Міністерство освіти і науки України
Тернопільський національний технічний університет імені Івана Пулюя

Факультет комп'ютерно-інформаційних систем і програмної інженерії
(повна назва факультету)
Кафедра програмної інженерії
(повна назва кафедри)

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття освітнього ступеня

магістр
(назва освітнього ступеня)

на тему: Аналіз та проектування C# бібліотеки для генерації текстів на основі мови розмітки даних (Markdown)

Виконав(ла): студент(ка) 6 курсу, групи СПм
спеціальності 121 інженерія
програмного забезпечення
(шифр і назва спеціальності)

	(підпис)	Борейко В. В. (прізвище та ініціали)
Керівник	(підпис)	Петрик М. Р. (прізвище та ініціали)
Нормоконтроль	(підпис)	Стоянов Ю. М. (прізвище та ініціали)
Завідувач кафедри	(підпис)	Петрик М. Р. (прізвище та ініціали)
Рецензент	(підпис)	(прізвище та ініціали)

Тернопіль
2021

Затвердження

Завдання

Календарний план

РЕФЕРАТ

Кваліфікаційна робота «Аналіз та проектування C# бібліотеки для генерації текстів на основі мови розмітки даних (Markdown)» написана Владиславом Борейко, студентом Тернопільського Національного Технічного Університету імені Івана Пулюя, факультет комп'ютерно-інформаційних систем і програмної інженерії, група СПм-61.

Мета роботи – це реалізація бібліотеки легкої мови розмітки, що відповідає загальновживаним стандартам Markdown Також створена бібліотека повинна відповідати оригінальним стандартам мови Markdown, що випливають із оригінальної бібліотеки Markdown.pl. Це дасть можливість використовувати бібліотеку із новими стандартами мови розмітки, які виникли після публікації офіційної бібліотеки Markdown.pl та розвивались незалежно один від одного.

Методом розв’язання задачі є попередній аналіз предметної області, постановка задач розробки, огляд засобів розробки, опис предметної ралізації біліотеки та її тестування для перевірки працездатності.

Отриманими результатами є робота, яку можна використовувати як зразок проведення аналітичного методу дослідження легких мов розмітки для форматування тексту та бібліотека на C# для Markdown. Також отримані результати мають практичне значення для варіантів використання мов розміток, принципу їх дії та сфер застосувань.

Ключові слова: MARKDOWN, МОВА РОЗМІТКИ, БІБЛІОТЕКА, СФЕРА ЗАСТОСУВАННЯ, ЗАВДАННЯ, АЛГОРИТМ, АРХІТЕКТУРА, ТЕСТУВАННЯ, РОЗРОБКА, ФОРМАТУВАННЯ, ПАРСЕР, ТЕКСТ.

ABSTRACT

Qualification work "Analysis and design of C # library for generating texts based on data markup language (Markdown)" was written by Vladislav Boreyko, a student of Ternopil National Technical University named after Ivan Pulyuy, Faculty of Computer Information Systems and Software Engineering, group SPm-61.

The purpose of the work is to implement a light markup language library that meets the commonly used Markdown standards. The created library must also meet the original Markdown language standards, which follow from the original Markdown.pl library. This will make it possible to use the library with the new markup language standards that emerged after the publication of the official Markdown.pl library and developed independently of each other.

The method of solving the problem is a preliminary analysis of the subject area, setting development tasks, review of development tools, description of the subject implementation of the library and its testing to verify performance.

The results are a work that can be used as an example of an analytical method for studying easy markup languages for text formatting and a library in C # for Markdown. The obtained results are also of practical significance for the options for using markup languages, the principle of their operation and areas of application.

Keywords: MARKDOWN, MARKING LANGUAGE, LIBRARY, FIELD OF APPLICATION, TASKS, ALGORITHM, ARCHITECTURE, TESTING, DEVELOPMENT, FORMATTING.

ЗМІСТ

ВСТУП

Markdown — це проста у використанні мова розмітки створена Джоном Грубером, яка використовується з простим текстом для додавання елементів форматування до звичайного тексту без використання формального текстового редактора або використання тегів HTML. Зараз вона є актуальною при написанні документації до програм та статей на інтернет ресурсах.

Офіційний опис синтаксису Markdown не визначає дану мову однозначно. Перші розробники звертались до реалізації бібліотеки на мові Markdown.pl написаної на мові Python. Але дана бібліотека була погано написана, мала велику кількість неоднозначних перетворень вихідного тексту. Такі проблеми свідчать про погане проектування програмного застосунку та відсутність якісних тестів. Це призвело до появи великої кількості варіантів реалізацій бібліотек для Markdown на різних мовах, різними людьми, що не узгоджували між собою єдиний стандарт розмітки мови. Це все лише ускладнювало ситуацію. Неоднозначність стандартів мови та її синтаксису зросла ще більше.

На даний момент офіційна бібліотека оновлювалась 15 років тому назад, а один зразок тексту з розміткою Markdown може по різному відображатись на різних онлайн чи офлайн ресурсах. Це стало наслідком нецентралізованого розвитку даної мови.

C# — мова програмування, що активно розвивається та використовується для створення програм для Windows та веб додатків. Цю мову було вибрано враховуючи її актуальність та перспективність. Існуючих рішень на мові C# дуже мало і вони не розкривають повного потенціалу цієї мови.

Темою даної роботи стало створення C# бібліотеки для генерації форматованих текстів використовуючи Markdown. Використання мови C# гарантує швидкодію та надійність коду при його використанні в інших проектах.

Мета роботи – це реалізація бібліотеки легкої мови розмітки, що відповідає загальновживаним стандартам Markdown Також створена бібліотека повинна відповідати оригінальним стандартам мови Markdown, що випливають із

оригінальної бібліотеки Markdown.pl. Це дасть можливість використовувати бібліотеку із новими стандартами мови розмітки, які виникли після публікації офіційної бібліотеки Markdown.pl та розвивались незалежно один від одного.

Для наукового обґрунтування результатів досліджень створення бібліотеки використано метод порівняльного аналізу. А саме підхід за яким було розшукано і виявлено схожі та розбіжні властивості однотипного характеру і порівняно їх, для отримання кращого. В даному випадку пошук переваг мови розмітки текстів Markdown у порівнянні з іншими конкурентними мовами, аналіз застосування даної мови в різних сферах, аналіз мови програмування C# та пошук необхідних бібліотек та функцій для швидкої обробки текстів.

Об’єктом дослідження даної роботи є легка мова розмітки Markdown для швидкого форматування текстів її синтаксис, властивості та особливості.

Предметом дослідження виступає реалізація синтаксису мови розмітки Markdown за допомогою мови програмування C#.

Така розробка забезпечена актуальністю. Оскільки дана мова програмування широко використовується при написанні програмного забезпечення серверів та на великій кількості домашніх комп’ютерів, ця бібліотека знайде своє використання. Наприклад, її можна використати для генерації текстів документації програм або оформлення текстових блоків на веб сторінках. Також її зручно застосовувати при веденні електронних нотаток .

Для того щоб досягнути мети, потрібно виконати ряд завдань, а саме: провести аналіз предметної області, визначити особливості та сфери застосування мови розмітки, дослідити синтаксис Markdown та можливості мови програмування C# в роботі з текстом, вибрати засоби для розробки бібліотеки, програмно реалізувати саму бібліотеку та провести її тестування.

Грамотне написання коду, з коментарями в блоках, що містять складну логіку дасть змогу доповнювати код в майбутньому. Це означає практичну підтримку бібліотеки в майбутньому навіть сторонніми розробниками.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Особливості мови розмітки даних

1.1.1 Семантика та форматування

Markdown — це легка мова розмітки, яку можна використовувати для додавання елементів форматування до текстових документів із відкритим текстом. Створена Джоном Грубером у 2004 році, Markdown зараз входить в рейтинг найпопулярніших мов для розмітки тексту в світі [1].

Основною метою Markdown було створення синтаксису для простого тексту, який можна було б легко конвертувати в HTML. Його початкова реалізація тепер вважається занедбаним програмним забезпеченням. Таким чином, канонічної реалізації Markdown немає. Зараз існує кілька різновидів Markdown, які часто мають незначні відмінності між їх реалізацією та синтаксисом.

Markdown не може замінити HTML. Його синтаксис дуже малий, відповідає лише дуже невеликій невіликій кількості тегів у HTML. HTML-теги досить легко вставляти, використовуючи сучасні редактори. Суть Markdown представлена в тім, щоб спростити читання, редагування та написання тексту. HTML – це формат веб сторінок та публікацій; Markdown — відповідно є форматом написання. В результаті, здатність Markdown форматувати текст вирішує лише проблеми, які можна передати у вигляді простого тексту.

Для будь-якої розмітки, яку синтаксис Markdown не зможе покрити своїм функціоналом, можна просто використовувати сам HTML. Повідомляти або розділяти такий документ, щоб вказати, що відбувається перехід з Markdown на HTML не потрібно. Можна просто використовувати теги HTML в Markdown.

При складанні документа Markdown важливо знати, які візуалізації доступні для використання, і переконатися, що використовується сумісний синтаксис. Як правило, дотримання Github-подібний Markdown зменшить ймовірність проблем.

Використання Markdown відрізняється від використання редактора WYSIWYG. У такій програмі, як Word, потрібно натискати кнопки, щоб від форматувати слова та фрази, і зміни видно відразу. Markdown працює за іншим принципом. Коли створюється файл у форматі Markdown, користувач додає синтаксис Markdown до тексту, щоб вказати, які слова та фрази мають виглядати інакше [2].

Наприклад, щоб позначити заголовок, потрібно додати перед ним знак числа (наприклад, # Heading One). Або, щоб зробити фразу напівжирним, необхідні дві зірочки перед і після неї (наприклад, **приклад тексту виділеного жирним**). Потрібно трохи часу аби звикнути бачити синтаксис Markdown у тексті, особливо після програм типу WYSIWYG. На рисунку нижче показано файл Markdown, який відображається в текстовому редакторі Atom.

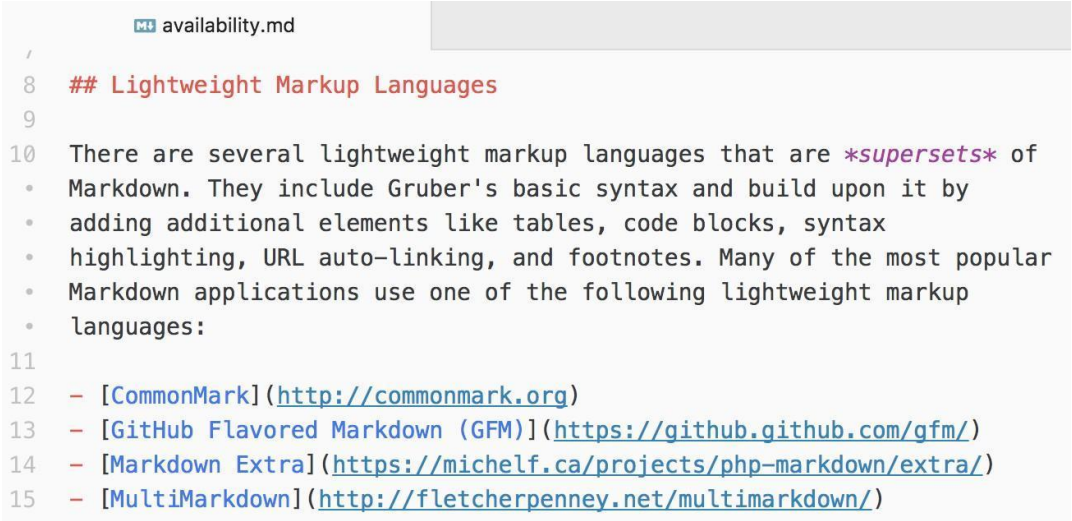


Рисунок 1.1 – Текст із розміткою Markdown в редакторі

Можна додати елементи форматування Markdown до текстового файлу за допомогою програми текстового редактора, або використовуючи одну з багатьох програм Markdown для операційних систем macOS, Windows, Linux, iOS та Android. Існує також кілька веб-додатків, спеціально розроблених для написання в Markdown. Більшість із них є безплатними.

Залежно від програми, яка використовується, попередньо неможливо переглянути відформатований документ. Проте це нормально. За словами Грубера, синтаксис Markdown розроблений таким чином, щоб його можна було

читати та ненав’язливо, тому текст у файлах Markdown можна прочитати, навіть якщо він не відтворений.

Головна мета дизайну синтаксису форматування Markdown — створити максиамльно читабельну мову. Відформатований у Markdown документ можна публікувати як є, як звичайний текст, не виглядаючи так, ніби він позначений тегами чи інструкціями щодо форматування [3].

Більшість текстових процесорів відокремлюють семантику від форматування. Якщо заздалегідь подбати про використання розділу форматування при роботі з документом Word, то семантична інформація, необхідна для зміни стилів, тобто візуальне представлення всіх семантичних одиниць (наприклад, заголовків), буде легко доступною. Відокремлення семантики документа від його форматування не є виключною властивістю мов розмітки. Однак, коли розділення тексту та семантики не виконується, існує ймовірність помилки. Наприклад, якщо змінити розмір шрифту заголовків розділів другого рівня, можна легко зробити це, але ще можна легко виділити один заголовок розділу та форматувати його, змінивши лише цей один заголовок. Що робить цей конкретний заголовок іншим і якщо пізніше змінити форматування заголовків другого рівня, цей один заголовок не зміниться.

Добре, якщо це навмисне; погано, якщо це не те, чого хотів користувач. За допомогою редакторів WYSIWYG можна відокремити семантику від форматування, але це розділення легко порушити. За допомогою таких мов також можна визначити деякі текстові елементи як особливі, а їхній формат відрізняється від пов’язаних елементів. Збереження основного тексту, що складається із семантичних елементів і окремо від форматування є життєво важливим у багатьох ситуаціях. Якщо необхідно перекласти текст як на паперові документи, так і на веб-сторінки, зазвичай потрібно, щоб формат у двох отриманих документах відрізнявся. Якщо основний текст містить лише семантичну структуру, це робиться швидко, використовуючи інше відображення від семантичних елементів до інформації про форматування, які зазвичай називають шаблонами або таблицями стилів. З різними таблицями стилів для

різних вихідних форматів, форматування є прив’язаним швидше до вихідного тексту, аніж до вхідного тексту.

1.1.2 Попередня обробка документів

Якщо документи містять звичайний текст, то існує багато варіантів, як обробити такий документ, перш ніж відформатувати його в кінцевий результат. Є багато програм, які добре працюватимуть із простим текстом і дозволять попередньо обробити документи.

Попередня обробка документів часто вимагає певних навичок програмування, користувача може цікавити лише написання тексту, але оскільки ця опція є, то можна написати свій текст, не турбуючись про його обробку спочатку, і додати такі кроки пізніше [4].

Після попередньої обробки можна використовувати багато різних мов розмітки. Для веб-сторінок використовується HTML. TeX і LaTeX використовуються для багато видів текстових документів, також вони особливо ефективні для верстки математики. Досить сильний функціонал попередньої обробки реалізує зв’язка Markdown і Pandoc.

Markdown — це просто звичайний текстовий документ, який можна переписати, перш ніж передати його через Pandoc. Будь-яке переписування файлу до того, як віддати його Pandoc, називається попередня обробка. Pandoc буде читати зі стандартного введення, тому ми можемо передати результат попередньої обробки в нього в командному рядку,як це зображено на рисунку 1.2:

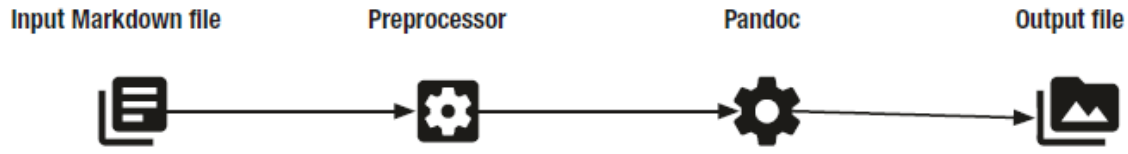


Рисунок 1.2 – Попередня обробка в алгоритмі Pandoc

Якщо припустити, що препроцесор приймає вхідний файл як аргумент і записує його результат у стандартний вихід, то команда може виглядати так:

```
preprocessor testfile.md |
\pandoc --from markdown ... -o outputfile
```

Потрібно лише повідомити Pandoc, що він отримує Markdown як вхідні дані, якщо він зчитує його зі стандартного in, і потім записує файл за допомогою параметра --from. Препроцесор може робити все, що знадобиться, якщо він виводить документ, який Pandoc може обробити. Вихідний файл може бути не лише Markdown — для цього користувач може змінити параметр --from.

Один із способів використання препроцесора — мати деяку інформацію, яку можна повторно використати в деяких файлах, і іншу вхідну інформацію для конкретного документа — наприклад, тіло документа — в іншому файлі. Ця ідея являє собою використання шаблонів, але є й інші випадки, коли можна використати такий підхід. Наприклад існує якийсь довільний клас і щотижня в ньому роздаються вправи. Деякі відомості, такі як назва класу та ім’я викладача, не змінюються від тижня до тижня, але інша інформація змінюється, наприклад, номер тижня. Можна створити файл header.yml із загальною інформацією:

```
class: Markfile and Editor
instructor: Name Suraname
```

Заголовок в даному випадку дуже простий. Можна включити лише файл будь-якої складності, даний приклад просто показує принцип. Для конкретного тижня можна вказати інформацію про тиждень, наприклад, номер тижня та фактичні вправи для цього тижня. Для прикладу файл exercises.md. Він містить вправи для 14 тижнів занять і наступним буде його структура:

```
---¶
#include·"header.yml"·week:·Week·14¶
---¶
#·This·is·an·exercise·Do·something·difficult¶
#·This·is·another·exercises¶
Do·something·even·more·difficult¶
```

Рисунок 1.3 – Вміст файлу exercises.md

Підключення `#include "header.yml"` - це місце, де препроцесор виконує свою роботу. Якщо ми передаємо документ через препроцесор:

```
gpp < exercises.md
```

то отримуємо такий результат:

```
---
class: Markdown and Pandoc
instructor: Thomas Mailund week: Week 14
---
# This is an exercise Do something difficult
# This is another exercises
Do something even more difficult
```

Рисунок 1.4 – Файл exercises.md з новим змістом

Також, цей приклад можна поєднати із шаблоном наступної структури:

```
\documentclass{article}
\usepackage{hyperref}
\title{$class$: $week$}
\author{$instructor$}
\begin{document}
\maketitle
\end{document}
```

Рисунок 1.5 – Новий шаблон форматування вмісту exercises.md

Поєднання препроцесора та Pandoc тепер дозволяє користувачеві створювати документ із власними вправами, даючи на виході готовий комбінований файл

```
gpp exercises.md | \
     pandoc --template exercises.tex \
          --from markdown \
          -o exercises.pdf
```

Рисунок 1.6 – Форматування файлу через препроцесор та Pandoc

Наступним цікавим підходом до комбінації файлів є умовне включення. Продовжуючи приклад вправи, можна уявити, що у файлі також є тести для якогось класу, і потрібно пояснити їм рішення для вправи. Розв’язки легше мати в тому самому документі, що й вправи, але нема сенсу передавати рішення своїм

учням. Тому треба включити відповіді, коли створюється документи до тестів, але виключити їх особливим чином. Це те, з чим GPP також добре справиться.

Достатньо перевірити, чи визначена змінна за допомогою #ifdef. Змінну в даномому випадку не слід плутати зі змінними, з якими працює Pandoc. Препроцесор бачить документ перед Pandoc і не обмінюється даними з Pandoc, крім того, що лише передає в нього вихідні дані.

Якщо необхідно включити або виключити блок тексту, його можна розмістити між #ifdef і #endif. Це також можна зробити для вирішення проблеми попереднього прикладу. Тепер код виглядатиме так:

```
---
#include "header.yml"
week: Week 14
---
# This is an exercise
Do something difficult
#ifdef SOLUTIONS
This is the solution to the exercise
#endif
# This is another exercises
Do something even more difficult
#ifdef SOLUTIONS
This is the solution to the exercise
#endif
```

Рисунок 1.7 – Відключення блоків тексту

Якщо побудувати документ як попередній, то в результаті відповіді не потраплять у вихідний файл.. Можна зробити це у файлі за допомогою оператора #define, але для цієї конкретної програми також можна передати їх у gpp у командному рядку. Це виконується за допомогою опції -D. Цей командний рядок створить PDF-файл, який містить як вправи, так і рішення.

```
gpp -DSOLUTIONS week14_exercises.md | \
pandoc --template exercises.tex \
--from markdown \
-o week14_exercises_solutions.pdf
```

Рисунок 1.8 – Створення PDF-файлу із рішеннями

1.1.3 Markdown та конкурентні мови розмітки текстів

Що робить Markdown особливо приємним у роботі, так це його простота. У HTML, наприклад, потрібно структурувати текст за допомогою тегів, які охоплюють кожен абзац, кожен заголовок, кожен список тощо. Коли відбувається редагування HTML-документу, важко відокремити анотації від самого вмісту. У LaTeX така ж проблема. Анотацію до тексту важко ігнорувати, коли необхідно зосередитися на написанні. Гірше того, якщо документи написані в HTML або LaTeX, велика частина тексту є кодами розмітки, які визначають форматування. Далі наведено приклад простого документу в кількох варіантах форматування. В першому випадку використовуючи Markdown:

```
# Це заголовок першого рівня
Це абзац тексту
## Це заголовок другого рівня
Ще один абзац, за яким слідує
* список
* без номеру
1. і пронумерований
2. список, що містить
3. три пункти
```

Рисунок 1.9 – Приклад тексту на Markdown

Приклад вище демонструє, що розмітки тут мінімальні і вони не заважають читанню чи написанню тексту. Для порівняння, HTML-версія того самого тексту виглядає так:

```
<h1>Це заголовок першого рівня</h1>
<p>Це абзац</p>
<h2>Це заголовок другого рівня</h2>
<p>Ось абзац, за яким слідує</p>
<ul>
     <li>ненумерований</li>
     <li>список</li>
</ul>
     <ol type="1">
     <li>і пронумерований</li>
          <li>список, який містить</li>
     <li>три елементи</li>
</ol>
```

Рисунок 1.10 – Приклад тексту на HTML

Це не дуже складно, і, трохи провівши часу за аналізом такого тексту користувач зможе простежити за структурою документа. Він далеко не такий чистий, як файл Markdown. Більша кількість тегів та їх складність зумовлена складнішими структурами веб сторінок, на відміну від звичайних текстових блоків. Тому такі сторінки можуть містити більше видів медіа інформації, при цьому зберігаючи свою структуру і зрозумілість для розробників.

Версію LaTeX трохи легше читати, ніж файл HTML, але все ще є кілька інструкцій щодо форматування, які заважають простому запису [5].

```
\section{Це заголовок першого рівня}
\subsection{Це заголовок другого рівня}
Ось абзац, за яким йде
\begin{itemize}
\item ненумерований
\item список
\end{itemize}
\begin{enumerate}
\item і нумерований
\item список, що містить
\item три елементи
\end{enumerate}
```

Рисунок 1.11 – Приклад тексту на LaTeX

Наступний приклад на мові reStructuredText. Вона була створена як формат для документів Docutils. Оскільки Docutils є набором інструментів для загальної обробки документів, reStructuredText є більш формалізованим, а згодом у кількох випадках більш докладним. Однак документ у reStructuredText можна легко перетворити на безліч форматів, не турбуючись про реалізацію, яка використовується для розбору документа.

Ця мова дає змогу форматувати крім тексту також різні додаткові структури типу посилань, таблиць, списків, програмного коду та інше. Ось виглядає таблиця на такій мові форматування:

```
+------------------------+------------+----------+----------+
| Заголовок 1, колонка 1 |Заголовок 2 |Заголовок 3|Заголовок 4|
| (заголовки опційні)    |            |          |          |
+========================+============+==========+==========+
| рядок тіла 1, колонка 1| рядок 2    | рядок3   | рядок 4  |
+------------------------+------------+----------+----------+
| рядок 2                | Клітинки можуть об'єднувати колонки|
+------------------------+------------+----------------------+
| рядок 3                |Клітинки    | - клітинки таблиці   |
+------------------------+об'єднують  | - містять            |
| рядок 4                |рядки       | - елементи таблиці   |
+------------------------+------------+----------------------+
```

Рисунок 1.12 – Приклад таблиці на reStructuredText

А тепер для порівняння проста табличка з форматуванням Markdown. Для неї буде використано простіший варіант. Але навіть в такому форматі, добре видно що на відміну від reStructedText потрібно набагато менша кількість символів для того щоб швидко організувати табличку. Це особливо пришвидшує роботу та допомагає при записуванні якоїсь структурованої інформації (вебінари, лекції, тощо).

```
|Ліве вирів.  |  Вирів. по центру  |  Праве вирів.|
|:---         |       :---:       |          ---:|
|git статус   |    git статус     |    git статус|
|git версія   |    git версія     |    git версія|
```

Рисунок 1.13 – Приклад таблиці на Markdown

Отже Markdown розроблено так, щоб користувачі могли анотувати свій текст семантичною інформацією з невеликими домішками анотацій. Читання введеного тексту є майже таким легким, як читання відформатованого тексту. З Markdown немає такої самої можливості для керування форматуванням, як у мові LaTeX. Також гнучкість форматування таблиць не досягає рівня reStructedText,

особливо можливість додаткового поділу клітинок чи їх об’єднання. Але зручність Markdown повністю компенсує це, враховуючи досить прості сфери застосування такої мови.

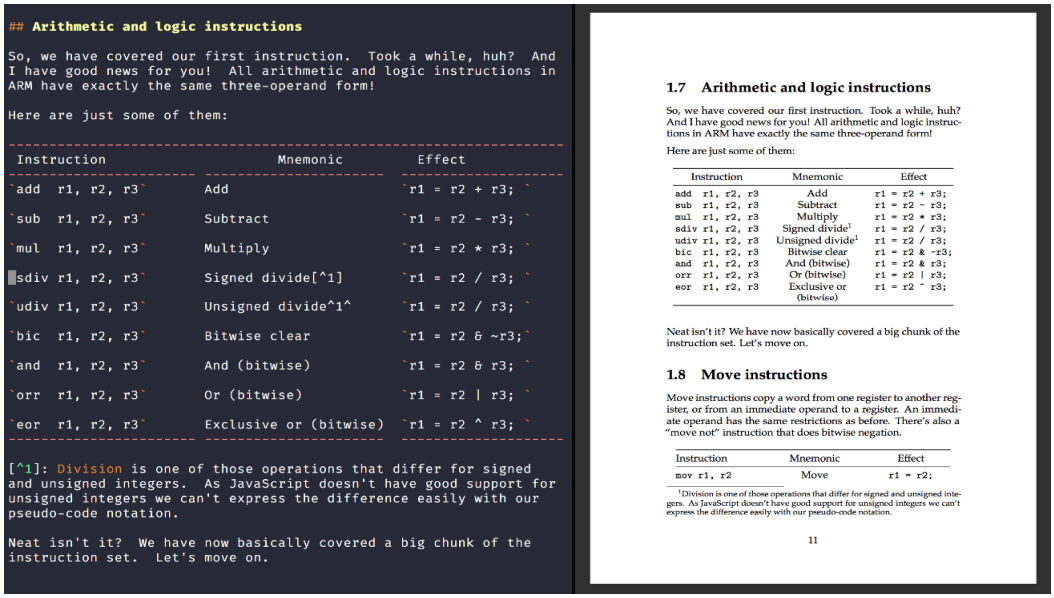
1.1.4 Вбудовані можливості Markdown в текстових редакторах

Markdown — це просто мова для додавання структури до тексту, вона не прив’язана до якогось конкретного інструменту. Багато платформ для ведення блогів дозволять писати текст у Markdown і автоматично форматувати його. Тепер багато текстових редакторів також підтримують Markdown і підтримуватимуть форматування в Markdown та експорт з перетворенням, як правило, з різними варіантами форматування та стилю, які визначають, як виглядатимуть вихідні файли, а також попередній перегляд в реальному часі. Якщо редактор може експортувати в різні формати файлів і в різних стилях, то це, очевидно, найпростіший спосіб для експортування тексту Markdown.

Pandoc — це безкоштовний конвертер документів, який широко використовується як інструмент для написання (особливо науковцями) і як основа для публікування робочих процесів. Його створив Джон Макфарлейн [1].

Формально це бібліотека Haskell для перетворення з одного формату розмітки в інший та середовище з командами, яке використовує цю бібліотеку. Pandoc може конвертувати між численними форматами розмітки та обробкою текстів, включаючи, але не обмежуючись, різні варіанти Markdown, HTML, LaTeX та Word docx. Pandoc також може виводити PDF. Розширена версія Markdown від Pandoc включає синтаксис для таблиць, списків визначень, блоків метаданих, виносок, цитат, математики та багато іншого. Pandoc має великий набір опцій щодо того, як повинні оброблятися документи і він є найбільш універсальним редактром. Він може прийняти документ в будь-якому форматі і провести перетворення розмітки файлу в іншу. Для власних форматів також можуть бути написані плагіни на Lua, який використовувався, наприклад, для реалізації інструменту експортування для набору тегів журналу. Тому якщо потрібно створити простий документ без дрібниць, в цьому допоможе Pandoc.

Pandoc має модульну конструкцію: в основі лежать парсери, які аналізують текст у заданому форматі та створюють оригінальне представлення документа (абстрактне синтаксичне дерево або AST), і набору програм для запису, які перетворюють це рідне уявлення в цільовий формат. таким чином, додавання формату введення або виведення вимагає лише додавання пристрою для зчитування або запису. Користувачі також можуть запускати спеціальні фільтри pandoc, щоб змінити проміжний AST. Оскільки проміжне представлення документа pandoc менш виразне, ніж багато форматів, між якими він перетворює, такі перетворення інколи не ідеальні. Він зберігає структуру файлу, але не форматує такі деталі, як розмір полів. І деякі елементи документа, такі як складні таблиці, можуть не вписуватися в просту модель документа pandoc. У той час як її конверсії Markdown до всіх форматів є майже ідеальними, можна очікувати, що перетворення з форматів, більш виразних, ніж Markdown, будуть з невеликими похибками.



```
## Arithmetic and logic instructions

So, we have covered our first instruction.  Took a while, huh?  And
I have good news for you!  All arithmetic and logic instructions in
ARM have exactly the same three-operand form!

Here are just some of them:

------------------------------------------------------------------
 Instruction                Mnemonic          Effect
-------------------- ---------------------  --------------------
`add  r1, r2, r3`     Add                   `r1 = r2 + r3; `

`sub  r1, r2, r3`     Subtract              `r1 = r2 - r3; `

`mul  r1, r2, r3`     Multiply              `r1 = r2 * r3; `

`sdiv r1, r2, r3`     Signed divide[^1]     `r1 = r2 / r3; `

`udiv r1, r2, r3`     Unsigned divide^1^    `r1 = r2 / r3; `

`bic  r1, r2, r3`     Bitwise clear         `r1 = r2 & ~r3;`

`and  r1, r2, r3`     And (bitwise)         `r1 = r2 & r3; `

`orr  r1, r2, r3`     Or (bitwise)          `r1 = r2 | r3; `

`eor  r1, r2, r3`     Exclusive or (bitwise) `r1 = r2 ^ r3; `
-------------------- ---------------------  --------------------

[^1]: Division is one of those operations that differ for signed
and unsigned integers.  As JavaScript doesn't have good support for
unsigned integers we can't express the difference easily with our
pseudo-code notation.

Neat isn't it?  We have now basically covered a big chunk of the
instruction set.  Let's move on.
```

1.7 Arithmetic and logic instructions

So, we have covered our first instruction. Took a while, huh? And I have good news for you! All arithmetic and logic instructions in ARM have exactly the same three-operand form!

Here are just some of them:

Instruction	Mnemonic	Effect
`add r1, r2, r3`	Add	`r1 = r2 + r3;`
`sub r1, r2, r3`	Subtract	`r1 = r2 - r3;`
`mul r1, r2, r3`	Multiply	`r1 = r2 * r3;`
`sdiv r1, r2, r3`	Signed divide[1]	`r1 = r2 / r3;`
`udiv r1, r2, r3`	Unsigned divide[1]	`r1 = r2 / r3;`
`bic r1, r2, r3`	Bitwise clear	`r1 = r2 & ~r3;`
`and r1, r2, r3`	And (bitwise)	`r1 = r2 & r3;`
`orr r1, r2, r3`	Or (bitwise)	`r1 = r2 | r3;`
`eor r1, r2, r3`	Exclusive or (bitwise)	`r1 = r2 ^ r3;`

Neat isn't it? We have now basically covered a big chunk of the instruction set. Let's move on.

1.8 Move instructions

Move instructions copy a word from one register to another register, or from an immediate operand to a register. An immediate operand has the same restrictions as before. There's also a "move not" instruction that does bitwise negation.

Instruction	Mnemonic	Effect
`mov r1, r2`	Move	`r1 = r2;`

[1] Division is one of those operations that differ for signed and unsigned integers. As JavaScript doesn't have good support for unsigned integers we can't express the difference easily with our pseudo-code notation.

11

Рисунок 1.14 – Робота з Markdown у Pandoc

GitBook — це інструмент спільної документації, який дозволяє будь-кому документувати будь-що — наприклад, продукти та API — і ділитися знаннями через зручну для користувача онлайн-платформу [6].

Згідно з цим GitBook — це гнучка платформа для різного контенту та співпраці. Він забезпечує єдиний уніфікований робочий простір для різних

користувачів, щоб створювати, керувати та обмінюватися вмістом без використання кількох інструментів. Наприклад:

- особи можуть використовувати GitBook для відстеження своїх особистих проектів, додавання нотаток чи ідей;
- команди можуть централізувати та ділитися своїми внутрішніми базами знань на GitBook, що покращує співпрацю та робить пошук інформації зручнішим;
- організації, у тому числі Arctype, можуть створювати чудові документи, щоб направляти та підтримувати своїх користувачів і учасників.

Багато популярних інструментів мають спільні з GitBook функції, зокрема підтримку користувацьких доменів, експорт PDF, пошук і можливості навігації. Однак GitBook перевершує ці рішення для документації в багатьох аспектах, включаючи, але не обмежуючись такими можливостями:

- відмінні можливості налаштування для відображення ідентичності будь-якого бренду;
- чудова система плагінів із майже 700 плагінами, які функціонально розширюють стандартний GitBook;
- найзручніша інтеграція GitHub, яку можна знайти, щоб легко синхронізувати документацію з GitHub і підтримувати все в актуальному стані.

Тому в сценарії, коли такі вимоги, як просунутий брендинг, настроюваний дизайн інтерфейсу користувача та розширюваність функцій, є важливими, існуючі альтернативи не можуть конкурувати з GitBook;

- у стандартній комплектації є чудовий онлайн-редактор WYSIWYG і підтримка Markdown;
- можна налаштувати відповідно до бренду будь-якої організації.

Є базова версія, яка є абсолютно безкоштовною для особистого використання, а платну преміальну версію також можна безкоштовно

ліцензувати. GitBook відображає вміст публічно чи приватно з ким завгодно, включно з користувачами, які не є GitBook. Його можна налаштувати для синхронізації вмісту з GitHub, а також для створення PDF-версій документації. Він підтримує інтеграцію з такими сервісами як Slack, Intercom, Google Analytics.

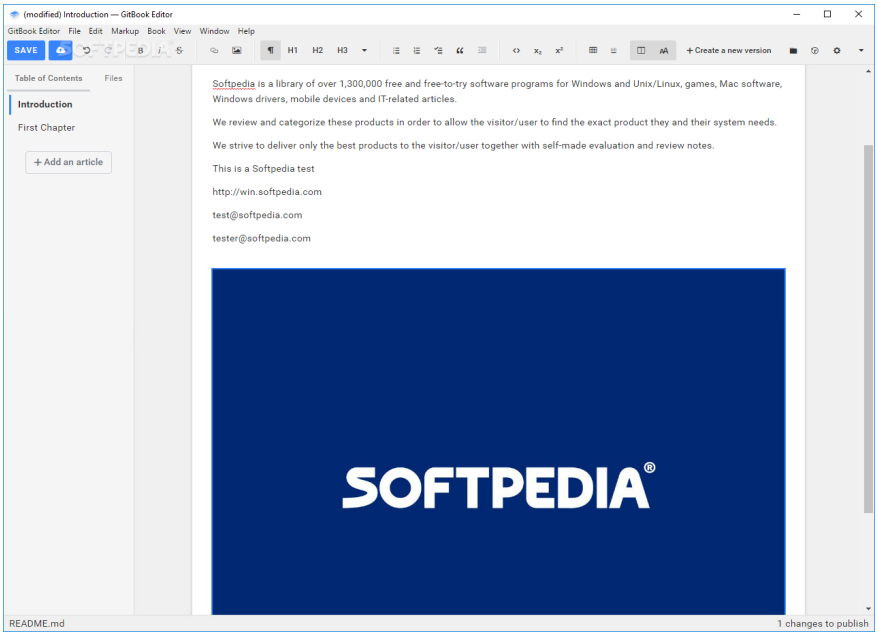


Рисунок 1.15 – Редактор Gitbook, що підтримує Markdown

MkDocs — це швидкий, простий і просто чудовий генератор для статичних сайтів, який призначений для створення проектної документації. Вихідні файли документації записуються в Markdown і налаштовуються за допомогою одного конфігураційного файлу YAML [7].

Роботу з таким інструментом можна почати з прочитання вступного посібника, а потім посібника користувача для отримання додаткової інформації. Є можливість налаштувати свою тему та/або встановивши деякі плагіни (для розширення функціоналу Markdown), щоб розроблювана проектна документація виглядала саме так, як цього хоче користувач. Змінити поведінку Markdown легко, за допомогою розширень Markdown. Доступно багато варіантів конфігурації. Є функція перегляду створюваного сайту. Вбудований сервер розробника дозволяє переглядати документацію в момент написання. Він навіть автоматично перезавантажить і оновить браузер щоразу, коли користувач зберігатиме зміни. MkDocs створює повністю статичні HTML-сайти, які можна розміщувати на

сторінках GitHub, Amazon S3 або де завгодно. Коли розробники працюють над будь-яким проектом, документація надзвичайно корисна, майже критична. На щастя, MkDocs створив гарний, ефективний метод створення документації, який виглядає професійним та лаконічним.

У документації до програмного забезпечення написано про розроблюваний код і власне проект, пояснюючи, про мету та функціонал. Це важливо для всі хвидів проектів. Документуючи такий код, розробники досягає наступних цілей:

- зробити свій код зрозумілим для інших, з ким відбувається співпраця (і для себе, коли в майбутньому будете переглянете свій код);
- слідкувати за всіма аспектами ПЗ;
- зробити налагодження потім простішим;
- організувати універсальний простір для зберігання файлів.

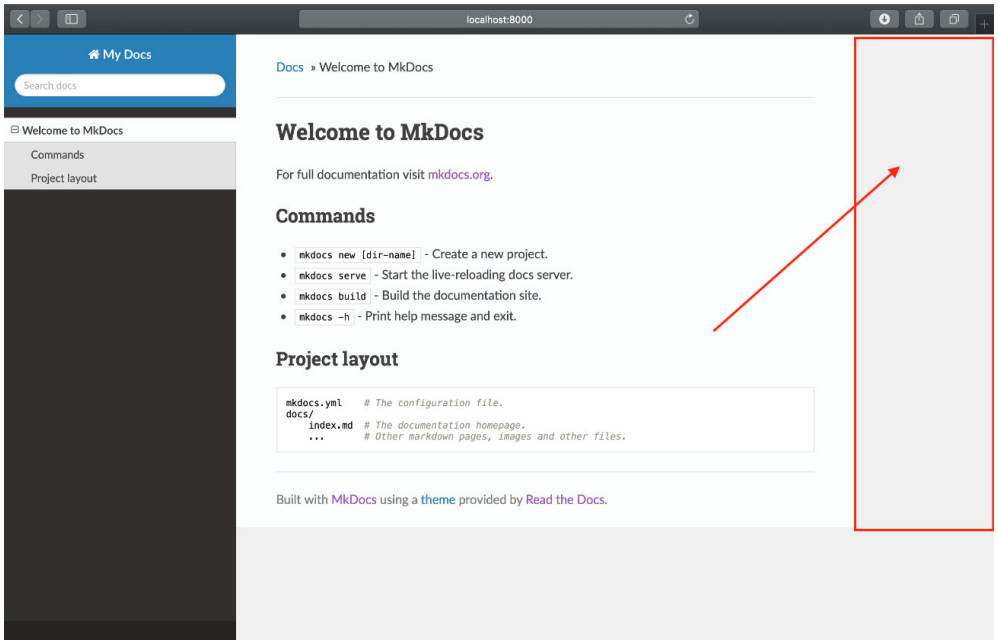


Рисунок 1.16 – Приклад сайту на основі Markdown в MkDocs

Створення скелета будь-якої документації за допомогою MkDocs має зайняти не більше 10 хвилин. Витрачати 10 хвилин на документування в момент розробки варто того, щоб потім не витратити незліченну кількість хвилин, намагаючись налагодити код і пояснити його колегам і для себе. Після створення

документації до проекту у вигляді не складних веб сторінок, MkDocs має вбудовану можливість розгорнути такі сторінки на сервісі GitHub Pages.

GitHub Pages — це статична служба хостингу сайтів, яка приймає документи JavaScript, HTML, CSS прямо зі сховища на GitHub, за бажанням запускає файли в процесі створення та публікує веб-сайт [8].

Користувач може розмістити свій сайт у домені GitHub github.io або у своєму власному домені. Існує три типи сайтів GitHub Pages: проект, користувач і організація. Сайти проекту підключаються до певного проекту, розміщеного на GitHub, наприклад бібліотеки JavaScript або колекції рецептів. Сайти користувачів і організацій підключені до певного облікового запису на GitHub.com. Щоб опублікувати сайт користувача, необхідно створити репозиторій, що належить до облікового запису користувача з назвою <username>.github.io. Щоб опублікувати сайт організації, потрібно створити репозиторій, що належить організації з назвою <organization>.github.io. Якщо не використовується користувацький домен, сайти користувачів і організацій доступні за адресою http(s)://<username>.github.io або http(s)://<organization>.github.io.

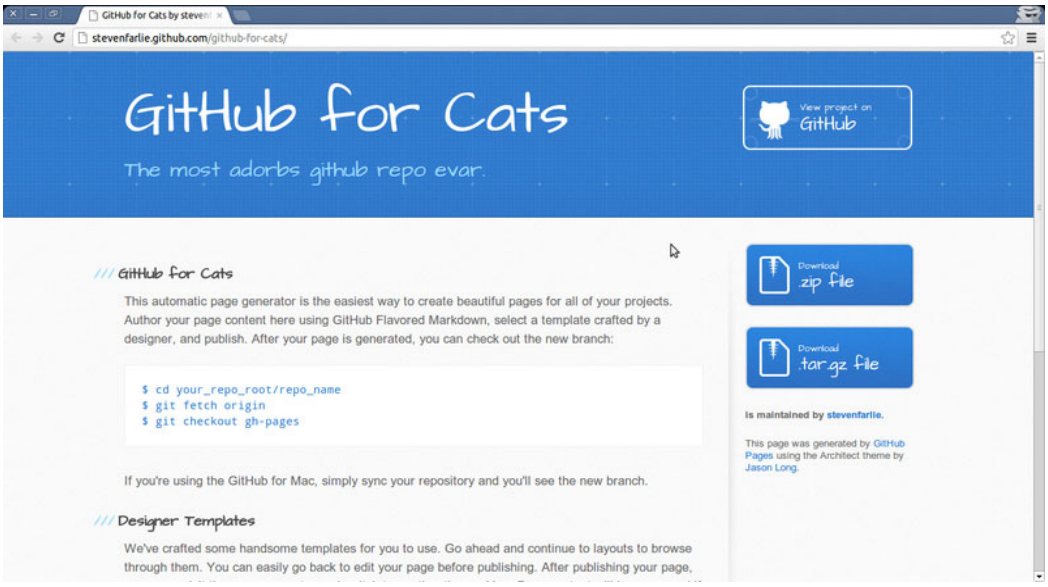


Рисунок 1.17 – Сторінка із MkDocs на GitHub Pages з використанням Markdown

Отже, суть філософії Markdown полягає в тім, що звичайні текстові документи повинні бути читаними без тегів, які все заплутують, але все одно повинні бути способи додавати текстові модифікатори, такі як жирний шрифт,

списки, курсив тощо. Це альтернатива WYSIWYG редакторам, які використовують форматований текст, який пізніше перетворюється на правильний HTML. На просторах інтернету можна зіткнутися з Markdown, не усвідомлюючи цього. Чат Facebook, Skype і Reddit дозволяють використовувати різні варіанти Markdown для форматування повідомлень. Практично всі гіганти редагування тексту та коду, Visual Code, Atom, Sublime Text містять вбудовані бібліотеки для роботи з Markdown. Та крім них, в даному розділі було згадано багато спеціалізованих застосунків для роботи з книгами та навіть сайтами, що також побудовані на цій мові. Markdown є надзвичайно актуальною темою в даний час, він активно використовується в багатьох редакторах тексту та коду.

1.2 Сфери застосування мови розмітки Markdown

В попередніх розділах було розглянуто Markdown та її конкурентів та зроблено висновки щодо цього. Подальший аналіз текстових редакторів та інструментів для форматування книг показав, що дана мова часто вже є вбудованою та застосовується в різних ситуаціях. Виникає враження, що цією мовою можна зробити майже все. Markdown — це дуже простий та швидкий спосіб робити нотатки, створювати вміст для веб-сайту та створювати документи, готові до друку.

Вивчення синтаксису Markdown не займе багато часу, і освоївши його використання її можна застосовувати майже скрізь. Більшість людей використовують Markdown для створення вмісту для Інтернету, але Markdown добре підходить для форматування всього, від електронної пошти до списків продуктів.

1.2.1 Документи

Markdown не володіє всіма перевагами текстових процесорів, на відміну від Microsoft Word, але він достатньо хороший для створення основних документів, таких як завдання та листи. Тому Markdown застосовують для створення швидких

документів, для створення та експорту документів з формату Markdown у формат файлів PDF або HTML [9].

Частина PDF є ключовою, оскільки, коли у наявності є PDF-документ, робити з ним можна що завгодно — роздрукувати його, надіслати електронною поштою, перевести у інший формат або завантажити на веб-сайт. Ось деякі програми для створення документів Markdown, які рекомендуються відомими IT працівниками та блогерами відповідної до кожної сфери чи ОС:

- Mac: iA Writer, Ulysses, Marked або MacDown;
- iOS / Android: iA Writer або Ulysses (лише iOS);
- Windows: MarkdownPad або ghostwriter (проект GitHub);
- Web: Dillinger або StackEdit на рисунку 1.6.

І iA Writer, і Ulysses надають шаблони для попереднього перегляду, друку та експорту документів у форматі Markdown. Наприклад, шаблон «Академічний – стиль MLA» від iA Writer робить абзаци відступами та додає подвійний інтервал між реченнями [3].

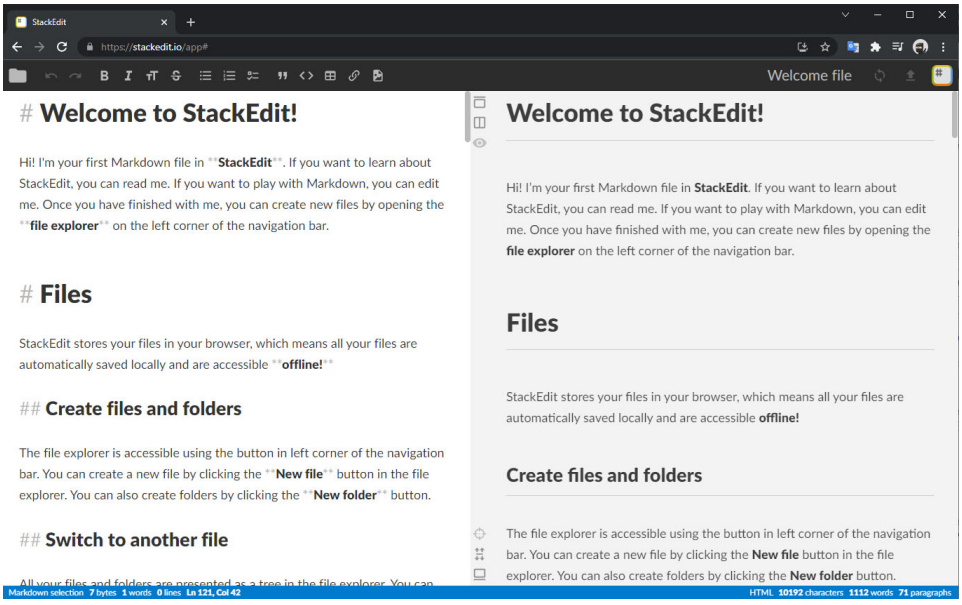


Рисунок 1.18 – Використання веб-редактора StackEdit

1.2.2 Нотатки

Майже в усіх відношеннях Markdown є ідеальним синтаксисом для конспектування. На жаль, Evernote і OneNote, дві з найпопулярніших програм для

нотаток, наразі не підтримують Markdown. Проте, багато інших програм для нотаток підтримують Markdown:

- Simplenote — це безкоштовна програма для створення нотаток, доступна для кожної платформи;
- Bear — це програма, схожа на Evernote, доступна для пристроїв Mac та iOS, вона не використовує виключно синтаксис Markdown за замовчуванням, але є можливість ввімкнути режим сумісності Markdown;
- Boostnote оголошується як «додаток для нотаток, створений програмістами для програмістів».

Якщо користувачу важко розлучитися з Evernote, можна переглянути Marxico, редактор Markdown на основі підписки для Evernote. Напотужнішою програмою для організування записок та задач є MyLifeOrganized для Windows та інших систем. Використовуючи її, можна оганізувати свої завдання на проекти та розбивати великі завдання на під задачі. Відкривати перегляд проектів, щоб легко переглянути всі свої проекти. На вкладки є функції для зберігання та швидкого доступу до певної конфігурації для вибраного подання, додаткового фільтра, масштабування, вибраного завдання тощо. І найголовніше для даної роботи, це легке та швидке додавання заголовків, списків, жирного, курсиву, посилань, зображення та інших елементів – використовуючи мову Markdown.

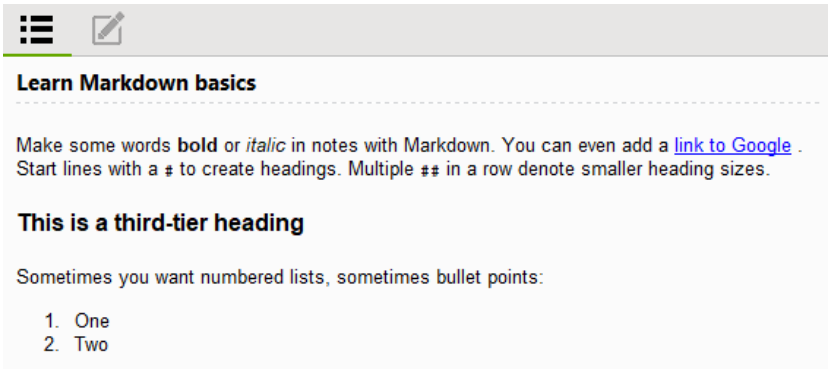


Рисунок 1.19 – Використання Markdown в MyLifeOrganized

1.2.3 Презентації

Не зважаючи на досить простий синтаксис та можливості мови Markdown користувач може створювати презентації з файлів, відформатованих Markdown. До створення презентацій у Markdown потрібно трохи звикнути, але як тільки з'явиться досвід з його використання, це стане набагато швидше та легше, ніж використовувати Keynote або ж PowerPoint. Remark (проект GitHub) є популярним інструментом для показу слайдів Markdown на основі браузера, як і Cleaver (проект GitHub). Якщо звичною у використанні є Mac і є потреба використовувати програму для презентацій на основі Markdown, можна скористуватись Deckset або Marked.

Яскравим прикладом такої програми для презентацій на ОС Windows є Marp. Це не просто програма, а ціла екосистема презентації на основі Markdown. Тут є можливість створювати красиві колоди слайдів, використовуючи інтуїтивно зрозумілий досвід Markdown. Marp (також відомий як екосистема презентації Markdown) забезпечує інтуїтивно зрозумілий досвід для створення красивих слайдів. Для цього потрібно лише зосередитися на написанні своєї історії в документі Markdown. Тут є можливість використовувати директиви та розширений синтаксис. Іноді простого текстового вмісту недостатньо, щоб підкреслити голос, тому Marp підтримує різноманітні прийоми, на кшталт: синтаксис зображення, математичний набір, автоматичне масштабування тощо. І це все для створення красивих слайдів. Вбудовані теми та CSS-теми. Основний механізм Marp має 3 вбудовані теми, які називаються за замовчуванням, gaia та uncover, щоб красиво розповісти презентовану історію. Якщо потрібно налаштувати свій дизайн, користувач може використовувати Marp, щоб налаштувати стилі за допомогою Markdown, або створити власну тему Marp за допомогою звичайного CSS. Сімейство Marp має офіційний набір інструментів. Екосистема Marp містить багатий набір інструментів, щоб допомогти у роботі. Marp для VS Code — це розширення, яке дозволяє редагувати та переглядати слайд Markdown та користувацькі теми в VS Code. Marp CLI — це застосунок, який дозволяє конвертувати Markdown за допомогою простого інтерфейсу CLI.

Також тут є можливість експортувати в HTML, PDF і PowerPoint. Після того як презентація написано, Marp може конвертувати Markdown у готові до презентації файли HTML, PDF та PowerPoint безпосередньо, на основі Google Chrome / Chromium. З додаткових особливостей можна згадати модульну архітектуру. Насправді, Marp по суті є просто конвертером для Markdown. Екосистема Marp побудована на фреймворку Marpit, вузькому фреймворку для створення слайдів HTML/CSS. Він побудований на модульній архітектурі, і будь-який розробник може розширити функції за допомогою плагінів.

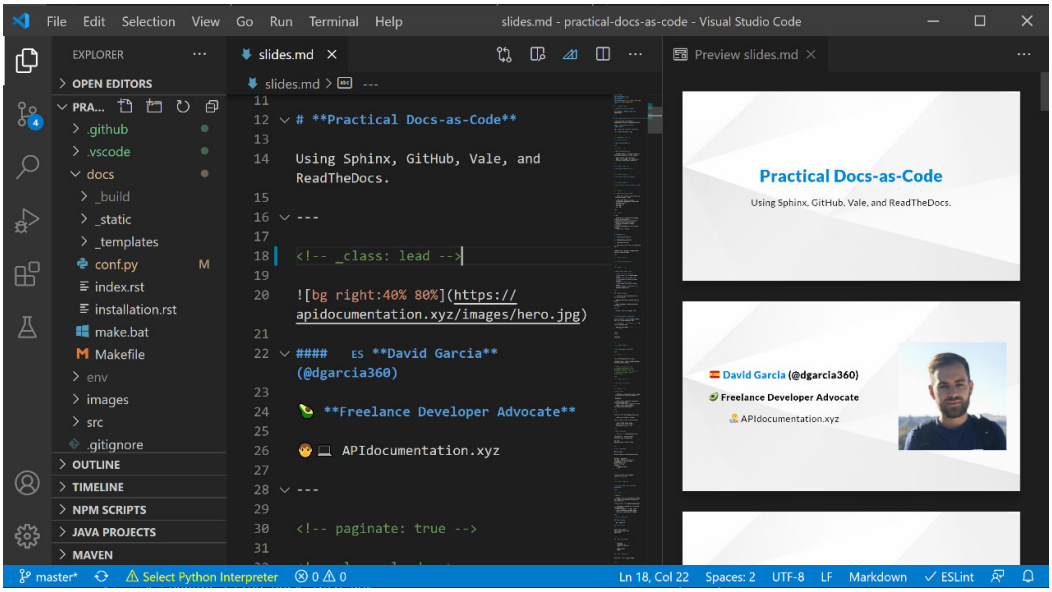


Рисунок 1.20 – Зразок написання презентації в Marp

1.2.4 Email

Електронне листування залишається все ще популярним засобом зв'язку чи інформування. Якщо потрібно надсилати багато електронних листів і від елементів керування форматуванням, доступних на більшості веб-сайтів постачальників електронної пошти можна лише втомитись, але не добитись бажаного дизайну, то є простий спосіб писати повідомлення email використовуючи Markdown.

Markdown Here – це безкоштовне розширення веб переглядача, яке перетворює відформатований Markdown текст у HTML, готовий до надсилання. Markdown Here допоможе писати електронні листи швидше та потужніше, ніж це пропонують стандартні інструменти. Без додаткових зусиль, Markdown Here

усуває всі клопоти щодо форматування електронної пошти. До цього часу форматування електронної пошти додавало клопоту: вибрати текст, натиснути кнопки форматування, а потім ще трохи ручних правок, щоб усе все правильно. Це нудно, розчаровує і повільно. Зате Markdown Here дозволяє писати складні електронні листи простим текстом, не відриваючи рук від клавіатури. Коли лист готовий, лише одним клацанням миші електронна пошта буде готова до надсилання.

Markdown дає додаткові спеціальні можливості, які допоможуть робити те, що майже неможливо в звичайних редакторах електронної пошти. Наприклад, додавати таблиці, фрагменти вихідного коду та навіть математичні формули в електронну пошту, як це зображено на рискну 1.9:

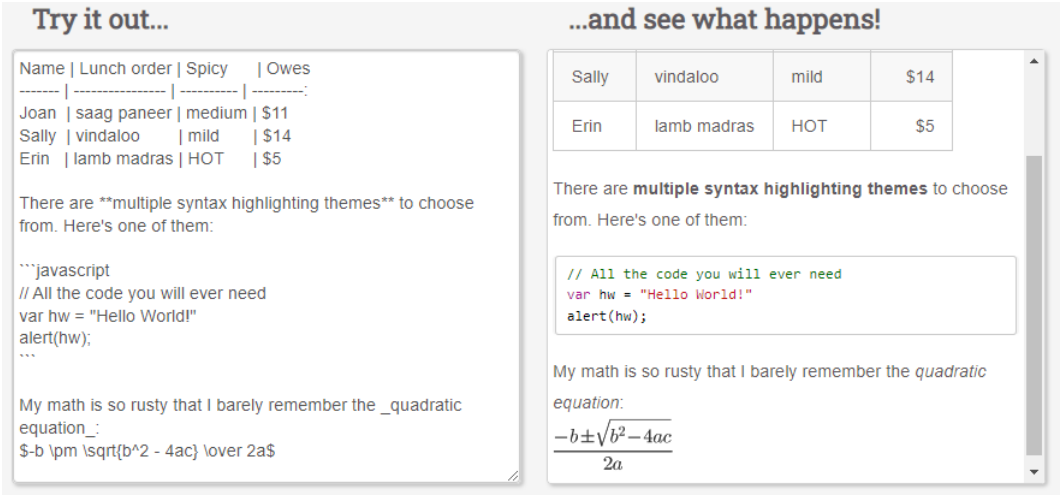


Рисунок 1.21 – Markdown форматування email листів

Markdown Here має розширення для всіх популярних браузерів: Opera, Firefox, Chrome, Safari, а також спеціалізованих програм для роботи з поштою: Thunderbird і Postbox. З Markdown Here можна написати електронну пошту в Markdown, а потім натиснути кнопку чи пункт контекстного меню, або скористатися гарячою клавішею і перетворити електронний лист у гарний, повністю відформатований текст, який технічно виглядає як HTML. Також Markdown Here доповнює стандарти мови. Він дає змогу змінювати кольори, шрифти, інтервали, відступи та вирівнювання. Організувати що-небудь настільки

маленьке, як налаштувати простір між маркерами, або таке велике, як створити цілу тему електронної пошти. І це все завдяки Markdown.

1.2.5 Документація

Markdown був розроблений для Інтернету, тому не дивно, що існує багато програм, спеціально розроблених для створення вмісту веб-сайту. Якщо потрібно знайти найпростіший спосіб організувати веб-сайт з файлами Markdown, відповідь є на сайтах blot.im і smallvictori.es. Після реєстрації в одній із цих служб вони створюють папку Dropbox на комп'ютері. Потім просто перетягнувши файли Markdown в папку вони автоматично з'являються на веб-сайті. Порівняно з такими технологія як WordPress чи просто створювання сторінок вручну – це набагато легше [10].

Якщо такі технології як HTML, CSS і контроль версій є знайомим, то достатньо ознайомитись з Jekyll, популярним генератором статичних сайтів, який приймає файли Markdown і створює веб-сайт HTML. GitHub Pages надає безкоштовний хостинг для веб-сайтів, створених Jekyll. Якщо Jekyll не підходить під конкретні задачі, то можна вибрати інші доступні генератори статичних сайтів. Наприклад Jekyll використовувався для веб-сайту Markdown Guide. Markdown Guide — це безкоштовний довідковий посібник із відкритим вихідним кодом. Він пояснює, як використовувати Markdown, просту й легку у використанні мову розмітки, яку можна застосувати для форматування практично будь-якого документа. Якщо є потреба використовувати систему керування вмістом (CMS) для підтримки свого веб-сайту, можна скористуватись Ghost. Це безкоштовна платформа для ведення блогів з відкритим вихідним кодом із гарною програмою редагуавння Markdown. Якщо зручніше користуватися WordPress, то для веб-сайтів, розміщених на WordPress.com, є підтримка Markdown. Сайти WordPress, розміщені на власному хостингу, можуть використовувати плагін Jetpack. Даний плагін дає змогу:

- резервне копіювання та відновлення в один клік;
- автоматичне сканування шкідливих програм і виправлення в один клік;

- потужний захист від спаму для коментарів і форм.

Markdown — дуже зручно підходить для технічної документації. Такі компанії, як GitHub, все частіше переходять на Markdown для своєї документації — можна ознайомитись з їхнім дописом у блозі про те, як вони перенесли свою відформатовану Markdown документацію на Jekyll. Якщо потрібно написати документацію для продукту чи послуги, то існують такі інструменти:

- Із Read the Docs можна створити веб-сайт документації з наявних файлів Markdown з відкритим вихідним кодом, достатньо підключити свій репозиторій GitHub до їхньої служби та вивантажити його;
- MkDocs – це застосунок для генерування статичних сайтів, який призначений для створення документації. Вихідні файли записані в Markdown і організовані за допомогою одного конфігураційного файлу YAML;
- Docusaurus – ще один генератор сайтів, розроблений виключно для створення веб-сайтів з документацією, він підтримує переклади, пошук та керування версіями;
- Jekyll згадувався раніше в розділі про веб-сайти, але це також хороший варіант для створення веб-сайту з документацією з файлів Markdown, виникне потреба скористатись цим шляхом, доведеться добре переглянути документацію Jekyll [11].

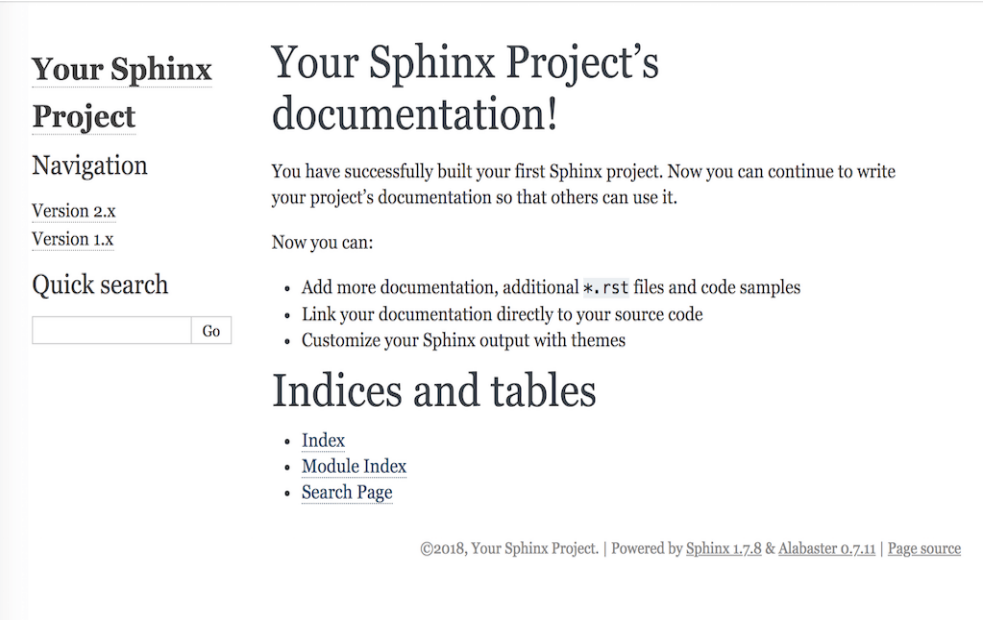


Рисунок 1.22 – Сайт документації на Read the Docs

2 ПОСТАНОВКА ЗАДАЧІ РОЗРОБКИ

2.1 Синтаксис мови форматування Markdown

Практично всі додатки на основі Markdown підтримують основний синтаксис, викладений в оригінальному документі Джона Грубера. Існують незначні відмінності та розбіжності між процесорами Markdown — вони відзначаються в рядках самого документа, де це можливо.

Загловок – достатньо додати кілька решіток (#) перед словом або фразою. Кількість знаків цифр, які використовується, має відповідати рівню заголовка. Для створення заголовока третього рівня (<h3>), використовують три цифрові знаки (наприклад, ### Мій заголовок):

```
# Heading level 1
## Heading level 2
### Heading level 3
#### Heading level 4
##### Heading level 5
###### Heading level 6
```

Рисунок 2.1 – Заголовки тексту у Markdown

Відповідно до логіки HTML, варіант з однією решіткою дає в результаті найвищий заголовок і візуально найбільший по розміру шрифту. Відповідно кожен наступний рівень все менший і менший від попереднього. Крім такого варіанту, до рядка під текстом можна додати будь-яку кількість знаків == для рівня заголовка 1 або символів -- для рівня заголовка 2:

```
Heading level 1
===============
Heading level 2
---------------
```

Рисунок 2.2 – Інший варіант заголовків

Також текст в Markdown можна розділяти на абзаци. Щоб створити абзаци, використовують порожній рядок, щоб розділити декілька рядків тексту:

```
this is a sample text on a markdown formatting

this is a sample text on a markdown formatting
```

Рисунок 2.3 – Поділ тексту на абзаци

Щоб створити розрив рядка (
), рядок закінчують двома або більше пробілами і після цього додають enter:

```
this is a sample text on a markdown formatting
this is a sample text on a markdown formatting
```

Рисунок 2.4 – Поділ тектсу на рядки

Майже в кожній програмі Markdown можна використовувати два або більше пробілів (які зазвичай називають «кінцевими пробілами») для розривів рядків, але це викликає незручності. Важко побачити кінцеві пробіли в редакторі, і багато людей випадково чи навмисно ставлять два пробіли після кожного речення. З цієї причини можна використовувати щось інше, ніж кінцеві пробіли для розривів рядків. Якщо використовувана програма Markdown підтримує HTML, то застосовують тег HTML
. Така версія Markdown як CommonMark і інші,

дозволяють вводити зворотну косу риску (\) в кінці рядка, але це не найкращий варіант з точки зору сумісності. І принаймні є кілька версій мови розмітки, що не вимагають нічого в кінці рядка — просто return.

Також текст можна виділити: жирним або курсивом. Щоб виділити жирний текст, треба додати дві зірочки або підкреслення до та після слова чи фрази. Щоб підкреслити середину слова, додають дві зірочки без пробілів навколо букв:

```
this is a **sample text** on a markdown formatting
this is a __sample text__ on a markdown formatting
this is a sample text on a m**ark**down formatting
```

Рисунок 2.5 – Виділення текстом жирним та курсивом

Програми Markdown не погоджуються щодо того, як обробляти підкреслення в середині слова. Для сумісності використовують зірочки для виділення середини слова для наголосу. Для тексту з курсивом, додають одну зірочку або підкреслення перед і після слова чи фрази. Щоб виділити середину слова для наголосу, слід додати одну зірочку без пробілів навколо літер:

```
another _sample_ of a text string
another sample of a *text* string
```

Рисунок 2.6 – Виділення частин слів курсивом

Щоб підкреслити текст жирним і курсивом одночасно, додають три зірочки або підкреслення до та після слова чи фрази. Як і в попередніх прикладах, для того щоб виділити лише частину слова, прийнято використовувати зірочки. Для жирного тексту та курсиву по середині слова для наголосу, слід додати три зірочки без пробілів навколо літер:

```
another ***sample of a text*** string
another s***amp***le of a text string
```

Рисунок 2.7 – Виділення частин слів жирним

Щоб створити блок-цитату, додають символ > перед абзацом тексту:

```
> Markdown is the best tool for noting and styling. Don’t trust
me? Just try it by your own!
```

Рисунок 2.8 – Створення блок-цитати в Markdown

Блокові цитати дозволяють вміщати кілька абзаців. Додаючи > на порожні рядки між абзацами можна розтягнути сам блок:

```
> Dorothy followed her through many of the rooms in her castle.
>
> The Witch bade her clean the pots and kettles and sweep the
floor and keep the fire fed with wood.
```

Рисунок 2.9 – Поділ блок-цитат на абзаци

Блокові цитати також можуть бути вкладеними. Для цього додають символ >> перед абзацом, який потрібно розмістити:

```
> Dorothy followed her through many of the rooms in her castle.
>
>> The Witch bade her clean the pots and kettles and sweep the
floor and keep the fire fed with wood.
```

Рисунок 2.10 – Вкладений абзац із блок-цитатою

Блокові цитати можуть вміщати інші відформатовані елементи Markdown. Не всі елементи можна використовувати — для цього треба проекспериментувати, щоб побачити, які з них працюють. Для сумісності вставляють порожні рядки до та після блокових лапок.

Markdown дає можливість упорядковувати елементи в упорядковані та невпорядковані списки. Спочатку приклад з використанням впорядкованих списків. Для того щоб створити впорядкований список, додають рядки з числами, за якими слідують крапки. Числа не обов’язково мають бути в порядку чисел, але список має починатися з числа один. Також використовуючи табуляцію в середині самих списків, можна створювати вкладені списки. Для вкладених списків діють ті ж правила, що і для основних. CommonMark та кілька інших легких мов розмітки дозволяють використовувати дужки ()) як роздільник

(наприклад, 1) Перший елемент), але це не найкращий варіант з точки зору сумісност:

```
1. First case
2. Second case
3. Third case
    1. Indented case
    2. Indented case
4. Fourth case
```

Рисунок 2.11 – Створення упорядкованого списку

Для створення невпорядкованого списку, додають тире (-), зірочки (*) або знаки плюс (+) перед позиціями рядка. А також можна використати табуляцію для одного або кількох елементів, щоб створити вкладений список:

```
+ First case
+ Second case
+ Third case
    * Indented case
    * Indented case
+ Fourth case
```

Рисунок 2.12 – Створення невпорядкованого списку

Блоки коду зазвичай мають відступ у 4 пробіли або одну табуляцію. Коли вони знаходяться в списку, для них роблять відступ на 8 пробілів або 2 табуляції:

```
1. відкрийте лістинг програми.
2. знайдіть наступний блок коду на рядку 21:
        <html>
          <head>
            <title>Test</title>
          </head>
3. Змініть назва, щоб вона відповідала імені веб-сайту
```

Рисунок 2.13 – Правила оформлення блоків з кодом

Також з Markdown можна вставляти в текст картинки. Використовують файли, що розміщуються локально або навіть з інтернету, для цього потрібно

лише вказати шлях, абсолютний або відносно директорії в якій розміщується текстовий документ в квадратних дужках, а також можна добавити підпис до картинки в круглих дужках зразу перед посиланням:

```
![This is the mascot of the Linux](/assets/images/tux.png)
```

Рисунок 2.14 – Використання картинок в тексті

Щоб швидко перетворити URL-адресу та адресу email на посилання, достатньо обгорнути їх у кутові дужки:

```
<https://www.markdownislife.org>
<testtest@mail.com>
```

Рисунок 2.15 – Створення гіперпосилань використовуючи Markdwon

Щоб створити горизонтальну лінію, використовують більше трьох зірочок (***), підкреслення (___) чи тире (---) окремо на рядку. Також для кращої сумісності порожні рядки вставляють до та після горизонтальних ліній:

```
Try to put a blank line before...

---

...and after a horizontal rule.
```

Рисунок 2.16 – Створення горизонтальних ліній в тексті

Далеко не всі Markdown додатки підтримують HTML у документах Markdown. Для цього варто перевірити документацію використовуваної Markdown програми. Багато з них підтримують тільки частину синтаксису HTML. Слід вставляти порожні рядки, щоб відокремити елементи HTML на рівні блоків, як-от , <table>, <pre> і <p>, від навколишнього вмісту. Не треба робити відступи в тегах табуляції чи пробіли — це може заважати форматуванню. Не

можна використовувати Markdown перемішуючи з тегами HTML. Наприклад, такий текст: italic and **bold** не працюватиме.

2.2 Застосування C# для обробки тексту

Операції з рядками в C# дуже оптимізовані і в більшості випадків не впливають істотно на продуктивність. Однак у деяких сценаріях, таких як жорсткі цикли, які виконуються багато сотень або тисячі разів, операції з рядками можуть вплинути на продуктивність. Оскільки робота над рядками лежить в основі бібліотеки для Markdown, то особливості її застосування в C# потрібно добре проаналізувати та систематизувати задля успішного виконання завдань.

Рядок – це об’єкт типу String, значення якого – текст. Внутрішньо текст зберігається як послідовна колекція об’єктів Char, доступна лише для читання. У кінці рядка C# немає символу закінчення нуля; тому рядок C# вміщає довільне число вбудованих нульових символів ('\0'). Властивість Length рядка представляє кількість об’єктів Char, які він містить, а не кількість символів Unicode. Щоб отримати доступ до окремих точок коду Unicode в рядку, використовуйте об’єкт StringInfo [12].

У C# ключове слово string є псевдонімом String. Таким чином, String і string є еквівалентними, незалежно від того, рекомендується використовувати наданий рядок псевдоніма, оскільки він працює навіть без using System. Клас String надає багато методів для безпечного створення, маніпулювання та порівняння рядків. Крім того, мова C# перевантажує деякі оператори, щоб спростити звичайні операції з рядком.

В мові C# є загалом 9 способів, якими можна оголосити і створити об’єкт типу рядка. Здебільшого не відбувається використання оператор new для створення об’єкта рядка, за винятком випадків, коли рядок ініціалізується масивом символів. Також ініціалізують рядок зі значенням константи Empty, для створення нового об’єкту String, рядок якого має нульову довжину. Рядковим літералом для рядка нульової довжини є "". Ініціалізуючи рядки значенням Empty замість null, можна зменшити ймовірність виникнення винятку

NullReferenceException. Тому використовують статичний метод IsNullOrEmpty(String), щоб перевірити значення рядка, перш ніж спробувати отримати до нього доступ.

Рядкові об’єкти незмінні: їх не можна змінити після їх створення. Усі методи String та оператори C#, які, здається, змінюють рядок, насправді повертають результати в новому об’єкті рядка. У наступному прикладі, коли вміст s1 і s2 об’єднується в один рядок, два вихідні рядки залишаються незмінними. Оператор += дає новий рядок, що містить об’єднаний вміст. Цей новий об’єкт призначається змінній s1, а вихідний об’єкт, який був призначений для s1, звільняється для збирання сміття, оскільки жодна інша змінна не містить посилання на нього, що і демонструється на рисунку 2.17:

```
string s1 = "A string is more ";
string s2 = "than the sum of its chars.";

// Concatenate s1 and s2. This actually creates a new
// string object and stores it in s1, releasing the
// reference to the original object.
s1 += s2;

System.Console.WriteLine(s1);
// Output: A string is more than the sum of its chars.
```

Рисунок 2.17 – Immutable поведінка string об’єктів

Оскільки «модифікація» рядка насправді є створенням нового рядка, треба бути обережними, створюючи посилання на рядки. Якщо створюються посилання на рядок, а потім «змінюється» вихідний рядок, посилання продовжуватиме вказувати на оригінальний об’єкт замість нового об’єкта, який був створений під час зміни рядка. Можна використовувати звичайні рядкові літерали, коли потрібно вставляти escape-символи, що наявні в C#. Дослівні рядки використовують для зручності та кращої читабельності, коли текст рядка містить символи зворотної косої риски, наприклад, у шляхах до файлів. Оскільки дослівні рядки зберігають символи нового рядка як частину тексту рядка, їх використовують для ініціалізації багаторядкових рядків. Подвійні лапки використовують, щоб вставити лапки в дослівний рядок. У наступному прикладі,

на рисунку 2.18, показано деякі поширені способи використання дослівних рядків:

```
string filePath = @"C:\Users\scoleridge\Documents\";
//Output: C:\Users\scoleridge\Documents\

string text = @"My pensive SARA ! thy soft cheek reclined
    Thus on mine arm, most soothing sweet it is
    To sit beside our Cot,...";
/* Output:
My pensive SARA ! thy soft cheek reclined
   Thus on mine arm, most soothing sweet it is
   To sit beside our Cot,...
*/
```

Рисунок 2.18 – Verbatim string

В момент компіляції дослівні рядки перетворюються на звичайні рядки з однаковими вихідними послідовностями. Тому, коли розробник переглядає дослівний рядок у вікні спостереження debugger, він бачить escape-символи, додані компілятором, а не дослівну версію з вашого вихідного коду. Наприклад, дослівний рядок @"C:\files.txt" з'явиться у вікні годинника як "C:\\files.txt".

Рядок форматування — це рядок, вміст якого визначається. Рядки формату створюються шляхом вбудовування інтерпольованих виразів або заповнювачів усередині дужок у рядку. Все, що знаходиться в дужках ({...}), буде розв’язано до значення та виведене як відформатований рядок. Існують два методи створення рядків форматування: інтерполяція рядків і композитне форматування.

Доступні в C# 6.0 і новіших версіях, інтерпольовані рядки ідентифікуються спеціальним символом $ і включають інтерпольовані вирази в дужках. Інтерполяцію рядків використовують, щоб покращити читабельність і придатність для підтримки коду. Інтерполяція рядків досягає тих же результатів, що й метод String.Format, але покращує простоту використання та вбудовану чіткість. Починаючи з C# 10, інтерполяцію рядків можна використовувати для ініціалізації константного рядка, коли всі вирази, що використовуються для заповнювачів, також є постійними рядками.

String.Format використовує заповнювачі в дужках для створення рядка форматування. Цей приклад дає результат, подібний до методу інтерполяції рядків, використаного вище [13].

Підрядок — це група символів, що є в рядку. Використовується метод Substring, щоб створити новий рядок із частини вихідного рядка. Можна шукати одне або кілька повторень підрядку з використанням методу IndexOf. Метод Replace, замінює всі входження вказаного підрядка новим рядком. Як і метод Substring, Replace фактично дає новий рядок. Приклад на рисунку 2.3:

```
string s3 = "Visual C# Express";
System.Console.WriteLine(s3.Substring(7, 2));
// Output: "C#"

System.Console.WriteLine(s3.Replace("C#", "Basic"));
// Output: "Visual Basic Express"

// Index values are zero-based
int index = s3.IndexOf("C");
// index = 7
```

Рисунок 2.19 – Використання підрядків в C#

Ще використовують позначення масиву зі значенням індексу, щоб отримати доступ лише для читання до окремих символів. Якщо методи String не забезпечують функціональність, необхідну для зміни окремих символів у рядку, завжди можна використати об’єкт StringBuilder, щоб змінити окремі символи «на місці», а потім створити новий рядок для збереження результатів за допомогою методи StringBuilder. Клас StringBuilder створює рядковий буфер, який реалізує кращу продуктивність, якщо програма виконує багато маніпуляцій з рядками. Рядок StringBuilder також дозволяє перепризначати окремі символи, чого вбудований рядковий тип даних не підтримує. Код на рисунку 2.4, наприклад, змінює вміст рядка без створення нового рядка:

```
System.Text.StringBuilder sb = new System.Text.StringBuilder("Rat: the ideal pet");
sb[0] = 'C';
System.Console.WriteLine(sb.ToString());
System.Console.ReadLine();

//Outputs Cat: the ideal pet
```

Рисунок 2.20 – Заміна символів без створення нового рядку

Порожній рядок — це екземпляр об’єкта System.String, він має нуль символів. Порожні рядки часто використовуються в різних сценаріях програмування для представлення порожнього текстового поля. В C# є можливість викликати методи для порожніх рядків, оскільки вони є дійсними об’єктами System.String. На відміну від цього, нульовий рядок не посилається на екземпляр об’єкта System.String, і будь-яка спроба викликати метод для нульового рядка викликає виключення NullReferenceException. Однак можна використовувати нульові рядки в операціях порівняння та конкатенації з іншими рядками.

Оскільки тип String реалізує IEnumerable<T>, розробник може використовувати для рядків методи розширення, визначені в класі Enumerable. Щоб уникнути візуального безладу, ці методи виключені з IntelliSense для типу String, але вони всеодно доступні. Ще використовують LINQ запити для рядків.

3 ОГЛЯД ЗАСОБІВ РОЗРОБКИ

3.1 Visual Studio 2019 для програмування

Інтегроване середовище розробки (IDE) — це програма з великим переліком функцій, що підтримує багато аспектів створення програмного забезпечення. Visual Studio IDE — це унікальна багатофункціональна програмна система, яку застосовують для налагодження, редагування та створення коду, а потім для публікації програми. Крім звичайного редактора і налагоджувача, які мають всі IDE, Visual Studio також містить інструменти завершення коду, графічні дизайнери, компілятори та багато інших функцій для покращення всього циклу розробки бібліотек та програмного забезпечення [14].

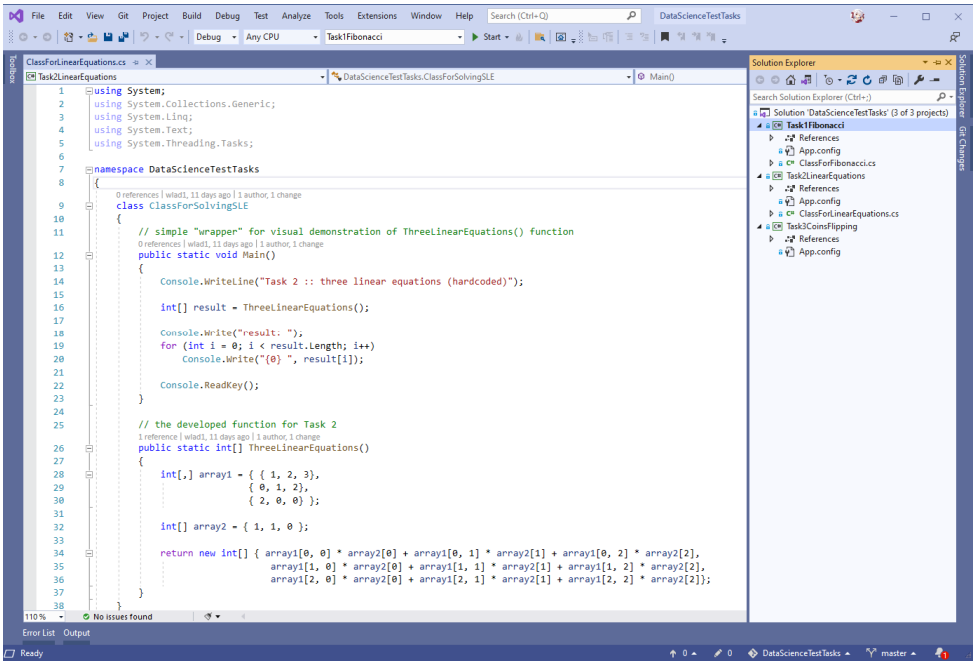


Рисунок 3.1 – Інструменти розробки VS 2019

Рисунок 3.1 зображує Visual Studio з активним проектом, який показує ключові вікна та їх функціональність:

- у провіднику рішень у верхньому правому куті можна переглядати, переміщатися і управляти файлами коду. Solution Explorer допомагає впорядкувати код, зібравши файли проекти та рішення;

- центральне вікно редактора, проводиться більша частину часу, відображає вміст файлу. У вікні редактора можна редагувати код або створювати візуальний інтерфейс;
- у розділі «Зміни Git», унизу праворуч, можна відстежувати елементи проекту та ділитися кодом за допомогою Git і GitHub.

В Visual Studio дуже багато популярних корисних функцій, які покращують продуктивність в час розробки програмних застосунків.

Підкреслення — це червоні хвилясті лінії, що виділяють помилки або можливі проблеми у коді, вони допоможуть негайно вирішити проблеми, не очікуючи виявлення проблем під час збірки або виконання.

Очищення коду – натиснувши кнопку, можна відформатувати свій код і застосувати будь-які виправлення коду, запропоновані налаштуваннями стилю коду, умовами .editorconfig та аналізаторами Roslyn, ця функція доступна лише для коду C#, що допоможе вирішити проблеми у коді.

Рефакторинг організовує такі операції, як розумне переіменування змінних, виділення одного або кількох рядків програми в нову функцію та перестановка порядку вхідних даних методу.

IntelliSense — система функціоналу, що відображає інформацію про код прямо у редакторі і, деколи, записує невеликі фрагменти коду, це працює як базова документація в редакторі, тому не доведеться шукати інформацію про типи в іншому місці.

Пошук у Visual Studio допомагає знайти меню, параметри та властивості Visual Studio, які іноді можуть здаватися надто значними, пошук Visual Studio або Ctrl+Q — найшвидший шлях знайти функції та код IDE в одному місці.

Live Share дає змогу спільно редагувати та налагоджувати з іншими розробниками в реальному часі, незалежно від типу програми чи мови програмування, миттєво та безпечно ділитися проектом, екземплярами терміналів, ділитися сеансом налагодження, веб-програмами localhost, голосовими дзвінками.

У вікні «Call Hierarchy» відображаються методи, які викликають обраний метод, така інформація є корисною, коли роздумується про редагування чи вилученння методу, чи коли користувач намагається відстежити поломку.

CodeLens знаходить посилання на код, пов'язані помилки, робочі елементи, зміни коду, огляди та різні тести, не виходячи з редактора.

Функція «Перейти до визначення» перенесе користувача безпосередньо до розташування визначення функції або типу в лістингу програми.

У вікні Peek Definition вказується визначення функції і типу без відкриття окремого файлу, що дає змогу розібратись з частиною іншого коду.

Налагодження коду ще одна корисна функція Visual Studio. Коли розробник пише код, він повинен запустити його та перевірити на наявність помилок. Система налагодження коду Visual Studio дає змогу покроковувати код по одному оператору та перевіряти змінні під час роботи. Ще є фунціонал для встановлення точок зупинки, що зупиняють код у визначеному рядку, і потім спостерігати, як змінюється значення змінних під час виконання коду. На рисунку 3.2 встановили точку зупинки, для перегляду значення змінної імені користувача. У стовпці з номерами при цьому з'являється червоне коло, і лінія виділяється:

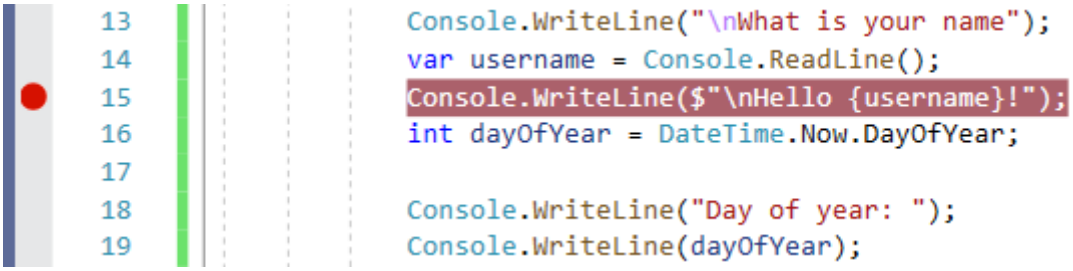


Рисунок 3.2 – Налагодження коду в Visual Studio

3.2 Visual Studio Code 2021 для тестування

VSCode — це блискавичний редактор, його пропонує компанія Microsoft. Це комплексний інструмент, який поєднує простий у використанні текстовий редактор з розширеною IDE, що дозволяє легко редагувати, створювати та налагоджувати тестові проекти з однієї інформаційної панелі. Інструмент підтримує платформи Mac, Windows, Linux. Підтримка JavaScript, Node.js і TypeScript є вбудованою, але він пропонує розширення для таких мов як Java,

Python, PHP, C++, C#, GO тощо. Він також підтримує інструменти виконання, такі як Unity та .NET.

VSCode не слід плутати з Microsoft Visual Studio. Середовище Visual Studio — це продукт Microsoft, який існує вже 20 років і працює на платформах Windows і MAC. Це не відкритий код. Однак VSCode заснований на Electron Framework. Його було створено GitHub для розробки додатків GUI за допомогою Node.js. Раніше він був відомий як Atom Shell. Microsoft перебрала цей продукт і випустила його з відкритим кодом у 2015 році. VSCode — це легкий текстовий редактор [15].

VSCode він пропонує розширення, які перетворюють його в потужну IDE, що дозволяє працювати з улюбленою мовою програмування. Однак він не є ресурсомістким і не вимагає складних процедур встановлення. Використовуючи цей єдиний редактор, запускають будь-які скрипти програмування.

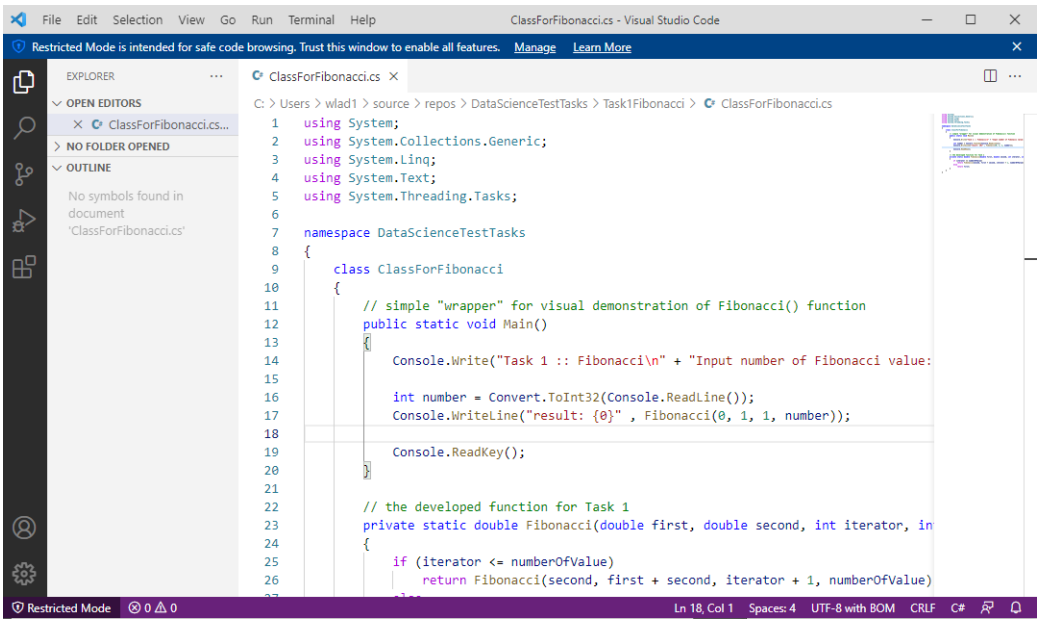


Рисунок 3.3 – Робота з C# в VS Code

Він підтримує всі основні платформи, такі як Windows, Linux, MAC тощо. Він не тільки легкий, але й працює набагато швидше, ніж інші IDE. Додавши розширення, можна користуватися багатофункціональним текстовим редактором. Крім того, він простий в установці та використанні. Встановлення функції — це простий клік. Це також безкоштовно. Інструмент швидко впроваджується в

інновації. Він пропонує корисні плагіни та розширення та підтримку налагодження. VSCode має активну спільноту, яка готова запропонувати допомогу, пов’язану з інструментом. Більше того, спеціальні команди Microsoft постійно працюють над цим інструментом, щоб покращити його. Вони випускають нову версію майже щомісяця, і продукт стабільний.

Selenium — це бібліотека автоматизації тестувань з відкритим вихідним кодом. Вона підтримує багато браузерів, таких як Chrome, Firefox, Edge тощо, і багато мов – python, java, C#, javascript тощо. Selenium можна використовувати із C# у коді VS.

Щоб запустити свої тести, користувач може використовувати налагоджувач або команду з терміналу. Щоб запустити тести, достатньо виконати наступну команду: dotnet run Якщо натиснути наведену вище команду, VS Code почне виконувати тести та запустить поточний проект для виконання кроків тестування.

Якщо помилки немає, то після виконання всіх кроків він закриє проект і надрукує повідомлення про успіх. І якщо він виявить будь-яку помилку, він покаже всі інформацію про помилку з номером рядка, де сталася помилка. Приклад на рисунку 3.4. Тут наведено приклади запусків тестувань над проектом Firefox, щоб показати варіанти пройдених та не пройдених тестів. Перший раз – без змін та повідомлень (правильна програма):

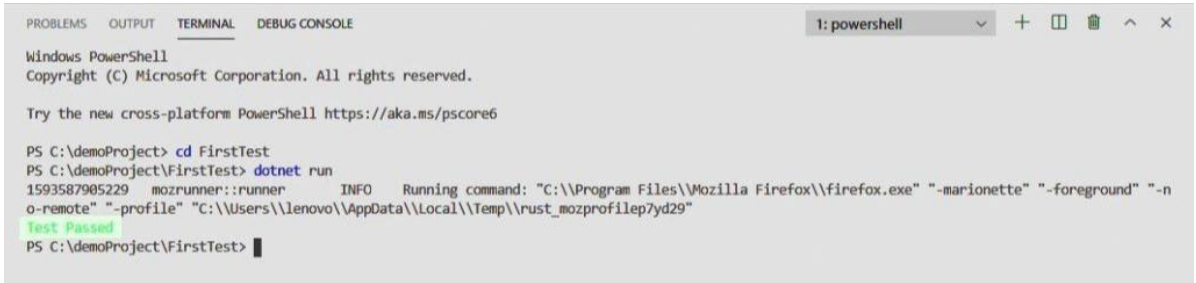


Рисунок 3.4 – Успішне завершення тестів в VS Code

А другий раз відбулося попереднє видаленням ідентифікатора локатора, Дана помилка була зроблена навмисне щоб показати повідомлення про помилку:

Рисунок 3.5 – Вивід помилки під час тестувань в VS Code

4 ОПИС ПРОГРАМНОЇ РЕАЛІЗАЦІЇ

4.1 Опис функціонування системи

Бібліотека для Markdown на C# призначена для перетворення тексту з використанням синтаксису легкої мови розмітки в форматований текст. Для того щоб організувати перетворення звичайного тексту в форматований, потрібен парсер, що буде аналізувати текст. Парсер або синтаксичний аналізатор — це компонент компілятора чи навіть інтерпретатора, що розбиває текст на менші елементи для легкого перекладу іншою мовою або іншої обробки. Синтаксичний аналізатор приймає вхідні дані у вигляді послідовності маркерів, інтерактивних команд або програмних інструкцій і розбиває їх на частини, які можуть використовуватися іншими компонентами в програмуванні.

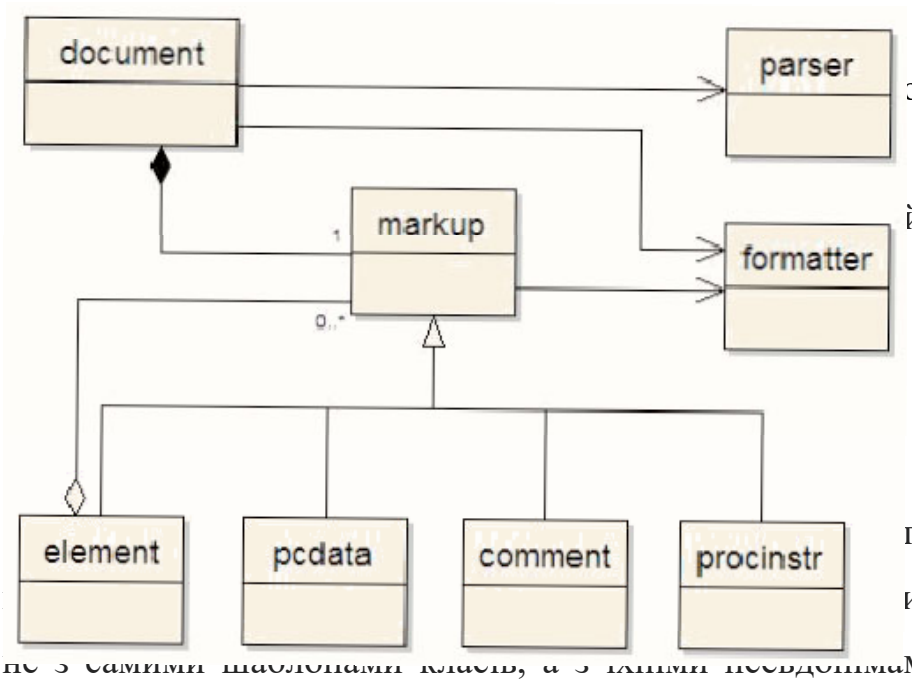
Синтаксичний аналізатор зазвичай перевіряє всі надані дані, щоб переконатися, що їх достатньо для побудови структури даних у формі дерева розбору чи навіть абстрактного дерева синтаксису [16].

В даному випадку парсер буде використовуватись для пошуку синтаксису мови Markdown та подальшого перетворення тексту в його очікуваний вигляд. Оригінальний парсер був написаний на Perl. Основні функції включають блочні елементи аналізу (такі як абзаци, розриви рядків, заголовки, блокові лапки, списки, блоки коду та горизонтальні правила) та елементи span (посилання, підкреслення, фрагменти коду та зображення). З тих пір мова не була розширена її творцем Джоном Грубером, тому з’явилася низка доповнень і реалізацій з різними синтаксичними аналізаторами, які додають підтримку різних реалізацій, як вони вважають за потрібне, або інтерпретують, як аналізуються певні елементи.

Парсер бібліотеки Markdown на C# вмістить в собі функції, що розпізнаватимуть оригінальний варіант синтаксису мови. Також буде підтримка всіх найбільш поширених варіацій мови розмітки, для того щоб бібліотека могла підтримуватись в майбутньому різними програмами та розробниками.

Для найбільш ефективного розбору тексту використаний метод часткового аналізу тексту FASTUS. У ньому закладено основні ідеї та п’ять етапів обробки мови загальні концепції, які добре стосуються даної проблеми [17].

Однак система FASTUS здебільшого використовувалася для вирішення завдань, які відрізнялися від згадуваних. Необхідно адаптувати ідеї до проблеми розбору слів, пошуку спеціальних символів та перетворення тексту у форматований.



ɔ парсера [18]

й бібліотеці:

п замість класу, оскільки

ичай обробка виконується

не з самими шаблонами класів, а з їхніми псевдонімами типів. На рисунку 4.1 показана діаграма класів для псевдонімів типів бібліотеки Markdown. Важливо зауважити, що розмітка є базовим типом для чотирьох типів, які будуть формувати вміст у текстовому. Сам елемент може містити нуль або більше об’єктів розмітки. Цей тип дизайну поширений в архітектурі класів і називається композитним шаблоном. Тип документа містить один об’єкт розмітки. Цей об’єкт розмітки буде використовуватися як корінь документа. І документ, і розмітка пов’язані з форматувальником, а документ — із синтаксичним аналізатором.

Хоча ітератори розмітки та ітератори-нащадки не показані на діаграмі, вони оголошуються в розмітці, тому ці ітератори під час оголошення потребують кваліфікації за допомогою розмітки, як у markup::iterator або markup::descendant_iterator. Хоча ці ітератори мають відношення лише до елементів, оскільки лише елементи можуть містити дочірні об’єкти розмітки,

ітератори оголошуються в розмітці, оскільки всі відкриті операції елемента також оголошуються в розмітці. Це дасть можливість появлятись таким випадкам, коли у є вказівник або посилання на об’єкт розмітки, що є або не є об’єктом елемента. Оскільки елемент є таким важливим і часто використовуваним об’єктом, більш вигідно мати можливість викликати операції елемента над об’єктом розмітки, щоб вимагати, щоб він спочатку був понижений до елемента. Будь-яка операція, пов’язана з елементом, що викликається на об’єкті розмітки, який не є елементом, виконуватиме деяку нульову поведінку. Наприклад, виклик markup::begin() для посилання на розмітку або вказівника, який насправді є коментарем, поверне markup::end(). Типи форматування та синтаксичний аналізатор були додані до бібліотеки Markdown замість того, щоб додавати операції write() чи parse(), щоб забезпечити більшу гнучкість аналізу та написання форматованих Markdown документів. І синтаксичний аналізатор, і формататор дозволяють встановлювати параметри, які визначать спосіб аналізу або форматування документів/об’єктів розмітки під час запису.

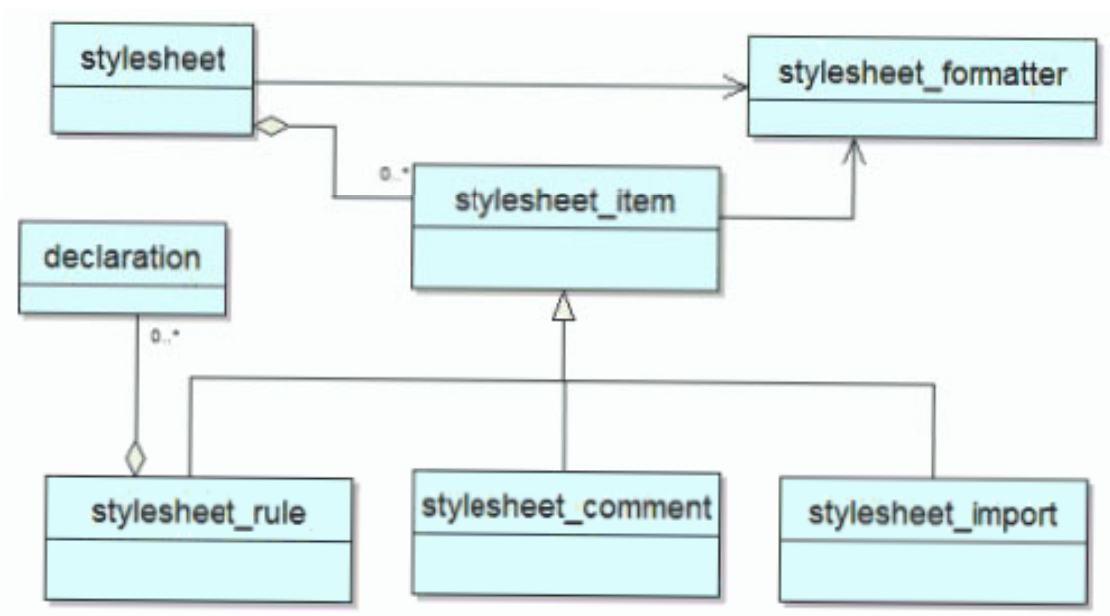


Рисунок 4.2 – UML діаграма класів синтаксису Markdown [18]

Типи таблиць стилів форматувань представлені на діаграмі класів на рисунку 4.2. На цій діаграмі варто зауважити, що stylesheet_item є базовим типом для stylesheet_rule, stylesheet_comment та stylesheet_import. Також треба підмітити,

що stylesheet_rule може містити нуль або більше декларацій. Також, як показано на схемі, таблиця стилів може містити нуль або більше об’єктів stylesheet_item. Stylesheet_formatter був створений, щоб забезпечити детальний контроль над форматом таблиці стилів, який записується у файл або потік попередній перегляд.

5 ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Забезпечення якості програмного забезпечення (SQA) – це процес, який гарантує, що всі процеси, методи, дії та робочі елементи контролюються та відповідають визначеним стандартам. Ці визначені стандарти можуть бути одним або комбінацією будь-яких, таких як ISO 9000, модель CMMI, ISO15504 тощо [20].

SQA включає всі процеси розробки ПЗ, починаючи від визначення вимог до кодування і закінчуючи випуском. Його головна мета – забезпечити якість ПЗ.

Рисунок 5.1 – Типовий цикл тестування програмних проектів [20]

Для даної роботи було застосовано функціональне тестування та white box testing, враховуючи відсутність інтерфейсу бібліотеки та одноразове виробництво.

Функціональне тестування — це етапи забезпечення якості і тип тестування чорного ящика, який базує свої тестові випадки на специфікаціях програмного компонента, що тестується. Функції перевіряються шляхом подачі вхідних даних та перевірки результатів, а внутрішня структура програми рідко розглядається Функціональне тестування проводиться для оцінки відповідності системи або

компонента визначеним функціональним вимогам. Функціональне тестування зазвичай описує те, що робить система.

Оскільки функціональне тестування є видом тестування чорного ящика, функціональність ПЗ можна перевірити, не знаючи внутрішньої роботи програми. Це означає, що тестувальникам не потрібно знати мови програмування або те, як програмне забезпечення було реалізовано. Таке може призвести до зменшення упередженості розробника (або упередженості підтвердження) у тестуванні, оскільки тестувальник не брав участі в розробці програми.

Функціональне тестування не означає, що тестується функцію (метод) модуля або класу. Функціональне тестування перевіряє частину функціональності всієї системи. Функціональне тестування відрізняється від тестування системи тим, що функціональне тестування «перевіряє програму, звіряючи її з ... проектним документом (документами) або специфікацією (специфікаціями)», тоді як тестування системи «перевіряє програму, перевіряючи її з опублікованими вимогами користувача чи системи».

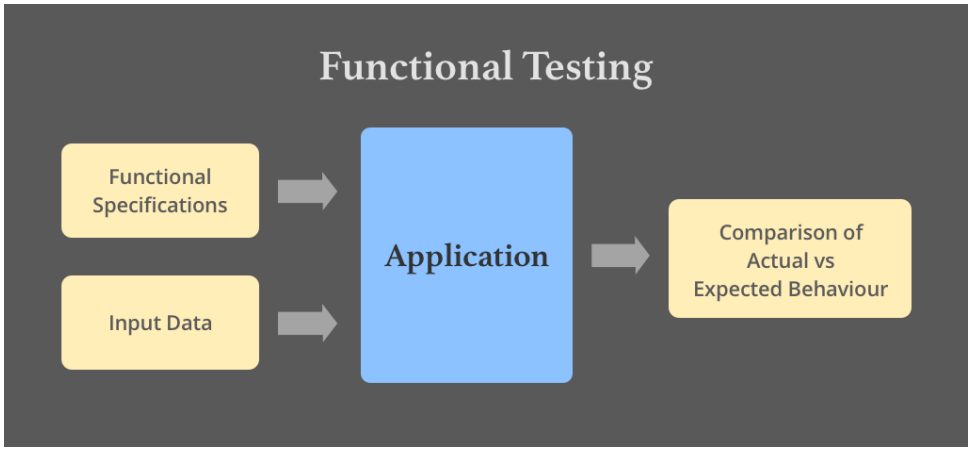


Рисунок 5.2 – Алгоритм функціонального тестування ПЗ [21]

Тестування чорного ящика означає проведення перевірки якості без знання архітектури або доступу до коду продукту, який тестується. Фахівцю з контролю якості ця інформація не потрібна. Замість бекенд-технологій вони орієнтуються на вимоги, описані в технічній документації. Отже, інженери QA перевіряють лише кінцевий продукт, а не базові технології.

Важливо перевірити сценарії, коли система може зламатися. Таким чином дізнаються, чи реагує програмний продукт на невдалі сценарії, як очікувалося. Недійсні введення можуть спричинити лише незручності (сповіщення про помилку не з'являється на екрані) або призвести до збою програми. Для всіх цих сценаріїв інженеру QA не потрібно звертатися до архітектури ПЗ або вивчати код. Можна побачити, чи функції працюють належним чином без нього. Таким чином, код залишається заблокованим у коробці, поки виконуються тести на поверхні.

Тест чорного ящика використовувався для оцінки отриманих результатів, порівняння їх з результатами із оригінальної бібліотеки Markdown.pl та результатами із інших бібліотек.

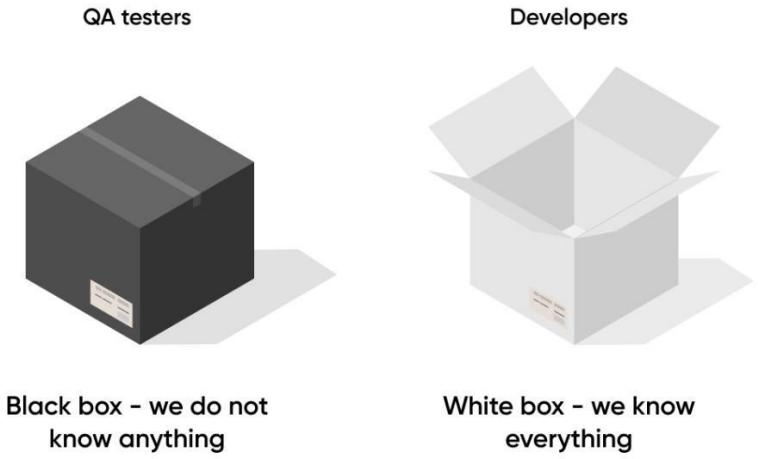


Рисунок 5.3 – Візуальне представлення BB і WB тестування [22]

Тестування білого ящика, навпаки, означає, що інженер QA знає архітектуру програмного забезпечення і ще розуміє код, який забезпечує всі ті функції, які потребують тестування. У цьому випадку спеціаліст з контролю якості може використовувати код як додаткове джерело інформації в момент тестування. Основна логіка допомагає спеціалісту з контролю якості відстежувати пов'язані випадки. Тести білого ящика проводять переважно розробники.

Даний метод передбачає тестування програми на рівні вихідного коду. Ці тестові приклади отримані за допомогою використання вищезгаданих методів проектування: тестування потоку керування, тестування потоку даних, тестування гілок, тестування шляху, покриття операторів і покриття рішень, а також

модифіковане покриття умов/рішень. Тестування білого ящика — використання таких методів як керівних принципів для створення середовища без помилок шляхом вивчення всього коду. Ці методи тестування «білих ящиків» є будівельними блоками тестування «білих ящиків», суть яких полягає в ретельному тестуванні програми на рівні вихідного коду для зменшення прихованих помилок у подальшому. Що стосується тестування білого ящика, то це ефективне рішення, яке дає технічну можливість знаходити та виправляти рядки коду з дефектами. Тестові випадки зосереджені на технічних деталях та особливостях коду.

Для бібліотеки Markdown на C# під час всього етапу розробки архітектури, логіки та функціоналу проводилось тестування білого ящика.

У розробці ПЗ test-case або тестовий приклад — це визначеннння вхідних даних, процедури тестування умов виконання та очікуваних результатів, які визначають один тест. Група test-case може бути створена для отримання бажаного охоплення ПЗ, що тестується. Тестові випадки дозволяють виконувати одні й ті ж тести багаторазово для всіх версій програми, що дозволяє проводити ефективне та послідовне регресійне тестування.

Під час розробки та тестування проект було неодноразово. Ось деякі test-case, що були пройдені під час тестування бібліотеки:

- перевірка коректності розпізнавання заголовків всіх рівнів;
- розмітка параграфів, багаторівневих вкладень параграфів;
- створення списків, нумерованих та ненумерованих, різних варіацій та поєднань цих списків разом з параграфами;
- виділення тексту в різних варіаціях: жирним, курсивом, поєднанням цих варіантів, використання зірочок, підкреслень та їх поєднань для виділення тексту в середині слів.

Тестування білого ящика та чорного ящика було проведено успішно. Всі знайдені помилки та невідповідності до поведінки мови розмітки Markdown було знайдено та успішно налагоджено. Пробне використання бібліотеки пройдено

вдало. Це означає, що бібліотека працює коректно, відповідає всім поставленим вимогам та готова до застосування в зв’язці з іншими проектами.

6 ОХОРОНА ПРАЦІ ТА БЕЗПЕКА В НАДЗВИЧАЙНИХ СИТУАЦІЯХ

ВИСНОВОК

Магістерська робота на тему аналіз та проектування C# бібліотеки для генерації текстів на основі мови розмітки даних (Markdown) успішно завершено. Під час написання диплому було проведено ґрунтовну роботу з аналізу предметної області, постановки завдання, огляду засобів розробки, опису програмної реалізації та тестування програмного забезпечення на коректність роботи.

В розділі предметної області розглянуто особливості мови розмітки даних, в першу чергу семантику та форматування. Було встановлено, що зараз існує кілька різновидів Markdown, які часто мають незначні відмінності між їх реалізацією та синтаксисом і в результаті канонічної реалізації бібліотеки не існує. Цей факт ліг в основу дипломної роботи. Розглянуто відмінності використання Markdown від використання редактора WYSIWYG. Встановлено, що перш ніж обробити текстовий документ з новим форматуванням Markdown можна виконати над ним масу попередніх обробок за допомогою препроцесора та інструменту Pandoc. Проведено порівняльний аналіз Markdown із конкурентними мова розмітки, що можуть використовуватись в одній сфері (HTML, LaTeX, reStructuredText) який показав однозначну перевагу Markdown. Після цього проведено пошук та аналіз існуючих текстових редакторів, що мають вбудоване застосування бібліотек для Markdown. Зокрема, проаналізовано роботу: універсального інструменту Pandoc, для створення документацій та конвертування файлів; програми GitBook для створення спільної документації; MkDocs – швидкого генератора статичних сайтів, який призначений для створення проектної документації та сервісу GitHub Pages, що дає можливість розміщувати такі сайти. Всі ці інструменти є сучасними проектами, шо активно розвиваються та успішно реалізують можливості Markdown. Наступним етапом став аналіз та дослідження всіх сфер застосування Markdown. Встановлено, що таку мову розмітки тексту можна успішно використовувати для: створення статичних сайтів; створення документів і в

подальшому перетворювати їх в інші формати; швидкого запису форматованих нотаток; створення презентацій та слайд-шоу; форматування електронних листів прямо в браузері з допомогою розширення MarkdwonHere.

Під час постановки задач розробки було ґрунтовно досліджено та систематизовано синтаксис Markdown для того, щоб результуюча бібліотека відповідала поширеним вимогам мови розмітки та могла успішно використовуватись в інших проектах. Наступним кроком став аналіз можливостей роботи мови програмування C# із текстовими об’єктами. З’ясовано що, операції з рядками в C# дуже оптимізовані і в більшості випадків не впливають істотно на продуктивність, а потужний клас Object.String має всі необхідні методи та інтерфейси для реалізації парсера та модифікатора тексту.

Огляд засобів розробки показав, що сучасні IDE мають всі необхідні функції та інструменти для якісного написання та аналізу коду. Зокрема використання комплексу VS Code та бібліотеки Selenium дає багато можливостей для тестування бібліотеки та її варіантів застосування.

Описано програмну реалізацію бібліотеки Markdown на C# та всі її етапи: опис функціонування системи, архітектуру бібліотеки та особливості роботи алгоритму. Для цього було створено модуль парсера та форматування тексту, а також типи представлення текстових файлів, таблиць стилів або ж синтаксису Markdown і типи ітераторів для перебору тексту. Їх структуру зображено у вигляді двох UML діаграм класів. В основі архітектури було закладено алгоритм часткового аналізу, який підключає модель мови до адаптованої системи типу Information Extraction (IE) у стилі FASTUS, що допомогло ефективно аналізувати текст та форматувати його.

Останнім етапом стало тестування бібліотеки Markdown, яке було успішно проведено. Тестування білого ящика та чорного ящика було проведено успішно. Всі знайдені помилки та невідповідності до поведінки мови розмітки Markdown було знайдено та успішно налагоджено. Пробне використання бібліотеки пройдено вдало. Це означає, що бібліотека працює коректно, відповідає всім поставленим вимогам та готова до застосування в зв’язці з іншими проектами.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Mailund T. Intrducing Markdown and Pandoc. Berkeley, CA: Apress, 2019.– 1 с.
2. Drupal: Usage statistics for Markdown [Електронний ресурс] – Режим доступу до ресурсу: https://www.drupal.org/project/usage/markdown
3. Cone M. The Markdown Guide. Independently published, 2020. – 10 с.
4. Introducing Markdown and Pandoc: Using Markup Language and Document Converter. O'Reilly Online Learning. [Електронний ресурс] – Режим доступу до ресурсу: https://www.oreilly.com/library/view/introducing-markdown-and/97814 84251492/html/486315_1_En_10_Chapter.xhtml
5. Kottwitz S. LaTeX beginner's guide: Create high-quality and professional-looking texts, articles, and books for business and science using LaTeX. Olton, Birmingham: Packt Pub., 2011. – 100 с.
6. Start exploring - GitBook Documentation. What is GitBook - GitBook Documentation.[Електронний ресурс] – Режим доступу до ресурсу: https://docs.gitbook.com/getting-started/start-exploring
7. Writing Your Docs - MkDocs. MkDocs. [Електронний ресурс] – Режим доступу до ресурсу: https://www.mkdocs.org/user-guide/writing-your-docs/
8. GitHub Pages: websites for you and your projects [Електронний ресурс] – Режим доступу до ресурсу: https://pages.github.com/getting-started
9. Dyer W. Using Markdown: A Short Instruction Guide Kindle Edition. Independently published, 2018. – 34 с.
10. Hogan B. P. Build Websites with Hugo: Fast Web Development with Markdown. The Pragmatic Programmers, LLC, 2020 – 5 с.
11. Quickstart. Jekyll • Simple, blog-aware, static sites. [Електронний ресурс] – Режим доступу до ресурсу: https://jekyllrb.com/docs/
12. String Class (System). Developer tools, technical documentation and coding examples | Microsoft Docs. [Електронний ресурс] – Режим доступу до ресурсу: https://docs.microsoft.com/en-us/dotnet/api/system.string?view=net-6.0

13. String.Format Method (System). Developer tools, technical documentation and examples | Microsoft Docs. [Електронний ресурс] – Режим доступу до ресурсу: https://docs.microsoft.com/en-us/dotnet/api/system.string.format?view=net-6.0
14. Welcome to the Visual Studio IDE | C# [Електронний ресурс] – Режим доступу до ресурсу: https://docs.microsoft.com/en-us/visualstudio/get-started/csharp/ visual-studio-ide?view=vs-2022
15. Overview for C# developers - Visual Studio (Windows). Developer tools, technical documentation and coding examples | Microsoft Docs. [Електронний ресурс] – Режим доступу до ресурсу: https://code.visualstudio.com/docs/supporting/FAQ
16. Thomas L. Beginning syntax. Oxford, UK : Blackwell, 1993. – 10 с.
17. FASTUS: Extracting Information from NL Texts. Artificial Intelligence Center @ SRI. [Електронний ресурс] – Режим доступу до ресурсу: http//www.ai.sri.com/natural-language/projects/fastus-schabes.html
18. XportPro, Design. XportPro, xhtml report generator and parser, for C++. [Електронний ресурс] – Режим доступу до ресурсу: http://www.xportpro.com/xport/design.php
19. Maciuszek D. Automating the Extraction of User Model Information from Consultation Dialogues: Final Thesis. Linkoping, 2001.. – 54 с.
20. What is Software Quality Assurance (SQA): A Guide for Beginners. Software Testing Help - Free Software Testing & Development Courses. [Електронний ресурс] – Режим доступу до ресурсу: https://www.softwaretestinghelp.com/software-quality-assurance/
21. What is Functional Testing? Definition, Test Cases & Examples. Insights on Latest Technologies - Simform Blog [Електронний ресурс] – Режим доступу до ресурсу: https://www.simform.com/blog/functional-testing/
22. Careerist. A Guide To: White Box, Black Box, and Gray Box testing. Get a High-Paying Job | Careerist. [Електронний ресурс] – Режим доступу до ресурсу:

https://www.careerist.com/insights/a-guide-to-white-box-black-box-and-gray-box-testing

ДОДАТКИ

ДОДАТОК А

Лістинг бібліотеки на C# для мови розмітки Markdown

```
using System;
using System.IO;
using System.Reflection;
using Markdown.Extensions.SelfPipeline;
using Markdown.Helpers;
using Markdown.Parsers;
using Markdown.Renderers;
using Markdown.Renderers.Normalize;
using Markdown.Syntax;
namespace Markdown
{
    /// <summary>
    /// Provides methods for parsing a Markdown string to a syntax
tree and converting it to other formats.
    /// </summary>
    public static partial class Markdown
    {
        public     static     readonly     string     Version     =
((AssemblyFileVersionAttribute)
typeof(Markdown).Assembly.GetCustomAttributes(typeof(AssemblyFileVer
sionAttribute), false)[0]).Version;

        internal static readonly MarkdownPipeline DefaultPipeline =
new MarkdownPipelineBuilder().Build();
        private        static        readonly        MarkdownPipeline
_defaultTrackTriviaPipeline                    =                    new
MarkdownPipelineBuilder().EnableTrackTrivia().Build();

        private                 static                MarkdownPipeline
GetPipeline(MarkdownPipeline? pipeline, string markdown)
        {
            if (pipeline is null)
            {
                return DefaultPipeline;
            }

            var                     selfPipeline                     =
pipeline.Extensions.Find<SelfPipelineExtension>();
            if (selfPipeline is not null)
            {
                return
selfPipeline.CreatePipelineFromInput(markdown);
            }
            return pipeline;
        }
        /// <summary>
        ///  Normalizes the specified markdown to a normalized
markdown text.
```

```
        /// </summary>
        /// <param name="markdown">The markdown.</param>
        /// <param name="options">The normalize options</param>
        /// <param name="pipeline">The pipeline.</param>
        /// <param name="context">A parser context used for the parsing.</param>
        /// <returns>A normalized markdown text.</returns>
        public static string Normalize(string markdown, NormalizeOptions? options = null, MarkdownPipeline? pipeline = null, MarkdownParserContext? context = null)
        {
            var writer = new StringWriter();
            Normalize(markdown, writer, options, pipeline, context);
            return writer.ToString();
        }
        /// <summary>
        /// Normalizes the specified markdown to a normalized markdown text.
        /// </summary>
        /// <param name="markdown">The markdown.</param>
        /// <param name="writer">The destination <see cref="TextWriter"/> that will receive the result of the conversion.</param>
        /// <param name="options">The normalize options</param>
        /// <param name="pipeline">The pipeline.</param>
        /// <param name="context">A parser context used for the parsing.</param>
        /// <returns>A normalized markdown text.</returns>
        public static MarkdownDocument Normalize(string markdown, TextWriter writer, NormalizeOptions? options = null, MarkdownPipeline? pipeline = null, MarkdownParserContext? context = null)
        {
            if (markdown is null) ThrowHelper.ArgumentNullException_markdown();
            pipeline = GetPipeline(pipeline, markdown);
            var document = MarkdownParser.Parse(markdown, pipeline, context);
            var renderer = new NormalizeRenderer(writer, options);
            pipeline.Setup(renderer);
            renderer.Render(document);
            writer.Flush();
            return document;
        }
        /// <summary>
        /// Converts a Markdown string to HTML.
        /// </summary>
        /// <param name="markdown">A Markdown text.</param>
        /// <param name="pipeline">The pipeline used for the conversion.</param>
        /// <param name="context">A parser context used for the parsing.</param>
        /// <returns>The result of the conversion</returns>
```

```
        /// <exception cref="ArgumentNullException">if markdown
variable is null</exception>
        public static string ToHtml(string markdown,
MarkdownPipeline? pipeline = null, MarkdownParserContext? context =
null)
        {
            if (markdown is null)
ThrowHelper.ArgumentNullException_markdown();

            pipeline = GetPipeline(pipeline, markdown);

            var document = MarkdownParser.Parse(markdown, pipeline,
context);
            return ToHtml(document, pipeline);
        }
        /// <summary>
        /// Converts a Markdown document to HTML.
        /// </summary>
        /// <param name="document">A Markdown document.</param>
        /// <param name="pipeline">The pipeline used for the
conversion.</param>
        /// <returns>The result of the conversion</returns>
        /// <exception cref="ArgumentNullException">if markdown
document variable is null</exception>
        public static string ToHtml(this MarkdownDocument document,
MarkdownPipeline? pipeline = null)
        {
            if (document is null)
ThrowHelper.ArgumentNullException(nameof(document));
            pipeline ??= DefaultPipeline;
            using var rentedRenderer = pipeline.RentHtmlRenderer();
            HtmlRenderer renderer = rentedRenderer.Instance;
            renderer.Render(document);
            renderer.Writer.Flush();
            return renderer.Writer.ToString() ?? string.Empty;
        }
        /// <summary>
        /// Converts a Markdown string to HTML and output to the
specified writer.
        /// </summary>
        /// <param name="markdown">A Markdown text.</param>
        /// <param name="writer">The destination <see
cref="TextWriter"/> that will receive the result of the
conversion.</param>
        /// <param name="pipeline">The pipeline used for the
conversion.</param>
        /// <param name="context">A parser context used for the
parsing.</param>
        /// <returns>The Markdown document that has been
parsed</returns>
        /// <exception cref="ArgumentNullException">if reader or
writer variable are null</exception>
```

```
        public static MarkdownDocument ToHtml(string markdown, TextWriter writer, MarkdownPipeline? pipeline = null, MarkdownParserContext? context = null)
        {
            if (markdown is null) ThrowHelper.ArgumentNullException_markdown();
            if (writer is null) ThrowHelper.ArgumentNullException_writer();
            pipeline = GetPipeline(pipeline, markdown);
            var document = MarkdownParser.Parse(markdown, pipeline, context);
            using var rentedRenderer = pipeline.RentHtmlRenderer(writer);
            HtmlRenderer renderer = rentedRenderer.Instance;
            renderer.Render(document);
            writer.Flush();
            return document;
        }
        /// <summary>
        /// Converts a Markdown string using a custom <see cref="IMarkdownRenderer"/>.
        /// </summary>
        /// <param name="markdown">A Markdown text.</param>
        /// <param name="renderer">The renderer to convert Markdown to.</param>
        /// <param name="pipeline">The pipeline used for the conversion.</param>
        /// <param name="context">A parser context used for the parsing.</param>
        /// <exception cref="ArgumentNullException">if markdown or writer variable are null</exception>
        public static object Convert(string markdown, IMarkdownRenderer renderer, MarkdownPipeline? pipeline = null, MarkdownParserContext? context = null)
        {
            if (markdown is null) ThrowHelper.ArgumentNullException_markdown();
            if (renderer is null) ThrowHelper.ArgumentNullException(nameof(renderer));
            pipeline = GetPipeline(pipeline, markdown);
            var document = MarkdownParser.Parse(markdown, pipeline, context);
            pipeline.Setup(renderer);
            return renderer.Render(document);
        }
        /// <summary>
        /// Parses the specified markdown into an AST <see cref="MarkdownDocument"/>
        /// </summary>
        /// <param name="markdown">The markdown text.</param>
        /// <param name="trackTrivia">Whether to parse trivia such as whitespace, extra heading characters and unescaped string values.</param>
```

```
        /// <returns>An AST Markdown document</returns>
        ///   <exception   cref="ArgumentNullException">if   markdown
variable is null</exception>
        public static MarkdownDocument Parse(string markdown, bool
trackTrivia = false)
        {
            if             (markdown             is             null)
ThrowHelper.ArgumentNullException_markdown();
            MarkdownPipeline?    pipeline    =    trackTrivia    ?
_defaultTrackTriviaPipeline : null;
            return Parse(markdown, pipeline);
        }
        /// <summary>
        ///  Parses  the  specified  markdown  into  an  AST  <see
cref="MarkdownDocument"/>
        /// </summary>
        /// <param name="markdown">The markdown text.</param>
        ///  <param  name="pipeline">The  pipeline  used  for  the
parsing.</param>
        /// <param name="context">A parser context used for the
parsing.</param>
        /// <returns>An AST Markdown document</returns>
        ///   <exception   cref="ArgumentNullException">if   markdown
variable is null</exception>
        public  static  MarkdownDocument  Parse(string  markdown,
MarkdownPipeline? pipeline, MarkdownParserContext? context = null)
        {
            if             (markdown             is             null)
ThrowHelper.ArgumentNullException_markdown();
            pipeline = GetPipeline(pipeline, markdown);
            return      MarkdownParser.Parse(markdown,      pipeline,
context);
        }
        /// <summary>
        /// Converts a Markdown string to Plain text and output to
the specified writer.
        /// </summary>
        /// <param name="markdown">A Markdown text.</param>
        ///     <param     name="writer">The     destination     <see
cref="TextWriter"/>  that  will  receive  the  result  of  the
conversion.</param>
        ///  <param  name="pipeline">The  pipeline  used  for  the
conversion.</param>
        /// <param name="context">A parser context used for the
parsing.</param>
        ///   <returns>The   Markdown   document   that   has   been
parsed</returns>
        /// <exception cref="ArgumentNullException">if reader or
writer variable are null</exception>
        public static MarkdownDocument ToPlainText(string markdown,
TextWriter    writer,    MarkdownPipeline?    pipeline    =    null,
MarkdownParserContext? context = null)
        {
```

```
            if (markdown is null) ThrowHelper.ArgumentNullException_markdown();
            if (writer is null) ThrowHelper.ArgumentNullException_writer();
            pipeline = GetPipeline(pipeline, markdown);
            var document = MarkdownParser.Parse(markdown, pipeline, context);
            // We override the renderer with our own writer
            var renderer = new HtmlRenderer(writer)
            {
                EnableHtmlForBlock = false,
                EnableHtmlForInline = false,
                EnableHtmlEscape = false,
            };
            pipeline.Setup(renderer);
            renderer.Render(document);
            writer.Flush();
            return document;
        }
        /// <summary>
        /// Converts a Markdown string to HTML.
        /// </summary>
        /// <param name="markdown">A Markdown text.</param>
        /// <param name="pipeline">The pipeline used for the conversion.</param>
        /// <param name="context">A parser context used for the parsing.</param>
        /// <returns>The result of the conversion</returns>
        /// <exception cref="ArgumentNullException">if markdown variable is null</exception>
        public static string ToPlainText(string markdown, MarkdownPipeline? pipeline = null, MarkdownParserContext? context = null)
        {
            if (markdown is null) ThrowHelper.ArgumentNullException_markdown();
            var writer = new StringWriter();
            ToPlainText(markdown, writer, pipeline, context);
            return writer.ToString();
        }
    }
}
---
// Copyright (c) Alexandre Mutel. All rights reserved.
// This file is licensed under the BSD-Clause 2 license.
// See the license.txt file in the project root for more information.

namespace Markdown
{
    /// <summary>
    /// This class is the Markdown pipeline build from a <see cref="MarkdownPipelineBuilder"/>.
```

```
    /// </summary>
    public sealed class MarkdownPipeline
    {
        // This class is immutable
        /// <summary>
        ///    Initializes    a    new    instance    of    the    <see
cref="MarkdownPipeline" /> class.
        /// </summary>
        internal MarkdownPipeline(
            OrderedList<IMarkdownExtension> extensions,
            BlockParserList blockParsers,
            InlineParserList inlineParsers,
            TextWriter? debugLog,
            ProcessDocumentDelegate? documentProcessed)
        {
            if            (blockParsers            is            null)
ThrowHelper.ArgumentNullException(nameof(blockParsers));
            if            (inlineParsers            is            null)
ThrowHelper.ArgumentNullException(nameof(inlineParsers));
            // Add all default parsers
            Extensions = extensions;
            BlockParsers = blockParsers;
            InlineParsers = inlineParsers;
            DebugLog = debugLog;
            DocumentProcessed = documentProcessed;
        }
        internal bool PreciseSourceLocation { get; set; }

        /// <summary>
        ///  The  read-only  list  of  extensions  used  to  build  this
pipeline.
        /// </summary>
        public OrderedList<IMarkdownExtension> Extensions { get; }
        internal BlockParserList BlockParsers { get; }
        internal InlineParserList InlineParsers { get; }
        // TODO: Move the log to a better place
        internal TextWriter? DebugLog { get; }
        internal ProcessDocumentDelegate? DocumentProcessed;
        /// <summary>
        /// True to parse trivia such as whitespace, extra heading
characters and unescaped
        /// string values.
        /// </summary>
        public bool TrackTrivia { get; internal set; }
        /// <summary>
        /// Allows to setup a <see cref="IMarkdownRenderer"/>.
        /// </summary>
        ///   <param   name="renderer">The   markdown   renderer   to
setup</param>
        public void Setup(IMarkdownRenderer renderer)
        {
            if              (renderer              is              null)
ThrowHelper.ArgumentNullException(nameof(renderer));
```

```
            foreach (var extension in Extensions)
            {
                extension.Setup(this, renderer);
            }
        }
        private HtmlRendererCache? _rendererCache;
        private HtmlRendererCache? _rendererCacheForCustomWriter;
        internal   RentedHtmlRenderer   RentHtmlRenderer(TextWriter?
writer = null)
        {
            HtmlRendererCache cache = writer is null
                ?  _rendererCache  ??=  new  HtmlRendererCache(this,
customWriter: false)
                :     _rendererCacheForCustomWriter     ??=     new
HtmlRendererCache(this, customWriter: true);

            HtmlRenderer renderer = cache.Get();

            if (writer is not null)
            {
                renderer.Writer = writer;
            }
            return new RentedHtmlRenderer(cache, renderer);
        }
        internal     sealed     class     HtmlRendererCache     :
ObjectCache<HtmlRenderer>
        {
            private const int InitialCapacity = 1024;
            private  static  readonly  StringWriter  _dummyWriter  =
new();
            private readonly MarkdownPipeline _pipeline;
            private readonly bool _customWriter;
            public HtmlRendererCache(MarkdownPipeline pipeline, bool
customWriter = false)
            {
                _pipeline = pipeline;
                _customWriter = customWriter;
            }
            protected override HtmlRenderer NewInstance()
            {
                var  writer  =  _customWriter  ?  _dummyWriter  :  new
StringWriter(new StringBuilder(InitialCapacity));
                var renderer = new HtmlRenderer(writer);
                _pipeline.Setup(renderer);
                return renderer;
            }
            protected override void Reset(HtmlRenderer instance)
            {
                instance.ResetInternal();
                if (_customWriter)
                {
                    instance.Writer = _dummyWriter;
                }
```

```
                else
                {
        ((StringWriter)instance.Writer).GetStringBuilder().Length    =
0;
                }
            }
        }
        internal readonly struct RentedHtmlRenderer : IDisposable
        {
            private readonly HtmlRendererCache _cache;
            public readonly HtmlRenderer Instance;

            internal   RentedHtmlRenderer(HtmlRendererCache   cache,
HtmlRenderer renderer)
            {
                _cache = cache;
                Instance = renderer;
            }
            public void Dispose() => _cache.Release(Instance);
        }    }}
```

Додаток Б

Лістинг коду тестування біліотеки C# для Markdown

```
using System;
using System.IO;
using System.Linq;
using System.Text.RegularExpressions;
using Markdown.Extensions.AutoLinks;
using NUnit.Framework;
namespace Markdown.Tests
{
    public class MiscTests
    {
        [Test]
        public void LinkWithInvalidNonAsciiDomainNameIsIgnored()
        {
                    // Valid IDN
            TestParser.TestSpec("[foo](http://ünicode.com)",   "<p><a
href=\"http://xn--nicode-2ya.com\">foo</a></p>");
            TestParser.TestSpec("[foo](http://ünicode.ünicode.com)",
"<p><a                        href=\"http://xn--nicode-2ya.xn--nicode-
2ya.com\">foo</a></p>");
            // Invalid IDN
            TestParser.TestSpec("[foo](http://ünicode..com)",  "<p><a
href=\"http://%C3%BCnicode..com\">foo</a></p>");
        }
        [TestCase("link          [foo          [bar]]")]          //
https://spec.commonmark.org/0.29/#example-508
        [TestCase("link [foo][bar]")]
        [TestCase("link [][foo][bar][]")]
        [TestCase("link [][foo][bar][[]]")]
        [TestCase("link [foo] [bar]")]
        [TestCase("link [[foo] [] [bar] [[abc]def]]")]
        [TestCase("[]")]
        [TestCase("[ ]")]
        [TestCase("[bar][]")]
        [TestCase("[bar][ foo]")]
        [TestCase("[bar][foo ][]")]
        [TestCase("[bar][fo[ ]o ][][]")]
        [TestCase("[a]b[c[d[e]f]g]h")]
        [TestCase("a[b[c[d]e]f[g]h]i foo [j]k[l[m]n]o")]
        [TestCase("a[b[c[d]e]f[g]h]i[]       [][foo][bar][]       foo
[j]k[l[m]n]o")]
        [TestCase("a[b[c[d]e]f[g]h]i foo [j]k[l[m]n]o[][]")]
        public     void     LinkTextMayContainBalancedBrackets(string
linkText)
        {
            string markdown = $"[{linkText}](/uri)";
            string           expected           =           $@"<p><a
href=""/uri"">{linkText}</a></p>";
            TestParser.TestSpec(markdown, expected);
```

```
            // Make the link text unbalanced
            foreach (var bracketIndex in linkText
                .Select((c, i) => new Tuple<char, int>(c, i))
                .Where(t => t.Item1 == '[' || t.Item1 == ']')
                .Select(t => t.Item2))
            {
          string brokenLinkText = linkText.Remove(bracketIndex, 1);
                markdown = $"[{brokenLinkText}](/uri)";
                expected                    =                $@"<p><a
href=""/uri"">{brokenLinkText}</a></p>";
                string actual = Markdown.ToHtml(markdown);
                Assert.AreNotEqual(expected, actual);  }    }
        [Theory]
        [TestCase('[', 9 * 1024, true, false)]
        [TestCase('[', 11 * 1024, true, true)]
        [TestCase('[', 100, false, false)]
        [TestCase('[', 150, false, true)]
        [TestCase('>', 100, true, false)]
        [TestCase('>', 150, true, true)]
        public   void   GuardsAgainstHighlyNestedNodes(char   c,   int
count, bool parseOnly, bool shouldThrow)
        {
            var markdown = new string(c, count);
            TestDelegate    test    =    parseOnly    ?    ()    =>
Markdown.Parse(markdown) : () => Markdown.ToHtml(markdown);
            if (shouldThrow)
            {
                Exception                       e                    =
Assert.Throws<ArgumentException>(test);
                Assert.True(e.Message.Contains("depth limit"));
            }
            else
            {
                test();            }        }
        [Test]
        public void IsIssue356Corrected()
        {
            string                          input                        =
@"https://foo.bar/path/\#m4mv5W0GYKZpGvfA.97";
            string           expected           =           @"<p><a
href=""https://foo.bar/path/%5C#m4mv5W0GYKZpGvfA.97"">https://foo.ba
r/path/\#m4mv5W0GYKZpGvfA.97</a></p>";
            TestParser.TestSpec($"<{input}>", expected);
            TestParser.TestSpec(input,                        expected,
"autolinks|advanced");        }
        [Test]
        public void IsIssue365Corrected()
        {
            // The scheme must be escaped too...
            string                          input                        =
"![image](\"onclick=\"alert&amp;#40;'click'&amp;#41;\"://)";
```

```
            string           expected           =           "<p><img
src=\"%22onclick=%22alert&amp;#40;%27click%27&amp;#41;%22://\"
alt=\"image\" /></p>";
            TestParser.TestSpec(input, expected);        }
        [Test]
        public void TestAltTextIsCorrectlyEscaped()
        {
            TestParser.TestSpec(
                @"![This is image alt text with quotation ' and
double quotation ""hello"" world](girl.png)",
                @"<p><img src=""girl.png"" alt=""This is image alt
text with quotation ' and double quotation &quot;hello&quot; world""
/></p>");        }
        [Test]
        public void TestChangelogPRLinksMatchDescription()
        {
            string                  solutionFolder                  =
Path.GetFullPath(Path.Combine(TestParser.TestsDirectory, "../.."));
            string   changelogPath   =   Path.Combine(solutionFolder,
"changelog.md");
            string changelog = File.ReadAllText(changelogPath);
            var   matches   =   Regex.Matches(changelog,   @"\(\[\(PR
#(\d+)\)\]\(.*?pull\/(\d+)\)\)");
            Assert.Greater(matches.Count, 0);
            foreach (Match match in matches)
            {
   Assert.True(int.TryParse(match.Groups[1].Value, out int textNr));
   Assert.True(int.TryParse(match.Groups[2].Value, out int linkNr));
                Assert.AreEqual(textNr, linkNr);       }        }
        [Test]
        public void TestFixHang()
        {
            var                          input                          =
File.ReadAllText(Path.Combine(TestParser.TestsDirectory,
"hang.md"));
            _ = Markdown.ToHtml(input);        }
        [Test]
        public void TestInvalidHtmlEntity()
        {
            var input = "9&ddr;&*&ddr;&de\uFFFD\uFFFD__";
            TestParser.TestSpec(input,
"<p>9&amp;ddr;&amp;*&amp;ddr;&amp;de\uFFFD\uFFFD__</p>");        }
        [Test]
        public void TestInvalidCharacterHandling()
        {
            var                          input                          =
File.ReadAllText(Path.Combine(TestParser.TestsDirectory,
"ArgumentOutOfRangeException.md"));
            _ = Markdown.ToHtml(input);        }
        [Test]
        public void TestInvalidCodeEscape()
        {
            var input = "```**Header**  ";
```

```
            _ = Markdown.ToHtml(input);        }
        [Test]
        public void TestEmphasisAndHtmlEntity()
        {
            var markdownText = "*Unlimited-Fun&#174;*&#174;";
            TestParser.TestSpec(markdownText,      "<p><em>Unlimited-
Fun®</em>®</p>");        }
        [Test]
        public void TestThematicInsideCodeBlockInsideList()
        {
        [Test]
        public void VisualizeMathExpressions()
        {
            string math = @"Math expressions
$\frac{n!}{k!(n-k)!} = \binom{n}{k}$
$$\frac{n!}{k!(n-k)!} = \binom{n}{k}$$
$$
\frac{n!}{k!(n-k)!} = \binom{n}{k}
$$
<div class=""math"">
\begin{align}
\sqrt{37} & = \sqrt{\frac{73^2-1}{12^2}} \\
 & = \sqrt{\frac{73^2}{12^2}\cdot\frac{73^2-1}{73^2}} \\
 & = \sqrt{\frac{73^2}{12^2}}\sqrt{\frac{73^2-1}{73^2}} \\
 & = \frac{73}{12}\sqrt{1 - \frac{1}{73^2}} \\
 & \approx \frac{73}{12}\left(1 - \frac{1}{2\cdot73^2}\right)
\end{align}
</div>";
            Console.WriteLine("Math Expressions:\n");
            var                pl                  =                  new
MarkdownPipelineBuilder().UseMathematics().Build();                //
UseEmphasisExtras(EmphasisExtraOptions.Subscript).Build()
            var html = Markdown.ToHtml(math, pl);
            Console.WriteLine(html);        }
        [Test]
        public void InlineMathExpression()
        {
            string math = @"Math expressions
$\frac{n!}{k!(n-k)!} = \binom{n}{k}$";
            var                pl                =                  new
MarkdownPipelineBuilder().UseMathematics().Build();                //
UseEmphasisExtras(EmphasisExtraOptions.Subscript).Build()
            var html = Markdown.ToHtml(math, pl);
            Console.WriteLine(html);
            Assert.IsTrue(html.Contains("<p><span
class=\"math\">\\("), "Leading bracket missing");
            Assert.IsTrue(html.Contains("\\)</span></p>"),  "Trailing
bracket missing");        }
        [Test]
        public void BlockMathExpression()
        {
            string math = @"Math expressions
$$
```

```
\frac{n!}{k!(n-k)!} = \binom{n}{k}
$$
";
            var                pl                =                new
MarkdownPipelineBuilder().UseMathematics().Build();                //
UseEmphasisExtras(EmphasisExtraOptions.Subscript).Build()
            var html = Markdown.ToHtml(math, pl);
            Console.WriteLine(html);
            Assert.IsTrue(html.Contains("<div
class=\"math\">\n\\["), "Leading bracket missing");
            Assert.IsTrue(html.Contains("\\]</div>"),       "Trailing
bracket missing");
        }
        [Test]
        public void CanDisableParsingHeadings()
        {
            var           noHeadingsPipeline           =           new
MarkdownPipelineBuilder().DisableHeadings().Build();
            TestParser.TestSpec("Foo\n===", "<h1>Foo</h1>");
            TestParser.TestSpec("Foo\n===",        "<p>Foo\n===</p>",
noHeadingsPipeline);
            TestParser.TestSpec("#    Heading    1",    "<h1>Heading
1</h1>");
            TestParser.TestSpec("# Heading 1", "<p># Heading 1</p>",
noHeadingsPipeline);
            // Does not also disable link reference definitions
            TestParser.TestSpec("[Foo]\n\n[Foo]:     bar",     "<p><a
href=\"bar\">Foo</a></p>");
            TestParser.TestSpec("[Foo]\n\n[Foo]:     bar",     "<p><a
href=\"bar\">Foo</a></p>", noHeadingsPipeline);        }
        [Test]
        public void CanOpenAutoLinksInNewWindow()        {
            var              pipeline              =              new
MarkdownPipelineBuilder().UseAutoLinks().Build();
            var           newWindowPipeline           =           new
MarkdownPipelineBuilder().UseAutoLinks(new    AutoLinkOptions()    {
OpenInNewWindow = true }).Build();
            TestParser.TestSpec("www.foo.bar",                  "<p><a
href=\"http://www.foo.bar\">www.foo.bar</a></p>", pipeline);
            TestParser.TestSpec("www.foo.bar",                  "<p><a
href=\"http://www.foo.bar\"   target=\"_blank\">www.foo.bar</a></p>",
newWindowPipeline);        }
        [Test]
        public void CanUseHttpsPrefixForWWWAutoLinks()        {
            var              pipeline              =              new
MarkdownPipelineBuilder().UseAutoLinks().Build();
            var            httpsPipeline            =            new
MarkdownPipelineBuilder().UseAutoLinks(new    AutoLinkOptions()    {
UseHttpsForWWWLinks = true }).Build();
            TestParser.TestSpec("www.foo.bar",                  "<p><a
href=\"http://www.foo.bar\">www.foo.bar</a></p>", pipeline);
            TestParser.TestSpec("www.foo.bar",                  "<p><a
href=\"https://www.foo.bar\">www.foo.bar</a></p>", httpsPipeline);
```

```
}       }}
```