-process communication is event-driven (PETZOLD, 1998).

According to the Windows API documentation at MSDN (MICROSOFT, 2017), messages are defined in the winuser.h header as:

```
typedef struct tagMSG
{
   HWND hwnd;
   UINT message;
   WPARAM wParam;
   LPARAM lParam;
   DWORD time;
   POINT pt;
} MSG, *PMSG;
```

In which *hwnd* is a handle (an opaque reference) to the addressed window, *message* contains an integer that indicates the message type (either a value defined by the system or by the user), and the contents of *wParam* and *lParam* are specific to each type of message. Additionally, *time* contains the moment when the message was placed in the message queue, and *pt* is the position of the cursor at the moment the message was generated.

According to Petzold (1998), the operating system is responsible for sending messages to windows, and maintains a specific message queue for each program. In turn, the window reads those messages using a loop in its window procedure. The procedure is always defined as follows:

```
LRESULT CALLBACK WndProc
        (HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
```

According to MSDN (MICROSOFT, 2017), applications may define their own messages and send them to other processes using the *SendMessage* function, but most messages are sent by the operating system and consist of commands to manipulate the window, close the application and others; additionally, all user input causes system messages. Messages are initially placed in the system's message queue, and the OS then directs them, one by one, to the correct thread's message queue.

3.3 KEYBOARD INPUT

3.3.1 Scancodes

Each physical key in the keyboard has an assigned value called the *scancode*, consisting of one or more bytes sent to the computer when a key is pressed (KAPLAN and WISSINK, 2003).

When the user presses a key on the keyboard, Windows generates an *interrupt* that is then handled by the device driver. The driver reads the hardware-generated scancode, and then sends a message to the focused window with information about the keypress (HOSKEN, 2003).

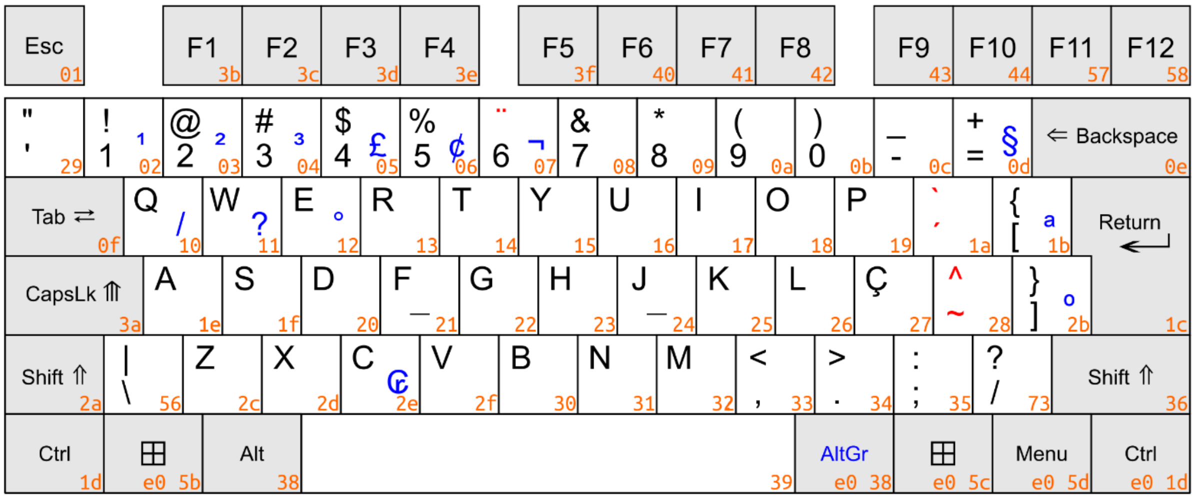
A scancode is called *make* when it represents a key being pressed down, and *break* when it represents a key release. There are mainly three scancode tables: scancode set 1, originally used in IBM PC and IBM XT; scancode set 2, introduced by IBM AT, and internally translated to set 1 in modern computers; and scancode set 3, introduced by IBM PS/2 but rarely used (CHAPWESKE, 2013).

The translation between sets 1 and 2 happens because set 2 (used in IBM AT) was incompatible with set 1 (used in IBM XT), motivating the use of the 8042 microprocessor, which translated set 2 scancodes into set 1 scancodes. This translation became standard practice in modern computers (BROUWER, 2009).

Each scancode corresponds to a physical key on the keyboard but doesn't necessarily represent the label printed to the key (CHAPWESKE, 2013). Because scancodes are associated to physical keys, their values are always the same, regardless of the system language; even still, there are different physical layouts present in the market, causing some inconsistency regarding the positioning of keys (KAPLAN and WISSINK, 2003).

Figure 3 displays the scancodes on the ABNT-2 keyboard as hexadecimal numbers. Note that it has extra keys compared to the ANSI and ISO layouts.

Figure 3: ABNT-2 layout scancodes



Source: Adapted from BROUWER (2009).

Keypresses in Windows causes *WM_KEYDOWN* and *WM_KEYUP* messages to be added to the system message queue, and those are later directed to the appropriate thread's message queue (usually corresponding to the window in focus). The system only directs keyboard messages to the correct thread after it finishes processing the previous message (PETZOLD, 1998).

3.3.2 Keyboard Layout

According to Kaplan and Wissink (2003), "a keyboard layout is a collection of data for each keystroke [...] within a particular keyboard driver".

3.3.2.1 Modifier Keys

The use of modifier keys, such as Shift, AltGr, Option and others, is a common way of assigning additional characters to the available keys of a keyboard (HOSKEN, 2003).

3.3.2.2 Dead Keys

Dead keys are common in many keyboard layouts; they don’t send characters and don’t cause visual feedback but modify the next character to be inserted (HOSKEN, 2003).

3.3.2.3 Operator Keys

Operator keys are similar to dead keys, but modify the previous character instead of the next, thus allowing for better feedback for the user. The greatest obstacle in this approach is the implementation, since the software needs to be capable of going back to edit the previously inserted character (HOSKEN, 2003).

3.3.3 Virtual Keys

Unlike scancodes produced by physical keys, virtual keys are independent of the key’s position on the keyboard and are translated from scancodes according to a specific layout (KAPLAN and WISSINK, 2003). Different layouts may have completely different maps from scancodes to virtual keys (HOSKEN, 2003).

According to the Windows API documentation at MSDN (MICROSOFT, 2017), virtual keys are independent of physical layout and may correspond to different characters depending on language; for example, the virtual key 0xba is translated from scancode 0x27 in the American English layout and produces a semicolon, while the same virtual key (0xba) is translated from scancode 0x1a in the German layout and produces the letter “U with Umlaut” (“Ü”).

Virtual keys have the fixed length of one byte, and may also represent various commands aside from letters, such as media keys, mouse clicks and language-specific commands (like 0x15: “VK_HANGUL”, present in Korean keyboards).

3.3.4 Keyboard Hooks for Message Manipulation

In addition to receiving messages through the window procedure, an application may intercept messages not addressed to it using a *hook*.

The Windows API documentation defines a *hook* as “a point in the system message-handling mechanism where an application can install a subroutine to monitor the message traffic in the system and process certain types of messages before they reach the target window procedure” (MICROSOFT, 2017). A function that intercepts a particular type of event is called a *hook procedure* (MICROSOFT, 2017).

According to Richer and Nasarre (2008), hooking is a way to inject a DLL into another process’ execution space; using it, it is possible for an application to execute its own code instead of another.

When intercepting a keystroke message, an application gains access to a message containing scancode, virtual key code and flags with additional information. However, keyboard hooks cannot identify the device that produced the message.

In order to identify the keyboard, it’s necessary to use another Windows resource, called the *Raw Input API*. Its documentation details that the interface makes it possible to intercept a message at an earlier point than a keyboard hook and extract the name of the device that generated the message. This API does not, however, offer the possibility of altering the content of messages (MICROSOFT, 2017).

Medek (2014) and Blecha (2014) demonstrate that it’s possible to use the Raw Input API to decide if a certain key should be blocked, and then use a keyboard hook to carry out that decision.

The keyboard hook intercept messages mainly by virtual key code (with a few edge cases), while Raw Input is used to make decisions depending on scancode. Therefore, combining these two mechanisms requires care in cases where scancodes and virtual keys do not correspond one-to-one, as that could lead to apparently conflicting information.

An example of a potentially problematic situation is the Right Alt key, which produces two virtual key events (Ctrl and Left Alt, in sequence) translated from a single scancode in layouts where the AltGr key is present. This sends two events to the keyboard hook while Raw Input detects only one.

3.4 INPUT SIMULATION

User input simulation must be done in Unicode, since this standard is widely adopted and enables inputting special characters (UNICODE CONSORTIUM, 2016).

This section goes into detail on the Unicode standard, and on keyboard input simulation in Windows.

3.4.1 **Unicode**

Unicode is a universal standard to encode characters internationally, defining a unique value to each character in several languages and for several purposes (UNICODE CONSORTIUM, 2016).

According to John Wells (2007), Unicode is a standard for encoding all written languages of the world, as well as for specialized disciplines. Unicode is multi-byte and capable of encoding a very large number of characters.

Unlike other character encodings, such as ASCII, the Unicode standard defines a universal set of characters for international use, created to resolve the needs caused by the many incompatible standards that existed then. The standard was created with the objective of providing a simple way of representing any writing system as 16-bit values, later extended to 32-bits (CONSTABLE, 2001).

Each Unicode character is represented by a positive integer called the codepoint, conventionally represented by “U+” followed by at least four hexadecimal digits. Currently, the standard occupies the U+0000..U+10FFFF range, divided into eleven *planes* of 65536 (=2^{16}) codepoints each. This means that a single 32-bit value suffices to represent any Unicode character (UNICODE CONSORTIUM, 2016).

3.4.2 UTF-16

When it became clear that 16-bit values would not be enough to cover all known writing systems, it was necessary to instead use 32-bit values. This change was formalized in Unicode 2.0 in 1996, which also defines UTF-8 and UTF-16 as unambiguous methods of representing any 32-bit codepoint as sequences of 8-bit values or 16-bit values.

Thus, UTF, meaning Unicode Transformation Format, was defined as a Unicode encoding that assigns a unique reversible sequence of bytes for each codepoint. The possible encodings, UTF-32, UTF-16 and UTF-8, use *code units* of 32, 16 and 8 bits, respectively (UNICODE CONSORTIUM, 2016).

According to the standard, in a Unicode text that uses the UTF-16 encoding, each character is represented by one or two 16-bit values, and each value is called a *code unit*. Codepoints up to U+FFFF are represented by one 16-bit code unit that is numerically equal to the code point, while higher codepoints are represented by a pair of code units, called a *surrogate pair*. The pair is calculated as follows (*lead* = first code unit, *trail* = second code unit):

```
// constants
const UTF32 LEAD_OFFSET = 0xD800 - (0x10000 >> 10);
const UTF32 SURROGATE_OFFSET = 0x10000 - (0xD800 << 10) - 0xDC00;

// computations
UTF16 lead = LEAD_OFFSET + (codepoint >> 10);
UTF16 trail = 0xDC00 + (codepoint & 0x3FF);

UTF32 codepoint = (lead << 10) + trail + SURROGATE_OFFSET;
```

The two values produced by this algorithm are called the *high surrogate* and the *low surrogate*. The algorithm always produces a high surrogate within the U+D800..U+DBFF range, and a low surrogate in the U+DC00..U+DFFF range. Codepoints in these ranges are reserved by the Unicode standard for use as surrogate pairs (they are not considered characters), and their isolated occurrence is considered an encoding error (CONSTABLE, 2001).

Operating systems of the Windows NT family internally use 16-bit characters, and support both ASCII and Unicode (PETZOLD, 1998). The Windows API documentation states that functions that work with 16-bit character strings expect them to be formatted in UTF-16 (MICROSOFT, 2017).

3.4.3 SendInput

The Windows API documentation describes the SendInput function, which allows for simulating user input as virtual keys or Unicode characters. Additionally, the wchar_t type defined in the C++ standard for supporting Unicode is implemented as a 16-bit type in Visual C++ (MICROSOFT, 2017).

SendInput makes it possible to simulate user input for any sequence of Unicode characters or virtual characters, including keyboard shortcuts.

3.5 EXISTING SOFTWARE

This section goes into existing software and tools that perform similar tasks as the one presented in this essay. We also point out Multikeys’ limitations.

3.5.1 Tavultesoft Keyman

Keyman is a multiplatform tool developed by Tavultesoft for the creation and use of custom layouts, with the goal of allowing the user to write in many writing systems with their keyboard (RAYMOND, 2014).

The software focuses on languages, and offers an intuitive and responsive layout, including modifier keys. It does not offer, however, the feature of programming more than one layout in different keyboards, and some users may be concerned about it being proprietary because of keyloggers and similar software.

3.5.2 AutoHotKey

AutoHotKey is a scripting language for Windows, available for Windows XP and above, and allows the user to assign the execution of scripts to keystrokes or sequences of keystrokes. Scripts written in this language can interact with the Windows API to carry out several kinds of tasks triggered by predetermined keys (GITHUB, 2017).

AutoHotKey can be used to create a custom layout through scripts that send Unicode characters to the focused window, but it does not provide a way to distinguish different keyboards. Besides, a user who wants to create their own keyboard mappings needs to learn to code them in the language, making adoption more difficult.

3.5.3 LuaMacros

LuaMacros, formerly called HidMacros, is a Windows application for the execution of macros that can identify different USB devices. The program is capable of opening executable files and sending simulated keystrokes, as well as assigning those actions to keys on specific keyboards. It was developed with flight simulators in mind, to allow using multiple keyboards as a controller (MEDEK, 2014).

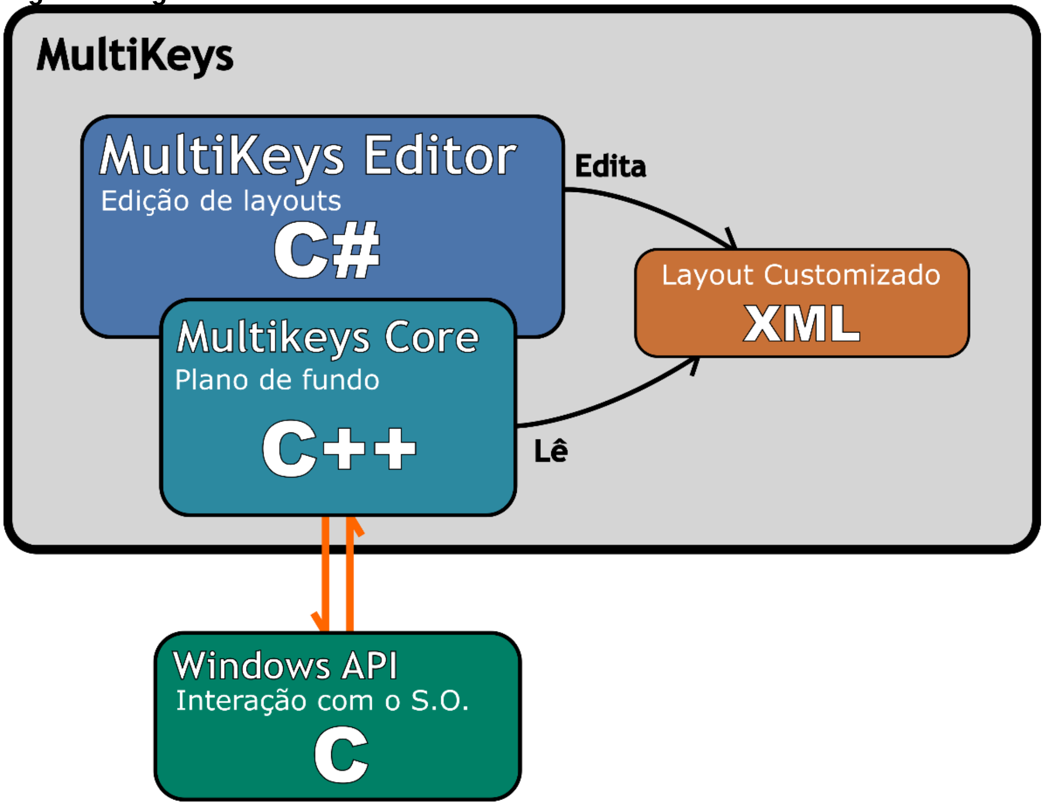
However, LuaMacros does not support Unicode input simulation, and isn’t built to emulate the behavior of a keyboard layout, which would include features such as modifier keys. This makes LuaMacros unsuitable for replicating the behavior of typical keyboard layouts, which require, among others, keeping internal state.

4 DEVELOPMENT

4.1 ARCHITECTURE

Figure 4 shows the high-level architecture of Multikeys. Multikeys Editor and Multikeys Core are the main modules of the system and have different features and requirements. As shown, Multikeys Editor depends on Multikeys Core, but not the opposite.

Figure 4: High level architecture

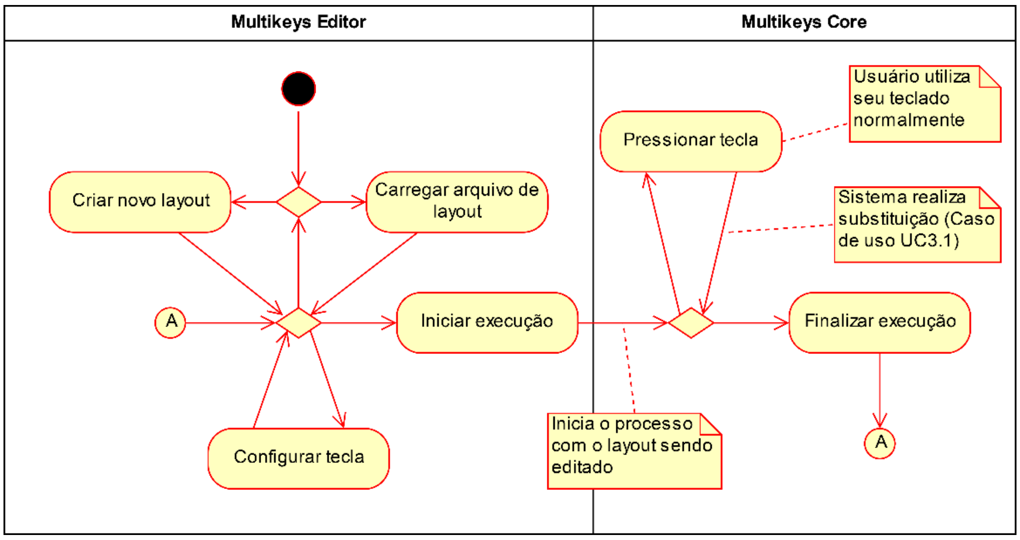


Source: Own authorship (2017)

Through Multikeys Editor, the user creates and edits layouts, which are used by Multikeys Core to remap keys.

Figure 5 shows an activity diagram with the main actions the user may take when using the system.

Figure 5: System's activity diagram


Multikeys Editor
Multikeys Core
Criar novo layout
Carregar arquivo de layout
A
Iniciar execução
Configurar tecla
Pressionar tecla
Usuário utiliza seu teclado normalmente
Sistema realiza substituição (Caso de uso UC3.1)
Finalizar execução
Inicia o processo com o layout sendo editado
A

Source: Own authorship (2017)

4.1.1 Multikeys Editor

This subsystem is a graphical application for editing custom layouts. The user can open, view and edit layouts through the UI, and assign layouts to specific keyboards connected to the keyboard.

The Multikeys Editor subsystem will contain all features related to configuring the system, including display language. The main goal is to offer the user an intuitive experience that does not require advanced knowledge about keyboard layouts or the logic behind layouts.

WPF was chosen as the technology for creating the graphical environment because of the relative ease to code custom forms. Therefore, this subsystem is written in the C# language and uses the .NET platform.

4.1.2 Multikeys Core

This subsystem is an application written in C++ for Windows and runs in the background (without a window) performing the key features of the system, such as intercepting keyboard messages and identifying their origin.

A Windows process needs a window controlled by a *Window Procedure* in order to receive system messages (which is central to Multikeys). However, Multikeys Core's window does not have visible graphic elements, and isn't visible in the taskbar either.

The process can be considered a background process because the user doesn’t have direct access to it.

The process will be started and finished according to the user’s decisions in Multikeys Editor. Multikeys Editor will have control over the instances of Multikeys Core.

4.1.3 Layout Persistence

Multikeys’ data persistence won’t use a database, since it’s not necessary to store large volumes of data, and there’s no need for large amounts of data retrieval either. Besides, the possibility of saving a custom layout as a file accessible to the user improves ease-of-use and doesn’t require a previous installation of a DBMS.

A configuration file in XML sufficiently fulfills the system’s requirements, as it’s well-suited for serializing hierarchical data structures. The Multikeys Editor subsystem will access custom layouts in the form of XML files (not necessarily using the xml extension, due to the Windows convention of using extensions to identify which program should open each file); Multikeys Core, on the other hand, will open those files to load layouts in memory.

4.1.4 Windows API

The Windows API is used in the Multikeys system to access several low-level operating system features. The official documentation for the Windows API is publicly available in the Microsoft Developer Network (abbreviated MSDN).

The functions and data structures exposed through the Windows API are written in the C language, making it possible for a C++ program to use them without additional configuration.

4.2 EMPLOYED TECHNOLOGIES

This section describes the technologies that were used in this project’s development.

4.2.1 Visual Studio

Visual Studio is an IDE developed and maintained by Microsoft, supporting development in a variety of platforms, including .NET. Because Multikeys makes extensive use of Microsoft platforms, in particular the WPF windows from .NET, Visual Studio was chosen as the main IDE for the project's development.

Additionally, Visual Studio has an integrated XML editor, including an XSD (XML Schema Definition) editor. This tool was used to create the documentation on configuration files.

4.2.2 WPF

Windows Presentation Foundation, or WPF, is a graphical platform for Windows, part of .NET, for the development of window applications. In .NET's version 4.5, WPF applications can be developed for Windows Vista or more recent. WPF offers a higher-level API in comparison to similar technologies, with features such as resolution independence (MACDONALD, 2012, p. 3-6).

This technology fulfills Multikeys' desktop interface requirements. Though it would be possible to code the visual interface in C++, using WPF has many advantages, both in convenience for programming and graphical functionalities.

4.2.3 Languages

4.2.3.1 C/C++

According to Petzold (1998, p. 7), using the C language to interact with the Windows API offers the best performance and versatility when it comes to carrying out OS-related tasks. A common alternative is to instead use the C++ language to interact with the Windows API.

The C++ language is an extension of the C language, with features from object orientation like inheritance and polymorphism (SAHAY, 2012, p. 7-8). We chose the C++ language for writing Multikeys Core due to the performance requirements and also the interaction with the Windows API, and also due to allowing for a reasonably comfortable implementation of object-oriented practices.

4.2.3.2 C#

C# is an object-oriented language developed by Microsoft which was inspired by many others, like C++ and Java; the language is integrated in the .NET platform, and has access to many resources and features of the platform, as well as interoperability with other .NET languages (PALMER and BARKER, 2008, p. 4-5).

We chose this language for implementing Multikeys Editor due to being part of the .NET platform, and also in order to use WPF for coding the interface.

4.2.3.3 XML / XSD

XML (Extensible Markup Language) is a markup language used in this project for data persistence.

XML is subset of SGML – Standard Generalized Markup Language, used for storing data and metadata, especially in a hierarchical structure. Its use facilitates interoperability between programs that don’t need to know each other’s details. Instead, XML can be used for the communication protocol. (FAWCETT, QUIN and AYERS, 2012, p. 6-13).

XML Schema is a W3C recommendation for representing the vocabulary and rules of an XML file; an XML file with the XSD (XML Schema Definition) extension is commonly used to contain the model with the schema. These files are used to define and validate syntaxes and vocabularies, as well as metadata definitions (FAWCETT, QUIN, AYERS, 2012, p. 117, 120).

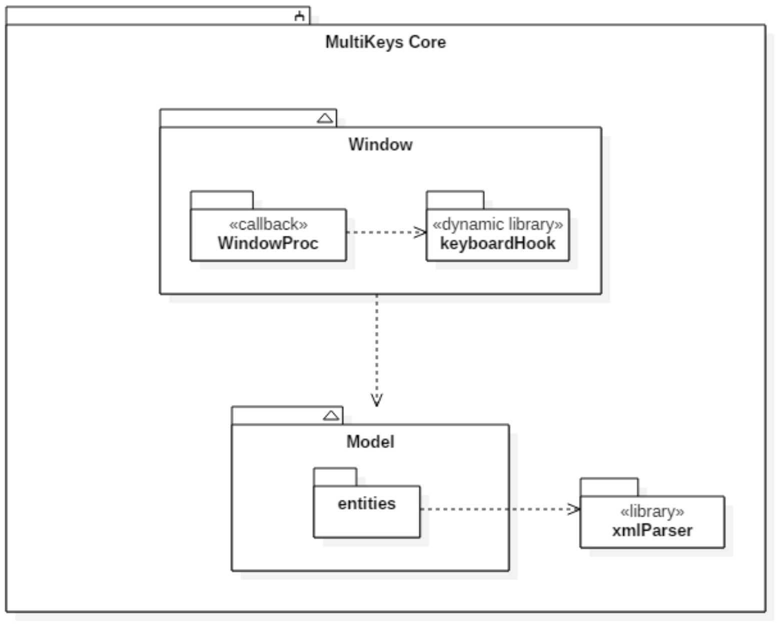
Using the markup language XML is justified by the hierarchical structure of a keyboard layout: each layout contains one or more levels (each activated by a specific combination of modifier keys), each level contains multiple maps from physical keys to actions, and each of those actions may contain one or more characters (usually just one). Additionally, the user may configure one or more keyboard devices and assign different layouts to them.

The advantages of using XML files are aligned to the program's requirements: user-accessible files on disk, interoperable, both human- and machine-readable, and capable of representing hierarchical data and metadata. An XSD schema formally defines the structure of XML files, which is highly desirable since XML will be used as the data protocol between the Multikeys modules.

4.3 MESSAGE MANIPULATION

The essential feature of Multikeys is its capacity to intercept and manipulate keyboard input. This feature is implemented in the Multikeys Core subsystem. The package diagram in Figure 6 describes the architectural structure of the application.

Figure 6: Application's package diagram

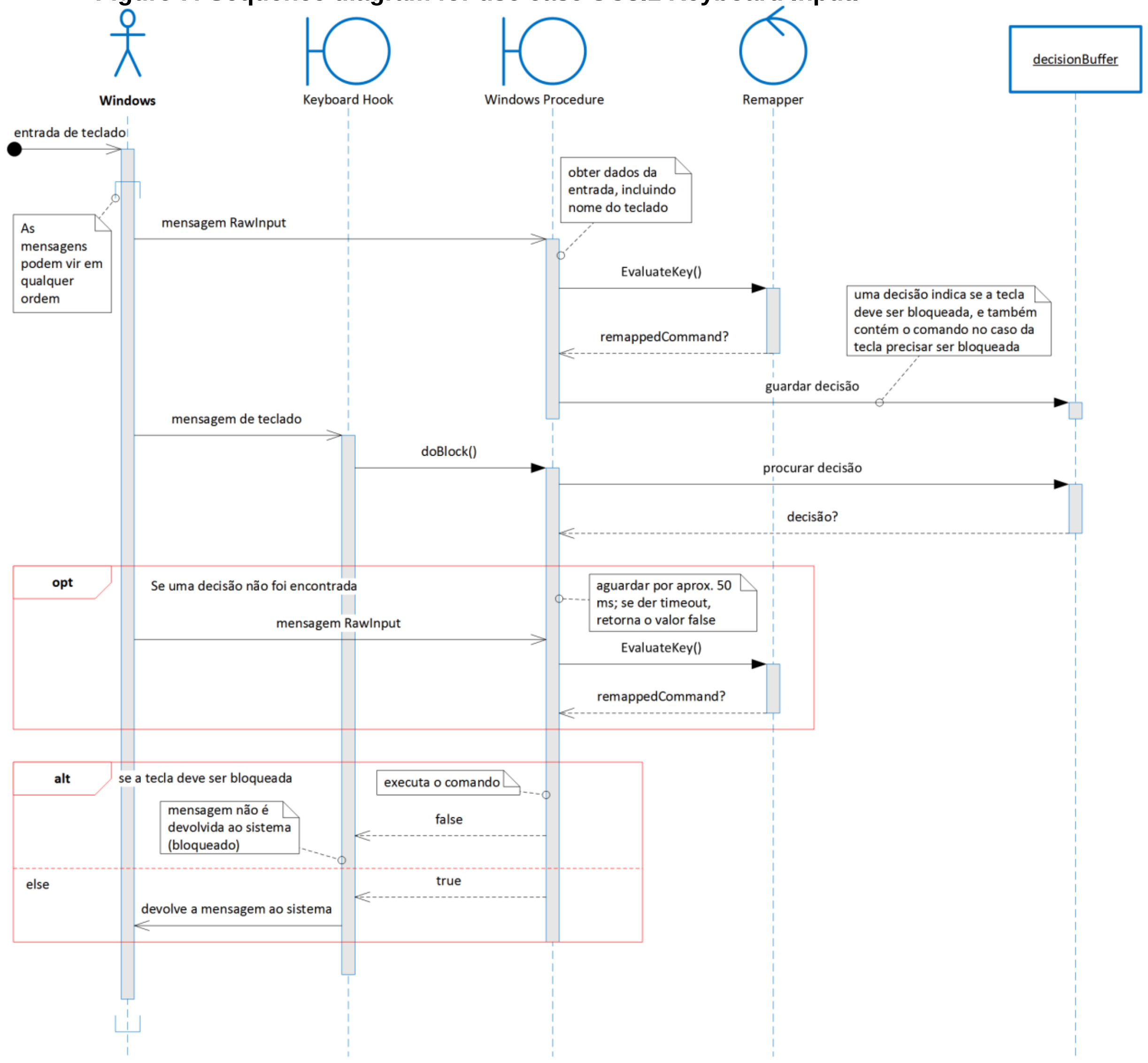


Source: Own authorship (2017)

The window procedure and the keyboard hook are included as components of the main window. In addition, the module responsible for emulating the behavior of a keyboard layout, including internal state, is implemented as a library.

The sequence diagram in Figure 7 shows the main flow of use case UC3.1 Keyboard Input.

Figure 7: Sequence diagram for use case UC3.1 Keyboard Input.


Windows
Keyboard Hook
Windows Procedure
Remapper
decisionBuffer
entrada de teclado
As mensagens podem vir em qualquer ordem
mensagem RawInput
obter dados da entrada, incluindo nome do teclado
EvaluateKey()
uma decisão indica se a tecla deve ser bloqueada, e também contém o comando no caso da tecla precisar ser bloqueada
remappedCommand?
guardar decisão
mensagem de teclado
doBlock()
procurar decisão
decisão?
opt
Se uma decisão não foi encontrada
aguardar por aprox. 50 ms; se der timeout, retorna o valor false
mensagem RawInput
EvaluateKey()
remappedCommand?
alt
se a tecla deve ser bloqueada
executa o comando
mensagem não é devolvida ao sistema (bloqueado)
false
else
true
devolve a mensagem ao sistema

Source: Own authorship (2017)

The Ю-like symbol represents an interface; in this case, the interface is Multikeys' interaction with the operating system through messages, since the background process never directly interacts with the user. The diagram illustrates that after receiving a keystroke message from the user, the operating system sends two messages to Multikeys. The first (not necessarily chronologically earlier) is a Raw Input message, necessary for detecting the device that sent the signal, and the second is a keyboard hook intercepted message.

Because the Raw Input message is used for decision making, it must be saved in memory. The decisionBuffer object (implemented as a queue) is used to hold those decisions until the hook message is received. The keyboard hook may return the message or block it, depending on the decision.

Multikeys’ WndProc is first prototyped as:

```
LRESULT CALLBACK    WndProc(HWND, UINT, WPARAM, LPARAM);
```

And later defined as:

```
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM
lParam)
{
    ...
}
```

WndProc is a procedure that must be called each time a message needs to be processed by the application window. In the program’s main function, there’s a loop that calls WndProc:

```
int APIENTRY wWinMain(_In_ HINSTANCE hInstance,
                     _In_opt_ HINSTANCE hPrevInstance,
                    _In_ LPWSTR lpCmdLine,
                    _In_ int nCmdShow)
{
    ...
    while (GetMessage(&msg, nullptr, 0, 0))
    {
            ...
            TranslateMessage(&msg);
            DispatchMessage(&msg);
    }
}
```

The DispatchMessage function sends a message to the appropriate WndProd; in this case, only one exists in the application.

Even though the window procedure’s main purpose is processing messages sent to the application, most messages to be processed in Multikeys’ windows procedure are messages addressed to other applications. The interception occurs both through the Raw Input API and the keyboard hook.

4.3.1 Raw Input API

The program’s main window receives messages from the Raw Input API through a subscription system. The following lines exist to inform the operating system that the caller requests to receive Raw Input messages to any window but limited to keyboard input.

```
// Register for receiving Raw Input for keyboards
   RAWINPUTDEVICE RawInputDevice[1];
   RawInputDevice[0].usUsagePage = 1;
   // usage page = 1 is generic and usage = 6 is for keyboards
   RawInputDevice[0].usUsage = 6;
   // (2 is mouse, 4 is joystick, 6 is keyboard, there are others)
   RawInputDevice[0].dwFlags = RIDEV_INPUTSINK;
   // Receive input even if the registered window is in the background
   RawInputDevice[0].hwndTarget = hWnd;
   // Handle to the target window (NULL would make it follow kb focus)
   RegisterRawInputDevices(RawInputDevice, 1, sizeof(RawInputDevice[0]));
```

From this point on, the window starts to receive WM_INPUT messages from the operating system, containing low-level information on every keystroke.

Being able to extract information about which device sent the keystroke is essential to the correct functioning of the system. The following lines are used to extract that information:

```
GetRawInputData
   ((HRAWINPUT)lParam,
   RID_INPUT,
   rawKeyboardBuffer,
   &rawKeyboardBufferSize,
   sizeof(RAWINPUTHEADER));

raw = (RAWINPUT*)rawKeyboardBuffer;
```

```
GetRawInputDeviceInfo
   (raw->header.hDevice,
   RIDI_DEVICENAME,
   keyboardNameBuffer,
   &keyboardNameBufferSize);
```

The first two lines extract data from the Raw Input message, while the next call extracts the name of the device that sent the keystroke. Other information, such as the scancode, is also extracted.

It is necessary to use the Raw Input API because that’s the only way to distinguish between different keyboards. Next, we call the Remapper, the module that holds the user configured layouts and answers with whether or not a key should be blocked, as well as a possible command that may execute a variety of actions, such as simulating Unicode input.

```
IKeystrokeCommand * possibleAction = nullptr;
BOOL DoBlock = remapper.EvaluateKey
   (&(raw->data.keyboard), keyboardNameBuffer, &possibleAction);
decisionBuffer.push_back
   (DecisionRecord(raw->data.keyboard, possibleAction, DoBlock));
```

Since it’s not possible to block keys using only the Raw Input API, this decision is simply stored. The DecisionRecord structure is used for storing the information on a key, whether or not it should be blocked, and the possible remapped command. The structure is defined as follows:

```
struct DecisionRecord
{
    RAWKEYBOARD keyboardInput;
    IKeystrokeCommand * mappedAction;
    BOOL decision;
};
```

4.3.2 *Keyboard Hook*

A keyboard hook intercepts keyboard input messages; unlike Raw Input messages, hooks allow a program to block messages, even those addressed at other programs, but it’s not possible to identify which device a message originated from. The hook must exist as a separate process to be able to intercept messages globally. Because of that, a separate dll, the following code is used to put the keyboard hook into the operating system’s *hook chain*:

```
BOOL InstallHook(HWND hwndParent)
{
    if (hwndServer != NULL)
    {
            // Already hooked
            return FALSE;
    }
    hookHandle = SetWindowsHookEx(WH_KEYBOARD, (HOOKPROC)KeyboardProc,
            instanceHandle, 0);
    if (hookHandle == NULL)          // security check
            return FALSE;
    hwndServer = hwndParent;
    return TRUE;
}
```

This function is exported in the dll’s header in order to be called from other processes:

```
__declspec(dllexport) BOOL InstallHook(HWND hwndParent);
```

The function is called from the WndProc:

```
InstallHook(hWnd);
```

The parameter is a handle to the main window, which is necessary to enable the dll’s hook to communicate with the rest of the Multikeys application.

From the moment the hook is injected, the KeyboardProc procedure is registered in the hook chain, and all keyboard input passes through it. Below is a simplified version of the KeyboardProc's definition:

```
static LRESULT CALLBACK KeyboardProc
   (int code, WPARAM wParam, LPARAM lParam)
{
   if (code != HC_ACTION)
          return CallNextHookEx(hookHandle, code, wParam, lParam);

   if (SendMessage(hwndServer, WM_HOOK, wParam, lParam))
          return 1;

   return CallNextHookEx(hookHandle, code, wParam, lParam);
}
```

This procedure also makes certain safety checks described in further detail in use case UC3.1, omitted from the samples here for brevity.

When the procedure returns 1, the message is not processed further by other programs; otherwise, we must call CallNextHookEx() to send the message to the next procedure in the hook chain; note that the message is blocked if any hook in the hook chain returns 1. Additionally, the call to SendMessage sends a message to the main window procedure, asking whether the intercepted key should be blocked, passing the key's data in the wParam and lParam parameters.

The main window procedure searches for the decision for this key, comparing by virtual key and scancode.

4.3.3 Deadlines and Blocks

As seen in Figure 7, the main flow starts when the keystroke message is intercepted and finishes when it's passed on (or not) to the original destination. All processing that occurs between these two moments increases latency, and risks degrading the user experience.

In addition, the code executed in this time window must be necessarily executed synchronously, since the result of this computation (whether or not to block the key, and a possible command) must occur in the same order as the input.

Because it's necessary to know what decision to carry out when blocking a key, finding the remapped command is a prerequisite for the keyboard hook executing its code. To keep any failures from causing delays of arbitrary length in the main flow, there's a deadline to every keystroke to keep the system responsive.

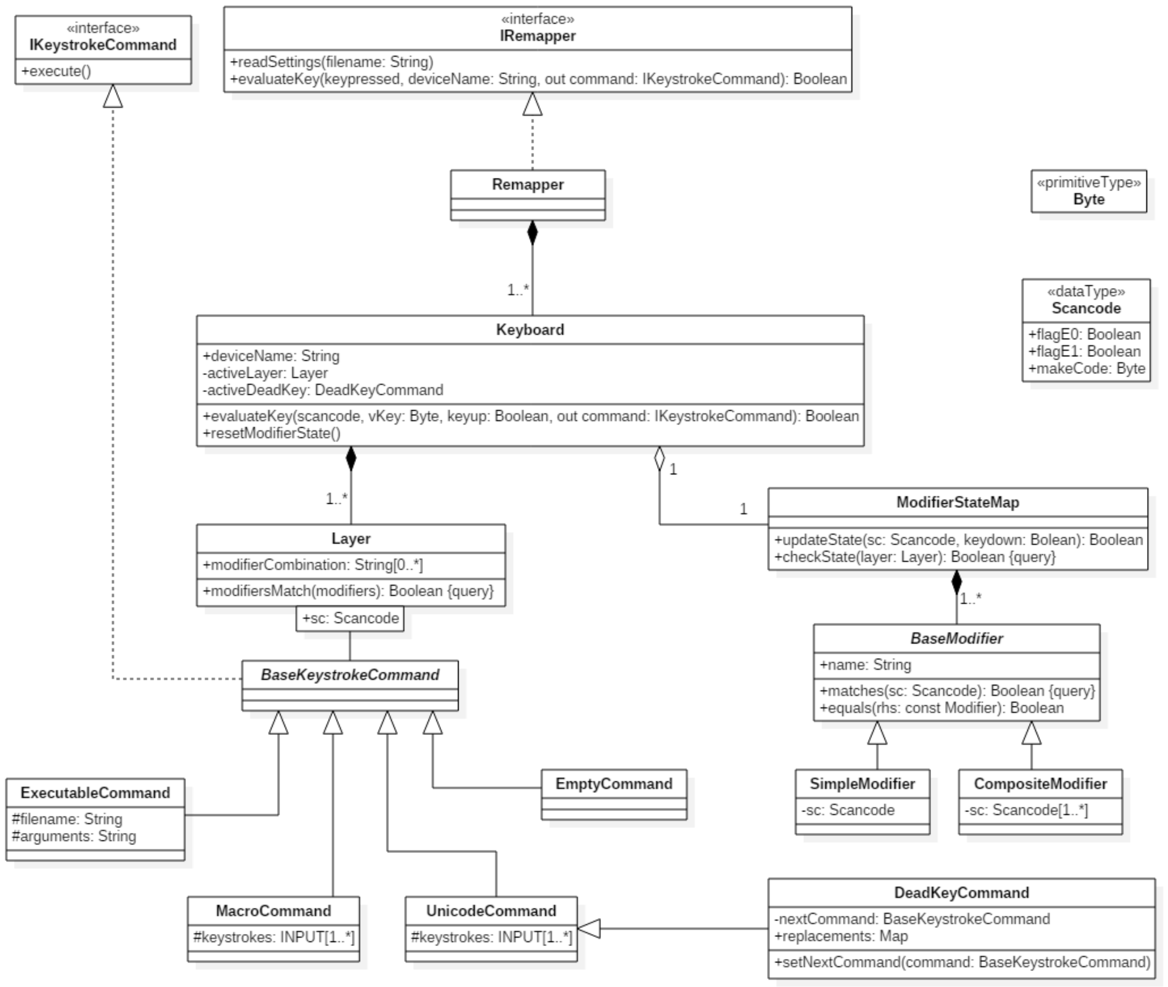
When the system can’t process a message within this deadline, we consider it a failure and the message is no longer processed; even though this process is soft real-time, we drop delayed messages to avoid accumulating them.

4.3.4 Remapper

The component responsible for keeping an internal model of the user’s keyboards, including modifier key status and dead key status, is called Remapper and is located in a separated library. The model was placed in a separated library to keep the project modularized, separating the primarily event-driven responsibilities in the WndProc from the object-oriented model in the Remapper. The Figure 8 contains the internal model’s class diagram.

The interfaces and commands are implemented in accordance to the Gang of Four’s design patterns.

Figure 8: Internal model's class diagram



«interface»
IKeystrokeCommand
+execute()
«interface»
IRemapper
+readSettings(filename: String)
+evaluateKey(keypressed, deviceName: String, out command: IKeystrokeCommand): Boolean
Remapper
«primitiveType»
Byte
1..*
«dataType»
Scancode
+flagE0: Boolean
+flagE1: Boolean
+makeCode: Byte
Keyboard
+deviceName: String
-activeLayer: Layer
-activeDeadKey: DeadKeyCommand
+evaluateKey(scancode, vKey: Byte, keyup: Boolean, out command: IKeystrokeCommand): Boolean
+resetModifierState()
1
1..*
1
ModifierStateMap
+updateState(sc: Scancode, keydown: Bolean): Boolean
+checkState(layer: Layer): Boolean {query}
Layer
+modifierCombination: String[0..*]
+modifiersMatch(modifiers): Boolean {query}
+sc: Scancode
1..*
BaseModifier
+name: String
+matches(sc: Scancode): Boolean {query}
+equals(rhs: const Modifier): Boolean
BaseKeystrokeCommand
ExecutableCommand
#filename: String
#arguments: String
EmptyCommand
SimpleModifier
-sc: Scancode
CompositeModifier
-sc: Scancode[1..*]
DeadKeyCommand
-nextCommand: BaseKeystrokeCommand
+replacements: Map
+setNextCommand(command: BaseKeystrokeCommand)
MacroCommand
#keystrokes: INPUT[1..*]
UnicodeCommand
#keystrokes: INPUT[1..*]

Source: Own authorship (2017)

The following methods and classes are exposed in the header RemapperAPI.h; this is a reduced and uncommented version for brevity:

```
namespace Multikeys
{
   typedef class IKeystrokeCommand
   {
   public:
           virtual BOOL execute(BOOL keyup, BOOL repeated) const = 0;
           virtual ~IKeystrokeCommand() = 0;
   } *PKeystrokeCommand;

   typedef class IRemapper
   {
   public:
           virtual bool loadSettings(
                   const std::wstring xmlFilename) = 0;
           virtual bool evaluateKey(
                   RAWKEYBOARD* const keypressed,
                   WCHAR* const deviceName,
                   OUT PKeystrokeCommand* const out_action) const = 0;

           virtual ~IRemapper() = 0;

   } *PRemapper;

   void Create(OUT PRemapper* instance);
   void Destroy(PRemapper* instance);
}
```

According to the principles of object orientation, IRemapper and IKeystrokeCommand are interfaces, implemented in C++ as pointers to purely abstract classes. The virtual destructors are required for the implementing classes to have their own destructors as well, as they're called from the superclass.

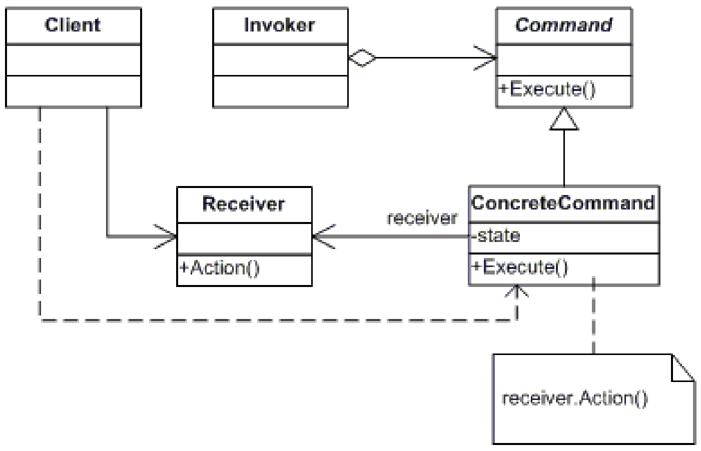
Also, constants are a way to implement immutability in the C language; the evaluateKey() method, for example, is marked as *const* to guarantee that the object where it's called will not be changed by the call. One of its commands, out_action, is a *const* argument, meaning that it can't be changed in the function's body.

Another point worthy of note is that the Create() and Destroy() methods, which encapsulate the creation of the actual concrete object, ensure that the implementing class is not visible from outside code. This implementation is based on the unparameterized factory method pattern, but in this case the methods have direct implementations instead of using entire classes dedicated to object creation.

IKeystrokeCommand is another class that can be seen in Figure 8; the application must be able to map keys to several kinds of actions, ranging from simulating Unicode input to opening executable files with parameters, and according to the GoF the command pattern is used to parameterize objects with actions to execute in different moments. The use of commands removes the callers' (in this case WndProc) responsibility of having any details on how to execute a certain task (actions like simulating Unicode input), promotes extensibility (in this case allows for adding various kinds of actions bindable to keys), and the tasks themselves can be inherited (such as the dead key command, which is a subtype of Unicode command).

Figure 9 shows a class diagram for the command pattern, according to Gamma et al. (1995).

Figure 9: Command design pattern



Source: http://www.dofactory.com/net/command-design-pattern (2017)

The Table1 below shows how the pattern in implemented in this project:

Table 1: Command pattern implementation

Component	Purpose	Implementation
Client	Creates concrete commands	Remapper
Invoker	Calls the command to carry out the task	Window procedure
Command	Interface for the execution of a task	IKeystrokeCommand
Concrete Command	Implements the execution of a task	UnicodeCommand, Executablecommand, etc.
Receiver	Executes the operations detailed in the command	Windows API

Source: Adapted from GAMMA et al (1995)

Since Multikeys Core’s architecture isn’t purely object-oriented, some implementations cannot be described as classes; even still; it’s possible to separate and identify the different components of the command pattern.

For a more detailed explanation: an instance of the Remapper class reads the remaps and actions from a configuration file and instantiates commands for each of them, parameterized with data from the file (for example, a UnicodeCommand is parameterized with the text the key will send when pressed). At a later moment, the window procedure queries the Remapper for a command corresponding to a key and executes it; the commands carry out those tasks through WinAPI calls. Therefore, Windows itself works as the receiver, executing tasks such as opening an executable file or sending Unicode characters to the focused window.

Gamma et al. (1995) also mention *callbacks* as an alternative to commands. Callbacks are function parameters to be called at a later time. The main problem in this context is that Multikeys needs to hold state in each action bound to a key, and that is not easily implementable with callbacks. As commands, the remaps are easy to parameterize, save and execute at any time.

4.4 OBSTACLES

The two mechanisms for capturing keystroke messages in Multikeys (Raw Input and the keyboard hook) are used in tandem to permit selectively blocking and manipulating keyboard messages depending on the origin device. This strategy was documented by Blecha (2014), and previously used By Petr Medek in the LuaMacros tool.

Blecha mentions many obstacles in dealing with keystroke interception this way; because Raw Input and the keyboard hook receive keyboard messages at different times, certain key signals cause inconsistent information to be received in the two APIs.

This section discusses some specific problematic cases, and what Multikeys does in those scenarios.

We observed and documented the system’s behavior in those odd cases in order to fix them. The results that follow were obtained through logging with the

OutputDebugString(); for example, the following logs information contained in a Raw Input message:

```
#if DEBUG
   WCHAR * text = new WCHAR[128];
   swprintf_s(text, 128,
             L"Raw Input: Virtual key %X scancode %s%s%X (%s)\n",
           raw->data.keyboard.VKey,          // virtual keycode
           (raw->data.keyboard.Flags & RI_KEY_E0 ? L"e0 " : L""),
           (raw->data.keyboard.Flags & RI_KEY_E1 ? L"e1 " : L""),
           raw->data.keyboard.MakeCode,      // scancode
           raw->data.keyboard.Flags & RI_KEY_BREAK ? L"up" : L"down");
           // keydown or keyup (make/break)
   OutputDebugString(text);
   memcpy_s(debugText, DEBUG_TEXT_SIZE, text, 128);
   delete[] text;
#endif
```

The topics in this section are also present as business rules in the business requisites document for Multikeys. This section lists only the most relevant cases.

4.4.1 AltGr

The AltGr key is problematic due to behaving in different ways depending on the active Windows layout. If the currently active Windows layout contains this key, Multikeys receives a combination of Ctrl and Alt instead of the Right Alt key (which normally has a scancode of 0xe0 0x38). Thus, the following sequence of messages is received:

Table 2: Order of received messages for the AltGr key

Order	API	Scancode	Virtual Key
1	*Hook*		L-Ctrl make
2	*RawInput*	0xe0 0x38 (right alt) make	
3	*RawInput*	0xe0 0x38 (right alt) break	
4	*Hook*	L-Ctrl make: timeout	
5	*Hook*		R-Alt make (2)
6	*Hook*		L-Ctrl break
7	*Hook*	L-Ctrl break: timeout	
8	*Hook*		R-Alt break (3)

Source: Own authorship (2017)

The keyboard hook receives an additional Left Control key, which times out because no corresponding message is ever received in the Raw Input, so there’s nothing to be found in the decision buffer. Multikeys continuously looks for a Left Control followed by a delayed Raw Input message containing a Right Alt and recognizes this pattern as an AltGr. This combination can only occur in this specific situation.

4.4.2 Pause/Break

The Pause/Break key does not have a use in modern programs and has the unique property of containing a three-byte scancode. According to Brouwer (2009), the 0xe1 prefix, which is only used for this key and no other, indicates that the break signal is sent immediately after the make signal; that is consistent with the data obtained during tests.

Table 3: Order of received messages for the Pause/Break key

Order	API	Scancode	Virtual Key
1	*RawInput*	0xe1 0x1d *make*	0x13 Pause *make*
2	*Hook*	0x45 (numlock) *make*	0x13 Pause *make*
3	*RawInput*	0x45 (numlock) *make*	0xff Error *make*
4	*RawInput*	0xe1 0x1d *break*	0x13 Pause *break*
5	*RawInput*	0x45 (numlock) *break*	0xff Error *break*
6	*Hook*	0x13 Pause *make* timeout	
7	*Hook*	0x45 (numlock) *break*	0x13 Pause *break*
8	*Hook*	0x13 Pause *break* timeout	

Source: Own authorship (2017)

Note that due to the three-byte scancode, the Raw Input API detects two messages, one containing the 0xe1 0x1d scancode, and another containing 0x45 (which by itself would correspond to the NumLock key). Neither of those correspond to the message intercepted in the keyboard hook (scancode 0x45, virtual key 0x13). The solution is to have the keyboard hook look for the 0xe1 0x1d scancode received at line 2 and interpret that as a Pause/Break. In this case, one Raw Input message is left ignored.

Another particularity of the Pause/Break key is that it sends a different scancode when the Ctrl modifier is being held down. However, in that case, the modified scancode is two bytes in length (0xe0 0x46) and is interpreted normally in Multikeys.

4.4.3 PrintScreen/System Request

The PrintScreen key sends two scancodes in sequence, a Shift (0xe0 0x2a) immediately followed by its own scancode (0xe0 0x37). The Shift message is omitted when the PrintScreen is pressed while Shift or Ctrl is held down. Those redundant Raw Input messages do not cause problems to Multikeys.

This key also has the curious behavior of never sending a make signal to the keyboard hook. This situation is easily solved by having the hook query for both the make and the break signals for the 0xe0 0x37 scancode. This process is further detailed in the Use Case Specification for UC3.1.

4.5 INPUT SIMULATION

The Remapper library is responsible for storing a map from physical keys (identified by scancode) and the tasks bound to them. During initialization, Multikeys Core reads an xml file containing the user's custom layout and creates all necessary instances of Keyboard, Layout and other classes. This also removes the need to allocate additional memory during execution.

We use the WinAPI's SendInput function to simulate keypresses and send Unicode characters. This function takes an array of INPUTs as parameter, each one being a data structure holding user input information. All such arrays are initialized when the xml file is read.

The following snippet maps the key of scancode 0x1f to send the Unicode character U+03C3, corresponding to the uppercase Greek letter sigma ('Σ').

```xml
<unicode Scancode="1F" TriggerOnRepeat="True">
   <codepoint>3C3</codepoint> <!-- Sigma -->
</unicode>
```

We use the Xerces library to read the xml file. An instance of UnicodeCommand is constructed and stored in a map.

Every command implements the execute() method; the following snippet shows its implementation in the UnicodeCommand class:

```
BOOL execute(BOOL keyup, BOOL repeated) const override
{
   if (keyup)
           return TRUE;
   else if (!repeated || (repeated && triggerOnRepeat))
           return  (SendInput(
                   inputCount,
                   keystrokes,
                   sizeof(INPUT)) == inputCount ? TRUE : FALSE);
   else return TRUE;
}
```

The UnicodeCommand instance stores its Unicode text in an array of INPUTs, as well as an integer with the number of characters to be sent.

It's worth noting that the INPUT structure is a union type that may represent one of different kinds of user input, including virtual keys and Unicode characters. In the case of the UnicodeCommand class, the structure is used to simulate Unicode text input.

The ExecutableCommand works differently, opening an executable in disk when activated. It can be configured as in the following example:

```
<execute Scancode="06">
        <path>C:\Program Files (x86)\Google\Chrome\Application\chrome.exe</path>
        <arguments>http://stackoverflow.com/</arguments>
</execute>
```

The WinAPI's ShellExecute function is used to carry out an operation on a file; in our case, the "open" operation is executed on the specified file, which causes its execution if it's an executable file.

```
BOOL execute(BOOL keyup, BOOL repeated) const override
{
   if (repeated || keyup) return TRUE;
   HINSTANCE retVal = ShellExecute(NULL,
                              L"open",
                              filename.c_str(),
                              arguments.c_str(),
                              NULL,
                              SW_SHOWNORMAL);
   if ((LONG)retVal <= 32)
          return FALSE;
   return TRUE;
}
```

These commands are obtained by the window procedure and executed when appropriate, as Figure 7 shows.

4.6 LAYOUT EDITING

One of Multikeys' main functionalities is creating and editing custom user layouts, allowing for easy configuration of actions. This feature is provided by Multikeys Editor, through a graphical user interface that allows the user to navigate through several layouts and edit commands.

According to the non-functional requirements NFR010 (Configuration Files) and NFR011 (Portable Execution), the main purpose of the program's GUI is to allow the user to edit layout files to be used by Multikeys Core for remapping keys. As discussed in the project's documentation, Multikeys is structured in layers.

4.6.1 Multikeys Editor

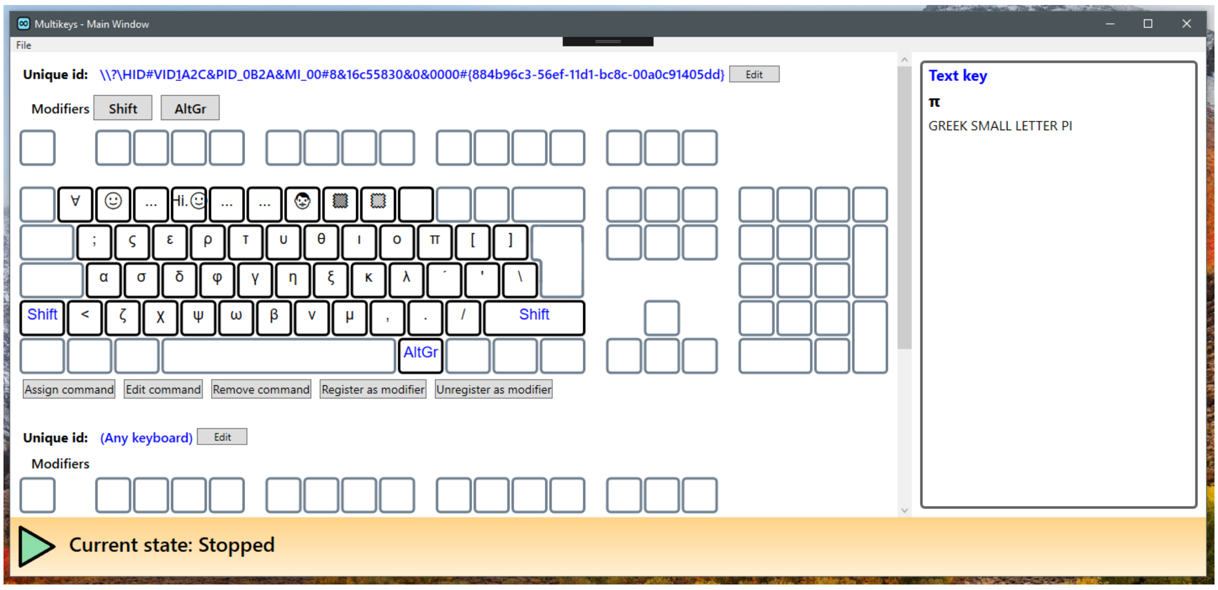
Figure 10 shows the logo created for Multikeys Editor, which will appear in the window title and in the taskbar.

Figure 10: Multikeys Editor's logo

Source: Own authorship (2017)

The program's graphical interface uses multiple custom user controls in order to graphically represent and organize the model. For example, a control called KeyControl represents a key and the action to be executed when it's pressed. Figure 11 shows a screenshot of the running program.

Figure 11: A screenshot of Multikeys Editor running



Multikeys - Main Window

File

Unique id: \\?\HID#VID1A2C&PID_0B2A&MI_00#8&16c55830&0&0000#{884b96c3-56ef-11d1-bc8c-00a0c91405dd} Edit

Text key

π

GREEK SMALL LETTER PI

Unique id: (Any keyboard) Edit

Current state: Stopped

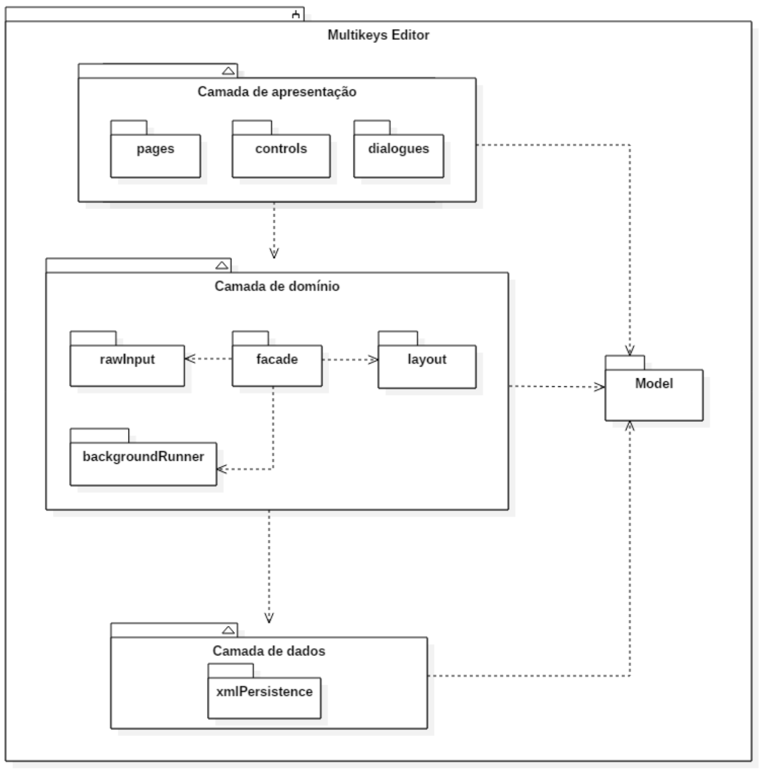
Source: Own authorship (2017)

Remaps for two keyboards are shown in the screenshot. The window shows relevant information on the selected key and options to edit the layouts; modifier keys, indicated in blue, can be clicked to access other layers of the same layout.

Through the control at the bottom of the window, the user can start and stop the execution of Multikeys Core.

Figure 12 shows Multikeys Editor's architecture. The architecture separates domain logic, presentation logic and xml persistence logic from each other.

Figure 12: Multikeys Editor's model diagram



Source: Own authorship (2017)

4.6.2 Structure of Controls

A WPF control is a graphical element that can be used as a component of the GUI. It's possible to define custom controls, called user controls, used in this project to represent entities like keys and keyboard layers.

As shown in Figure 12, the controls defined by the application are defined in the *controls* namespace. The controls approximately correspond to the domain model; for example, the LayerControl control corresponds to a keyboard layer, and contains a collection of KeyControl instances that may take different graphical shapes on screen, such as the shape of an ISO return key.

Similar to the domain model, controls are organized in a hierarchical structure; the main window contains a control representing a layout, which contains many keyboard controls, and so on.

Each control may be initialized with an existing domain object read from a file, and also has methods to create a new domain object from its content. For example, the main window may initialize a LayoutControl with the model, and later use it to obtain the updated domain object.

4.6.2.1 Events in Controls

The events available in the .NET platform are used in WPF for many types of notifications, such as button clicks and text input; the controls implemented in the *controls* namespace use events to notify other controls of updates in the model; for example, when a key is selected in the Layer control, the Layout control is notified to display a side panel containing a summary on the selected key. This side panel can be seen in Figure 11.

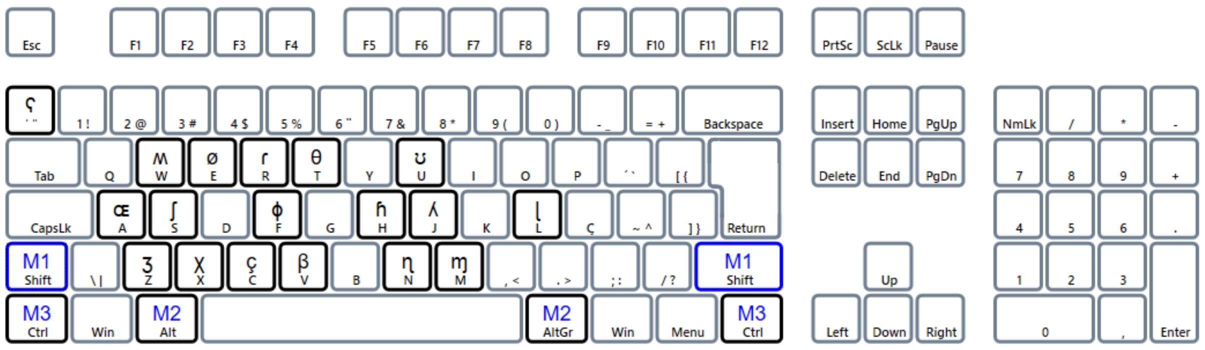
4.6.3 Localization

An important feature of WPF controls is the possibility of loading text from a separate localizable file. This means the program's localization can be implemented through dictionaries changeable in execution time, affecting window titles, label text, and so on.

This technology allows for adding new translations for the program.

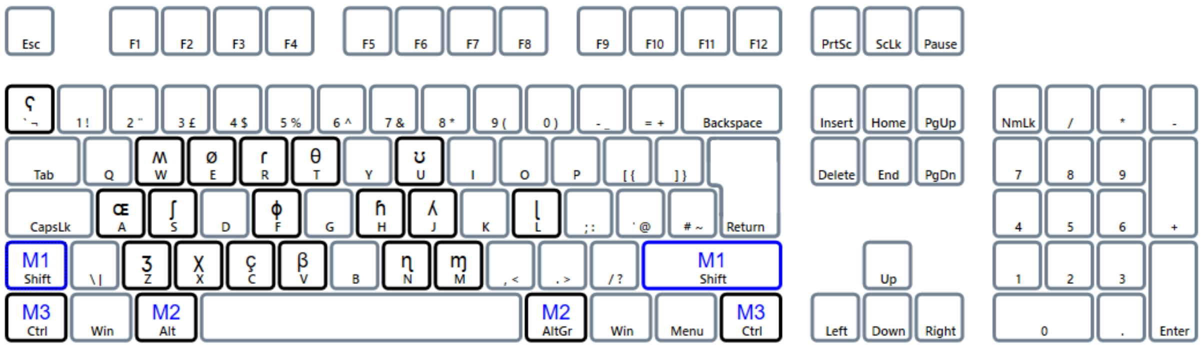
Additionally, the visual components shown on screen can be changed to correspond to the user's keyboard by choosing among physical and logical keyboard standards. The following screenshots show, respectively, keyboards configured to render with the ABNT-2 physical layout with ABNT-2 logical layout, ISO physical layout with En-UK logical layout and ISO physical layout with DE logical layout.

Figure 13: Layout configured to display as ABNT-2



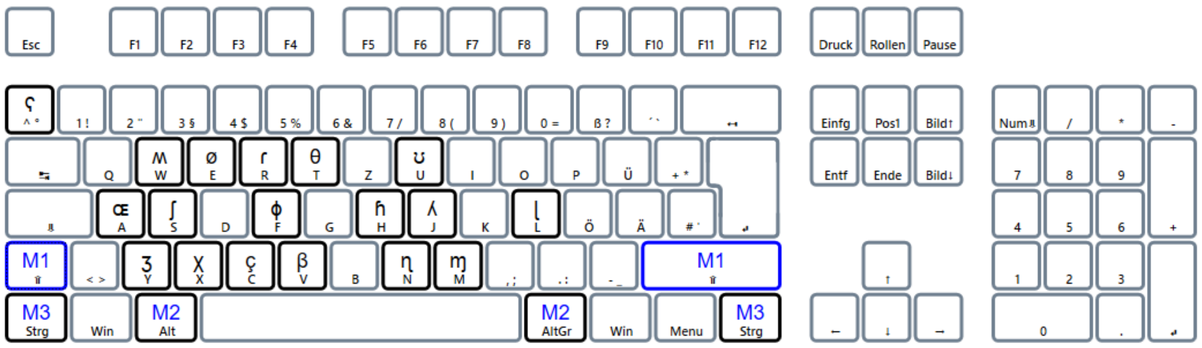
Source: Own authorship (2017)

Figure 14: Layout configured to display as ISO (En-UK)


Esc F1 F2 F3 F4 F5 F6 F7 F8 F9 F10 F11 F12 PrtSc ScLk Pause
ʕ `¬ 1 ! 2 " 3 £ 4 $ 5 % 6 ^ 7 & 8 * 9 (0) - _ = + Backspace Insert Home PgUp NmLk / * -
Tab Q ʍ W ø E ɾ R θ T Y ʊ U I O P [{] } Delete End PgDn 7 8 9
CapsLk œ A ʃ S D ɸ F G ɦ H ʎ J K ɭ L ; : ' @ # ~ Return 4 5 6 +
M1 Shift \ | ʒ Z χ X ç C β V B ɳ N ɱ M , < . > / ? M1 Shift Up 1 2 3
M3 Ctrl Win M2 Alt M2 AltGr Win Menu M3 Ctrl Left Down Right 0 . Enter

Source: Own authorship (2017)

Figure 15: Layout configured to display as ISO (DE)


Esc F1 F2 F3 F4 F5 F6 F7 F8 F9 F10 F11 F12 Druck Rollen Pause
ʕ ^ ° 1 ! 2 " 3 § 4 $ 5 % 6 & 7 / 8 (9) 0 = ß ? ´ ` Einfg Pos1 Bild↑ Num / * -
Q ʍ W ø E ɾ R θ T Z ʊ U I O P Ü + * Entf Ende Bild↓ 7 8 9
œ A ʃ S D ɸ F G ɦ H ʎ J K ɭ L Ö Ä # ' 4 5 6 +
M1 < > ʒ Y χ X ç C β V B ɳ N ɱ M , ; . : - _ M1 ↑ 1 2 3
M3 Strg Win M2 Alt M2 AltGr Win Menu M3 Strg ← ↓ → 0 .

Source: Own authorship (2017)

Physical layouts determine which physical keys are rendered, and where they are rendered. For example, the ABNT-2 physical layout contains two keys not present in the ISO layout: the key to the left of Right Shift, and the thousands separator in the numpad.

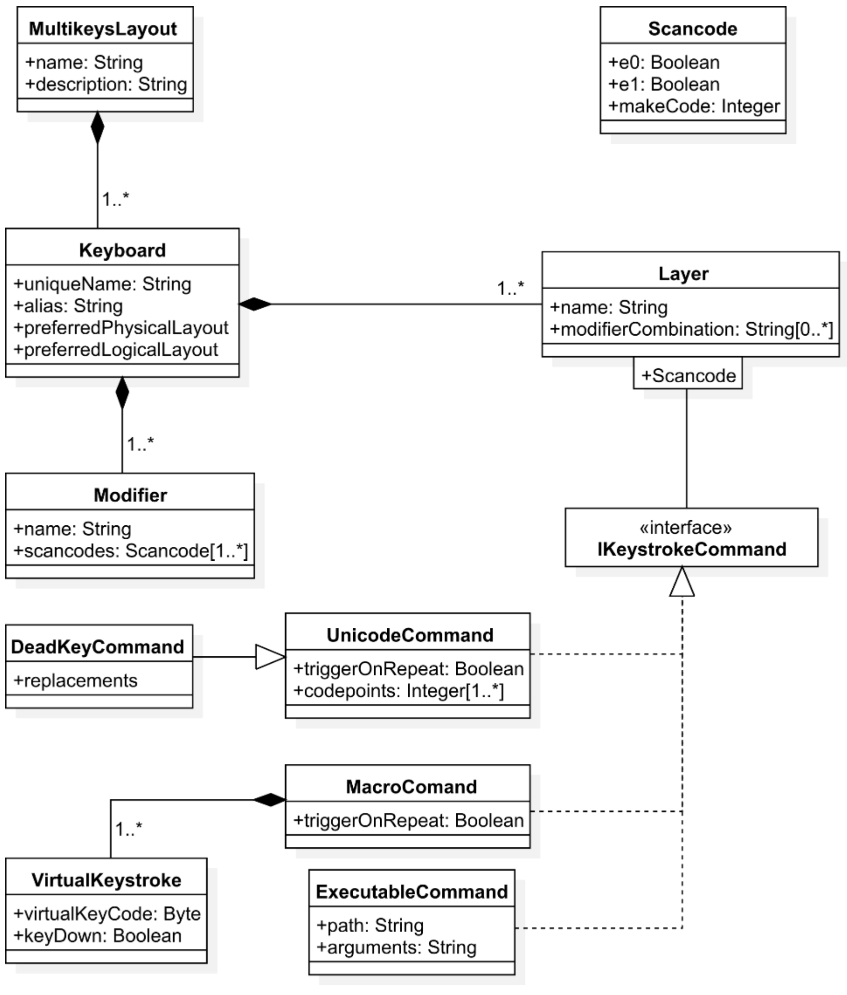
The logical layout is used to determine what text appears labeled on the keys. Figure 14 and Figure 15 use the same physical layout but differ in the labels printed on the keys because the logical layout is different.

The layouts are stored in the project's resource files. Adding a logical layout to the project is simply a matter of adding a new file there.

4.6.4 Domain Model

Multikeys Editor's domain model (shown in Figure 16) has many similarities to the one in Multikeys Core. However, this model does not have any behaviors; rather, it simply stores data. A single instance of the MultikeysLayout class stores all necessary information read from a configuration file.

Figure 16: Class diagram from Multikeys Editor's *model* namespace



Source: Own authorship (2017)

Because XML files are used to persist layout information, Multikeys Editor's persistence layer is relatively simple; it is responsible primarily for two tasks:

1. Reading a configuration file and creating an instance of MultikeysLayout with all the file's information in it, and

2. Persisting an instance of the MultikeysLayout class in a file at a specified path.

In addition to simplifying data persistence, using XML allows for validating the model using the XSD that defines the program's XML rules. The schema file is used to validate a file before attempting to parse it, and to determine if an instance of MultikeysLayout generates a valid xml before writing it into disk.

4.6.4.1 Possible Amount of Keyboard Layers

Because each combination of modifier keys in a keyboard may correspond to a valid layer full of key remaps, the number of possible layers grows exponentially with the amount of modifier keys registered in a keyboard.

For layouts that use a small number of modifiers, this doesn't affect the system's performance; a keyboard with only Shift and AltGr keys registered as modifiers has four valid layers (no modifiers, Shift, AltGr and both). However, in Multikeys, any key on the keyboard can be used as a modifier, resulting in 2^n layers for n modifiers.

It would be possible to implement a limit on the number of registered modifiers for each keyboard, or a limit of nonempty layers in a keyboard.

However, instead of imposing artificial limits, we chose an alternate solution. The list of keyboard layers only contains actual instances for remapped layers, while empty layers are presented to the user normally, available for remapping; the subjacent objects are only instantiated when at least one key is remapped.

4.6.5 Interaction with Multikeys Core

As illustrated in Figure 4, the Multikeys Editor subsystem is one layer above Multikeys Core and encapsulates its functionalities. The communication between the two modules is minimal; there aren't for example, any method calls between them.

Instead of communicating through method calls, the interaction happens through system messages. Multikeys Editor is capable, for example, of sending a message to Multikeys Core to request its termination. Sending messages is possible because the Editor starts the Core process and keeps a reference to it.

In addition to starting the background process, Multikeys Editor also interacts with Multikeys Core to detect the name of a connected keyboard.

For that, a process is started to only listen to keyboard input and output its device name. Because processes can only return integers on termination, the device name is sent to Multikeys Editor using standard output. The output is redirected to obtain that information.

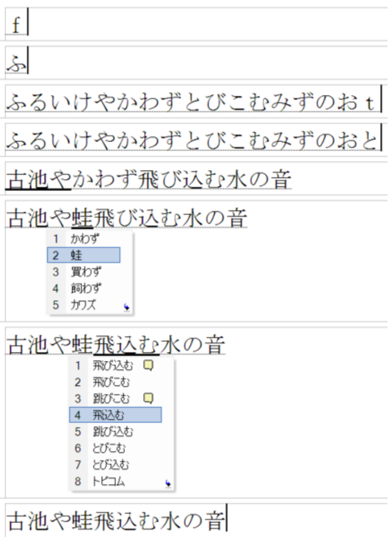
4.7 LIMITATIONS

4.7.1 IMEs

An Input Method Editor, abbreviated IME, is used to process text input for languages that need more characters than can be placed in a keyboard.

Figure 17 illustrates an IME being used to input text; an IME is typically used in conjunction with a specific layout; for example, the additional keys present in the JIS physical layout (Figure 18) have the primary purpose of interacting with an IME for inputting Japanese text.

Figure 17: Demonstration of an IME for inputting text



f

ふ

ふるいけやかわずとびこむみずのおｔ

ふるいけやかわずとびこむみずのおと

古池やかわず飛び込む水の音

古池や蛙飛び込む水の音

1 かわず
2 蛙
3 買わず
4 飼わず
5 カワズ

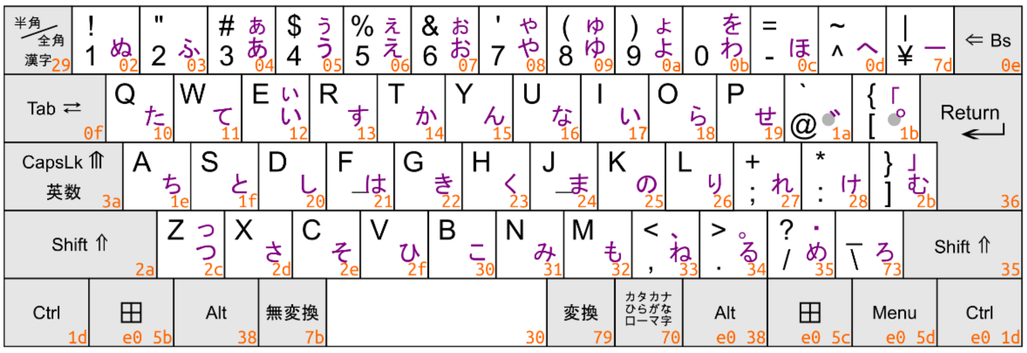
古池や蛙飛込む水の音

1 飛び込む
2 飛びこむ
3 跳びこむ
4 飛込む
5 跳び込む
6 とびこむ
7 とび込む
8 トビコム

古池や蛙飛込む水の音

Source: Adapted from https://commons.wikimedia.org/wiki/File:IME_demonstratie_-_Matsuo_Bashou_-_Furu_ikeya_kawazu_tobikomu_mizuno_oto.png (2007)

Figure 18: Alphanumeric part of the JIS layout



Source: Adapted from BROUWER (2009).

Because the implementation of an IME is outside of this project’s scope (as specified in the vision document), it is not possible to use Multikeys to replicate the behavior of layouts that depend on IMEs. It is possible, however, to use an IME in a non-remapped keyboard, together with one or more keyboard remapped through Multikeys.

Note that during normal usage, language input keys can be remapped through Multikeys.

4.7.2 Fn Key

The function key, often abbreviated Fn, is present as a modifier in many keyboards of reduced size. However, as described in the business rule BR3.4 Fn Modifier Key, this key does not send a scancode to the operating system, and as a consequence it cannot be detected by Multikeys.

As a result, the user cannot remap the Fn key, or any combination of keys that include it. The complete description of business rule BR3.4 can be found in this project’s business rules document.

4.8 PUBLISHING

The code was published in a public repository hosted in GitHub, accessible at the url https://github.com/rafaelktakahashi/multikeys.git. The code was published under the modified BSD license, available as a LICENSE file at the root of the project. A README is also provided with an introduction to the project.

FINAL CONSIDERATIONS

Even though handling keyboard input is often a trivial feature to implement, manipulating and selectively blocking input according to device requires much more in-depth knowledge about message processing and the operating system.

The Multikeys software implemented this feature by using both the Raw Input API and a keyboard hook from the Windows API. However, using those two mechanisms at once results in many incorrect behaviors that must be individually handled. Correcting these cases required knowledge on the PS/2 keyboard standard and the Windows operating system.

Constructing the system's architecture was an interesting step of this project because it contains traits from both event orientation (due to Windows' message system) and object orientation (Remapper's implementation and Multikeys Editor).

Additionally, many architectural decisions, such as storing data in XML and using the C++ and C# languages, were directly influenced by the requirements.

The resulting application has a unique set of functionalities, as it allows for configuring and editing different custom keyboard layouts in different keyboards. The objectives proposed at the beginning of this essay have been achieved, since the functionalities mentioned are currently functioning, including the program's graphical editor.

We expect Multikeys to be able to meet the needs laid out at the start of this essay, and that it may continue to be developed, possibly by others (whether or not for similar motivations); publishing the source code openly reflects this sentiment.

REFERENCES

ACETO, Luca et al., ***Reactive Systems: Modelling, Specification and Verification***, Cambridge University Press, 2007.
Available at: < http://www.cs.ioc.ee/yik/schools/win2007/ingolfsdottir/sv-book-part1.pdf>.
Accessed Oct. 12th, 2017.

GITHUB. ***AutoHotKey_L***. Repositório GitHub, 2017.
Available at: <https://github.com/Lexikos/AutoHotkey_L>.
Accessed Jun. 5th, 2017.

BONÉR, Jonas; KLANG, Viktor, ***Reactive Programming vs. Reactive Systems***. O'Reilly Media, www.oreilly.com, 2016.
Available at: <https://www.oreilly.com/ideas/reactive-programming-vs-reactive-systems>.
Accessed Oct 21st, 2017.

BROUWER, Andries. ***Keyboard Scancodes***, Fakulteit Wiskunde en Informatica, Eindhoven, Netherlands, 2009.
Available at: <https://www.win.tue.nl/~aeb/linux/kbd/*scancode*s.html#toc1>.
Accessed Oct. 5th, 2017.

BLECHA, Vít. ***Combining Raw Input and Keyboard Hook to Selectively Block Input from Multiple Keyboards***, Code Project, 2014.
Available at: <https://www.codeproject.com/Articles/716591/Combining-Raw-Input-and-keyboard-*Hook*-to-selective>.
Accessed Oct. 5th, 2017

CHAPWESKE, Adam. ***The PS/2 Keyboard Interface***, Computer-Engineering.org, 2013. Available at:
<http://www.computer-engineering.org/ps2keyboard/>.
Accessed Jun. 6th, 2017.

CONSTABLE, Peter. ***Understanding Unicode***, SIL International, 2001.
Available at: <http://scripts.sil.org/IWS-Chapter04a>.
Accessed Jun. 6th, 2017.

FARINES, Jean-Marie; FRAGA, Joni da Silva; OLIVEIRA, Rômulo Silva, **Sistemas de Tempo Real**, Universidade Federal de Santa Catarina. Florianópolis, 2000.
Available at: <http://www.romulosilvadeoliveira.eng.br/livro-tr.pdf>.
Accessed Oct. 17th, 2017.

FAWCETT, Joe; QUIN, Liam R. E.; AYERS, Danny. ***Beginning XML***. 5th ed. John Wiley and Sons: Indiana, EUA, 2012

FOWLER, Martin**,** ***Focusing on Events***, *www.martinfowler.com*, *Thoughtworks*, 2006a.

Available at: <https://martinfowler.com/eaaDev/EventNarrative.html>. Accessed Oct. 20th, 2017.

FOWLER, Martin, ***Event Collaboration***, www.martinfowler.com, *Thoughtworks*, 2006b.
Available at: <https://martinfowler.com/eaaDev/EventCollaboration.html>.
Accessed Oct. 20th, 2017.

FOWLER, Martin, ***What do you mean by "Event Driven"?***, *Martinfowler*.com*,* *Thoughtworks*, 2017.
Available at: <https://martinfowler.com/articles/201701-event-driven.html>.
Accessed Oct. 8th, 2017.

GAMMA et al., ***Design Patterns: Elements of Reusable Object-Oriented Software***, Reading, Mass: Addison-Wesley, 1995.

HOHPE, Gregor. ***Programming without a Call Stack****: Event-Driven Architectures*. *Enterprise Integration Patterns*, 2006.
Available at: <http://www.enterpriseintegrationpatterns.com/docs/EDA.pdf>.
Accessed Oct. 6th, 2017.

HOSKEN, Martin. ***An Introduction to Keyboard Layout Design Theory***: What Goes Where?, SIL *International*, 2003.
Available at: <http://scripts.sil.org/keybrddesign>
Accessed Jun. 7th, 2017.

JUVKA, Kanaka, ***Real-Time Systems****. Carnegie Mellon University,* 1998.
Available at: <https://users.ece.cmu.edu/~koopman/des_s99/real_time/>.
Accessed Oct. 20th, 2017.

KAPLAN, Michael S.; WISSINK, Cathy. ***Unicode and Keyboards on Windows***. *23rd Internationalization and Unicode Conference*, República Checa, 2003.
Available at:
<https://sites.google.com/site/talapornmon/Unicode-KbdsonWindows.pdf>.
Accessed Jun. 7th, 2017.

LEVINA, Olga; STANTCHEV, Vladmir, *A Model and an Implementation Approach for Event-Driven Service Orientation*. **International Journal on Advances in Software**, vol. 2, 2009.
Available at:
<http://www.academia.edu/993297/A_Model_and_an_Implementation_Approach_for_Event-Driven_Service_Orientation>.
Accessed Jun. 7th, 2017.

LIPARI, Giuseppe; PALOPOLI, Luigi, ***Real-Time Scheduling:*** *From Hard to Soft Real-Time Systems, Cornell University,* arXiv, 2015,
Available at: <https://arxiv.org/pdf/1512.01978.pdf>.
Accessed Oct. 12th, 2017.

MACDONALD, Matthew, ***Pro WPF 4.5 in C#***: *Windows Presentation Foundation in .NET 4.5, 4th Ed*. Apress Press, 2012.

MEDEK, Petr, ***HID macros*****,** GitHub repository, *2014.*
Available at: <https://github.com/me2d13/luamacros>.
Accessed Jun. 7th, 2017.

MICROSOFT, ***Windows API Index***, Microsoft, 2017.
Available at:
<https://msdn.microsoft.com/en-us/library/windows/desktop/ff818516(v=vs.85).aspx>.
Accessed Nov. 5th, 2017.

PALMER, Grant; BARKER, Ken, ***Beginning C# 2008 Objects***, 1st Ed. Apress Press, 2009.

PETZOLD, Charles, ***Programming Windows****, 5th ed. Microsoft Press,* Washington, 1998.

RAYMOND, David. ***Keyman*** *– Keyboard Systems for Windows, Mobile Devices and Web*. Tavultesoft Pty Ltd (SIL International) 2014.
Available at:
<http://scriptsource.org/cms/scripts/page.php?item_id=entry_detail&uid=vsphmhqr6h>.
Accessed Jun. 7th, 2017.

RICHTER, Jeffrey; NASARRE, Christopher. ***Windows via C/C++***, 5th ed. *Microsoft Press*, Washington, 2008.

SAHAY, Sourav, ***Object Oriented Programming with C++***, *2nd ed. New Delhi: Oxford University Press*, 2012.

STANKOVIC, John A. ***Real-Time Computing***. University of Massachusetts. Amherst, MA, 1992.
Available at:
<https://pdfs.semanticscholar.org/9f6a/3616dc9d9e1a8481681131dd7ae42aac3ac3.pdf>.
Accessed Oct. 17th, 2017.

UNICODE CONSORTIUM. ***The Unicode Standard****, Version 9.0.0*. *Mountain View,* CA: *The Unicode Consortium*, 2016.
Available at:
< http://www.unicode.org/versions/Unicode9.0.0/UnicodeStandard-9.0.pdf>.
Accessed Jun 8th, 2017.

WELLS, John. ***An Update on Phonetic Symbols in Unicode***. University College London, Saarbrücken, 2007.
Available at: <http://www.icphs2007.de/conference/Papers/1357/1357.pdf>.
Accessed Jun 8th, 2017.

APPENDIX A: VISION DOCUMENT

Multikeys

FATEC
Faculdade de Tecnologia de Mogi das Cruzes

Version History

Date	Version	Description	Author	Reviewer
2017-05-02	0.1	Draft	Rafael Kenji	-
2017-05-27	1.0	First version	Rafael Kenji	-
2019-07-28	1.1	English translation	Rafael Kenji	-

Index

1 Objectives

This document is an overview of the project, stating the problem, current needs and scope. It provides a high-level definition of the project that will meet those needs.

The document aligns the expectations of interested parties and formalizes the beginning of the project, defining needs and the high level features of the Multikeys system.

1.1 Scope

This project's scope is the development of a desktop application for creating custom layouts to be used in specific keyboards connected to the computer, though the interception and modification of user input using the Windows API. A graphical interface must exist to facilitate user customization.

The customized layouts must replicate the functionality of a traditional keyboard layout in the Windows operating system, including modifier keys and dead keys; additional technologies, such as input method editors (IMEs) are outside of this project's scope.

1.2 References

This document makes reference to other documentation in this project:

- Requirement analysis document: Formalizes the functional and non-functional requirements of the project and describes them in depth.
- Business rules document: Formalizes the project's business rules.
- Use case document: Lists all use cases for the software, as well as each activity flow.

2 Project's Needs

There are users who frequently need to insert special characters in their computers. For example, the following areas of knowledge make frequent use of special symbols:

- Mathematics, where symbols are used to convey many different concepts; the symbols used in mathematics are numerous and include "→" (implies), "Δ" (Delta), "∊" (is an element of), and others;

- Linguistics, where the International Phonetic Alphabet (IPA) is used to represent all pronounceable sounds, and includes symbols like "ʔ" (glottal stop) and "ɹ" (alveolar aproximant);

Most of these characters are used in specific areas, and aren't included in layouts built into the operating system. For that reason, inputting special characters depends on features of text editors and third-party software.

One alternative would be to use an additional keyboard remapped to input special characters.

Various pieces of software allow the user to extend keyboard functionality, as evidenced by the use of macros in computer games and scripting like AutoHotKey. Even still, the programs that exist for the creation of custom keyboard layouts have different problems, like closed-source, unintuitive interface and lack of support for remapping multiple keyboards.

This last problem in particular makes it impossible to remap different input devices for different purposes (for example, programming a second keyboard to insert mathematical symbols) without affecting all connected keyboards the same way.

The Multikeys system will allow the creation of custom layouts. Its main features will enable the user to apply multiple keyboard layouts simultaneously to different keyboards makes it possible to use more than one keyboard, which is normally impossible.

For users who make frequent use of special characters, this would mean easy access to any Unicode-representable symbol; aside from specialized areas (like mathematics and linguistics), the system would also enable the insertion of text in multiple languages without switching keyboard layouts. A keyboard dedicated to macros would also be possible. A graphical user interface is also planned to allow use with little background knowledge.

3 Project's Objectives

Develop a desktop application capable of:

- Allowing the user to create and apply custom layouts that replicate the functionality of built-in Windows keyboard layouts; this includes modifier keys (e.g. Shift, Ctrl, Alt) and dead keys, without the need for installing additional layouts in the operating system;
- Providing easy and intuitive creation of custom layouts;
- Intercepting, analysing and altering signals from the keyboard in order to simulate a custom keyboard layout;

- Assign different layouts to different keyboards connected to the computer, allowing for simultaneous usage.
- Additionally, assigning sequences of keystrokes and opening executables in response to a keystroke.

4 Preliminary Statement on Scope

4.1 Description

The Multikeys keyboard layout creation system will enable the user to assign any character or sequence of characters, as well as program execution, to the keys of a conventional computer keyboard, in order to extend the functionality of said devices. The capability of inserting arbitrary characters into any program in the user's desktop will increase convenience for users who frequently need to insert symbols that are absent from conventional keyboard layouts provided by the operating system.

Aside from layout customization, another key feature of the proposed system is recognizing specific input devices, in order to respond differently to different keyboards. A possible application for this feature is using two or more keyboard layouts simultaneously, possibly for writing text in multiple languages.

4.2 Product Overview

The following items are considered products that will result from this project:
A public repository, containing the source code and the following:
- An installer for the application, including a graphical user interface;
- All the documentation generated during the project's development;
- A User's guide.

4.3 Requirements

This section gives a high-level overview of the system's requirements. A more in-depth description will be available in the requirements document.

4.3.1 Functional Requirements

4.3.1.1 Application

The main application must be capable of simulating the typical features of keyboard layouts built into the operating system. That is:

- Mapping physical key signals (*scancodes*) to Unicode characters;
- Support several modifier states, depending on the combination of pressed modifiers (Shift, AltGr and possibly others);
- Support dead keys, which modify the character assigned to the key pressed next;

The program also has the following additional requirements for offering the features described above:

- Mapping physical keys to sequences of Unicode characters or simulated keystroke signals;
- Mapping physical keys to executable files on disk;
- Discriminating keyboard signals by the origin device, in order to behave differently in response to different keyboards;

Though Input Method Editors (IME) are used for inputting certain languages, like chinese and japanese, where the number of required characters is much larger than representable in a keyboard, the technology is outside of the proposed project's scope.

4.3.1.2 Graphical Inteface

The functional requirements pertaining to the graphical user interface are as follows:

- View custom keyboard layouts (including modifier states), read from an xml configuration file;
- Create, edit, save and delete custom keyboard layouts;
- Edit the key mappings for each layout;
- Apply existing keyboard layouts to the keyboards connected to the computer;
- Start and stop the execution of layouts;
- Change the display language.

4.3.2 Non-Functional Requirements

Platform: All of the system's features must function correctly in the Windows operating systems starting with Windows 7.

Response time: Because the proposed system will need to respond to user input in real time, response time must be sufficiently quick, as perceptible delays in the program's response during execution are undesirable for the user.

Robustness: In case of error, it is undesirable to interrupt the user with error popups and similar; those actions must be avoided. Execution error must be handled in such a way as to not cause collateral effects, and normal operation must be resumed as soon as possible.

Resources: Because the program is proposed as a background process, it should have a low system resource usage, and must not block the use of resources by other applications, besides the keyboard. In particular, memory leaks are not acceptable, and the program must be capable of running for very long times without any performance degradation whatsoever.

Availability: Due to its event-oriented nature, the program must be always ready to respond to user input. In addition, starting and stopping the execution of a custom keyboard layout must not require interruptions in other user activies (for example, restarting the computer), and must be able to happen seamlessly and at any moment.

Open source: Given that the main features of the proposed system include intercepting, processing and modifying user input, it is expected for many users to be concerned about the system's internal operations. With that in mind, the program's source code will be openly published, under a free software license.

4.3.3 Business rules:

This section describes the typical behavior of keyboard input, which must be replicated during the implementation of the system's requirements.

A more in-depth description will be available in the business rules document.

4.3.3.1 Physical Layouts

- Different physical layouts should be available to the user; physical layouts are defined by standards, and specify the positioning and shape of the physical keys on the keyboard.

4.3.3.2 Aphanumeric keys

- Every key sends a signal upon being pressed or released; when held down, the key continues to send the same keypress signal repeatedly with an interval configurable in the operating system's settings;
- Alphanumeric keys send one or more Unicode characters to the focused window.

4.3.3.3 Modifier Keys

- A modifier key changes the layout's state until it is released;
- Combinations of modifier keys also count as different keyboard states;
- Modifier keys can correspond to more than one physical key, like the Shift modifier, which correspond to both scancodes 2a and 36, one on either side of the keyboard;
- In layouts that use the AltGr modifier, the Right Alt key (scancode e0 38) sends the scancodes 1d (Left Ctrl) followed by 38 (Left Alt) instead of its own scancode.

4.3.3.4 Dead Keys

- A dead key does not send a character to the focused window, as an alphanumeric key would do, but instead places the keyboard in a state that changes the next key according to predetermined substitution rules; the state is reverted as soon as the next key is pressed;
- Dead keys cannot be combined; when two dead keys are pressed down, the second keypress is considered a candidate for substitution;
- The Tab, Esc and modifier keys are not considered as candidates for substitution, and do not revert the keyboard state;
- A dead key has an independent representation which usually, but not necessarily, corresponds to the standalone representation of a diacritic symbol (such as the tilde "~");
- An invalid sequence (which occurs when the keypress after the dead key is not suitable for replacement) causes the dead key's independent representation to be sent, followed by the normal (non-substituted) representation of the second keypress. If the second key is also a dead key, the independent representation of both is used.

5 Assumptions

Operating systems of the Windows NT family will be supported, starting from Windows 7. Support for Windows XP and Vista is not planned, but may occur futurely;

other operating systems will not be supported, due to the high dependency on the Windows API.

The program will be open-sourced. This does not extend to third party components, tools, libraries and platforms, but open-source dependencies are preferred.

6 Stakeholder Influence

The target audience of this application is very broad, and as such, we expect concerns with the program’s internal behavior, given the necessity to intercept user input and its potential use as a keylogger for similar purpose. It is due to those concerns that the source code for the proposed application will be freely available as open source.

The broadness of the target audience also makes localization in several languages a desirable feature.

REQUIREMENTS DOCUMENT

Multikeys

FATEC
Faculdade de Tecnologia de Mogi das Cruzes

Tecnologia em Análise e Desenvolvimento de Sistemas | Tecnologia em Agronegócio | Tecnologia em Recursos Humanos

Version History

Date	Version	Description	Author	Reviewer
2017-05-02	1.0	First version	Rafael Kenji	-
2017-09-24	1.1	Additional requirements	Rafael Kenji	-
2017-10-01	1.2	Requirements review	Rafael Kenji	-
2017-11-28	1.3	Final version	Rafael Kenji	-
2019-07-28	1.4	English translation	Rafael Kenji	-

Index

1 Introduction

1.1 Purpose of this Document

The purpose of this requirements document is to describe the functional and non-functional requirements for the Multikeys software for keyboard remapping and custom layout creation. These requirements include a detailed description concerning features, performance, data and other relevant traits of the system.

1.2 Target Audience

The target audience for the Multikeys software are users of the Windows NT family operating system starting with Windows 7, with no limitations in regards to region or language.

1.3 Scope of the Project under Development

Multikeys' scope includes its distinct features, its potential applications and its limitations.

Its features are:

- Keyboard layout customization;
- Remapping keys in specific keyboards;
- Assigning Unicode characters or macros to specific keys;
- Executing and opening files in response to keyboard input;
- Replicating the behavior of a typical keyboard layout built into the operating system, with features such as modifier keys and dead keys.

Potential applications for the Multikeys software include:

- Using two or more keyboard layouts simultaneously, for translators, language learners, and other users who make frequent use of multiple keyboard layouts;
- Creating a macro keyboard to increase user productivity in software that makes frequent use of keyboard shortcuts or sequences thereof;
- Using a dedicated keyboard for inserting special Unicode characters for mathematicians, linguists and other users who frequently need to insert less common Unicode characters not present in keyboard layouts built into the operating system.

However, the Multikeys software will not implement the following features:

- Automating user input in games (a.k.a. bots);

- Replicating or imitating the behavior of IMEs (Input Method Editors), also called input method editors.

1.4 Definitions, Acronyms and Abbreviations

This section defines terms used throughout this document, as to clear up their meaning in this context.

Term	Definition
API	Application Programming Interface.
C++	The C++ programming language; its use in this document refers specifically to Microsoft's Visual C++.
C#	Programming language developed by Microsoft.
Unicode	Text encoding standard capable of representing written content in numerous languages, as well as special characters like mathematical symbols and Emoji.
Scancode	Signal received from the keyboard device representing a keystroke.
WinAPI	WinAPI – Shortened form of Windows API; Refers to the API for operating systems of the Windows NT family.
WPF	WPF – Windows Presentation Foundation, a graphic subsystem made by Microsoft for a window-based UI;

1.5 References

This document complements and references the following documents:
Vision Document
Business rules document

1.6 General Overview

The remaining sections of this document contain the following sections:

External Interface Requirements, describing technologies that are external to the Multikeys system and are required for its correct functioning;

System Requirements, describing graphical user interface requirements, functional requirements pertaining to the Multikeys background process, hardware and software interfaces, data requirements and Multikeys' non-functional requirements, in this order.

2 External Interface Requirements

This section describes external resources necessary for Multikeys' proper functioning.

2.1 Hardware Interfaces

Even though the system frequently interacts with user keyboards, this interaction is accomplished through the WinAPI, and therefore doesn't require specific input devices.

2.2 Software Interfaces

The system makes direct use of WinAPI functions, and by extension, it must run on the Windows operating system. Due to specific features used by Multikeys, the minimum required version of Windows is Windows 7.

Because communication with input devices is done through the WinAPI, there are no specific drivers or other programs required for Multikeys to perform correctly.

In order to minimize software dependencies, the subsystem written in C++ must be compiled statically with the redistributable VC++ libraries, so that no packages are required to exist in the user's computer.

Multikeys' graphical interface, named Multikeys Editor, will be implemented in .NET, and version 4.5 of the platform will be necessary for executing the program. An installer is to be provided as a product, but the system must also be made available as a portable installation, capable of being executed without changing the user's machine.

3 System Requirements

This section lists the system's functional and non-functional requirements and feature descriptions.

3.1 Functional Requirements

This section gives descriptions on the system's functional requirements; the requirements are divided into the program's graphical user interface and the background process.

The GUI requirements refer to the system's features for creating and editing custom keyboard layouts. When these layouts are active, a background process is started; the second group of requirements refer to this process, which is responsible for intercepting and interpreting keyboard input.

The functional requirements listed below are annotated with the following labels:

N – Necessary feature;

D – Desirable feature;

Group 1: Graphical Inteface	
FR1.1 (N)	View layouts
Details: View custom layouts, visually displayed in such a way as to resemble a keyboard layout.	
FR1.2 (N)	Edit layouts
Details: Edit a layout, assigning one or more Unicode characters, or opening executable files on disk, to keys on the keyboard. Editing layouts must also allow for defining modifier keys, layout levels for each modifier combination, and dead keys with replacement tables.	
FR1.3 (N)	Read and save layouts
Details: The system must be able to create, read, save and edit custom layouts stored as configuration files visible to the user.	
FR1.4 (N)	Apply layouts to keyboards

Details: The user must be able to assign custom layouts to specific keyboards; it must also be possible to assign different layouts to different keyboards at a time. A set of remapped keyboards must be stored in a single configuration file.	
FR1.5 (N)	Start and stop layout execution
Details: The must be able to start and stop the background process, which applies the remaps. When in activity, keystrokes will be processed by the system according to the requirements in group 2: Application.	
RF1.6 (D)	Change language
Details: Select the display language of the GUI.	
RF1.7 (D)	Configure physical layout
Details: Change the physical layout displayed for a keyboard; the positioning and shape of keys must change according to the user’s choice.	
RF1.8 (D)	Configure logical layout
Details: Change the logical layout used to display the key labels on screen.	
Group 2: Application	
FR2.1 (N)	Identify keyboard input
Details: Intercept and block keyboard input based on scancodes, and carry out the remapped action according to the active layout; devices must also be distinguished by name or port, allowing for different behavior depending on the device of origin.	
FR2.2 (N)	Simulate keyboard input
Details: The system must be able to simulate Unicode text input to any other application.	
FR2.3 (N)	Simulate keystroke signals
Details: The system must be able to simulate keypresses and sequences thereof.	
RF2.4 (D)	Open executable files
Details: The system must be able to execute files at previously set paths in response to keypresses. Note that executables that run in administrator mode should normally require user permission.	
FR2.5 (N)	Support modifier keys
Details: Replicate the behavior of modifier keys; refer to business rules 3.1 to 3.4.	
FR2.6 (N)	Support dead keys
Details: Replicate the behavior of dead keys; refer to business rules 4.1 to 4.6	
RF2.7 (D)	Support toggle keys
Details: Support configuring toggle keys, with behavior similar to Capslock and Numlock, where pressing and depressing the key once toggles the keyboard state.	

3.2 Non-Functional Requirements

The following list refers to the non-functional requirements of both the graphical user interface and the background process, except where noted.

NFR001	**Hardware Requirements**
Description: The system must not have significant hardware requirements, aside from the operating system’s hardware requirements (refer to requirement NFR002).	
Justification: Additional hardware requirements must be avoided to keep the target audience as broad as possible. Besides, high resource usage could degrade response time, as further detailed in NFR003.	
NFR002	**Software Requirements**
Description: Multikeys must work at least on Windows versions 7, 8.1 and 10.	
Justification: These versions of Windows are used by a large portion of Windows users. Support for older version will be kept a possibility but will not be prioritized during development.	
NFR003	**Response time (background process):**
Description: The background process’ response time must be short enough to be unnoticeable. However, delays when opening an executable is the action bound to a key is inevitable and acceptable.	
Justification: The background process must repond to the user’s input in real time, and noticeable delays in response time would disrupt the user’s workflow.	
NFR004	**Robustness (background process):**
Description: Execution errors must be handled in such a way as to not cause collateral effects, like popup messages. Given any errors, normal operation must be resumed as soon as possible.	
Justification: It is undesirable to ever interrupt the user. Alerting the user about recoverable errors would interrupt their normal workflow, which is not expected of a background process.	
NFR005	**Resource usage (background process):**
Description: The system must never block access to any resources for the user, the operating system or other applications, except for the keyboard, in accordance to Multikeys’ normal operation. A particular problem to avoid is memory leaks, which would block memory from being used by other applications; the system must be able to run for very long lengths of time without performance degradation.	
Justification: Since the program is running as a background process, it must not interrupt the user’s normal workflow outside of normal interaction.	

NFR006	**Availability (background process):**
Description: During execution, the system must be always ready to respond to user input. In addition, starting and stopping the execution of a custom keyboard layout must not require interruptions in other user activies (for example, restarting the computer), and must be able to happen seamlessly and at any moment.	
Justification: Multikeys is largely event-driven, since during execution, processing occurs in response to user input instead of function calls. The system must be able to respond to input messages at any moment.	
NFR007	**Open sourced code**
Description: The program's source code must be freely available under a free software license.	
Justification: Multikeys' features include intercepting, processing and modifying user input. It is understandable for users to be concerned about the system's internal operations.	
NFR008	**Internationalization (GUI)**
Description: The graphical user interface must be available at least in English and Portuguese, and preferrably in other languages as well. The system's architecture must allow for easily adding new translations.	
Justification: Given Multikeys' broad target audience, being available in multiple languages will make it easier to reach a larger amount of users.	
NFR009	**Ease of use (GUI)**
Description: The graphical user interface must be reasonably easy to use, with an intuitive and responsive interface. Using the system must not require previous knowledge of any other system (except the operating system) or programming language.	
Justification: Though there is existing software with similar features, few existing tools offer both features comparable to Multikeys and an easy-to-use interface that doesn't require previous knowledge in specific areas.	
NFR010	**Configuration files**
Description: The user must be able to save their custom layouts in configuration files. These configuration files must be available for the user to manipulate and move to other computers.	
Justification: The system's ease of usage includes manipulating configuration files; this requirement also allows users to share their custom layouts with other users.	
NFR011	**Portable execution**

Description: The program must be able to be executed without installation. A portable installation must be provided so that users may use Multikeys without installing files to the user's computer; configuration files must also not depend on any pre-installed files.
Justification: Users that wish to use their layouts on other computers should not need to install Multikeys on other machines as well.

Business Rules Document

Multikeys

Faculdade de Tecnologia de Mogi das Cruzes

Tecnologia em Análise e Desenvolvimento de Sistemas | Tecnologia em Agronegócio | Tecnologia em Recursos Humanos

Version History

Date	Version	Description	Author	Reviewer
2017-05-09	0.1	First draft	Rafael Kenji	-
2017-05-27	0.2	Second draft	Rafael Kenji	-
2017-09-24	1.0	Complete version	Rafael Kenji	-
2017-10-01	1.1	Rules review	Rafael Kenji	-
2017-11-25	1.2	Final version	Rafael Kenji	-
2019-07-29	1.3	English translation	Rafael Kenji	-

Index

1 Introduction

1.1 Purpose of this Document

This business rules document explains business rules related to the Multikeys system, referring to the typical behavior of built-in keyboard layouts in Windows.

1.2 References

This document complements and references the following documents:
- Vision Document
- Requirements document;

2 Business rules:

This section lists and details business rules and restrictions, related to the normal behavior of keyboard layouts. The following business requirements are defined for the system to conform to the corresponding business rule.

The business rules listed in sections 2.2, 2.3 and 2.4 refer to the usual workflow when using a keyboard. In order to replicate the behavior of a built-in keyboard layout, its precise behavior and different situations must be understood.

2.1.1 Keyboard Layouts

BR1.1	**Physical keyboard standards**
Description: There exist many physical keyboard standards, which define the positioning and shape of keys on the keyboard. Additionally, different physical layouts may contain different amounts of keys, or keys not present in other layouts.	
Business requirement: The graphical user interface must let the user configure the physical layout displayed on screen to match their device.	
BR1.2	**Logical keyboard standards**
Description: A logical layout describes the map from scancodes (physical signals sent by each key) to virtual keys, corresponding to the function of each key; different logical	

layouts are used according to the operating system's localization. The printed labels on each key correspond to a logical layout.
Business requirement: When editing a layout, the visual components on screen must label each key with text corresponding to a logical layout of the user's choice.

2.2 Aphanumeric keys

An alphanumeric key is one that produces one or more characters; most keys on the keyboard are alphanumeric; they do not include, for example, the F1 - F12 function keys.

BR2.1	***Scancodes***
Description: Every key sends a signal upon being pressed or released; when held down, the key continues to send the same keypress signal repeatedly with an interval configurable in the operating system's settings;	
Business requirement: The system must be capable of detecting keyboard signals in the form of scancodes (see functional requirement FR2.1).	
BR2.2	**Unicode representation**
Description: Alphanumeric keys can be identified by a sequence of one or more Unicode characters, which are sent to the focused window. Operating systems of the Windows NT family use the UTF-16 encoding.	
Business requirement: The system must be able to simulate Unicode text input, encoded in UTF-16 (see functional requirement FR2.2).	

2.3 Modifier keys

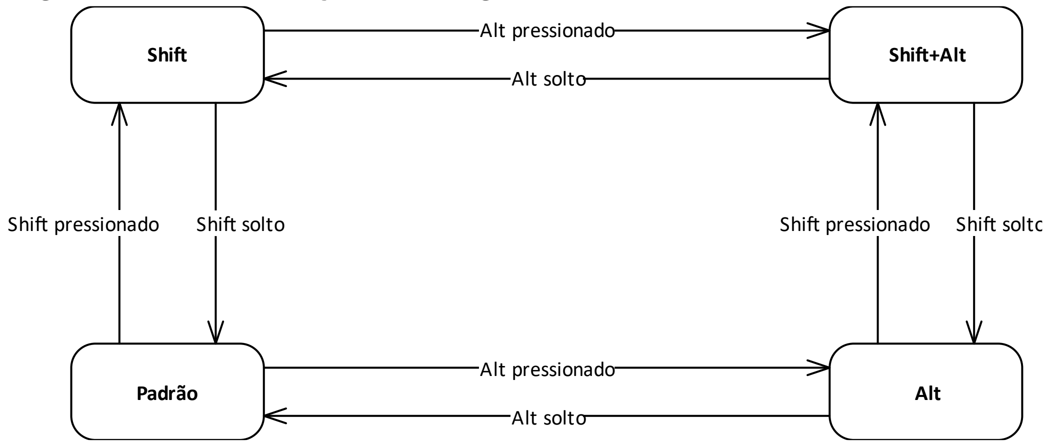
A modifier key is a strategy for increasing the amount of available characters or actions on the user's input device without needing to increase the amount of physical keys. Different keyboard layouts use different modifiers; for example, the AltGr key is not present in all layouts.

BR3.1	**Modifier key state**
Description: A modifier key changes the layout's state until it is released. Combinations of modifier keys also correspond to modifier key states (for example, Shift + Alt puts the keyboard into a different state).	

Business requirement: The system must support editing multiple keyboard layers, one for each modifier combination. For example, key remaps for the Shift + AltGr state must be kept separate from key remaps for the Shift state.	
BR3.2	**Modifier key composition**
Description: Modifier keys can correspond to more than one physical key, like the Shift modifier, which corresponds to both scancodes 2a and 36, one on either side of the keyboard. In this case, the modifier key is considered active while at least one corresponding physical key is being held down.	
Business requirement: The internal representation of a modifier key may contain more than one scancode.	
BR3.3	**AltGr Modifier Key**
Description: In layouts that use the AltGr modifier, the Right Alt key (scancode e0 38) sends the scancodes 1d (Left Ctrl) followed by 38 (Left Alt) instead of its own scancode. Pressing down the Ctrl + Alt combination has the same effect as pressing down AltGr.	
Business requirement: The system must be capable of correctly identifying an AltGr keypress, even when the scancodes are different, and treat it as a Right Alt. Implementing this may possibly require knowing which layout is currently active.	
BR3.4	**Fn modifier key**
Description: The Fn (“Function”) key is present in numerous keyboards of reduced size, usually for accessing numpad keys, media keys and other additional features. However, the Fn key is implemented on a hardware level, changing the physical signals sent by other keys on the keyboard. This means the operating system does not receive any signal for the Fn key.	
Business requirement: Because the Fn key is essentially invisible to the operating system, it cannot be remapped by Multikeys.	
BR3.5	**Scancode map independence**
Description: The state of a modifier key determines which scancode map is used; however, changing the modifier states must not alter where the modifier keys themselves are on the keyboard. That is, keys like Shift and AltGr never change in function of modifier state.	
Business requirement: Keys registered as modifiers must not depend on the current scancode map.	

The state diagram in figure 1 illustrates a layout with two modifier keys. Though built-in keyboard layouts in Windows only allow for Shift, AltGr and Ctrl to be modifiers, it is theoretically possible to allow an unlimited number of modifier states.

Figure 1: Modifier key state diagram



Each modifier state has a map of physical keys (identified by scancode) to characters; the shifted state, for example, has uppercase letters in most layouts.

A keyboard layout that uses more than one modifier key does not necessarily need to use all available space; the canadian multilingual layout, for example, does not have a character map for the Shift+AltGr state; in this case, the keys do not have any effect.

2.4 Dead Keys

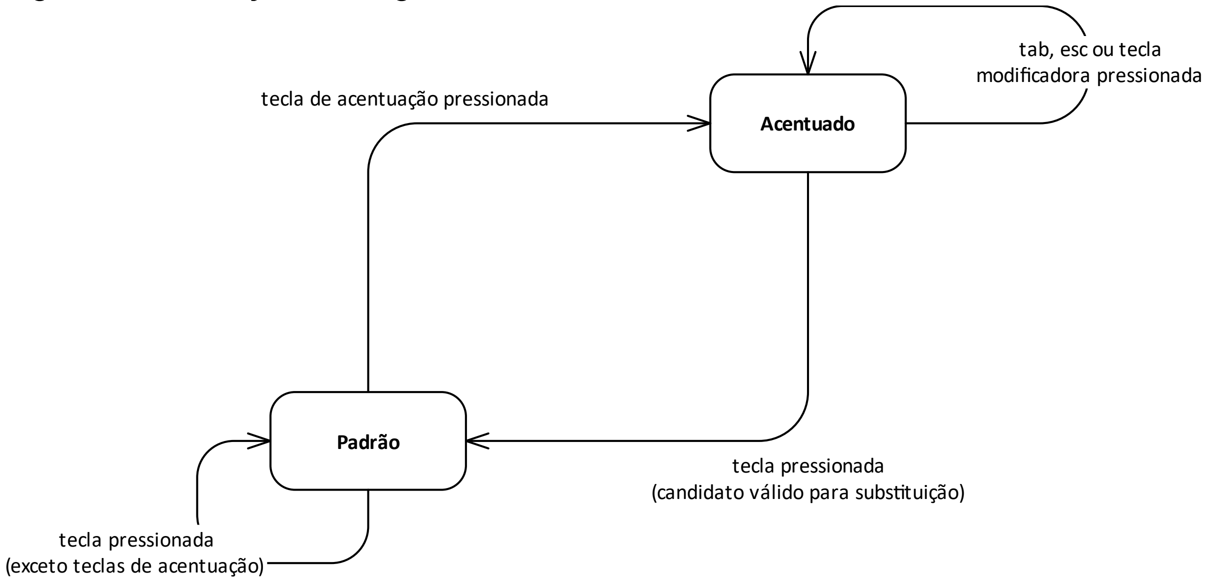
Dead keys serve a similar purpose to modifier keys, in that they both expand the amount of available characters available to the user without extra physical keys. Dead keys modify the next character to be typed by placing the keyboard in a dead key state; for example, the dead key ‘~’, places the keyboard into a state that will replace the next character such that if ‘a’ is typed, ‘ã’ is used instead.

BR4.1	**Dead key state**
Description: dead key does not send a character to the focused window, as an alphanumeric key would do, but instead places the keyboard in a state that changes the next key according to predetermined substitution rules; the state is reverted as soon as the next key is pressed.	
Business requirement: The system must support defining dead keys containing substitution rules; also, the internal representation of remapped keyboards must support dead key states.	
BR4.2	**Dead key combinations**
Description: Dead keys cannot be combined; when two dead keys are pressed down, the second keypress is considered a candidate for substitution.	

BR4.3	Keys invisible to dead keys
Description: The Tab, Esc and modifier keys are not considered as candidates for substitution, and do not revert the keyboard state.	
BR4.4	**Unicode representation of a dead key**
Description: A dead key has an independent representation which usually, but not necessarily, corresponds to the standalone representation of a diacritic symbol (such as the tilde "~").	
BR4.5	**Invalid dead key substitution**
Description: An invalid sequence (which occurs when the keypress after the dead key is not suitable for replacement) causes the dead key's independent representation to be sent, followed by the normal (non-substituted) representation of the second keypress. If the second key is also a dead key, the independent representation of both is sent in sequence.	
BR4.6	**Dead keys and modifier states**
Description: Dead key states persist between modifier states. That is, a dead key pressed while the key is in a certain modifier key state is still valid in other modifier key states.	

The state diagram in figure 2 illustrates how dead keys work. Unlike modifier keys, dead key states don't alter the scancode map of the layout, but makes the next key go through a list of substitution candidades and possibly send another character instead of its own.

Figure 2: Dead key state diagram


tab, esc ou tecla modificadora pressionada
tecla de acentuação pressionada
Acentuado
Padrão
tecla pressionada (candidato válido para substituição)
tecla pressionada (exceto teclas de acentuação)

2.5 Dead Keys

This section explains the behavior of non-alphanumeric keys such as F1-F12, Delete, Pause/Break and others.

BR5.1	**Pause/Break**
Description: The pause/break key generates two Raw Input messages (scancodes 0xe1 0x1d and 0x45), but only one message is received by the keyboard hook (scancode 0x45). This key's scancode is 0xe1 0x1d 0x45, and it's the only key with a three-byte scancode.	
Business requirement: The system must identify this pattern to correctly detect a pause/break key input.	
BR5.2	**Ctrl + Pause/Break**
Description: Another particularity of the Pause/Break key is that it sends a different scancode when the Ctrl modifier is being held down. In this case, the scancode becomes 0xe0 0x46.	
Business requirement: The system must take into consideration that the pause/break produces a different scancode if pressed while the Ctrl key is pressed down.	
BR5.3	**PrintScreen**
Description: The PrintScreen key, when pressed by itself (without any active modifiers) produces a fake Shift keypress (see business rule BR5.5 Fake Shift), followed by the 0xe0 0x37 scancode. Curiously, the Shft + PrintScreen combination does not cause the fake shift message to be sent. The PrintScreen key has another problematic particularity, in that it only generates a break signal, without ever sending the make signal.	
Business requirement: The particular behavior of the PrintScreen key must be taken into consideration when receiving key messages; the system must be aware that there's no make signal for this key.	
BR5.4	**Alt + PrintScreen**
Description: The Alt + PrintScreen combination corresponds to the System Request key, which is often ommited from keyboard labels and has no standard usage. Its scancode is 0x54.	
Business requirement: The system must take into consideration that the pause/break produces a different scancode if pressed while the Alt key is pressed down.	
BR5.5	**Fake Shift**

Description: A fake shift is a signal that didn't originate from a physical keypress. It has scancode 0xe0 0x2a, and it's sent by the driver in certain situations to manipulate the Shift modifier state.	
Business requirement: The system must be aware that this scancode does not correspond to a physical key signal. Care must be taken with keys and key combinations that send a fake shift message.	
BR5.6	**Eisuu / Alphanumeric**
Description: The Eisuu (英数) key, present in the JIS layout, occupies the same space on the CapsLock key. While the japanese keyboard layout is active, this key sends a different message to the keyboard hook.	
Business requirement: The system must be aware that the Eisuu key corresponds to the same physical key as the CapsLock.	
BR5.7	**Korean language keys**
Description: The Hanja (한자, scancode = 0xf1) and Han/Yeong (한/영,scancode = 0xf2) keys present in the Dubeolsik layout send one scancode when pressed down, but none when released. These keys do not repeat scancodes when kept pressed down.	
Business requirement: The system must be aware that the scancodes 0xf1 and 0xf2 do not send break signals.	

ARCHITECTURE DOCUMENT

Multikeys

FATEC
Faculdade de Tecnologia de Mogi das Cruzes

Version History

Date	Version	Description	Author	Reviewer
2017-05-09	0.1	Draft	Rafael Kenji	-
2017-05-27	1.0	First version	Rafael Kenji	-
2017-09-24	1.1	Update	Rafael Kenji	-
2017-10-01	1.2	Data vision update	Rafael Kenji	-
2017-10-08	1.3	Logic vision update	Rafael Kenji	-
2017-11-25	1.4	Final version	Rafael Kenji	-
2019-07-29	1.5	English translation	Rafael Kenji	-

Index

1 Introduction

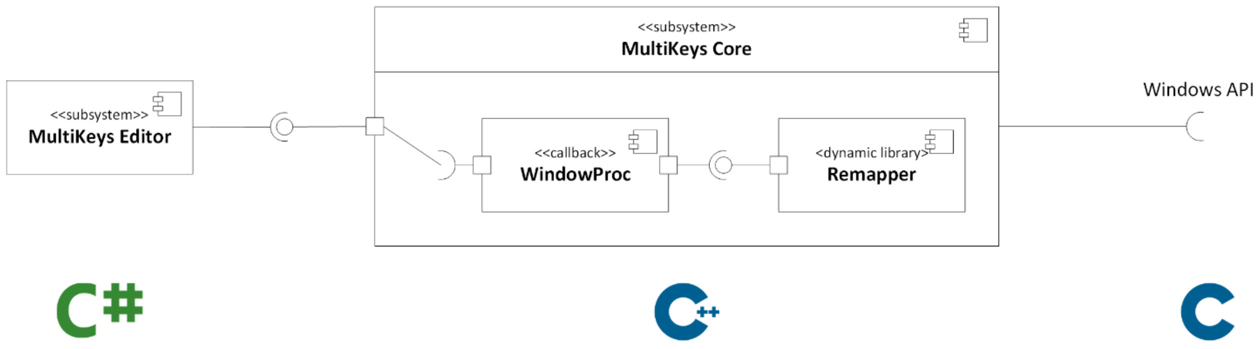
1.1 Purpose of this Document

This document describes Multikeys' architecture, providing a number of architectural views to illustrate different aspects of the system. This document presents and justifies the significant architectural decisions make during the project's development.

2 Architectural Representation

The system will be developed according to the architecture shown in figure 1. This diagram separates the project into subsystems.

Figure 1: Subsystem component diagram



The presented architecture shows that the system is separated into the graphical user interface and the application's core, which are called subsystems. The communication between them is minimal, through configuration files and system messages; in addition, only the application's core will make direct use of the Windows API.

The separation into subsystems is justified by the different needs of the Editor in relation to the core, which actually remaps the user's keystrokes. The Editor exists to provide easy use, eliminating the need for the user to have specific knowledge in order to use and configure Multikeys.

This separation is based on the concept of software layers, where a certain layer can only use resources from lower layers. In this case, the Editor is responsible for starting and stopping the core's execution, among other operations. In turn, the core uses the

OS's resources through calls to the Windows API, and is independent of any graphical interface. Lastly, the operating system itself is obviously independent of the Multikeys system.

The main application is divided into three parts: the window procedure, the business model and the hook, responsible for intercepting keyboard messages.

The Editor contains the presentation layer, the domain logic layer and the model (which consists of domain entities).

To maximize modularity between the two components, there's no common code between the Editor and the Core; instead, they communicate only through the configuration file containing the structure of one or more custom layouts, and also through system messages in execution time.

2.1 Architectural Restrictions

Since the core application will be written in C++, certain components must be implemented accordingly. For example, the interfaces shown in figure 4 are implemented as pointers to purely abstract classes, and the components shown as packages are implemented as namespaces.

These implementations are based on the Gang of Four's design patterns, whose original work provides examples and explanations in C++.

3 Use Case Vision

Preliminarily, the main and most significant use case to the architecture is the application's response to keyboard input. This activity occurs in response to each intercepted keyboard message, and only occurs when the core is active in the background.

Furthermore, configuration use cases are also fundamental to this project's objectives - the ability to create, view and carry out operations in custom layouts, as well as apply these layouts to specific keyboards connected to the keyboard.

Detailed descriptions about each use case, as well as use case diagrams, can be found in this project's use case document. Additionally, the keystroke interception use case can be found in the use case specification for UC3.1.

4 Logic Vision

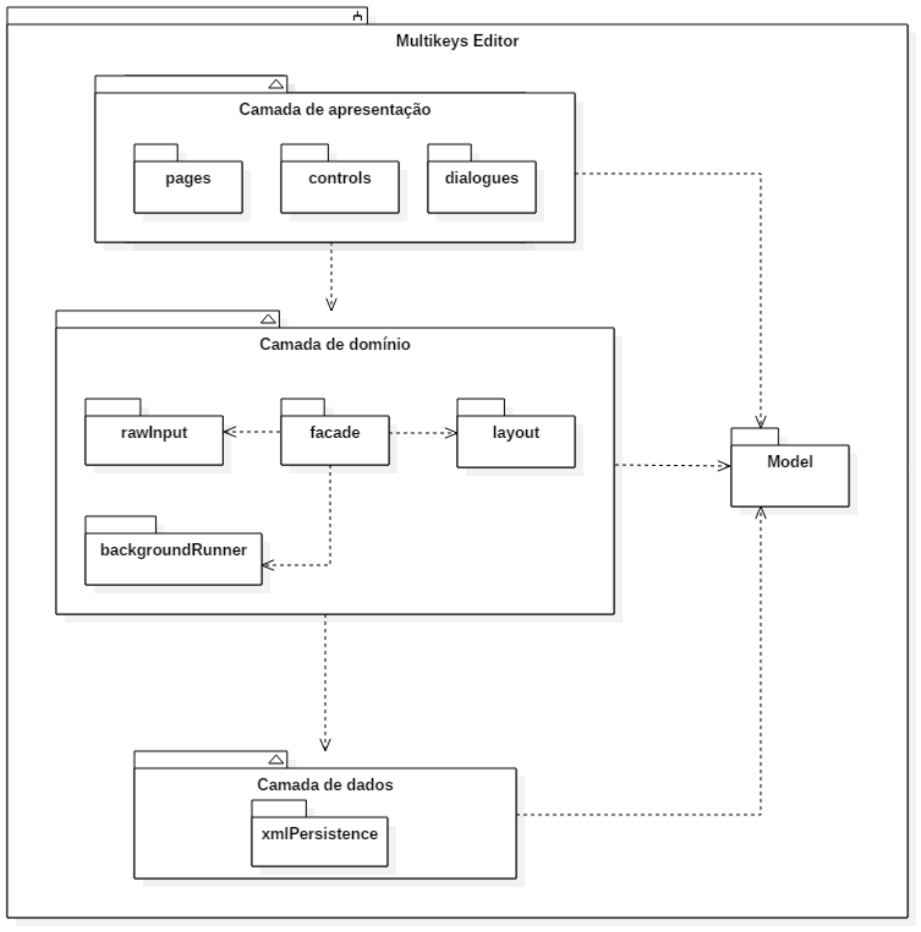
This section details design elements of Multikeys' architecture, and the organization of its components.

4.1 Graphical Inteface

The graphical interface (Multikeys Editor) is the subsystem responsible for interacting with the user through desktop windows, allowing them to edit custom layouts. Considering that the program will be written for Windows, the graphical interface will use the .NET platform and the C# programming language, using WPF (Windows Presentation Foundation) to separate interface from logic.

Even though the Editor by itself works as the whole system's view layer, this subsystem is further divided into its own view layer, domain layer and data layer. The reason for that is the need for manipulating custom layouts in the form of xml files; the business model in this subsystem exists only to model data in a layout, and doesn't provide any behaviors. Additionally, many features and business rules in this system (like switching physical layout standards and detecting a keyboard's name) a grouped in the domain layer.

Figure 2: Multikeys Editor's model diagram



4.1.1 Presentation Layer

This layer contains the graphical components, consisting of XAML windows, as well as their controller classes containing presentation logic. WPF custom components, called *user controls*, are also present in this layer.

4.1.2 Domain Layer

This layer contains the domain objects, representing the structure of Multikeys' layouts.

The package called *backgroundRunner* is responsible for communicating with the core application, as well as managing its lifecycle. The domain layer's functionalities are exposed to the presentation layer through a facade.

Due to the FR1.7 (Configure Physical Layouts) and FR1.8 (Configure Logical Layout), there exists a package named *layout*, containing files describing different physical and logical layouts to determine in execution time where to place the keys and which labels to use depending on user configuration.

4.1.3 Data Layer

The data layer is responsible for reading and writing xml configuration files in disk. Data persistence is not done with a database, but with configuration files in disk that the core can read them and work independently of the editor.

The xml format was chosen due to the hierarchical organization of the data structures, i.e. each layout has many layers, each layer has many remaps and so on.

4.1.4 Model

The model is shown separately in figure 2's diagram because its entities may be used by any of the editor's layers. The model represents the content of a Multikeys configuration file; the entities are data classes and do not encapsulate any behavior.

Figure 3: Class diagram for Multikeys Editor's domain

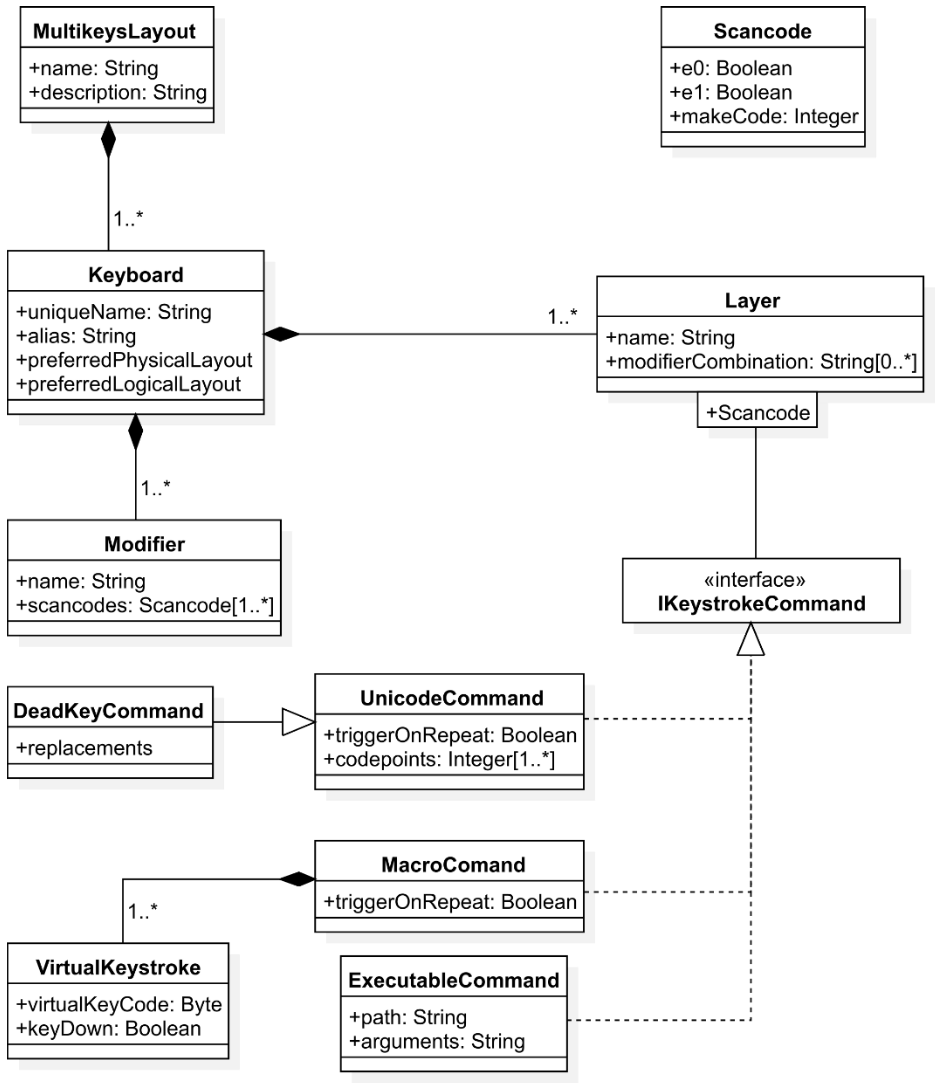


Figure 3 illustrates the classes in the Editor's *model* package. These classes are organized in a tree structure, with the MultikeysLayout at the root. This means a single instance of MultikeysLayout can store all information from a Multikeys configuration file.

4.2 *Multikeys Core*

The Multikeys Core subsystem will be executed in the background, receiving, processing and modifying keyboard system messages. This component is written in C++

due to frequent interactions with the Windows API and also because of performance and responsivity requirements.

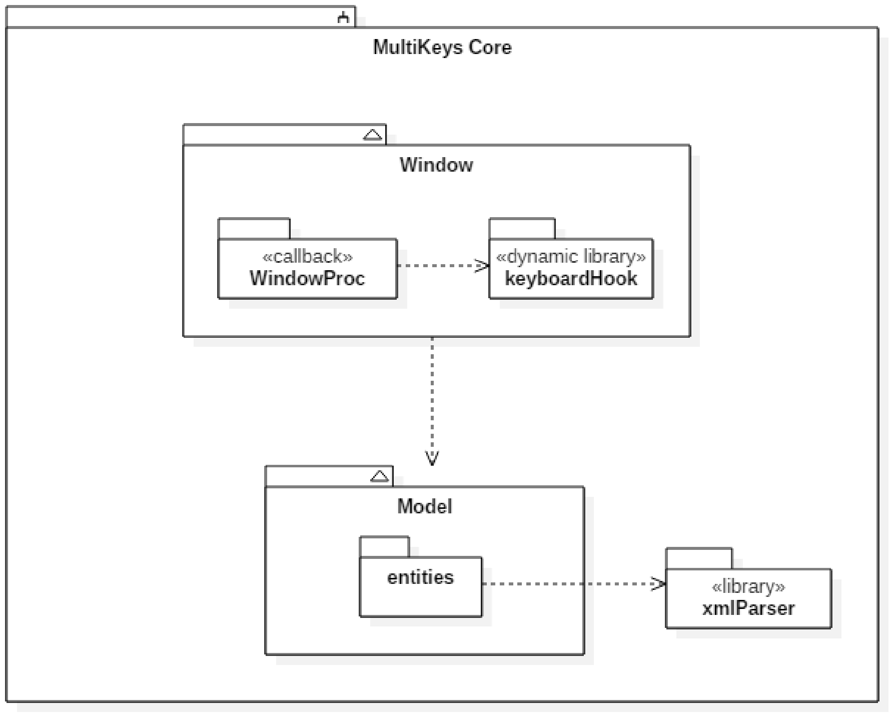
While it is true that many of the features used by this system can be accessed at a higher level by using the .NET framework, this project requires many low level resources (such as identifying the device that sent a certain signal) that justified using C++ to directly use the Windows API.

The core application will have its lifetime managed by the Multikeys Editor subsystem, communicating with it only through system messages. The core will respond to messages to load messages from a configuration file, or to finish execution. Configuration files are primarily manipulated through Multikeys Editor.

The application will have an event-driven architecture, in which a window procedure registers a hook through the Windows API in order to receive system messages referring to keyboard input.

It’s worth mentioning that in Windows it’s possible to write *services*, which are similar to *daemons* in Linux and *agents* in MacOS/OS X, as they all execute in the background, without directly interacting with the user. However, installing services in a computer requires administrator rights, while running window applications (even without a visible window) do not, which is an advantage for usability.

Figure 4: Multikeys Core's model diagram



4.2.1 *Window Procedure*

The Window Procedure, sometimes called WindowProc, is a callback function that processes messages sent to a window (even if the window is not visible). The Window Procedure is necessary to intercept and process data from keyboard devices.

Two mechanisms for intercepting messages are used in this procedure: One is the Raw Input API, provided by the operating system for obtaining low level data, and the other is a keyboard hook, also provided by the system API, installed as a separate dynamic library to intercept messages globally. The dynamic library is pictured in figure 3 as the *keyboardHook* module.

4.2.2 *Business Layer*

The business layer is responsible for keeping a mutable internal model of each of the user's keyboards, updating it as keyboard input is received. This model also contains all the user's remaps, loaded from the configuration file, complete with the actions to execute (which are constructed when the file is read).

4.2.3 Xerces

Multikeys Core uses the Xerces library, developed by Apache, to read the configuration file in xml and parse it into a hierarchical structure and then load it into the internal model.

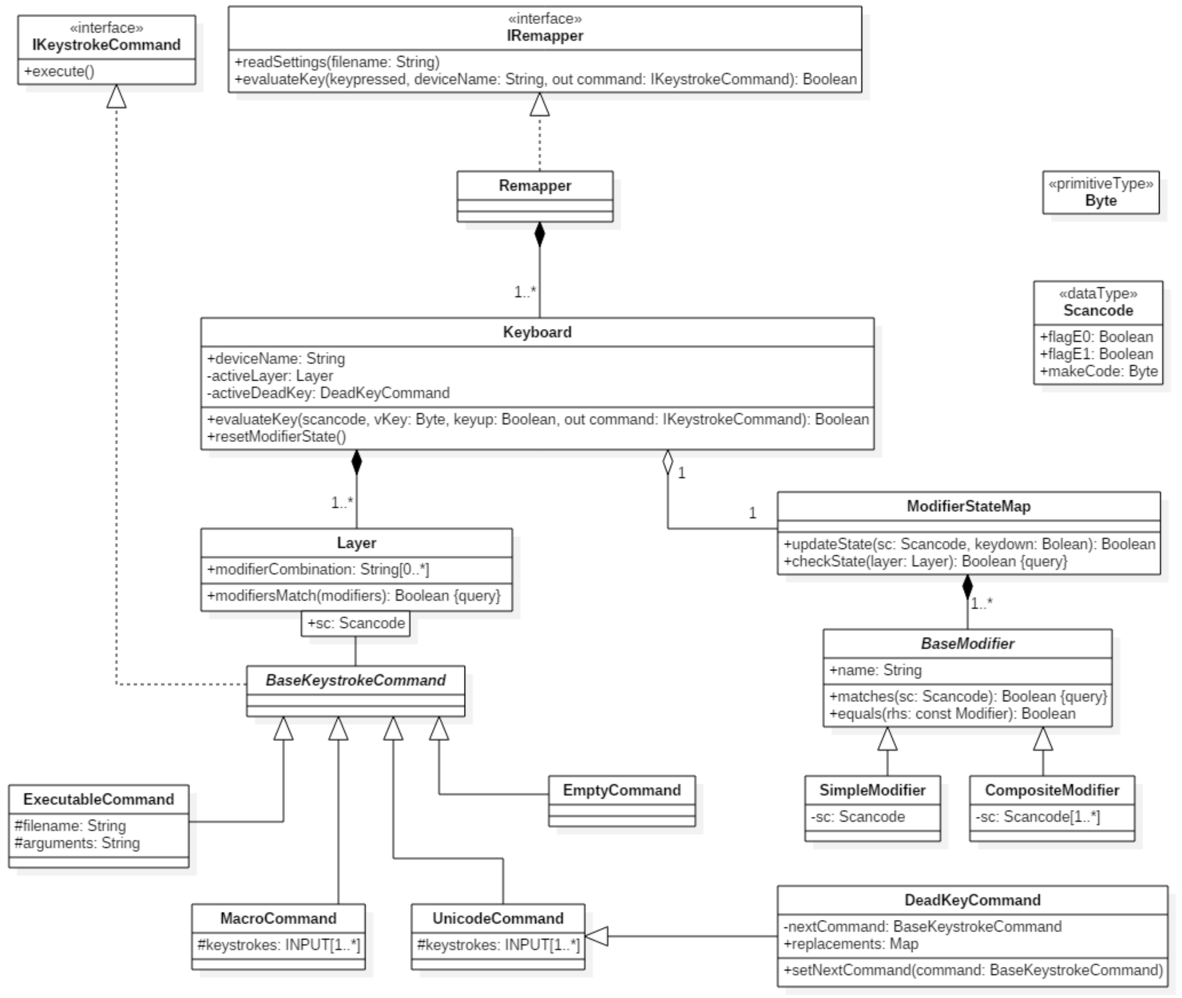
4.2.3.1 Entities

This package contains the internal model of the user’s keyboards, exposed through interfaces accessible by the window procedure. An object of type IRemapper is instantiated and used by the window procedure; this object is responsible for receiving information on each user keypress and returning ICommand instances corresponding to the remapped actions.

The model is encapsulated by functions exposed in a header; those functions instantiate and destroy instances of IRemapper using opaque pointers. As is common in C++, interfaces are implemented as pointers to purely abstract classes.

The choice of using functions to control construction and destruction of instances was done to make future modifications to the entity package less problematic.

Figure 5: Internal model's class diagram

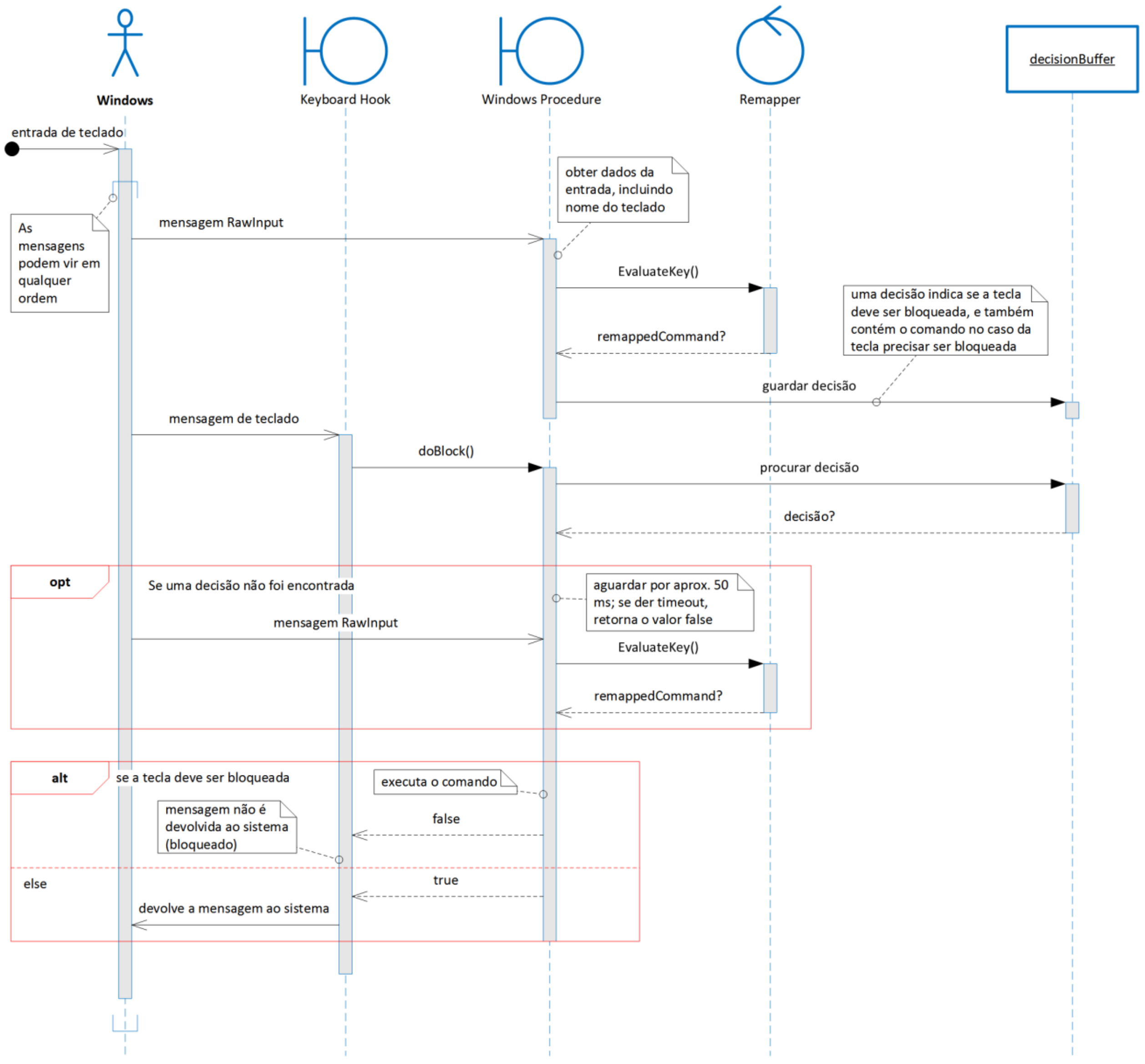

«interface»
IKeystrokeCommand
+execute()
«interface»
IRemapper
+readSettings(filename: String)
+evaluateKey(keypressed, deviceName: String, out command: IKeystrokeCommand): Boolean
Remapper
«primitiveType»
Byte
«dataType»
Scancode
+flagE0: Boolean
+flagE1: Boolean
+makeCode: Byte
1..*
Keyboard
+deviceName: String
-activeLayer: Layer
-activeDeadKey: DeadKeyCommand
+evaluateKey(scancode, vKey: Byte, keyup: Boolean, out command: IKeystrokeCommand): Boolean
+resetModifierState()
1
1..*
1
Layer
+modifierCombination: String[0..*]
+modifiersMatch(modifiers): Boolean {query}
+sc: Scancode
ModifierStateMap
+updateState(sc: Scancode, keydown: Bolean): Boolean
+checkState(layer: Layer): Boolean {query}
1..*
BaseModifier
+name: String
+matches(sc: Scancode): Boolean {query}
+equals(rhs: const Modifier): Boolean
BaseKeystrokeCommand
ExecutableCommand
#filename: String
#arguments: String
EmptyCommand
SimpleModifier
-sc: Scancode
CompositeModifier
-sc: Scancode[1..*]
MacroCommand
#keystrokes: INPUT[1..*]
UnicodeCommand
#keystrokes: INPUT[1..*]
DeadKeyCommand
-nextCommand: BaseKeystrokeCommand
+replacements: Map
+setNextCommand(command: BaseKeystrokeCommand)

The IRemapper and IKeystrokeCommand interfaces are exposed and used in the window procedure. All necessary model instances are allocated when a configuration file is read with the *IRemapper::readSettings()* method.

4.3 Use Case Realization

Figure 6 contains a sequence diagram illustrating the processing done in the window procedure in response to a keystroke. This processing occurs in what we label "Window" in figure 4.

Figure 6: Sequence diagram for responding to a keystroke


Windows
Keyboard Hook
Windows Procedure
Remapper
decisionBuffer
entrada de teclado
As mensagens podem vir em qualquer ordem
mensagem RawInput
obter dados da entrada, incluindo nome do teclado
EvaluateKey()
remappedCommand?
uma decisão indica se a tecla deve ser bloqueada, e também contém o comando no caso da tecla precisar ser bloqueada
guardar decisão
mensagem de teclado
doBlock()
procurar decisão
decisão?
opt
Se uma decisão não foi encontrada
aguardar por aprox. 50 ms; se der timeout, retorna o valor false
mensagem RawInput
EvaluateKey()
remappedCommand?
alt
se a tecla deve ser bloqueada
executa o comando
mensagem não é devolvida ao sistema (bloqueado)
false
else
true
devolve a mensagem ao sistema

In this diagram, the question mark is used to represent nullable objects, which in C++ are implemented as pointers, taking a value of NULL to represent a certain state. This notation is not part of the UML language specification, but is used here to convey behavior that is relevant to the sequence.

The decisionBuffer object is a queue containing decisions on whether to block the key. The class name is not shown in the diagram because its implementation details are not important for the architecture. Each decision in the queue also contains a reference to the command to be executed.

5 Deployment Vision

The system will not work with networking, and an installer (.msi or .exe) will be provided for user convenience. The system will run on any computer running Windows, starting from Windows 7.

A desirable feature for the program would be a portable installation, so that it may be easily used without being installed.

The broad audience means there’s no specific target for testing; hardware specifications are defined in Multikeys’ requirements document as a non-functional requirement.

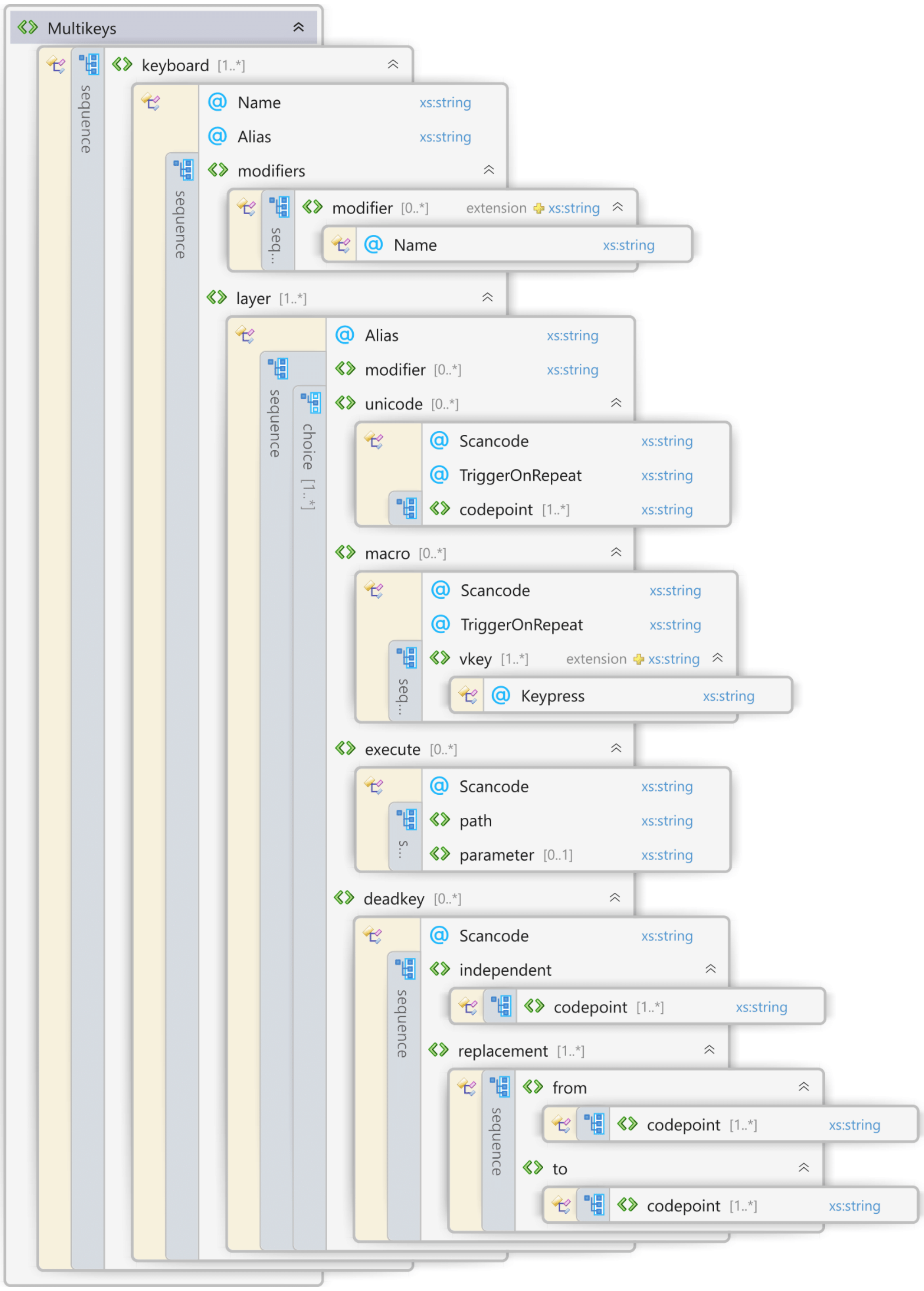
6 Data View

Custom layouts are the data to be persisted between uses of Multikeys. These layouts contain information about the different modifiers, keyboard layers (modifier states), actions attributed to keys, and so on.

Multikeys does not use a DBMS (Database Management System). This is justified by the lack of benefits that this application could extract from using a DBMS, relational or not, such as storing large amounts of data, and data confidentiality. Instead, XML files are used to represent each layout; an XSD schema file is also used to formally define and validate the xml file’s vocabulary and structure.

Figure 7 contains the Multikeys data model, generated in Visual Studio’s XML viewer.

Figure 7: XML configuration file structure



7 Size and Performance

In accordance to Multikeys' non-functional requirements, one of the system's priorities are avoiding perceptible delays during execution. Ideally, all the necessary information for the background process' correct functioning should be loaded in memory as soon as a layout is read from a configuration file, without dynamic memory allocations later on. The objective of this is to ensure that all processing is sufficiently fast as to be imperceptible to the user.

8 Quality

The architecture brought forward in this document fulfills the requirements listed in the requirements document. The separation into subsystems means that each part of the system can be developed with focus on the relevant requirements; that is, the GUI is developed with focused on usability, as to facilitate use without requiring training, and the core is developed with focus on performance and responsivity.

USE CASE DOCUMENT

Multikeys

Faculdade de Tecnologia de Mogi das Cruzes

Version History

Date	Version	Description	Author	Reviewer
2017-05-09	0.1	Draft	Rafael Kenji	-
2017-05-27	1.0	First version	Rafael Kenji	-
2017-10-21	1.1	Finishing up use cases	Rafael Kenji	-
2017-11-25	1.2	Final version	Rafael Kenji	-
2019-07-29	1.3	English translation	Rafael Kenji	-

Index

1 Introduction

1.1 Purpose of this Document

This document provides a list of all use cases for the Multikeys system, save for those that are described in their own documents.

1.2 System Description

The Multikeys system for remapping keys in Windows allows the user to create custom layouts, assigning any Unicode character or keystroke sequences to keys on the keyboard. Furthermore, the user may set these layouts to distinguish between physical keyboard devices, which is not normally possible in the Windows operating system, where multiple keyboards are treated the same way and share state with each other.

2 Use Cases

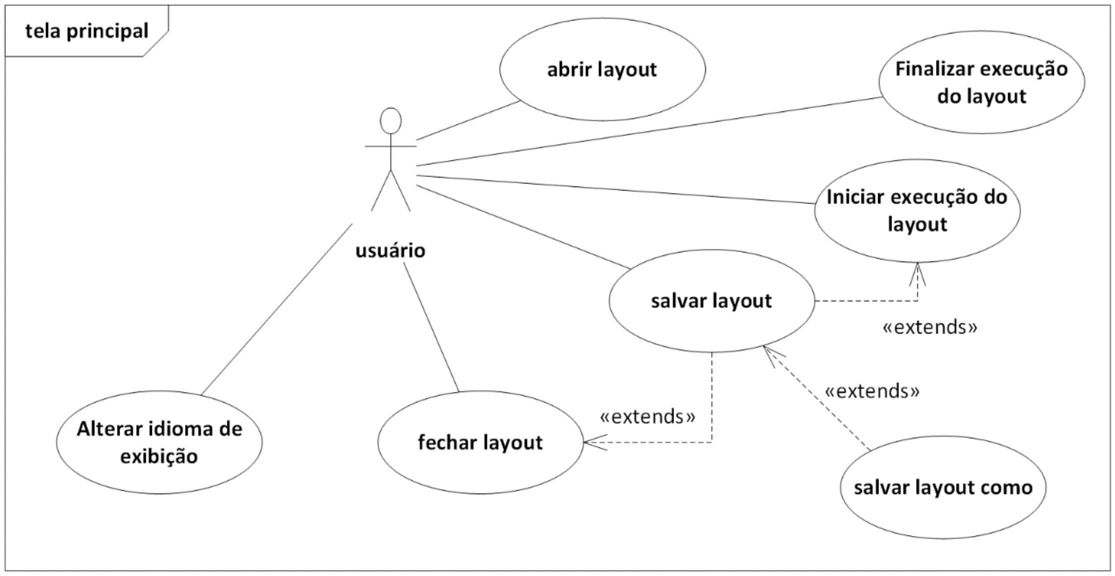
2.1 Multikeys Editor Subsystem

This subsystem enables the user to configure custom layouts. It's the system's GUI, where the user can manipulate layouts and choose remaps between physical keys (identified by scancode) and several actions, such as sending Unicode characters, simulating keypresses or opening executable files in disk.

2.2 Main Window

Error! Reference source not found. displays a use case diagram showing all activities the user may perform from the system's main window.

Figure 1: Main window use case diagram



2.2.1 Open layouts

Code: UC1.1
Description: User opens a layout file to edit using the system.
Pre-conditions: None
Post-conditions: Layout is open and ready to be edited on screen.
Extensions: None
Main Flow:
1. User clicks the button to open a layout.
2. System opens a window for file selection.
3. User navigates to the desired file and selects it.
4. System opens and reads the configuration file.
5. System shows the file contents on screen.
6. System stores the filename of the file being edited.

Alternative Flows:
1a. There is already an open layout being edited.
1a.1. System informs the user that changes will be lost and asks if the user wants to continue.
1a.2. If the user wants to continue, the flow moves onto step 2; otherwise, the flow in interrupted.
4a. System cannot open the selected file.

4a1. System informs the user that the selected file is invalid or corrupted.

2.2.2 Save Layout

Code: UC1.2
Description: User wants to save their changes to the file in disk.
Pre-conditions: There is already an open layout being edited.
Post-conditions: Layout changes are saved to disk.
Extensions: UC1.3 Save Layout As; UC1.4 Close Layout
Main Flow:
1. User tells the system to save a layout (through the toolbar or a shortcut).
2. System looks for the file in disk at the stored path.

Alternative Flows:
2a. The layout still does not exist in disk (was created in this session).
2a.1. Move to step 2 of use case 2.2.3 Save Layout As.
2b. System cannot save the open layout, possibly because the file is open in another application.
2b1. System informs the user of the error.
2c. The file in disk was not found, possibly because it no longer exists in disk.
2c.1. System informs the user that the file was not found and asks if the user wants to save a new layout.
2c.2. If the user wants to save a new layout, move to step 2 of use case 2.2.3 Save Layout As.

2.2.3 Save Layout As

Code: UC1.3
Description: User wants to save a layout that is not yet stored in disk or wants to save the current layout to a different file.
Pre-conditions: There is already an open layout being edited.
Post-conditions: Layout changes are saved to disk at the specified location.
Extensions: None
Main Flow:
1. User tells the system to save as... (through the toolbar or a shortcut).
2. System shows a dialog for the user to choose the file’s name and path.
3. User chooses and confirms the file’s name and path.
4. System creates a new file containing the open layout’s data.

5. From this point on, the system will consider that file as the currently open file.
Alternative Flows:
4a. The is already a file with the specified path and name.
4a1. System asks the user if they want to replace the existing file.
4a2. User answers and the system acts accordingly.

2.2.4 Close Layout

Code: UC1.4
Description: User closes an open layout.
Pre-conditions: There is already an open layout being edited.
Post-conditions: Open layout is closed.
Extensions: UC1.2 Save Layout
Main Flow:
1. User tells the system to close the layout (through the toolbar or a shortcut).
2. System tells the user that unsaved changes will be lost and asks for confirmation.
3. System closes the layout being edited, and no longer uses its path as the currently open layout.
Alternative Flows:
2a. User does not confirm closing the currently open layout.
2a.1. The flow is interrupted.

2.2.5 Start layout execution

Code: UC1.5
Description: User starts the background process (Multikeys Core subsystem) for applying the current layout.
Pre-conditions: The background process is not in execution.
Post-conditions: The background process is started.
Extensions: UC1.2 Save Layout
Main Flow:
1. User gives the command to start layout execution.
2. System starts the background process, passing the currently open layout’s path as a parameter.
Alternative Flows:
1a. The currently open layout has unsaved changes, and therefore, the file in disk is not synced to the information displayed on-screen.

1a.1. System informs the user that there are unsaved changes and asks if the user wants to save the layout.

1a.2. If the user wants to save the layout, use case UC1.2 Save Layout is executed from step 2, and then returns to this flow.

1a.3. Flow returns to step 2 of this use case's flow.

2.2.6 Stop layout execution

Code: UC1.6
Description: User stops the background process (Multikeys Core subsystem).
Pre-conditions: The background process is started.
Post-conditions: None
Extensions: None
Main Flow:
1. User gives the command to stop layout execution.
2. System sends a message to the background process to stop.

Alternative Flows:
2a. The background process has already been stopped by other means.
    2a.1. The process is considered finished, and the flow is interrupted.
2b. The background process has failed to finish or has stopped responding.
    2b.1. The process is forcibly stopped and is considered as finished.

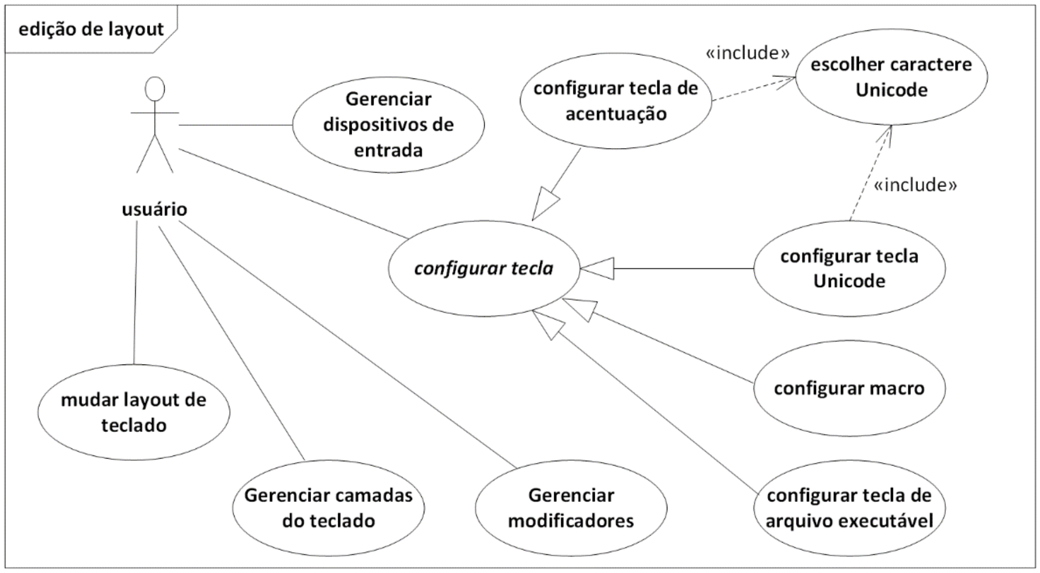
2.2.7 Change Display Language

Code: UC1.7
Description: User wants to change the GUI's display language.
Pre-conditions: None
Post-conditions: The page is reloaded according to the user's chosen display language.
Extensions: None
Main Flow:
1. User gives the command to change the interface to a certain language.
2. System shows the list of supported languages.
3. User chooses the desired language and confirms.
4. System reloads the controls according to the chosen language and stores the new language in the user's preferences.

Alternative Flows:

3a. User does not want to change language.
    3a1. The flow is interrupted, and the display language is unchanged.

Figure 2: Layout editing use case diagram


edição de layout
usuário
Gerenciar dispositivos de entrada
configurar tecla de acentuação
«include»
escolher caractere Unicode
«include»
configurar tecla
configurar tecla Unicode
configurar macro
mudar layout de teclado
Gerenciar camadas do teclado
Gerenciar modificadores
configurar tecla de arquivo executável

2.2.8 Configure Key

Code: UC2.1
Description: Abstract use case; user wants to assign an action to a certain key.
Pre-conditions: A layout is open.
Post-conditions: The selected key has been mapped to an action.
Extensions: None
Main Flow:
1. User selects a certain key for editing.
Alternative Flows:

2.2.9 Configure Unicode Character

Code: UC2.2.
Description: User wants to assign a physical key to send one or more Unicode symbols.
Pre-conditions: A layout is open.

Post-conditions: The physical key has been remapped to send one or more Unicode symbols.

Extensions: None

Main Flow:

1. User selects a certain key for editing.
2. **Include**: Use case UC2.6 Choose Unicode Character.
3. System assigns the key to the Unicode characters chosen by the user.

Alternative Flows:

2a. User cancels the activity.

2a1. System closes the window for inserting characters; the key is not remapped.

2.2.10 Configure Dead Keys

Code: UC2.3

Description: User wants to configure a physical key as a dead key.

Pre-conditions: A layout is open.

Post-conditions: The selected key has been mapped to a dead key.

Extensions: None

Main Flow:

1. User selects a certain key for editing.
2. System displays a window for inputting dead key data.
3. User inputs dead key data (independent characters and the replacement map). **Include**: Use case UC2.6 Choose Unicode Character.
4. System assigns the key to a dead key with the informed data.

Alternative Flows:

3a. User cancels the activity.

3a1. System closes the window for inserting characters; the key is not remapped.

2.2.11 Configure Macro

Code: UC2.4

Description: User wants to configure a physical key with a sequence of simulated keystrokes.

Pre-conditions: A layout is open.

Post-conditions: The selected key has been mapped to a sequence of simulated keystrokes.

Extensions: None

Main Flow:

1. User selects a certain key for editing.
2. System shows a window for selecting virtual keys
3. User makes a list of virtual keys.
4. User gives a name to the macro, to visually display on screen.
5. System assigns the key to a dead key with the informed data.

Alternative Flows:

3a. User cancels the activity.

3a1. System closes the window for inserting virtual keys; the key is not remapped.

2.2.12 Configure Executable Key

Code: UC2.5

Description: User wants to configure a physical key to open an executable in disk.

Pre-conditions: A layout is open.

Post-conditions: The physical key has been remapped to open an executable file in disk.

Extensions: None

Main Flow:

1. User selects a certain key for editing.
2. System shows a window for selecting an executable in disk.
3. User selects an executable file.
4. System gives the option of opening the executable file with parameters.
5. User inserts the parameters for opening the executable with.
6. System assigns the key to opening the executable with the informed parameters.

Alternative Flows:

3a. User does not select a file.

3a1. The activity is canceled; the key is not remapped.

4a. User informs the system that they do not want to open the executable file with parameters.

4a1. Flow skips to step 6.

2.2.13 Choose Unicode Character

Code: UC2.6

Description: System must give the user some way of inserting one more Unicode characters, displaying a summary of the inserted characters.

Pre-conditions: None

Post-conditions: None

Extensions: None

Main Flow:

1. System displays a window for inserting Unicode text, with a field for inserting characters and a panel showing a summary on the inserted characters (codepoints, Unicode name and others).
2. User inserts one or more Unicode characters.
3. System continuously updates the panel with the inserted characters' data. The user may continue to edit the Unicode characters.
4. User confirms the inserted text.

Alternative Flows: None

2.2.14 Change Keyboard Layout

The visual representation of a layout on screen depends on the physical layout to determine the positioning of keys. Additionally, the keyboard control chooses the printed labels on the on-screen controls depending on a logical layout. The user must be able to change which physical layout and which logical layout each configured keyboard uses.

2.2.14.1 Change Physical Layout

Code: UC2.7.1

Description: If the physical layout of any of the user's keyboard is not correctly represented by the on-screen visual arrangement, the user must be able to change the visual arrangement of keys to better represent their physical keyboard (see business rule BR1.1).

Pre-conditions: A layout is open.

Post-conditions: None

Extensions: None

Main Flow:

1. User selects the option to change the layout of one of the keyboards in the currently open layout.

2. System displays a list of physical layout standards for the user to choose from.

3. User chooses a physical layout standard.

4. From this point on, the system will display the on-screen keyboard's key arrangement according to the selected physical standard.

Alternative Flows:

3a. User cancels the activity.

3a1. The activity is canceled; the keyboard is not updated with a different physical layout standard.

2.2.14.2 Change Logical Layout

Code: UC2.7.2

Description: If the on-screen labels do not correspond to the labels on the user's physical keyboard, the user must be able to change the labels displayed on screen to better correspond to their physical device.

Pre-conditions: A layout is open.

Post-conditions: None

Extensions: None

Main Flow:

1. User selects the option to change the layout of one of the keyboards in the currently open layout.

2. System displays a list of logical layouts, also indicating the most suitable physical layout for each option (e.g. logical layout ISO - DE / physical layout ISO).

3. User chooses a logical layout.

4. From this point on, the system will show the keyboard's labels on screen according to the user's chosen logical layout.

Alternative Flows:

3a. User cancels the activity.

3a1. The activity is canceled; the keyboard is not updated with different labels.

2.2.15 Manage Keyboards

2.2.15.1 Add Keyboard

Code: UC2.8.1
Description: User wants to configure a new keyboard to remap its keys.
Pre-conditions: None
Post-conditions: The newly inserted keyboard is available for remapping.
Extensions: None
Main Flow:
1. User gives the command to add new keyboard.
2. System adds a new empty keyboard control to the list of keyboards.
(Note: After this action, the user must configure the keyboard.)
Alternative Flows: None

2.2.15.2 Edit Keyboard

Code: UC2.8.2
Description: User wants to edit an existing keyboard in the currently open layout.
Pre-conditions: Layout is open and has at least one configured keyboard.
Post-conditions: None
Extensions: None
Main Flow:
1. User clicks the icon to edit an existing keyboard.
2. System displays a dialog containing information on the keyboard, automatically filled in with existing information. The user must also have the option to automatically detect the keyboard's device name.
3. User makes all the desired changes.
4. System saves the changes.
Alternative Flows:
3a. User cancels the activity.
3a1. The flow in interrupted and no changes are saved.

2.2.15.3 Delete Keyboard

Code: UC2.8.3
Description: User removes a configured keyboard from the layout; the layout will no longer identify the removed keyboard.
Pre-conditions: Layout is open and has at least one configured keyboard.
Post-conditions: None
Extensions: None

Main Flow:

1. User clicks the option to remove a keyboard from the layout.

2. System asks for confirmation, and also gives the option to dissociate the keyboard from the layout without deleting all information.

3. User confirms the keyboard's deletion.

4. System removes the keyboard and all its remaps.

Alternative Flows:

3a. User does not confirm the keyboard's deletion.

3a1. The flow is interrupted and the keyboard is not removed from the layout.

3b. User decides to dissociate the keyboard without deleting it.

3b1. System removes the keyboard's configuration, and its remaps become inactive.

2.2.16 Manage Keyboards

2.2.16.1 Add Modifier Key

Code: UC2.9.1

Description: User wants to add a modifier to a certain keyboard on the currently open layout.

Pre-conditions: Layout is open and has at least one configured keyboard.

Post-conditions: None

Extensions: None

Main Flow:

1. User gives the command to register a certain key as a modifier.

2. System displays a dialog for editing modifiers.

3. User gives a name to the modifier or adds the key to an existing modifier (see business rule BR3.2 Modifier Key Composition for further information on modifiers bound to more than one physical key).

4. System registers then new modifier for the keyboard.

Alternative Flows:

1a. The selected key has remaps in other layers, and registering it as a modifier would cause those other remaps to be lost (according to the business rule 3.5 Scancode Map Independence).

1a.1. System informs the user that remaps in other layers will be lost and asks for confirmation to continue.

1a.2. If the user wants to continue, the flow moves onto step 2. Otherwise, the flow is interrupted and the key is not registered as a modifier.

3a. User cancels the activity.

3a1. The flow is interrupted, and the modifier is not saved.

3b. User associates the modifier to a key that is already remapped in other layers.

3b1. System confirms the action, with possible information loss.

2.2.16.2 Edit Modifier Key

Code: UC2.9.2

Description: User edits an already registered modifier.

Pre-conditions: Layout is open and has at least one configured keyboard with at least one key registered as a modifier.

Post-conditions: None

Extensions: None

Main Flow:

1. User gives the command to edit an existing modifier in a keyboard.
2. System displays a dialog for editing modifiers.
3. User edits modifier key information.
4. System saves the changes to the modifier.

Alternative Flows:

3a. User cancels the activity.

3a1. The flow is interrupted; changes are not saved.

2.2.16.3 Unregister Modifier Key

Code: UC2.9.3

Description: User wants to unregister a physical key that’s currently registered as a modifier.

Pre-conditions: Layout is open and has at least one configured keyboard with at least one key registered as a modifier.

Post-conditions: None

Extensions: None

Main Flow:

1. User gives the command to unregister a modifier key.
2. System asks for confirmation, informing that layers that depend on this modifier key will be lost.

3. User confirms the action.

4. System will no longer consider the selected key as a modifier key.

Alternative Flows:

3a. User does not confirm the action.

3a1. The flow is interrupted. The modifier key is not unregistered

3a. System determines that unregistering the physical key as a modifier will result in a modifier without assigned physical keys.

3a1. When asking the user for confirmation, the system adds that any layers that depend on this modifier will be lost.

3a2. The flow continues normally into step 4.

2.2.17 Manage Keyboard Layers

The layers of a keyboard layout are different maps from physical keys to actions, and each layer is accessed with a combination of modifier keys. For example, pressing the Shift key in many built-in layouts will access a layer containing uppercase letters.

Layers must be shown in the same control on screen, and the user must be able to navigate between layers by selecting modifier keys on screen. The use cases in this section are illustrated in figure 2.

2.2.17.1 Add Layer

Code: UC2.10.1

Description: User wants to add a new layer to a keyboard. The layer is activated through a specific combination of modifiers registered to the keyboard.

Pre-conditions: There exists at least one modifier key configured in the keyboard to which the user wishes to add a layer.

Post-conditions: None

Extensions: None

Main Flow:

1. User selects the correct combination of modifier keys on the on-screen keyboard control to access the layer.

2. System shows the layer, without any remapped keys.

3. User may map this new layer.

(Note: The system creates instances of layer objects only when necessary, without informing the user).

Alternative Flows: None

2.2.17.2 Edit Layer

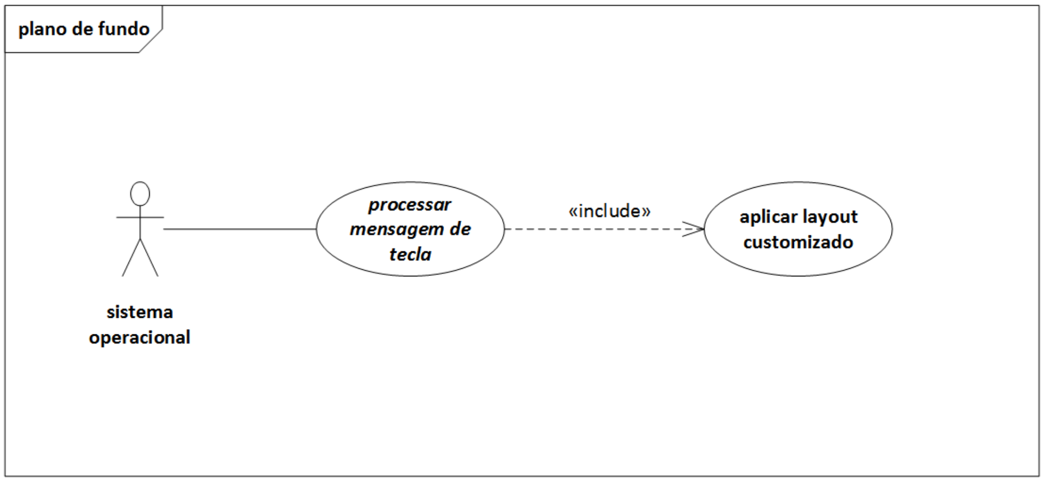
Code: UC2.10.2
Description: User wants to edit a specific layer.
Pre-conditions: There exists at least one modifier key configured in the keyboard.
Post-conditions: None
Extensions: None
Main Flow:
1. User accesses a layer by selecting its modifier combination.
2. System shows a side window with an editable summary of the selected layer.
3. User edits the data.
4. System saves the changes to the currently open layer.

Alternative Flows:

2.3 Multikeys Core Subsystem

This subsystem is responsible for applying the custom keyboard layouts that the user created with Multikeys Editor. Though the user doesn't interact with this subsystem directly, the subsystem responds to keystroke signals from the user.

Figure 3: Multikeys Core's use case diagram



2.3.1 Process Keyboard Input

The use case specification for this use case can be found in the use case document for UC3.1.

2.3.2 Apply Custom Layout

The use case specification for this use case can be found in the use case document for UC3.2.

USE CASE SPECIFICATION

Multikeys

Use Case UC3.1: Process Keyboard Input

FATEC
Faculdade de Tecnologia de Mogi das Cruzes

Tecnologia em Análise e Desenvolvimento de Sistemas | Tecnologia em Agronegócio | Tecnologia em Recursos Humanos

Version History

Date	Version	Description	Author	Reviewer
2017-05-09	0.1	Draft	Rafael Kenji	-
2017-05-27	1.0	First version	Rafael Kenji	-
2017-10-20	1.1	Update	Rafael Kenji	-
2017-11-28	1.2	Final version	Rafael Kenji	-
2019-07-29	1.3	English translation	Rafael Kenji	-

Index

1 Use Case Name

UC3.1 – Process Keyboard Input

2 Objective

The aim of this use case specification is providing a computational solution capable of intercepting and identifying keystroke signals received by the operating system and substituting actions according to a custom layout.

3 Description

Keyboard input from the user is interpreted by the device driver, creating a message that is then processed by the operating system. The message is intercepted by the Multikeys background process, which extracts information about the keypress. The interception is first done through the Raw Input API for obtaining low level data, and again through a keyboard hook, for manipulating the message and possibly blocking it.

4 Requirements

4.1 Functional Requirements

FR2.1 – Identify Keyboard Input

4.2 Non-Functional Requirements

NFR003 – Response Time
NFR004 – Robustness
NFR005 – Resource usage
NFR006 – Availability

4.3 Business Requirements

BR2.1 – Scancodes
BR3.3 – AltGr Modifier Key
BR5.1 – Pause/Break
BR5.2 – Ctrl + Pause/Break
BR5.3 – PrintScreen
BR5.4 – Alt + PrintScreen
BR5.5 – Fake Shift
BR5.6 – Eisuu / Alphanumeric

5 Use Case Type

This use case is concrete, directly initiated by the actor.

6 Actors

6.1 Operating System

In this use case, the Windows operating system is considered the only actor. Even though the user is responsible for causing keystroke messages, they do not interact directly with the Multikeys Core Subsystem, which is a background process.

This interaction occurs though the Windows API. The operating system sends Raw Input messages to all registered receivers, and keyboard input messages are received by any code in the hook chain. Code in the hook chain is also responsible for passing the messages onwards. Furthermore, additional actions like sending a Unicode charcter also happen through the Windows API.

Because the user is never notified of these transactions, they are excluded as an actor in this use case.

7 Pre-Conditions:

7.1 Background Process

The Multikeys background process must be running.

7.2 Keyboard Message

The operating system has received a keyboard input message from the keyboard.

8 Event Flow

8.1 Main Flow

1. Windows sends a Raw Input message to the Multikeys Core Subsystem.

2. Multikeys extracts data from this message, including the device name.

3. Multikeys queries the Remapper to evaluate what must be done with this keystroke, according to the currently loaded custom layout.

Include: The processing done in the Remapper object is described in the UC3.2 use case document.

4. If the keystroke is remapped, a command object is received and stored together with the decision of whether to block it or not.

Note: Decision objects are stored in a decision buffer; each decision is composed of a flag indicating whether or not the keystroke must be blocked, and a command containing the action to be performed. Because the Raw Input API does not allow for blocking user input or manipulating it at all, this decision needs to be stored to be acted on later. However, using the Raw Input API is necessary because hooks do not have access to the name of the device that sent the message.

5. The decision is stored to be read later.

6. Windows sends a keybord message that reaches Multikeys through a keyboard hook registered in the hook chain.

7. The hook queries Multikeys Core, asking whether the intercepted key must be blocked.

Note: The keyboard hook exists as a separate dll in order to be applied globally, intercepting keyboard input originally sent to any window. The method call occurs through Windows' messaging system.

8. Multikeys looks for a stored decision corresponding to the received input.

9. Multikeys retrieves the decision and executes the command if there is one; then, decisions in the decision buffer are removed.

10. The system returns the decision to the hook, which passes the message further up the chain if it's not to be blocked.

8.2 Alternative Flows:

8.2.1 OS Modifier Key

In the operating system, keystroke messages are interpreted together with the state of the Shift, Alt, Ctrl and Windows (sometimes called *WinKey*) modifier keys. In the case of those keys, intercepting the message and not returning it is not enough for the OS to consider them not pressed; this can interfere with other keyboard input messages.

In order to resolve this situation, Multikeys injects release keystrokes when a system modifier key is pressed.

4a. The message is identified as a system modifier key (Shift, Alt, Ctrl or WinKey) which is also remapped in the currently active layout.

4a.1. A message corresponding to a release keystroke for the modifier key is simulated.

4a.2. The flow skips to step 5.

8.2.2 Injected Message

Remapped keys can simulate keypress messages. If not handled, keystroke input simulation would cause Multikeys to intercept its own messages, resulting in infinite loops.

The system handles this by adding information to simulated messages, as to identify them later when they're intercepted by Multikeys.

6a. The keyboard hook interceps a message identifies as having been injected by Multikeys.

Note: The simulated keystrokes sent by the command carry additional data not normally present in keyboard input messages in order to be identifiable by Multikeys itself.

6a.1. The message is ignored by the hook, without blocking it; the message is not sent to Multikeys Core for evaluation.

8.2.3 Pause/Break Message

The Pause/Break message is unique in that it generates a three-byte scancode (0xe1 0x1d 0x45). The keyboard hook intercepts a single message containing the 0x45 scancode, and two messages are received through the Raw Input API (see business rule BR5.1 – Pause/Break).

7a. The intercepted message is identified as Pause/Break.

Note: When the system looks for the Pause/Break key, it should not consider Ctrl + Pause/Break, which sends a different scancode (see business rule BR5.2). This combination does not need special handling.

7a.1. The system must look for the decision for the 0xe1 0x1d key. The second Raw Input message is ignored and has no effect.

7a.2. Flow returns to step 8 of this use case’s main flow.

8.2.4 PrintScreen Message

The PrintScreen key sends a fake Shift message, and also has a different scancode when pressed while the Alt key is pressed down. However, the PrintScreen key has another problematic behavior, in that it only generates a break signal, without ever sending the make signal. This behavior must be corrected, since key remaps in Multikeys are bound to make signals.

7b. The intercepted message is identified as PrintScreen break.

Note: This check must consider that the scancode will be different if the Alt key is currently pressed down.

7b.1. A copy of the received message is simulated, except that the copy is a make message instead of a break message. The message must not be marked as injected by Multikeys.

7b.2. The flow is suspended until the message is interpreted in the main flow.

7b.3. Flow returns to step 8 of this use case’s main flow.

8.2.5 Incorrect Order of Messages

Messages received by Multikeys from the operating system can arrive in any order. Usually, Raw Input messages arrive before the hook intercepts the message, but that’s not guaranteed. If the hook message is received before the Raw Input message, the main flow will start at step 6, causing the hook to not find a corresponding decision; if this happens, the following alternative flow is carried out from step 8:

8a. Decision was not found.

8a.1. The system enters a loop for approximately 50ms, waiting for a delayed Raw Input message.

8a.2. When the Raw Input message is received, steps 2 and 4 of the main flow are executed, in order to extract data from the message and obtain the decision.

8a.3. Then decide:

- If the Raw Input message corresponds to the hook intercepted message, go to the next step.

- If the Raw Input message does not correspond to the hook intercepted message, the system stores the decision in the decision buffer and returns to step 8a.1 to keep waiting. The time until timeout is not reset.

8a.4. Flow returns to step 10 of this use case’s flow.

8.2.6 AltGr Message

The AltGr key behaves differently than other keys (see business rule BR3.3 – AltGr Modifier Key). A message received by the hook containing a Left Ctrl key is received, followed by a delayed Raw Input message containing a Right Alt. A second hook message is then received containing a Right Alt.

8a.1a. If a Left Ctrl message did not find its corresponding decision, the system must instead wait for a delayed Right Alt message.

8a.1a.1. When the Alt message is received, a flag is activated for the hook to ignore the next Right Alt message.

8a.1a.2. The flow is redirected to step 8a.2, to evaluate the Raw Input message corresponding to the Right Alt key.

8.3 Exception Flows

8.3.1 Timeout

The system enters a time-limited loop in step 8a.1. If the loop is finished by time, the following alternative flow is carried out from step 8a.1:

8a.1a. The loop has existed without the expected Raw Input message being received.

8a.1a.1. The system tells the hook that the key is not to be blocked.

9 Post-conditions:

9.1 Availability:

After processing keyboard input, the system must be ready to receive further keystroke message immediately.

10 References

For details about Multikeys' requirements, see the requirements document and the business rules document for this project.

Details concerning other use cases can be found in the use case document for this project, and in UC3.2's use case specification document. Further information on implementations are available in the architecture document.

USE CASE SPECIFICATION

Multikeys

Use Case UC3.2: Apply Custom Layout

FATEC
Faculdade de Tecnologia de Mogi das Cruzes

Tecnologia em Análise e Desenvolvimento de Sistemas | Tecnologia em Agronegócio | Tecnologia em Recursos Humanos

Version History

Date	Version	Description	Author	Reviewer
2017-05-27	1.0	Draft	Rafael Kenji	-
2017-10-21	1.1	Review	Rafael Kenji	-
2017-11-28	1.2	Final version	Rafael Kenji	-
2019-07-29	1.3	English translation	Rafael Kenji	-

Index

1 Use Case Name

UC3.2 – Apply Custom Layout

2 Objective

The aim of this use case specification is providing a computational solution capable of intercepting and identifying keystroke signals received by the operating system and substituting actions according to a custom layout.

3 Description

After Multikeys intercepts a keyboard message, it uses its internal model to know if there's a remapped action for that message. The system is capable of interpreting registered modifier keys, dead keys and other typical features of a conventional keyboard layout.

The task of finding a remapped command assigned to a key is the Remapper module's responsibility.

If there's an action mapped to a key, the action is returned by the Remapper in the form of a command, which uses the Windows API to perform various actions.

4 Requirements

4.1 Functional Requirements

RF2.2 – Simulate Keyboard Input
RF2.3 – Simulate Keystroke Signals
RF2.4 – Open Executable Files
RF2.5 – Support Modifier Keys
RF2.6 – Support Dead Keys

4.2 Non-Functional Requirements

RNF003 – Response Time
NFR004 – Robustness
NFR005 – Resource usage
NFR006 – Availability

4.3 Business Requirements

BR2.1 – Scancodes
BR2.2 – Unicode Representation
BR3.1 – Modifier Key State
BN3.2 – Modifier Key Composition
BR4.1 – Dead Key State
BR4.2 – Dead Key Combinations
BR4.3 – Keys Invisible to Dead Keys
BR4.4 – Unicode Representation of a Dead Key
BR4.5 – Invalid Dead Key Substitution
BR4.6 – Dead Keys and Modifier States

5 Use Case Type

This use case is included by UC3.1. Therefore, this use case is concrete, initiated under the same circumstances as UC3.1.

6 Actors

6.1 Operating System

In this use case, the Windows operating system is considered the only actor. Even though the user is responsible for causing keystroke messages, they do not interact directly with the Multikeys Core Subsystem, which is a background process.

This interaction occurs though the Windows API. The operating system sends Raw Input messages to all registered receivers, and keyboard input messages are received by any code in the hook chain. The keyboard hook is also responsible for passing the messages onwards. Furthermore, additional actions like sending a Unicode character also happen through the Windows API.

Because the user is never notified of these transactions, they are excluded as an actor in this use case.

7 Pre-Conditions:

7.1 Background Process

The Multikeys background process must be running.

7.2 Function Call

The Multikeys function call queries the Remapper for the command assigned to a certain intercepted key. The system has already extracted keystroke information, including the scancode and the name of the keyboard that sent the signal.

8 Event Flow

8.1 Main Flow

These steps are included between steps 3 and 4 of UC3.1's main flow.

1. Information concerning the keypress arrive at the Remapper, Multikeys' internal model.

2. The system looks for the keyboard (in the model) with the same name as the one contained in the message and queries the keyboard instance for evaluation.

3. The keyboard instance evaluates the keypress message and calls the currently active layer with the data.

4. The command corresponding to the received key is returned by the layer instance.

5. The Remapper module returns the command object.

8.2 Alternative Flows:

8.2.1 Keyboard Configured to Respond to All Messages

In a custom Multikeys layout, it is possible to configure a keyboard whose remaps apply to all keyboard devices. This feature's intended usage is remapping all keyboards that aren't explicitly configured.

2a. System finds a keyboard object configured to respond to any keyboard device.

2a.1. Remapper considers the keyboard as appropriate to evaluate the message.

2a.2. The flow skips to step 3.

8.2.2 Keyboard not Used in the Current Layout

The user may leave one or more keyboard unremapped in a custom layout in order to use it normally. In this case, the system will return the message without executing any action.

2b. System does not find the keyboard in the active layout.

2b.1. Remapper returns an answer saying that there isn't a command remapped to this key.

8.2.3 Lack of Matching Layer

Remaps from physical keys to actions are organized in layers, each corresponding to a modifier key combination. If the matching keyboard does not have a layer that matches the currently pressed modifier key combination, the message is returned to the Operating System to be normally processed, without executing any remapped action.

3a. The keyboard instance does not find a suitable layer in its collection of layers.

3a.1. Remapper returns an answer saying that there isn't a command remapped to this key.

8.2.4 Modifier Key

If the received key is registered as a modifier key, the model must be updated accordingly.

3b. The received keystroke message corresponds to a key that's registered as a modifier key.

3b.1. The keyboard instance updates its internal data to reflect the currently active modifiers in the user's keyboard.

3b.2. The keyboard instance searches its collection of layers to see which corresponds to the currently active modifier combination.

3b.3. The layer is then set as the currently active layer for that keyboard. If the current modifier combination does not correspond to any layer, a null reference is used to represent the lack of suitable layers.

8.2.5 Active Dead Key

If the keyboard is in a dead key state (because a dead key was previously pressed), any command found in the model goes through the stored dead key. Even if a command is not found, the dead key is called upon the next keypress. The dead key's command is returned instead of whatever command was found, and it becomes responsible for deciding what action to perform.

4a. The keyboard is in a dead key state (each keyboard keeps a reference to the currently active dead key, which is null when there's no active dead key).

4a.1. The dead key receives the keypress, together with a command if one was found.

4a.2. The dead key's command evaluates the command that was just found and checks it against its own substitution map to determine its execution algorithm.

4a.3. Remapper returns the modified dead key command.

8.2.6 Received Key is not Mapped in the Current Layer

If the current layer is not empty but has no remap for the keypress, the system lets the message through without executing any action.

4b. System does not find a suitable command in the active layer.

4b.1. Remapper returns an answer saying that there isn't a command remapped to this key.

8.2.7 Dead Key Command

The user may define keys as dead keys, which have a Unicode representation and a collection of substitutions to match with the next keypress.

Note: This alternative flow is only checked after the 4a alternative flow. Therefore, if two dead keys are pressed in sequence, the 4a alternative flow is executed and this one isn't.

4c. System has found a dead key command.

4c.1. Instead of returning the command, the keyboard's internal data is updated to store the dead key.

4c.2. Remapper returns an empty command (which performs no action when executed).

8.3 Exception Flows

This use case does not include any exception flows because all possible situations are already covered by alternative flows, with the purpose of never interrupting the user experience. Upon any unexpected errors, the Remapper must return an answer saying that no remap was found, without interrupting the flow.

9 Post-conditions:

9.1 Updated State

If the received keystroke message requires any type of update to the internal model, such as a dead key, the model must respond to the next keystroke with the model already updated.

10 References

For details about Multikeys' requirements, see the requirements document and the business rules document for this project.

Details concerning other use cases can be found in the use case document for this project, and in UC3.1's use case specification document. Further information on implementations are available in the architecture document.