System Design Document for Proton Text

Version: 1.2

Date: August 13, 2017

This version overrides all previous versions.

Ludvig Ekman
eludvig@student.chalmers.se
Institutionen för Informationsteknik

Anton Levholm
levholm@student.chalmers.se
Institutionen för Informationsteknik

Mickaela Södergren
micsod@student.chalmers.se
Institutionen för Informationsteknik

Stina Werme
stinawe@student.chalmers.se
Institutionen för Informationsteknik

1 Introduction

This document describes the system of Proton Text, a text editor that uses markdown to fast and easily take notes. The main use case is to take lecture notes and format the text with a simple syntax, and the application has features in mind for such notes.

1.1 Design goals

This document describes the text application specified in the requirements and analysis document. Some design goals:

- The design must be loosely coupled, preferably in a MVC design, to make it possible to switch GUI.
- The design must be testable, so it should be possible to isolate parts.
- For usability see the requirements and analysis document.

2 System architecture

The application is written in java and uses a MVC structure.
The application is run on a single computer and is decomposed into the following top level packages (see arrows for dependencies):

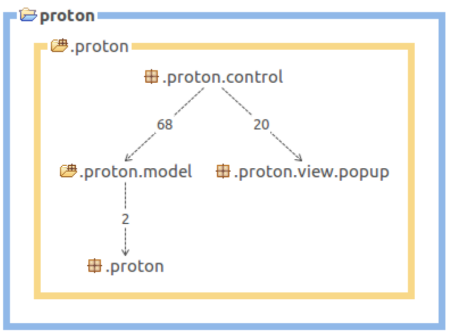

Figure 1: Package dependencies of Proton Text

- **Control** is the use case controller, where the user's actions is interpreted and translated.
- **View** is the top level package for all GUI related classes including the main window.
- **Model** is where all the functionality and data is kept.

The MVC structure avoids circular dependencies by only making the controller dependent on the model and view but not the other way around.

2.1 General observations

2.1.1 Parsing

During runtime, the application uses Java's Pattern and Matcher to find markdown syntax in the text written by the user. This is done using regular expressions to find a specific pattern of different styles of markdown syntax. Whenever a match is found, the match is then handled and formatted.

2.1.2 Tabs and view

To easily both write and get the output at the same time, the application mainly has two views on each tab, where each tab has their own controller. One that the user writes in, and one that previews the syntax translation. To easily change between different documents the user can choose to show a file tree, and there is always a horizontal list of the active tabs at the top of the window. When clicking on a file in the file tree the chosen file is opened in a new tab. When creating a new file from the tree it is not opened in a new tab. The user can manually change the amount of space each view takes.

2.1.3 Save files

The standard Java file saving system is used in order to save files. The file in use, the one in the active tab, is the one being saved. Document implements Observable to match with the tab that is in use. The save method is called on by the current document with the file path as input parameter.

2.2 Compatibility with computers

The application runs on Linux (Ubuntu 16.10), Windows (Windows 10) and MacOS (Sierra 10.12.5), (tested for the versions in parenthesis). The application is used by one computer and person at a time.

2.3 Design patterns

The application uses design patterns such as the factory pattern, the strategy pattern and a MVC model, with Java's Observer and Observable. The application uses different kinds of libraries to handle more complex use cases.

2.4 External dependencies

2.4.1 JFoenix JavaFX

JFoenix JavaFX is a library with graphical components. Its components are visually more impressive than those of Java FX's, which is why it was used in the application (JFoenix,).

2.4.2 iText PDF

iText PDF handles the use case of saving a file as a .pdf file. It is a complex use case and thus it was better to have an external library handle it (iText PDF,).

2.4.3 jsoup HTML parser

jsoup is a Java library for working with real-world HTML. In the application it's used for removing the HTML from the text before the text is sent to the Markdown class for parsing (jsoup,).

2.4.4 JUnit Test Framework

JUnit Test Framework is a framework used for writing repeatable tests (JUnit,).

2.5 UML

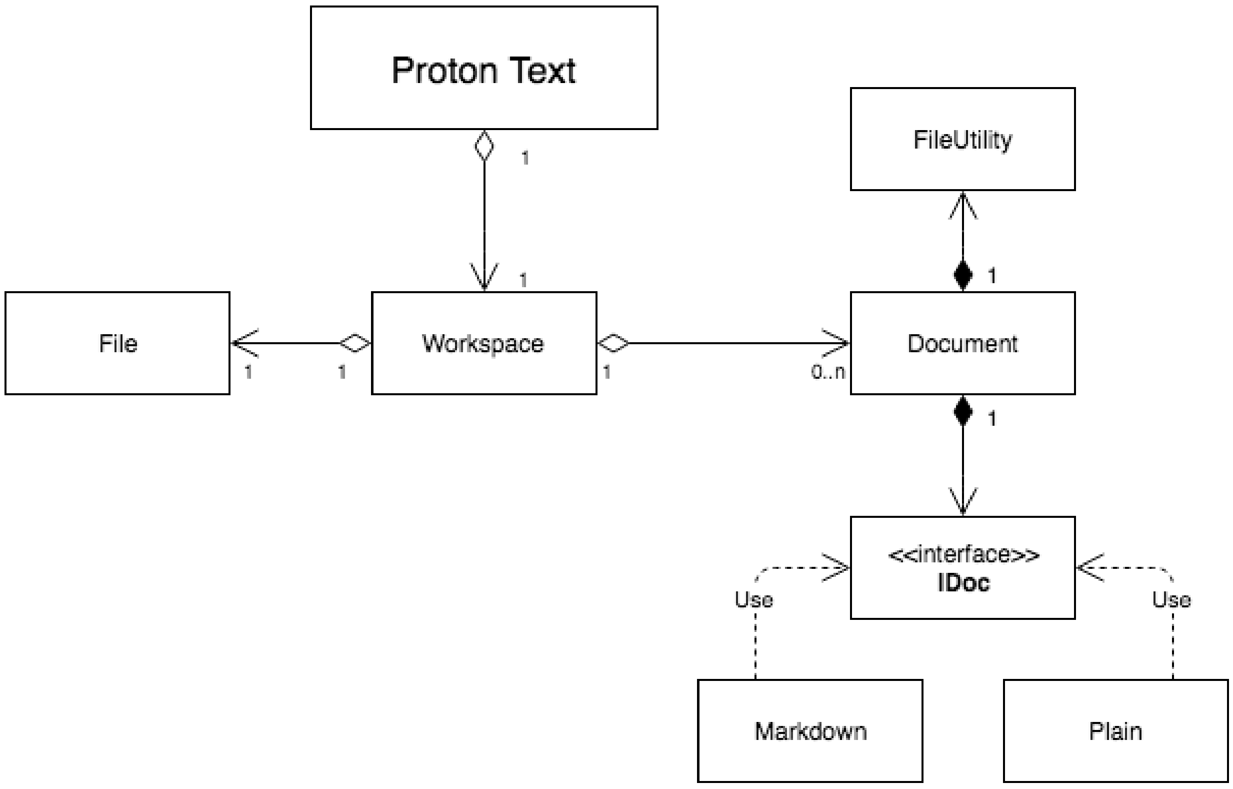

Figure 2: UML of the application

ProtonText is responsible for starting up the application. Workspace is responsible for creating new documents (via a factory for documents) and keeping track of them. Workspace also makes use of File (which is part of Java's own library) to keep track of which folder you are currently located in. FileUtility is used to extend the functionality of Java's File.

Every tab's property is handled by Document. Document also handles the text itself. To know how the text should be formatted, Document has an IDoc variable. This variable is set to any class that implements IDoc whenever you instantiate Document. When you call on a method, the variable delegates the call to the set class. In this case, either Markdown or Plain. Markdown is responsible for parsing the text and formatting it appropriately, while Plain does not format the text in any way.

3 Subsystem decomposition

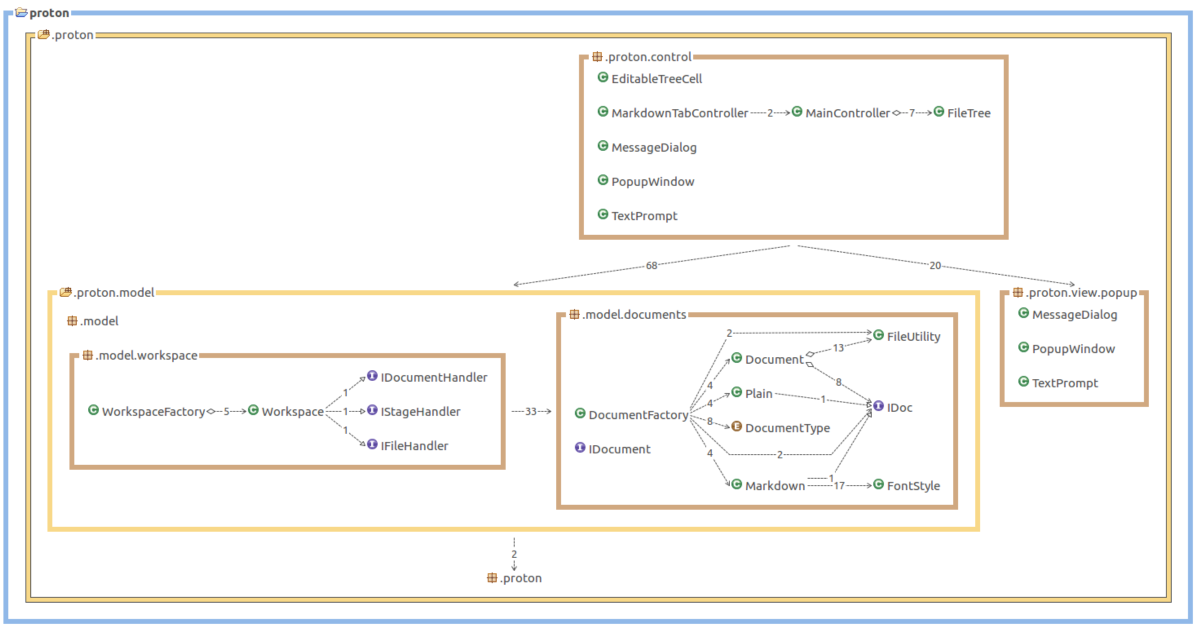

Figure 3: Inner dependencies of Proton Text

3.1 Control

The 'Control' package is using classes in the Model package and sends information to the View package, e.g. the text input in the main document and translating it to the view. How different button functions and looks is directly implemented in the controller.

3.2 View

The 'View' package contains all files related to the GUI, such as FXML and Java files that describes a GUI element. The purpose of the FXML files is to give the user an interface to interact with; it only knows which methods should be called from the corresponding controller class.

3.2.1 Popup

The 'Popup' package contains Java files that creates pop-up windows at specified instances during the user's interaction with the application; for example when the application requires input from the user.

3.3 Model

The 'Model' package in the application acts as the model from the MVC design pattern. It is where all the functionality and data is kept. In other words, the model directly manages the data, logic and rules of the application.

The model is asked to perform different actions by the controller. This is done thanks to different kinds of getter-methods in the model package, which allows the controller to communicate what it wants to be done. The model has no direct communication with the view, it simply does things and leaves the communication up to the controller.

3.3.1 Documents

The 'Documents' package contains the tabs and all the tabs' properties.

3.3.2 Markdown

The 'Markdown' package contains the classes and interfaces related to the document type Markdown.

3.3.3 Util

The 'Util' package contains tools to handle files and text formatting.

3.3.4 Workspace

The 'Workspace' package contains the classes and interfaces related to the workspace in the application.

4 Persistent data management

The data stored is text documents (of any sort). The documents are saved when a user activates any save command, either through "Save", "Save as" or "Rename". The application stores the data via a normal file handler.

5 Access control and security

NA. Application starts and exits as a normal desktop application (script).

References

iText PDF. (2017). Retrieved 2017-07-30, from http://itextpdf.com

JFoenix. (2017). *Javafx material design library.* Retrieved 2017-07-30, from http://www.jfoenix.com

jsoup. (2017). Retrieved 2017-07-30, from https://jsoup.org

JUnit. (2017). Retrieved 2017-07-30, from https://junit.org