Department of Informatics, University of Zürich

Software Project Report

Development of a student-friendly code editor

Luka Lapanashvili
13-934-062
luka.lapanashvili@uzh.ch

June 10, 2017
supervised by
Prof. Dr. Prof. Dr. Michael Böhlen
and
Katerina Papaioannou

University of Zurich^UZH
Department of Informatics

Table of Contents

1 Introduction

Monolith Code is a light weight, fast and intuitive code editor that allows the user to tinker around with small algorithms efficiently. It provides a set of powerful feature while maintaining a low-profile user interface that is not intimidating for beginners. In its simplest form, it looks like a stock text editor provided by the operating system manufacturers (Lapanashvili, GithHub, 2017).

There are many integrated development environments like Eclipse or Intellij that contain a rich feature set and integrated tools for productivity. Unfortunately, none of those IDEs are tailored to tinkering projects, which are especially important for beginners. The main problem with the big IDEs is, that while providing a variety of tools, they take a long time to boot up. Furthermore, for every code snippet that a user wants to test, a new project, packet and class needs to be created. Even when creating a dedicated project for testing standalone scripts, most IDEs will not run the code if the project contains other non-compiling classes.

On the other hand, the alternative option would be to use a combination of a plain text editor with a terminal. Even though this is the preferred working environment for most professional programmers, a lot of beginners are missing some key features with this workflow. Syntax highlighting is very important for programmers regardless of the skill level. Also, working with a terminal can be intimidating for certain novice programmers.

Monolith Code aims to bridge the gap between the bare minimum text editor terminal setup and a fully fleshed out IDE. The idea is to provide the most useful and important tools of an IDE while maintaining the simplicity of a plain text editor, which makes it great for educational purposes. The main features of Monolith Code towards this direction are: syntax highlighting, support for multiple languages, code completion, code compilation, inline math calculation, search and replace and customizable themes.

In this project, I enhance Monolith Code with: (a) a backup system, (b) automated updates, (c) custom execution commands and (d) native console integration. The rest of this report is structured as follows: In Section 2 I describe the structure and implementation behind Monolith Code. In section 3, I will describe in detail the changes required to the base application to achieve a successful implementation of the new features. Section 4 contains a brief description of the carried-out testing to ensure the stability and reliability of Monolith Code. The end of the report is marked by the conclusion in section 5.

2 Monolith Code Implementation

The class hierarchy of Monolith Code is very flat because there is no real need for inheritance. Instead, it is based on a component based architecture. There is one main class `MonolithFrame.java` that initializes different subcomponents like the `LanguageFactory.java` and `StatusBar.java`. This modular structure allows for new features to be developed independently from the core as well as selective activation of features. Additionally, the architecture allows for modding support to be implemented further down the road.

`MonolithFrame.java` is the core component of the entire code editor. The class extends from `JFramePlus.java` which is a slightly modified subclass of java's GUI library "Java Swing". It defines a window containing all graphical as well as logical components.

Graphical components are as follows:

- Built in menu bar `JMenu.java` on top of the window with a range of menu items (`JMenuItem.java`)
- Different kinds of panels (`JPanel.java`) which are used to display information about the current open file as well as modified panels that are used to display status messages (`StatusBar.java`)
- Hidden inside a divider (`JSplitPaneWithZeroSizeDivider.java`) is the integrated console (`Console.java`) which is used to inform the user about important notifications as well as errors. The console also accepts a limited range of commands that activate certain features.
- The main component of `MonolithFrame.java` is the text field (`RSyntaxTextArea.java`). It is derived from the built-in swing component `JTextArea.java`, but extends it with powerful features like syntax highlighting, code completion and other functionalities that are common in modern IDEs. `RSyntaxTextArea.java` is distributed in the RSyntaxTextArea library developed by "bobbylight" since 2005 (bobbylight, 2008). After trying out different libraries like jsyntaxpane (Unknown, 2008), SyntaxPane by Sciss (Sciss, 2011) and a self-attempt to create a parser, I came to the conclusion that RSyntaxTextArea was the best library to use in this project for the following reasons: It is open source, it is still actively developed and the developer is active on GitHub fixing issues, reacting to reports and accepting merge requests.

Logical Components:

- The `LanguageFactory.java` is a component responsible for everything related to the different programing languages. On initial start, it generates a list of strings and definitions that correspond to the operating system and language, like the extension filter and the compile and run commands for the languages.
- Monolith Code stores all user settings on the hard drive, which allows persistent configuration of the editor throughout sessions. This is managed by the `Settings.java` class.
- `CodeBuilder.java` is the class responsible for assembling a command and executing it via the integrated Runtime executer. A command alongside with the language reference is passed from `MonolithFrame.java` to the `CodeBuilder.java`. It is determined, if the

passed language supports execution. Following the proper command according to the operating system is chosen and executed.

- Alongside with the `CodeBuilder.java` there needs to be a second class that handles the In and Out streams of the created process. This task is realized by `BuildConsole.java`. It is derived from the previously mentioned `Console.java` and extends it with an additional toolbar for quickly accessing the most important process relevant functionality.
- `Expression.java` is an external math library that handles evaluations of mathematical equations. It is directly integrated into `MonolithFrame.java`. The currently selected line in the code editor can be directly sent to be evaluated. The result is directly appended to the end of the selection. Therefore, the user can do mathematical calculations directly inline. `Expression.java` is part of the GitHub repository EvalEx created by uklimaschewski (uklimaschewski, 2012).

3 New Features

During the software project, I implemented crucial functionality according to the scheduled task list as well as some additional features that were necessary for a public release. All these features are essential for Monolith Code to be a viable tool for computer science students. Among other things, the task list includes a backup system, an application updater and most importantly the possibility to provide custom compile and run commands alongside, with the option to bypass the provided terminal and instead utilize the native terminal.

3.1 Backup System

From the beginning of the development, there were multiple instances of unexpected crashes. Even though this is to be expected for a software in development, it is nevertheless very frustrating to experience. A clean code structure and many fail safes can reduce the occurrence of crashes, but unfortunately there is no way to guarantee a bug free application. The backup system is designed to further mitigate the severity of loss of important documents by automatically managing backups without any specific input from the user. Upon application start, a separate thread is spawned and the backup system is allocated to that thread. This approach grants multiple advantages, such as a seamless experience while the backups are written to the hard drive and it also makes sure that in case of an unexpected problem on another application thread, the backup will still be executed due to the nature of multithreading.

The `BackgroundSave.java` class defines an integer `SAVE_INTERVAL` which defines in what interval the backups should be generated. To avoid unnecessary write actions, the system checks the last saved file against the current file to be saved. If there are no changes, the backup is not triggered for this interval. All backup files are written to a local directory of the java application called “autosave”. The name of the backup file is made up by the exact timestamp of the occurred save and the name, that was defined by the user. To avoid a large number of old backups, the system checks on every application start the “autosave” folder and determines which files are older than the maximal allowed days (`DEF_BACKUP_MAX_SAVE_DAYS)` and deletes them if necessary. This setting can be accessed by the user in the settings file and is set to 30 days by default. To avoid a total erase of backups after an extended period of inactivity, for example a holiday season, there is an override that protects the latest few files from being erased. This value is controlled by `MIN_KEEP_NUMBER`.

3.2 Update Installer & Downloader

A key aspect of an application that is in rapid development, while being publicly available, is the possibility to automatically update the application. Often it is very crucial to deploy critical hotfixes to maintain an overall satisfied userbase. The update system prompts the user on application start for the installation of a recent version. After the confirmation, the update is automatically downloaded and installed. Fortunately, due to the small size of the application, this process does not take longer than a few seconds.

The updater system works by comparing the `BUILD` number against a remote file on a webserver. A `VERSION` number is displayed alongside the `BUILD` number, but serves purely for decorative purposes. The `BUILD` number is a unique id and is incremented on every build. This approach allows to circumvent complicated parsing of the version numbers and implementation of custom comparison functions and works with simple integer comparison. If the update prompt is confirmed by the user, the `Updater.jar` starts up. A big advantage of Java is that it does not block the .jar files even when executing them. This allows the Updater process to delete the old jar files including the source files and replace them with the updated ones, while the application is still running.

The update process itself is fairly simple. A zipped container with the updated jar files and resources is downloaded and unzipped, which causes the old files to be overwritten. Finally, the zip file gets deleted. As mentioned previously, this process takes no longer than a few seconds. To engage the update, the user simply needs to restart the application.

The Updater can also act as a standalone installer. Since it is a separate application, it hast no information about the version of the main application. When looking for updates, the main application passes its version number to the Updater via an Inter Process Communication (IPC) call. If no arguments are passed to the updater, it acts as a first-time installer.

3.3 Custom Commands

The alpha version of Monolith Code was missing an essential component to be an effective tool not only for students, but for every user in general. This component being an interface to set user defined commands for building and executing code. Even though the feature itself was always present in the application and was set to a meaningful default setting, it was inaccessible for the user. The ability to customize the command, has many advantages. For once, it gives the user a way to adjust the default command, in cases where the default command would not work on that specific instance of hardware or the specific configuration of the operating system. Furthermore, it allows more advanced users to set up automated procedures. For example, a user can specify a build command for the HTML language. Even though HTML is internally marked as a non-compliant language, the user can still activate the compilation. While the compile code does not necessarily have to be a real compile code, but instead can be any sort of process script. In the example with the HTML project, the compile code could be misused to upload the html document to an ftp server. Finally, the most obvious application case of the custom command, is to pass arguments to the compiled executable.

As mentioned above, the basic architecture for compilation of code was already present in the application. The procedure was as follows: The user requests to build and run the current document via a shortcut or by the menu bar. `MonolithFrame.java` first checks if the file already has a path and name assigned to it. If not, a save file prompt is displayed. At this point, the prompt also suggests a name according to a customizable algorithm defined for each language in the `LanguageFactory.java`. This is especially useful for languages, where the class name must correspond to the filename, like in Java. After the save procedure is successfully completed, an instance of `CodeBuilder.java` and `BuildConsole.java` is created where the current text and language is passed. The Code Builder would then determine according to the constants established in the `LanguageFactory.java`, what command needs to be executed, to compile the current code.

This procedure has now changed to allow for user defined commands. The main difference is, that now the code builder doesn't directly access the constants from the `LanguageFactory.java`, but instead gets the appropriate command from the `Settings.java` class. This change allows for users to define their own default commands. Further down the pipeline, the command will be overridden, if a custom command is declared by the newly introduced `CustomCommandEntity.java`. This class is a container for custom commands and is initialized by the `CustomCommandSerializer.java`. The idea being, that via a dialog screen, the user can define custom commands, that are stored on disc in a xml format. The `CustomCommandSerializer.java` is responsible for writing and reading the xml file and creates a set of custom command objects. All these changes give the user the chance to define default commands as well as togglable custom commands which persist throughout the sessions.

3.4 Native Console Options

For convenience and consistency reasons, I have decided to create a simple class similar to a terminal, that would serve as a hub for Input/Output of all streams that the Code Builder process generates. It contains basic functionality and some convenient features such as the ability to change the output color according to the type of stream that is displays. This makes system output, output of the compiled application and error outputs easier to distinguish. Unfortunately, as I discovered later, there is a big hurdle when implementing terminal like applications. Every programming language and every compiler implement their output streams differently. For example, a simple Java system print:

```
System.out.println(“Hello World”);
```

creates a normal output stream, that can be read without issues. On the contrary, a basic C print code:

```
printf("Hello World");
```

does not result in any signal flow in the stream, since the print command in C is buffered. Meaning, it must be explicitly flushed (`fflush(stdout);`) and post fixed with a “\n”, to generate any output in the stream. This inconsistency in behavior makes it nearly impossible to create one unified terminal, that results in correct behavior. I was able to work out an approach that works for 90% of all use cases, but unfortunately couldn’t entirely solve the issue.

For this reason, the decision was made to give the user the option to execute the commands in a native terminal of the operating system. The native terminal was realized by inserting an escape branch from the previously mentioned code building pipeline. Additionally, the custom command prompt interface got an additional checkbox to enable the native terminal option. Furthermore, a `NativeConsole.java` class was created. Its purpose it to assemble a command, which when executed in a native terminal, prints all the necessary information. The goal was to create a similar experience with the native terminal to the integrated terminal of Monolith Code. This involved the detection of the operating system and adjustment of the command according to it. For windows and Linux based operating systems, this was an easy task, since windows and Linux allow to start a terminal with command argument. For the Mac operating system, this task was a bit more complex, since MacOS does not allow to start the terminal with arguments. I was able to formulate a workaround, that involved Osascript and an abuse of the password mode in the Mac terminal.

4 Testing

To ensure a flawless experience for the upcoming courses, the application was tested for multiple course exercises, including:

- Multiple exercises of Informatics I – Exercises in Java
- All tasks of Informatics II: Algorithms and Data Structures – Exercises is C
- A handful of exercises of Informatik und Wirtschaft – Exercises in Python
- A small sample of multi class projects from my personal code collection – Code in Java

The execution of the tasks worked in all cases. However, a small amount of C exercises resulted in unexpected behavior because of the already mentioned limitation of the integrated terminal. The issues were resolved by explicitly using the native terminal. Alternatively, inserting a `fflush(stdout);` after every print, resolved the issue as well.

5 Conclusion

In this report, I presented you the "Monolith Code” and I described how I extended it with features such as the backup system, the automated update system, custom commands and native console integration. The editor is now a very stable, simple to use and approachable base platform for students and programmer in general. Future work will include a smarter detection of the languages alongside with automated imports for libraries.

6 Bibliography

bobbylight. (2008, August 16). *GitHub*. Retrieved from GitHub: https://github.com/bobbylight/RSyntaxTextArea

Lapanashvili, L. (2017, April 6). *GithHub*. Retrieved from GitHub: https://github.com/Haeri/Monolith-Code

Sciss. (2011, December 24). *GitHub*. Retrieved from GitHub: https://github.com/Sciss/SyntaxPane

uklimaschewski. (2012, December 15). *GitHub*. Retrieved from GitHub: https://github.com/uklimaschewski/EvalEx

Unknown. (2008, June 18). *Google Code*. Retrieved from Google Code: https://code.google.com/archive/p/jsyntaxpane/