

ТЕХНОЛОГИЧНО УЧИЛИЩЕ ЕЛЕКТРОННИ СИСТЕМИ  
към ТЕХНИЧЕСКИ УНИВЕРСИТЕТ - СОФИЯ

## ДИПЛОМНА РАБОТА

Тема: Библиотека за създаване на текстов потребителски интерфейс

Дипломант:  
*Ивайло Генчев*

Научен ръководител:  
*маг. инж. Владимир Алексиев*

С О Ф И Я

2 0 2 3



## **Становище на дипломния ръководител**

Дипломантът Ивайло Генчев се е справил отлично. Изпълнил е поставените задачи. Моля да бъде допуснат до защита пред държавната квалификационна комисия.

Предлагам за рецензент Веселин Русинов.

Дата:  
гр. София

Подпис:  
/ Владимир Алексиев/

# Използвани съкращения

**DOM** Document Object Model

**SRP** Single-responsibility principle

**REGEX** Regular Expression

**TUI** Text User Interface

**CSS** Cascading Style Sheet

**HTML** Markup Language

**STDIN** Standard Input

**STDOUT** Standard Output

**ОД** Обхождане в дълбочина

**ТПИ** Текстов потребителски интерфейс

**ГПИ** Графичен потребителски интерфейс

# Съдържание

<b>1</b>	<b>Проучване</b>	<b>8</b>
1.1	Терминали . . . . .	8
1.1.1	Телетайп терминал - TTY . . . . .	8
1.1.2	Псевдотелетайп терминал - PTY . . . . .	9
1.1.3	Терминален емулатор . . . . .	10
1.2	Контролни символи . . . . .	10
1.2.1	ANSI контролна последователност . . . . .	10
1.3	Преглед на съществуващи TUI библиотеки . . . . .	11
1.3.1	ncurses . . . . .	11
1.3.2	Textual . . . . .	12
<b>2</b>	<b>Проектиране на библиотека за създаване на текстов потребителски интерфейс</b>	<b>14</b>
2.1	Функционални изисквания . . . . .	14
2.2	Подбор на средства за разработка . . . . .	15
2.2.1	Програмен език . . . . .	15
2.2.2	Статичен анализатор . . . . .	16
2.2.3	Компонентно тестове . . . . .	16
2.2.4	Система за управление на версиите . . . . .	16
2.2.5	Непрекъсната интерграция . . . . .	17
2.2.6	Среда за разработка . . . . .	18

<b>3</b>	<b>Реализация на библиотека за създаване на текстов потребителски</b>	<b>19</b>
3.1	Структура на проекта . . . . .	19
3.2	Файлове с общо предназначение . . . . .	20
3.3	Йерархия на компоненти . . . . .	21
3.4	Вътрешно представяне на компонент . . . . .	23
3.5	Видове компоненти . . . . .	25
3.6	Композитор . . . . .	26
3.7	Визуализация на приложение . . . . .	29
3.8	Персонализиране на компоненти . . . . .	31
3.9	Издаване и обработка на събития . . . . .	32
3.10	Абониране за събития . . . . .	32
<b>4</b>	<b>Ръководство за потребителя</b>	<b>38</b>
4.1	Инсталиране на библиотеката . . . . .	38
4.1.1	Сдобиване с кода на библиотеката . . . . .	38
4.1.2	Инсталиране на библиотеката . . . . .	39
4.2	Ръководство за обучение . . . . .	40
4.2.1	Hello World! . . . . .	40
<b>5</b>	<b>Заклучение</b>	<b>44</b>

# Увод

Първоначално, когато компютрите започват масово да се разпространяват, те могат да се достъпват единствено през терминал - текстови интерфейс, с който се работи с клавиатура. Не след дълго, поради бързия технологичен напредък, графичния потребителски интерфейс (ГПИ) бива създаден заедно с първата компютърна мишка. Това развитие значително увеличава достъпността и лекотата на използване на компютърните устройства.

Днес, около половин век по-късно, почти всеки притежава устройство с ГПИ. Текстовият потребителски интерфейс (ТПИ) се използва доста по-рядко - предимно в среди, където няма ГПИ, например при сървъри с ограничени ресурси или от потребители с големи възможности. Поради тази причина повечето съществуващи програми за ТПИ съществуват от десетилетия, а новите са принудени да използват софтуер, разработен преди десетки години. Това не намалява функционалността на програмите, но увеличава времето им за разработка и може да ги направи по-недостъпни за потребителя. Този проект решава именно този проблем.

За цел на проекта е поставена разработката на лесна за ползване библиотека, която да позволява създаването на модерен потребителски интерфейс.

## Задачи

- Логическо отделени елементи на потребителския интерфейс.
- Оформление на компонентите на потребителския интерфейс.
- Обработка на събития от мишка и клавиатура.

В **Глава 1** е направен обзор на средите на работа на ТПИ и методите за управлението им. Също така е направен преглед на съществуващи библиотеки, които се използват от приложения и разработчици по целия свят.

# Глава 1

## Проучване

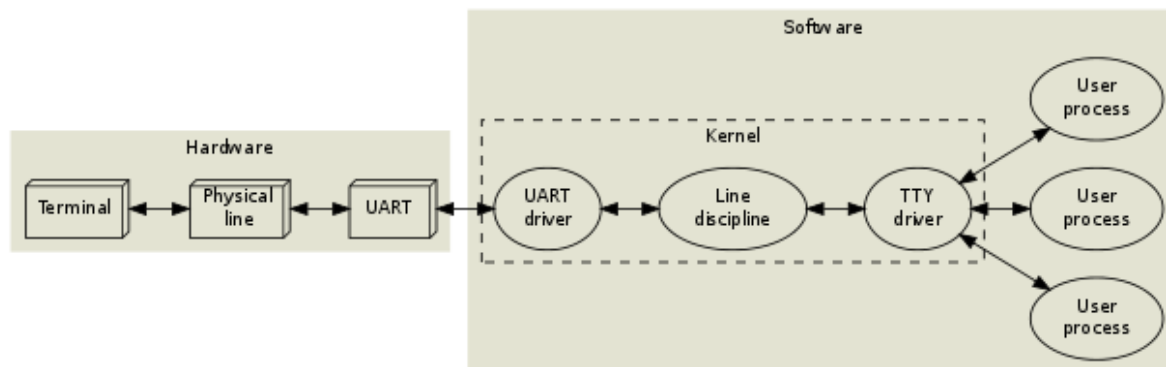
### 1.1 Терминали

От библиотеката се изисква създаването на текстови потребителски интерфейси, което зависи от средата, в която се изпълнява програмата. Разгледани са възможните среди без съсредоточаване в детайлите.

#### 1.1.1 Телетайп терминал - TTY

Този вид терминал се нарича така, поради сходността си с телеграфния апарат. На фигура 1.1 е показана блокова схема на принцип на работа на TTY терминал. Потребителят пише на хардуерния терминал, който е свързан към UART (универсален асинхронен приемник и предавател) на компютъра. Операционната система на компютъра съдържа UART драйвер, който управлява физическото предаване на байтове. "Дисциплината на линията" определя командите за редактиране на терминала - например изтрий линия, препечатай линия, изтрий дума.



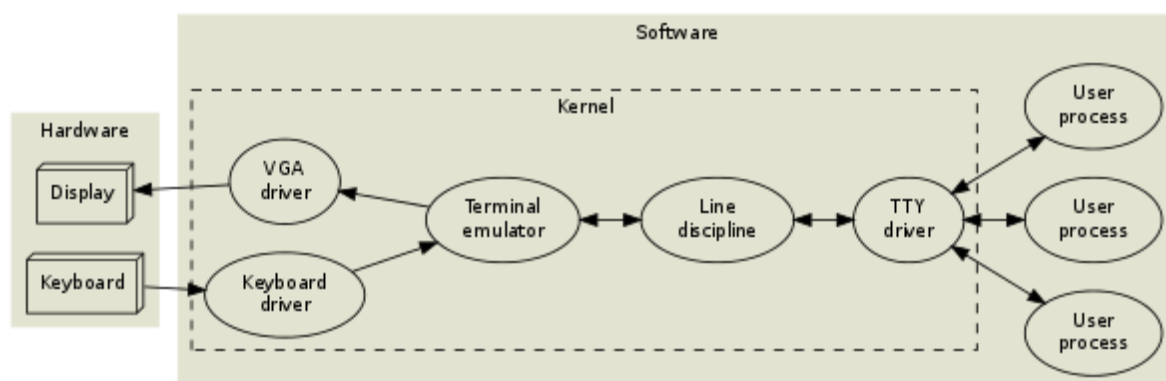


Фигура 1.1: Принцип на работа на хардуерен TTY

### 1.1.2 Псевдотелетайп терминал - PTY

Фигура 1.2 постига същия резултат като фигура 1.1, като разликата е, че хардуерен терминал вече не се използва. Linux ядрото използва софтуерно реализиран краен автомат, който емулира същите функционалности.

[1]



Фигура 1.2: Принцип на работа на PTY

### 1.1.3 Терминален емулатор

Терминалните емулатори използват PTY подсистемата, но за разлика от PTY работят на ring 3 (потребителско пространство). Това ниво на абстракция добавя много повече гъвкавост, но и усложнява имплементацията си, защото един PTY може да се отвори в друг PTY.

Тази гъвкавост води до разлики в терминалните емулатори. Част от тези, които са програмирани преди въвеждането на ISO 6429, не го имплементират, а други имплементират разширени CSI кодове

## 1.2 Контролни символи

Контролните символи, също наричани непечатни, нямат графично представяне, а се използват за управление на устройства и предаване на данни. POSIX стандартът дефинира само осем контролни символа, от които по-често използваните са:

- `\0` - край на символен низ
- `\n` - нов ред
- `\t` - табулация
- `\r` - връщане на каретата

### 1.2.1 ANSI контролна последователност

ANSI кодовете могат да въздействат на поведението на терминала, като променят позицията на курсора, цвета на фона и символите, стилизирането на шрифта или други настройки.

Структурата на всички ANSI кодове е следната:

CSI [байтове на режима] n1;n2... [междинни байтове] финален байт

ANSI последователността винаги започва с CSI и приключва с финален байт. Байтовете на режима могат да бъдат в обхват от '0' (0x30) до '?' (0x3f), а междинните байтове в обхват от ' ' (0x20) до '/' (0x2f). Числата n1, n2, ... не са задължителни и, ако стойността им не е посочена, се приемат за 0 или 1 в зависимост от операцията, която ще се изпълни заради ANSI кода.

Въпреки че е позволено наличието на повече от един междинен байт и байт на режима, това не се ползва.

#### **Идентификатор на контролна последователност (CSI)**

Преамбюл, с който се разпознава началото на ANSI кода. Има големина 2 байта, като започва с ESC символ (0x1b) и "["(0x5b)

## **1.3 Преглед на съществуващи TUI библиотеки**

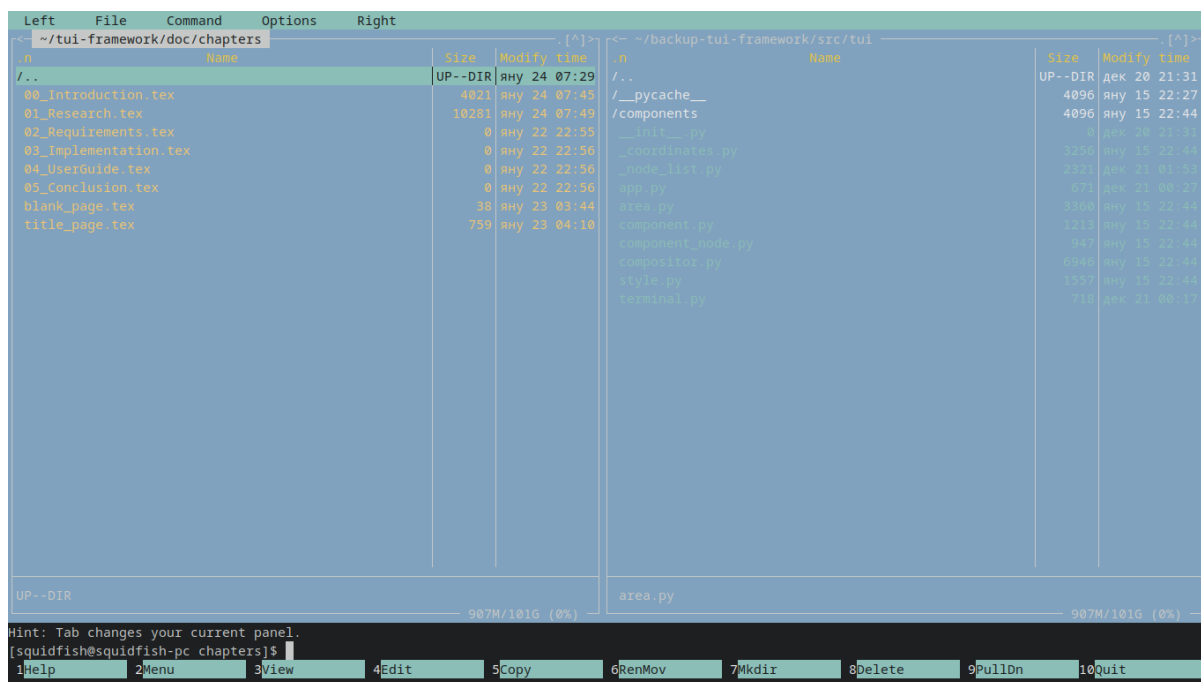
### **1.3.1 ncurses**

ncurses е най-използваната библиотека за управление на терминали в UNIX-подобни среди, позволяваща създаване на приложения с текстов потребителски интерфейс.

С ncurses програмистите не трябва да се грижат за използването на правилните контролни символи при различните терминали. Библиотеката предоставя и абстракция за логическо отделение на компоненти - прозорци. Прозорците се съпоставят с координатите на терминала, като

всеки прозорец представлява масив от символи, в който програмистът може да промени външния вид на прозореца.

Една от най-силните страни на ncurses е съвместимостта на библиотеката. Тя може да работи във всякакви среди при всякакви условия



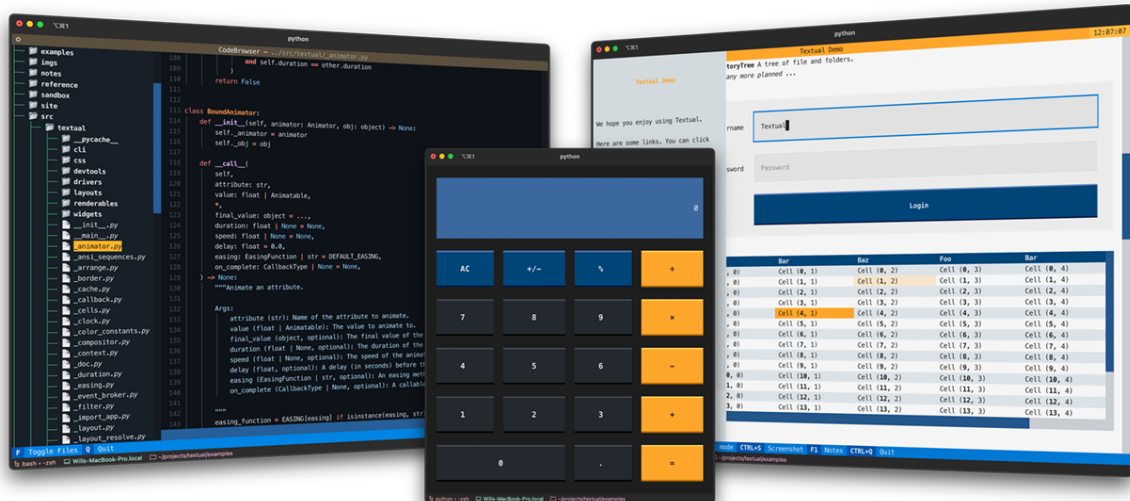
Фигура 1.3: Midnight Commander - файлов мениджър написани с ncurses

### 1.3.2 Textual

Понастоящем Textual е авангардна библиотека за създаване на модерни текстови потребителски интерфейси.

Textual взема вдъхновение от уеб разработката. В Библиотеката нивото на абстракция е много по-високо отколкото при ncurses. Използват се компоненти, които са подредени в дървовидна структура и силно напо-

добяват HTML елементи. Компонентите могат да се стилизират със CSS. Има поддръжка на мишка и могат да се създават плавни анимации с над 60 кадъра в секунда (в зависимост от терминалния емулатор).



Фигура 1.4: Примерни приложения написани с Textual

## Глава 2

# Проектиране на библиотека за създаване на текстов потребителски интерфейс

### 2.1 Функционални изисквания

Изискванията, поставени върху проекта са следните:

- Подредба и структуриране на елементи на потребителски интерфейс
  - Елементите трябва да имат стриктна йерархична структура.
  - Елементите трябва да могат да се подреждат спрямо себе си и други компоненти.
- Елементи на потребителски интерфейс с добавена функционалност
  - Трябва да има елементи, които изпълняват някаква конкретна функционалност. Те могат да бъдат:
    - \* Етикет - елемент, който показва текст.
    - \* Бутон - изпълнява зададена функционалност при натискане.
    - \* Въвеждане - позволява на потребителя да въвежда текст.
- Обработка на събития от мишка и клавиатура

- Приложенията, трябва да могат да отчитат и реагират на събития от клавиатура и мишка.

## 2.2 Подбор на средства за разработка

### 2.2.1 Програмен език

Текущата дипломна работа е реализирана на програмния език Python.

Езикът за програмиране Python е създаден в началото на 90-те години на миналия век от холандския програмист Гуидо ван Росум. По това време той иска да създаде лесен за изучаване език с общо предназначение, който да преодолява ограниченията на езици като С. Поради синтактичната си близкост до английския език и гореизброените причини Python бързо набира популярност.

Python е обектно ориентиран и строго типизиран език от високо ниво, като типовете на данните се определят по време на изпълнението. Поради скриптовия си характер, липсата на необходимост от задаване на типа на променливи и вградените сложни полиморфни типове (списъци и речници), когато сравнявано с други езици с общо предназначение, времето за разработка на програми написани на Python е значително по-малко. Те са 3-5 пъти по-кратки от еквивалентите на Java и 5-10 пъти от тези на C++. [5]

От 2003 г. насам Python се класира в топ 10 на най-популярните езици за програмиране. В момента на писане на дипломната работа е определен за най-популярен според ‘TIOBE Programming Community Index’. Езикът намира обширна употреба, като най-често се използва при анализ на данни, научни изчисления, изкуствен интелект както и в сферата на информационната сигурност. [3]

Основните недостатъци на приложения, написани на Python, са ско-

ростта им на изпълнения и паметта, която заделят. Това не е проблем за текущия проект, защото скоростта е ограничена от бодовете на терминала, в който се изпълнява, а паметта е пренебрежима за модерните компютри.

### 2.2.2 Статичен анализатор

Много софтуерни проекти използват инструменти, които сигнализират за грешки в програмирането, бъгове, стилистични грешки и подозрителни конструкции.

Текущият проект използва **flake8** и **pylint** за статичен анализ на код. И двата инструмента проверяват за спазването на официалното ръководство за стила на кода на Python (PEP8). Въпреки че се препокриват на някои места, двата статични анализатора проверяват и за спазването на допълнителните практики при писане на код, които се различават.

### 2.2.3 Компонентно тестване

При компонентното тестване (Unit Testing) се проверява дали отделни единици код (методи или класове) работят правилно. За разработката на текущия проект е използвана библиотеката **pytest**. Използвани са плъгините **pytest-cov** и **pytest-asyncio** за проверка на покритието на тествания код и съвместимост при тестване на асинхронни методи.

### 2.2.4 Система за управление на версиите

Всеки един софтуерен проект в днешно време използва някакъв вид система за управление на версиите. Най-разпространената такава система е **git**. Тя позволява проследяването на всички промени по кода, като по този начин подпомага сътрудничеството на големи екипи от хора.

По време на разработката на текущия проект бе използвана стратегия на разклоненията (branching strategy), при която всяко изискване по

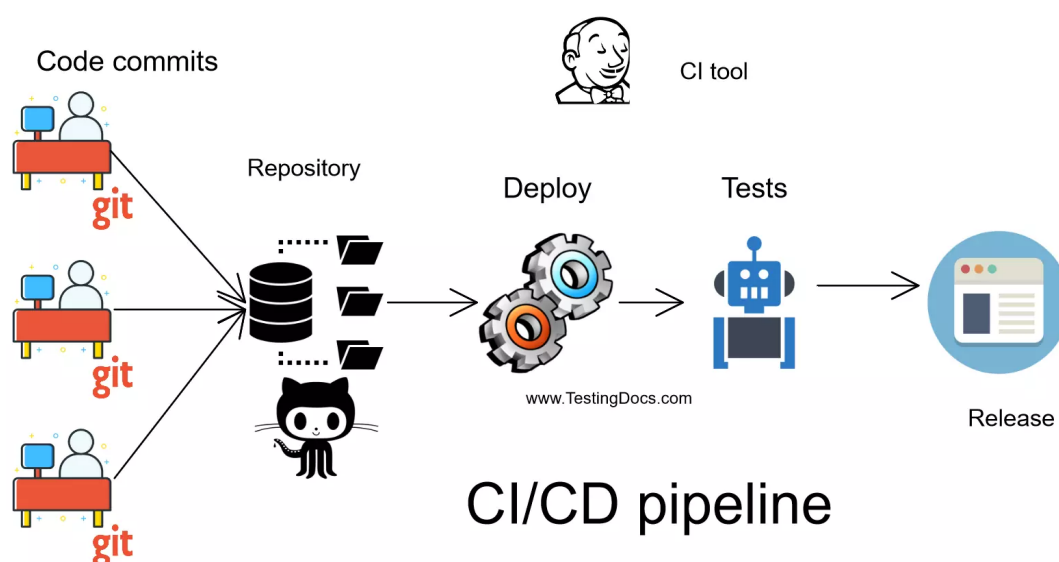


проекта има свое отделно разклонение:

- area - разработка на вътрешното растеризиране на компонентите
- compositor/button - разработка на елемент 'бутон'
- compositor/input - разработка на елемент 'въвеждане'
- compositor/label - разработка на елемент 'надпис'
- compositor - разработка на алгоритъм за композиране на елементи
- events - работа със събития от периферни устройства
- styles - разработка на настройки за персонализиране на елементи

### 2.2.5 Непрекъснатата интеграция

Непрекъснатата интеграция (Continuous Integration) (фиг. 2.1) е практика при разработката на софтуер при която работата на много разработчици се обединява на едно място, след което се задейства автоматизирана компилация/интерпретация и тестване на софтуерния продукт.



Фигура 2.1: Непрекъснатата интеграция

По време на разработката на текущия проект бяха използвани предоставените от платформата **GitHub** средства за автоматизирани тестване и статичен анализ на кода (GitHub Actions).

#### **2.2.6 Среда за разработка**

Текущата дипломна работа бе разработена на операционната система Arch Linux, като за редакция на кода и документацията бе използвана конфигурация на текстовия редактор Neovim - Astronvim. Кодът на проекта бе публикуван онлайн в платформата GitHub. Настоящата документация бе написана с помощта на  $\text{\LaTeX}$ .

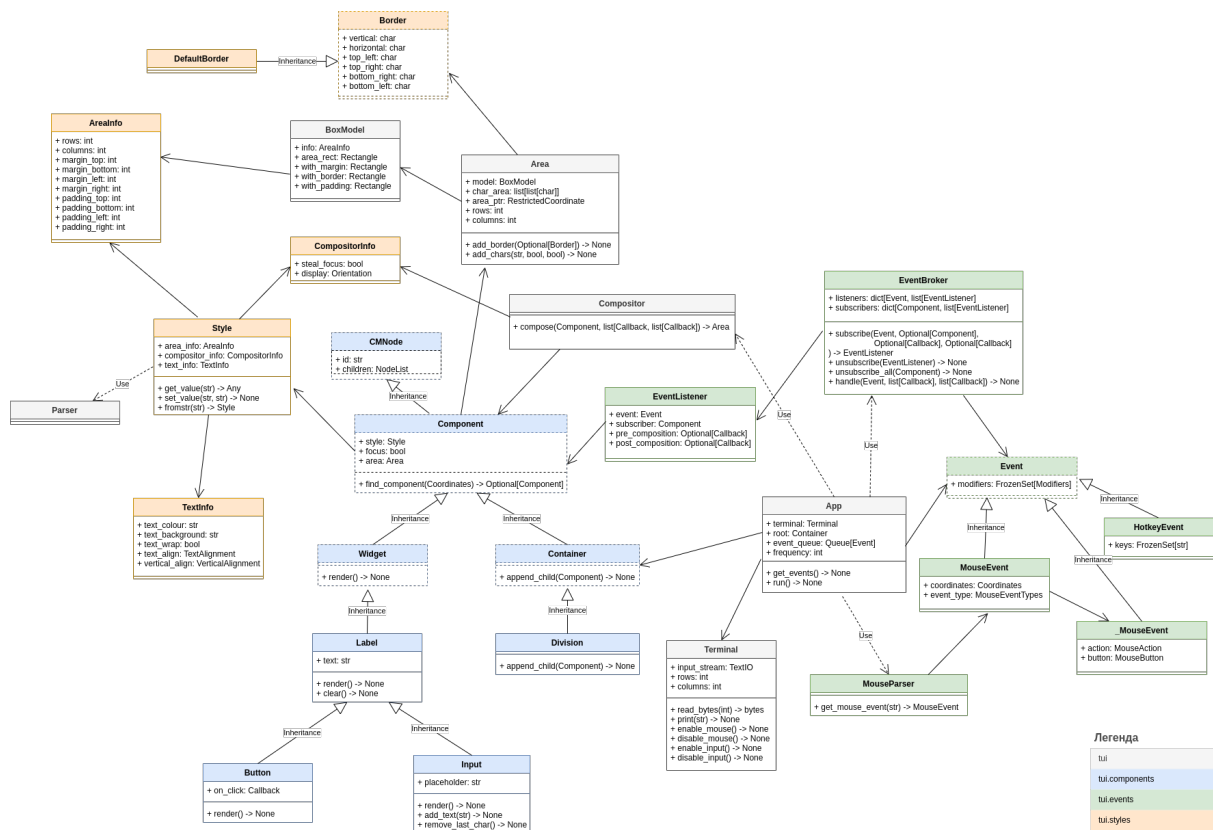
## Глава 3

# Реализация на библиотека за създаване на текстов потребителски

### 3.1 Структура на проекта

Всички части на библиотеката се намират в пакета `tui`, който от своя страна е разделен на подпакети:

- стилизация - `styles`
- елементи - `components`
- събития - `events`



Фигура 3.1: UML диаграма на архитектурата на библиотеката

## 3.2 Файлове с общо предназначение

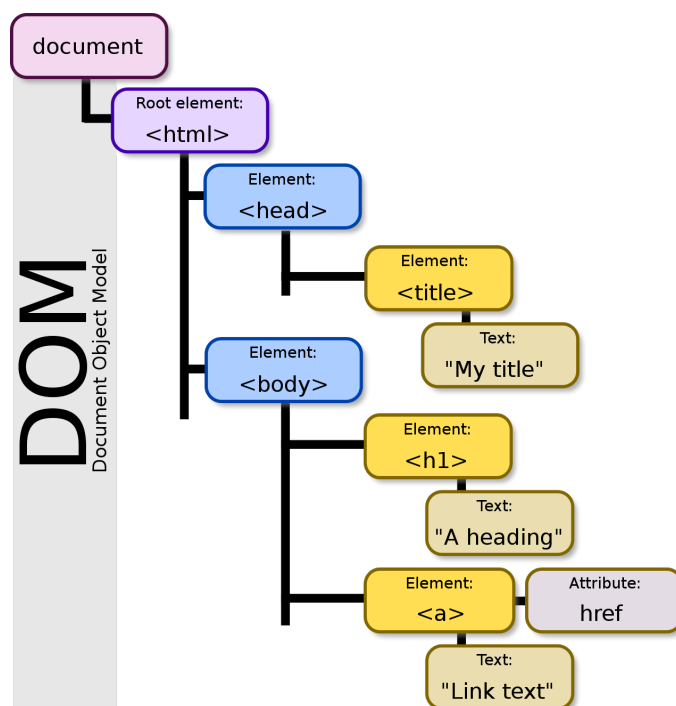
Файловете `_coordinates.py` и `_parser.py` са с общо предназначение.

Във файла `_parser.py` са дефинирани регулярни изрази (regex), които се използват в целия проект.

Файлът `_coordinates.py` предоставя абстракция за точка в декартова координатна система, правоъгълник и точка, която не може да излезе извън границите на правоъгълник.

### 3.3 Йерархия на компоненти

Всяко приложение има главен (root) компонент, който обикновено заема цялата резолюция на терминала. Към този компонент могат да се добавят деца - други компоненти, които описват как изглежда главният компонент - родителят им. Този процес може да се повтори за всички компоненти, които могат да имат деца, като по този начин се образува дървовидна структура наподобяваща документен обектен модел (ДОМ).



Фигура 3.2: Пример за ДОМ дърво

Компонентното дърво е имплементирано посредством класа **NodeList**.  
Компонентите се записват в:

- масив - запазва реда на компонентите
- хеш таблица - откриването на компонент според идентификатора му отнема константо време.

```

1  class NodeList(Sequence):
2      """Responsible for providing a structure for accessing an ordered component list"""
3
4      def __init__(self) -> None:
5          self._nodes_list: list[Component] = [] # preserve order
6          # components with no id aren't in the dict
7          self._nodes_dict: dict[str, Component] = {} # fast search by index
8
9      def __len__(self) -> int:
10         return len(self._nodes_list)
11
12     def __getitem__(self, index: int | slice) -> Component | list[Component]:
13         return self._nodes_list[index]
14
15     def __setitem__(self, index: int, new_component: Component) -> None:
16         prev_component = self._nodes_list[index]
17         self._nodes_list[index] = new_component
18
19         if prev_component.id is not None:
20             self._nodes_dict.pop(prev_component.id)
21         if new_component.id is not None:
22             self._nodes_dict[new_component.id] = new_component
23
24     def __contains__(self, component: Component) -> bool:
25         return component in self._nodes_list
26
27     def get_by_id(self, identifier: str) -> Optional[Component]:
28         """Get component (search by id)"""
29         return self._nodes_dict.get(identifier)
30
31     def append(self, component: Component) -> None:
32         """Add component at the end of the list"""
33         if component in self._nodes_list:
34             raise ValueError("Component already exists")
35
36         if component.id in self._nodes_dict:
37             raise KeyError("Id already exists")
38
39         self._nodes_list.append(component)
40
41         if component.id is not None:
42             self._nodes_dict[component.id] = component
43
44     def pop(self, index: int) -> Component
45         """Remove component (search by id)"""
46         component = self._nodes_list.pop(index)
47
48         if component.id is not None:
49             self._nodes_dict.pop(component.id)
50
51         return component
52
53     def remove(self, component: Component) -> None:
54         """Remove component"""
55         self._nodes_list.remove(component)
56         if component.id is not None:
57             self._nodes_dict.pop(component.id)

```

### 3.4 Вътрешно представяне на компонент

Както се забелязва (фиг. 3.1), компонентите зависят от класа **Area**. Това е задължителна част от всеки компонент. В основата си, класът представлява двуизмерна матрица от символи, която определя как трябва да изглежда даден компонент сам по себе си - преди да бъде композиран.

```
1  class Area:
2      """
3      Responsible for rendering how a component should look.
4      This includes the box model and any dynamic changes that may happen within it
5      """
6      def __init__(self, area_info: AreaInfo, border: Optional[Border] = None) -> None:
7          self.model = BoxModel(info=area_info)
8          # char_area is where every change is reflected
9          # rows x columns
10         self.char_area: list[list[str]] = [
11             [' ' for _ in range(self.columns)]
12             for _ in range(self.rows)
13         ]
14
15         # the area pointer is used for easier navigating and writing to
16         # char_area as it automatically keeps track of a lot of variables
17         self.area_ptr = RestrictedCoordinates(
18             _row=0,
19             _column=0,
20             _restriction=self.model.with_padding,
21             relative=True
22         )
23
24         self.add_border(border)
25
26     def __str__(self):
27         return "\n".join(["".join(list(row)) for row in self.char_area])
28
29     def add_border(self, border: Optional[Border]) -> None:
30         """Apply border to the area. This will shrink the space the component
31         can use depending on the border's size"""
32
33         ... # code omitted
34
35     def add_chars(
36         self,
37         string: str, # string to be added to the area
38         column_preserve: bool = False, # should new line start at the current area pointer column
39         ptr_preserve: bool = True # reset area pointer to initial position
40     ) -> None:
41         """Write to char_area starting from area_ptr coordinates.
42         Any '\n' in the string translate to a row increment.
43         Default behaviour doesn't mutate the area pointer."""
```

```

44
45     # remember initial coordinates (for column preserve and to roll-back the area pointer)
46     initital_coords = Coordinates(_row=self.area_ptr.row, _column=self.area_ptr.column)
47
48     # check if string can fit
49     if not self._verify_str(
50         string=string,
51         column_preserve=column_preserve
52     ):
53         raise IndexError("String is too large")
54
55     for count, char in enumerate(string):
56         if char == '\n':
57             # if area_ptr is on the last row and the last char is a newline
58             # area_ptr restriction would activate, hence the latter check
59             if count >= len(string) - 1:
60                 break
61
62             self.area_ptr.row += 1
63             if column_preserve:
64                 self.area_ptr.column = initital_coords.column
65             else:
66                 self.area_ptr.column = (
67                     self.area_ptr.restriction.top_left.column
68                 )
69
70             continue
71
72         self.char_area[self.area_ptr.row][self.area_ptr.column] = char
73
74         # Required check to prevent a RestrictedCoordinates exception
75         if (self.area_ptr.column < self.area_ptr.restriction.bottom_right
76             .column):
77             self.area_ptr.column += 1
78
79     if ptr_preserve:
80         self.area_ptr.row = initital_coords.row
81         self.area_ptr.column = initital_coords.column
82     else:
83         # column is incremented one last time after adding the last char
84         # this can force area_ptr to go out of bounds hence returning the pointer one back
85         self.area_ptr.column -= 1
86
87     def _verify_str(self, string: str, column_preserve: bool) -> bool:
88         """Verify that string can fit in char_area - used in add_chars.
89         It mimics the behaviour of add_char to make sure the string never goes out of bounds."""
90
91         ... # code omitted
92
93
94     ... # methods omitted

```



### 3.5 Видове компоненти

Компонентите могат да се разделят на две категории, в зависимост от поведението си:

- **Container** - Отговаря единствено за собствените си деца. Управлява реда, в които ще се композират и позициите, в които ще застанат едни спрямо други. Примерна имплементация на контейнер е **Division** (разделение)

```
1 class Container(Component):
2     """
3     Abstract component that's responsible for containing and organizing other components.
4     '@property def children(self)' should be implemented in child classes
5     """
6
7     def __init__(
8         self,
9         *children: Component, # Child components
10        style: str | Style = Style(), # Style properties for the component
11        identifier: Optional[str] = None, # Unique identifier
12    ) -> None:
13        super().__init__(identifier=identifier, style=style)
14
15        for child in children:
16            # Components that inherit Container could pass *children to
17            # super().__init__() which is seen as an empty tuple here
18            if isinstance(child, tuple):
19                continue
20
21            self.append_child(child)
22
23    def append_child(self, component: Component):
24        """Add a child at the end of the component list of the container.
25        You can manipulate this by adding other components for positioning or composition"""
26        self.children.append(component)
```

- **Widget** - Абстрактен компонент, който не може да има дъщерни компоненти. Може да променя поведението на повечето събития и се съсредоточава върху изпълняването на конкретна функционалност. Примери за виджети са класовете **Label** (надпис) и **Button** (бутон).

```

1  class Widget(Component):
2      """An abstract component that can't hold other components and can override event behaviour."""
3      def __init__(
4          self,
5          style: str | Style = Style(), # Style properties for the component
6          identifier: Optional[str] = None, # Unique identifier
7      ) -> None:
8          super().__init__(identifier=identifier, style=style)
9
10     @abstractmethod
11     def _render_to_area(self) -> None:
12         """Render the component's content to its area"""
13
14     @property
15     def children(self) -> list:
16         """Widgets can't have child components.
17         Return an empty list for compatibility"""
18         return []

```

Благодарение на разграничението на компоненти, всеки изпълнява само една роля (SRP). Това улеснява имплементирането на нови компоненти, като композиция от вече съществуващи компоненти.

### 3.6 Композитор

Композиторът е съставна част от приложението. Той се грижи за сглобяването на крайния му изглед. Процесът се изпълнява във всеки кадър при условие, че има промяна на изгледа. Алгоритъмът се основава на ОД (обхождане в дълбочина).

Методът **compose** се извиква рекурсивно с всички деца на главния (root) компонент, докато не се стигне до най-крайното дете - дете, което няма деца. Неговият родител получава вътрешното му представяне и го подрежда според стила си. Текущият проект поддържа два вида ориентация на контейнери:

- **inline** - децата на контейнера се подреждат на същия ред
- **block** - децата на контейнера се подреждат в същата колона

```

1  class Compositor:
2      """Responsible for compositing components into a single area"""
3
4      @staticmethod
5      def compose(
6          root: Component, # the component which's area is being composed
7          pre_composit: list[Callback],
8          post_composit: list[Callback]
9      ) -> Area:
10         """Compose a component with its children components recursively. Run
11         pre-compose and post-compose hooks"""
12         for callback in pre_composit:
13             callback()
14
15         # set root rect mapping to itself
16         root._rect_mapping = root.area.model.area_rect
17         new_area = Compositor._compose(root=root)
18
19         for callback in post_composit:
20             callback()
21
22         return new_area
23
24     @staticmethod
25     def _compose(root: Component) -> Area:
26         """Compose a component with its children components recursively
27
28         root: the component's area that's being composed
29         next_component: rectangular area which the next component should occupy
30         """
31
32         new_area = copy.deepcopy(root.area)
33
34         # the rectangle the previous child was in
35         prev_rect: Optional[Rectangle] = None
36
37         for child in root.children:
38             try:
39                 prev_rect = Compositor._get_next_rectangle(
40                     parent=root,
41                     prev_rect=prev_rect,
42                     component=child
43                 )
44                 child._row_mapper = prev_rect
45             except CoordinateError as exc:
46                 raise InsufficientAreaError(
47                     "Component area isn't large enough"
48                 ) from exc
49
50         # recursion ends when there are no more children
51         child_area = Compositor._compose(child)
52         new_area.area_ptr.row = (
53             prev_rect.top_left.row
54             + new_area.model.with_padding.top_left.row
55         )
56         new_area.area_ptr.column = (
57             prev_rect.top_left.column

```

```

58         + new_area.model.with_padding.top_left.column
59     )
60
61     # draw child component
62     try:
63         new_area.add_chars(str(child_area), column_preserve=True)
64     except IndexError as exc:
65         raise InsufficientAreaError(
66             "Component area isn't large enough"
67         ) from exc
68
69     return new_area
70
71 @staticmethod
72 def _get_next_rectangle(
73     parent: Component, # the parent component this one will reside in
74     component: Component, # the component calculations are done for
75     prev_rect: Optional[Rectangle] = None
76 ) -> Rectangle:
77     """Helper function that decides where components are placed when
78     compositing"""
79     if parent.style.compositor_info.display == Orientation.INLINE:
80         return Compositor._get_next_rectangle_inline(
81             parent=parent,
82             component=component,
83             prev_rect=prev_rect
84         )
85
86     return Compositor._get_next_rectangle_block(
87         parent=parent,
88         component=component,
89         prev_rect=prev_rect
90     )
91
92 @staticmethod
93 def __get_next_rectangle_inline(
94     parent: Component, # the parent component this one will reside in
95     component: Component, # the component calculations are done for
96     # rectangle for the previous component
97     prev_rect: Optional[Rectangle] = None
98 ) -> Rectangle:
99     """Helper function for inline compositing. Returns the area the next
100    component should be placed in
101
102    Default prev_rectangle is a rectangle outside of the parent's area
103    that's derived to place the component in the top left of the parent
104    component's area:
105    top_left: row = 0 && column <= -1
106    bottom_right: row >= 0 && column = -1
107    """
108    if prev_rect is None:
109        prev_rect = Rectangle(
110            top_left=Coordinates(_row=0, _column=-1),
111            bottom_right=Coordinates(_row=0, _column=-1)
112        )
113
114    return Rectangle(
115        top_left=Coordinates(
116            _row=prev_rect.top_left.row,

```

```

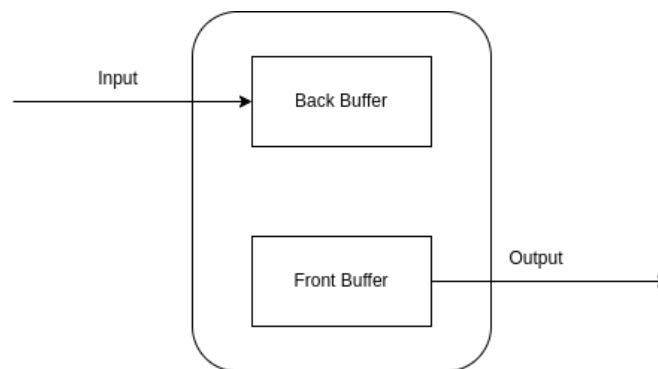
117         _column=prev_rect.bottom_right.column + 1
118     ),
119     bottom_right=Coordinates(
120         _row=parent.area.rows - 1,
121         _column=prev_rect.bottom_right.column +
122         component.area.columns
123     )
124 )
125
126 @staticmethod
127 def __get_next_rectangle_block(
128     parent: Component, # the parent component this one will reside in
129     component: Component, # the component calculations are done for
130     # rectangle for the previous component
131     prev_rect: Optional[Rectangle] = None
132 ) -> Rectangle:
133     """Helper function for block compositing. Returns the area the next
134     component should be placed in
135
136     Default prev_rectangle is a rectangle outside of the parent's area
137     that's derived to place the component in the top left of the parent
138     component's area:
139         top_left: row = -1 && column <= 0
140         bottom_right: row >= -1 && column = 0
141     """
142     if prev_rect is None:
143         prev_rect = Rectangle(
144             top_left=Coordinates(_row=-1, _column=0),
145             bottom_right=Coordinates(_row=-1, _column=0)
146         )
147
148     return Rectangle(
149         top_left=Coordinates(
150             _row=prev_rect.bottom_right.row + 1,
151             _column=prev_rect.top_left.column
152         ),
153         bottom_right=Coordinates(
154             _row=prev_rect.bottom_right.row + component.area.rows,
155             _column=parent.area.columns - 1
156         )
157     )

```

### 3.7 Визуализация на приложение

Тъй като приложението е предвидено да се използва в терминал, кадрите в секунда, които могат да се постигнат, са ограничени от бодовете на терминала. Писането в стандартния изход (stdout) трябва да използва минимално системни извиквания, както и да се пишат минимален брой байтове. За целта се използва техника подобна на двойно буфериране.

(фиг. 3.3).



Фигура 3.3: Принцип на работа на двойно буфериране

Използвайки аналогията с двойно буфериране (фиг. 3.3) - задният буфер запазва предишното състояние - **Area**, на терминала. Сравнява се с текущото състояние, като на предния буфер се записва низ, който описва промените с минимален брой байтове - използват се ANSI кодове за манипулиране на позицията на курсора.

```
1 class Terminal:
2     """Responsible for providing an interface to the terminal the program is being ran in"""
3
4     def __init__(self, input_stream: TextIO = stdin) -> None:
5         ... # code omitted
6         self._prev_state: Optional[Area] = None
7         ... # code omitted
8
9     def print(self, area: Area) -> None:
10        """Print a string to the terminal. Overwrites content in the terminal
11        for higher response time and no stuttering."""
12        # handle first print
13        if self._prev_state is None:
14            print(str(area), end="", flush=True)
15            self._prev_state = area
16
17        return
18
19        print(self._mutate_on_diff(self._prev_state, area),
20              end="",
21              flush=True)
22        self._prev_state = area
23
24    def _mutate_on_diff(self, area1: Area, area2: Area) -> str:
25        """Find the difference between 2 strings and return a string which
26        describes how the first one should change to become the second. Assume
27        that both strings have the same dimensions. Expected strings are
28        unnecessarily bloated for this sole reason (a lot of whitespaces can be
29        followed by '\n') """
```

```

30     mutater_str = ""
31
32     # are the char differences consecutive
33     streak = False
34
35     for line_count, line1 in enumerate(area1.char_area):
36         for char_count, char1 in enumerate(line1):
37             char2 = area2.char_area[line_count][char_count]
38             if char1 != char2:
39                 if streak is False:
40                     # move cursor
41                     mutater_str += self._move_cursor(row=line_count,
42                                                         column=char_count)
43                     streak = True
44
45                     mutater_str += char2
46                 else:
47                     streak = False
48
49     return mutater_str
50
51     def _move_cursor(self, row: int, column: int) -> str:
52         """Return an ANSI code which moves the string to the specified
53         coordinates."""
54         return f"\x1b[{{row}};{{column}}f"
55
56     ... # methods omitted

```

### 3.8 Персонализиране на компоненти

Всеки компонент може да бъде стилизиран посредством класа **Style**. Това най-често се случва, като се подаде символен низ, на метода **fromstr**. Символният низ се обработва с регулярен израз (regex) и превръща в композиция от следните подстилове:

- **AreaInfo**
- **CompositorInfo**
- **TextInfo**

Валиден символен низ следва следния формат: 'property=value, ... '

### 3.9 Издаване и обработка на събития

Текущият проект разпознава натискането на единични клавиши, комбинация от клавиши, когато е поддържана емуляция на VT100 терминал, и събития от мишка в xterm терминални емулатори.

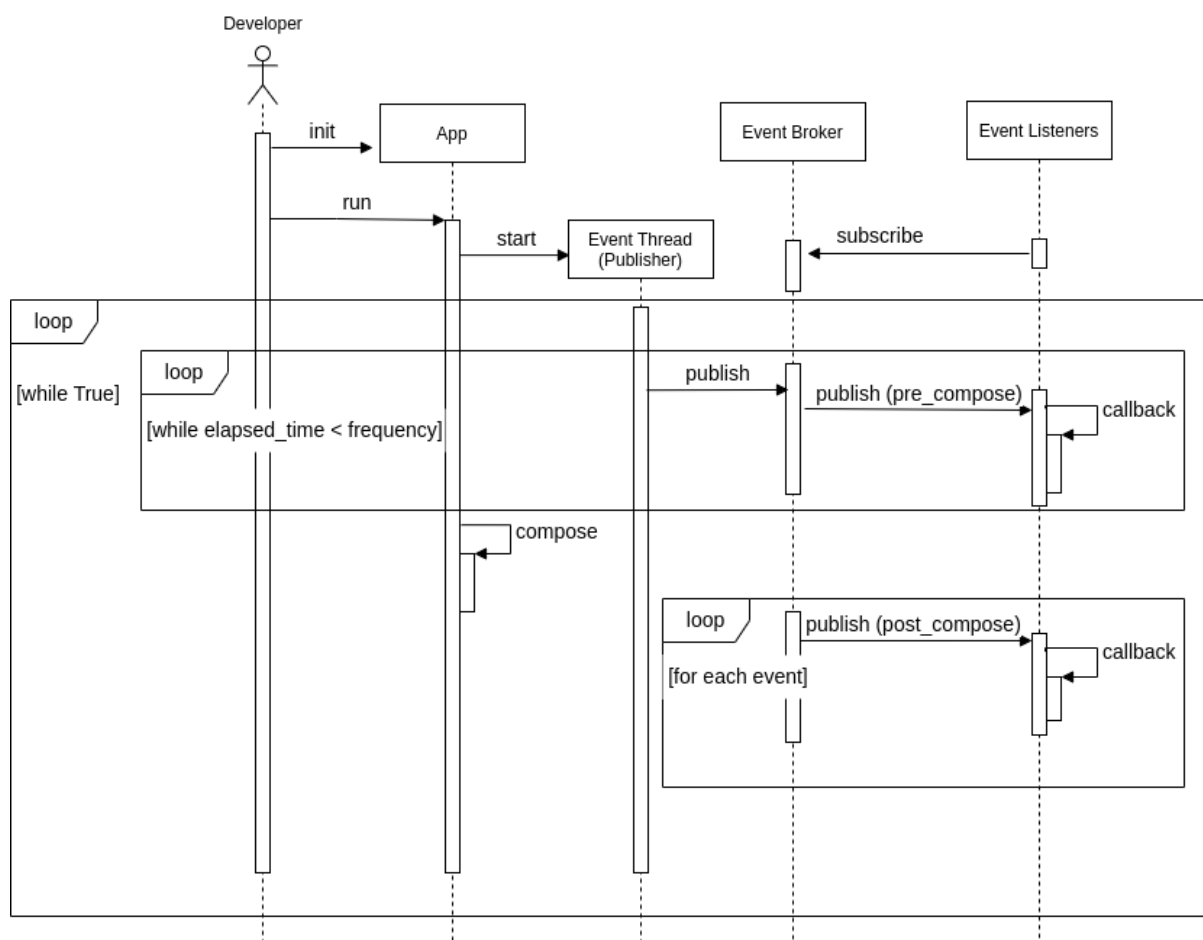
Тези събития се прочитат от стандартния вход (stdin) и обработват с помощта на речници и регулярни изрази. Това се случва в отделна нишка по следния начин:

```
1 class App:
2     """Contains everything necessary for running the application."""
3
4     ... # methods omitted
5
6     def get_events(self) -> None:
7         """Continuously fetch events and put them in the event queue."""
8         from tui.events._mouse_parser import MouseParser
9         from tui.events.key_event import HotkeyEvent
10        from tui.events.keys import ANSI_SEQUENCES_KEYS
11
12        while True:
13            # read from stdin
14            control_code = self.__terminal.read_bytes(_bytes=16).decode()
15            # check if it's a control sequence or a simple key press
16            if len(control_code) == 1:
17                self.event_queue.put(HotkeyEvent(control_code))
18                continue
19
20            # check if it's a mouse control code and add to the event queue
21            try:
22                self.event_queue.put(HotkeyEvent(ANSI_SEQUENCES_KEYS[control_code]))
23            except KeyError:
24                self.event_queue.put(MouseParser.get_mouse_event(control_code))
25
26        ... # methods omitted
```

### 3.10 Абониране за събития

Принципът на работа със събития е много сходен до модела издател-абонат (pub-sub):





Фигура 3.4: Sequence Диаграма на модела на работа със събития

Всеки компонент може да бъде абонат, но за да слуша за събития, компонентът трябва да има фокус. Единствено глобалните абонати - абонати без посочен компонент, не се нуждаят от фокус. Фокусът може да се придобие посредством следния глобален абонат:

```

1  class App:
2      """Contains everything necessary for running the application."""
3      def __init__(
4          self,
5          fps: int = 60, # frames per second
6          # static row and col amount (in case terminal fails)
7          rows: Optional[int] = None,
8          columns: Optional[int] = None,
9      ) -> None:
10
11      ... # code omitted
12
13      def set_focus(event: MouseEvent) -> None:
14          """Set focus to a component, once it's clicked"""

```

```

15         focus_component = self.root.find_component(event.coordinates)
16         if focus_component is None or focus_component == set_focus.prev_component:
17             return
18
19         set_focus.prev_component._focus = False
20         focus_component._focus = True
21         set_focus.prev_component = focus_component
22
23     set_focus.prev_component = None
24     EventBroker.subscribe(
25         event=MouseEventTypes.MOUSE_LEFT_CLICK,
26         subscriber=None,
27         pre_composition=set_focus
28     )
29
30     ... # methods omitted

```

Структурата на **EventListener** (абонат) съдържа следните полета:

- **event** - събитието, за което абонатът ще бъде известяван
- **subscriber** - компонентът, който трябва да има фокус, преди да започне да бъде известяван за събития. Глобалните абонати са имплементирани посредством компонент **subscriber**, който има идентификатор равен на -1.
- **pre\_composition** - Функция, която ще се извика при известяване от събитие. Функцията се извиква преди композирането на компоненти.
- **post\_composition** - Функция, която ще се извика при известяване от събитие, след като е извършена композицията. Не може да се очаква синхроност с **pre\_composition**, защото при възникването на еднакви събития в период по-малък от един кадър, **pre\_composition** ще се извика два пъти преди **post\_composition** да се извика два пъти. За обработването на повечето събития се предпочита **pre\_composition**. Глобалният абонат за показване на курсора на мишка използва именно **post\_composition**.

**EventBroker** класът има ролята на посредник между нишката, която обработва събития, и абонатите, които очакват да бъдат известени.

Той е отговорен за управлението на абонати, като предоставя методи за абониране и отписване. При подаване на събитие към метода **handle** определя кои **pre\_composition** и **post\_composition** функции трябва да бъдат извикани.

```
1  class EventBroker:
2      """Responsible for managing event subscription and event callbacks"""
3
4      # a 'subscriber' with id == -1 is used to signify a global event
5      __global_component = Division(identifier='-1')
6
7      listeners: dict[Event, list[EventListener]] = {}
8      subscribers: dict[Component, list[EventListener]] = {}
9
10     ... # methods omitted
11
12     @staticmethod
13     def handle(
14         event: Event,
15         pre_composit_hook: list[Callback],
16         post_composit_hook: list[Callback]
17     ) -> None:
18         """Find the corresponding listeners for an event and prepare their
19         callbacks. The callbacks are appended to the pre/post composite hooks.
20         """
21
22         def handle_listener(listener: EventListener, event: Event) -> None:
23             if listener.pre_composition is not None:
24                 pre_composit_hook.append(
25                     lambda: listener.pre_composition(event)
26                 )
27             if listener.post_composition is not None:
28                 post_composit_hook.append(
29                     lambda: listener.post_composition(event)
30                 )
31
32         # Convert MouseEvent to _MouseEvent, since that is what listeners are
33         # subscribed to
34         if isinstance(event, MouseEvent):
35             _event = event.event.value
36         else:
37             _event = event
38
39         try:
40             for listener in EventBroker.listeners[_event]:
41                 handle_listener(listener, event)
42
43         except KeyError:
44             # Do nothing if event isn't listened to
45             return
46
47         # handle listeners to all keys
48         try:
49             if not isinstance(event, HotkeyEvent):
50                 return
```

```

51         for listener in EventBroker.listeners[HotkeyEvent(Keys.Any)]:
52             handle_listener(listener, event)
53     except KeyError:
54         # Do nothing if event isn't listened to
55         return

```

В основния цикъл на текущия проект се заемат събития от опашка, в която пише нишката, която чете и обработва събития. **EventBroker** дава достъп до всички абонати на текущото събитие. **pre\_composition** функциите се извършват между всеки кадър. За всеки кадър се композира **root** компонента, който след това се принтира по максимално ефективен начин.

```

1
2 class App:
3     """Contains everything necessary for running the application."""
4
5     ... # methods omitted
6
7     def run(self) -> None:
8         """Start and constantly update the app."""
9         input_thread = threading.Thread(
10             target=self.get_events,
11             args=(),
12             daemon=True
13         )
14
15         input_thread.start()
16
17         try:
18             pre_composit_hook = []
19             post_composit_hook = []
20             timer_start = perf_counter()
21             while True:
22                 while self.event_queue.qsize() > 0:
23                     event = self.event_queue.get()
24                     EventBroker.handle(
25                         event=event,
26                         pre_composit_hook=pre_composit_hook,
27                         post_composit_hook=post_composit_hook
28                     )
29
30                 if perf_counter() - timer_start >= self.frequency:
31                     self.__terminal.print(
32                         str(Compositor.compose(
33                             self.root,
34                             pre_composit=pre_composit_hook,
35                             post_composit=post_composit_hook))
36                     )
37                     timer_start = perf_counter()
38                     pre_composit_hook = []
39                     post_composit_hook = []

```

```
40
41     except KeyboardInterrupt:
42         self.__terminal.disable_mouse()
43         self.__terminal.disable_input()
44     except BaseException as e:
45         self.__terminal.disable_mouse()
46         self.__terminal.disable_input()
47         raise e
```

## Глава 4

# Ръководство за потребителя

### 4.1 Инсталиране на библиотеката

Библиотеката е съвместима с операционната система Linux. Преди да започнете процеса на инсталация ще са Ви нужни следните допълнителни програми:

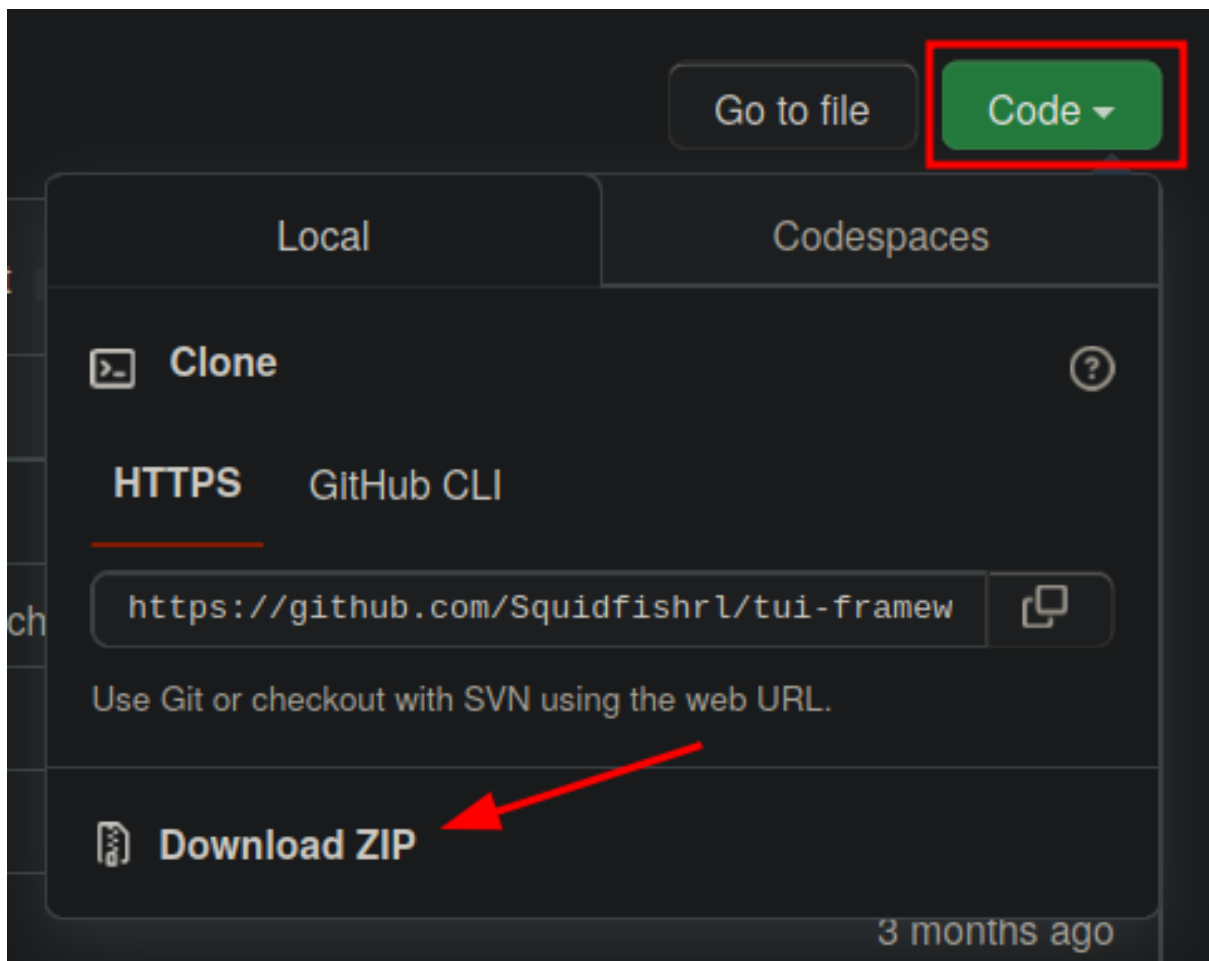
- git
- Python интерпретатор (версия 3.10 или 3.11)

#### 4.1.1 Сдобиване с кода на библиотеката

Кодът на програмата може да бъде изтеглен от GitHub хранилището на проекта <https://github.com/Squidfishrl/tui-framework>. Най-удобният и бърз начин е да копирате хранилището директно:

```
$ git clone https://github.com/Squidfishrl/tui-framework
```

Като алтернатива може да изтеглите кода под формата на zip архив (фиг. 4.1):



Фигура 4.1: Теглене на кода под формата на zip архив

#### 4.1.2 Инсталиране на библиотеката

След като сте се сдобили с кода може да преминете към инсталацията на библиотеката. Стъпките за това са следните:

1. (по желание) Създайте виртуална среда, за да изолирате библиотеката от общосистемните пакети

```
$ mkdir ./venv  
$ python3 -m venv ./venv  
$ source ./venv/bin/activate
```

2. Инсталирайте пакета и неговите зависимости

```
$ pip install -e ./tui-framework  
$ pip install -r requirements.txt
```

## 4.2 Ръководство за обучение

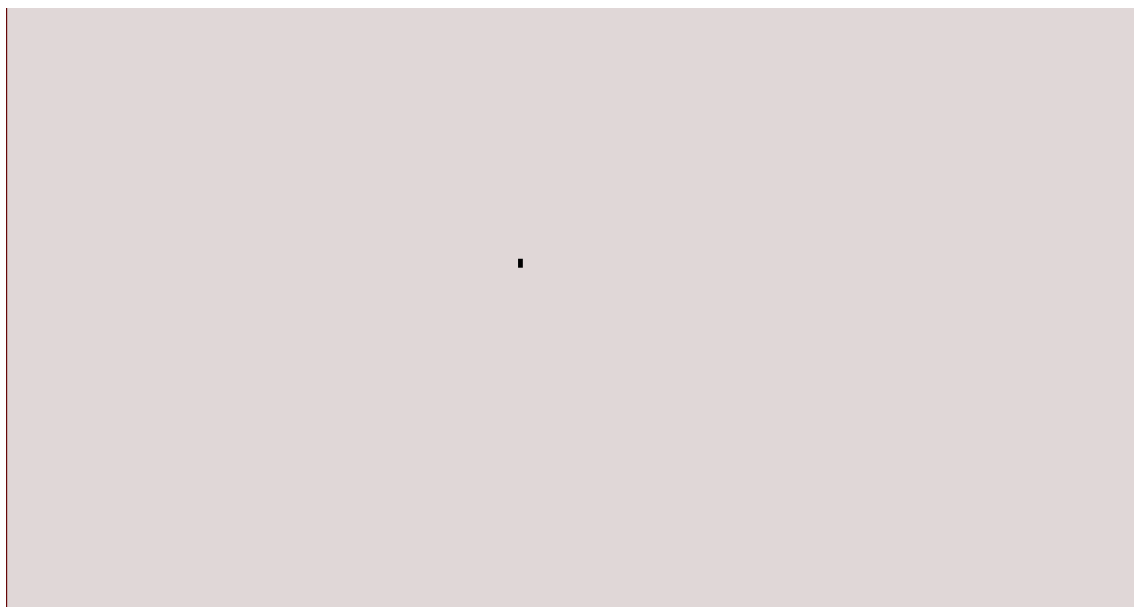
Най-добрият начин да се научиш да програмираш е, като седнеш да програмираш. В този раздел от ръководството ще имплементираме проста програма.

### 4.2.1 Hello World!

Възможно най-простата програма, която можем да напишем е следната: създаваме приложението, използвайки класа **APP** и го стартираме с **run** метода.

```
1 from tui.app import App
2
3 app = App()
4 app.run()
```

Когато го пуснем, виждаме празен екран с изключение на черен правоъгълник, който следва курсора на мишката ни. (фиг. 4.2):



Фигура 4.2:

Следващата ни стъпка е да принтираме текста "Hello, World!" на екрана. Това се случва посредством класа **Widget**, а именно **Label** (надпис).

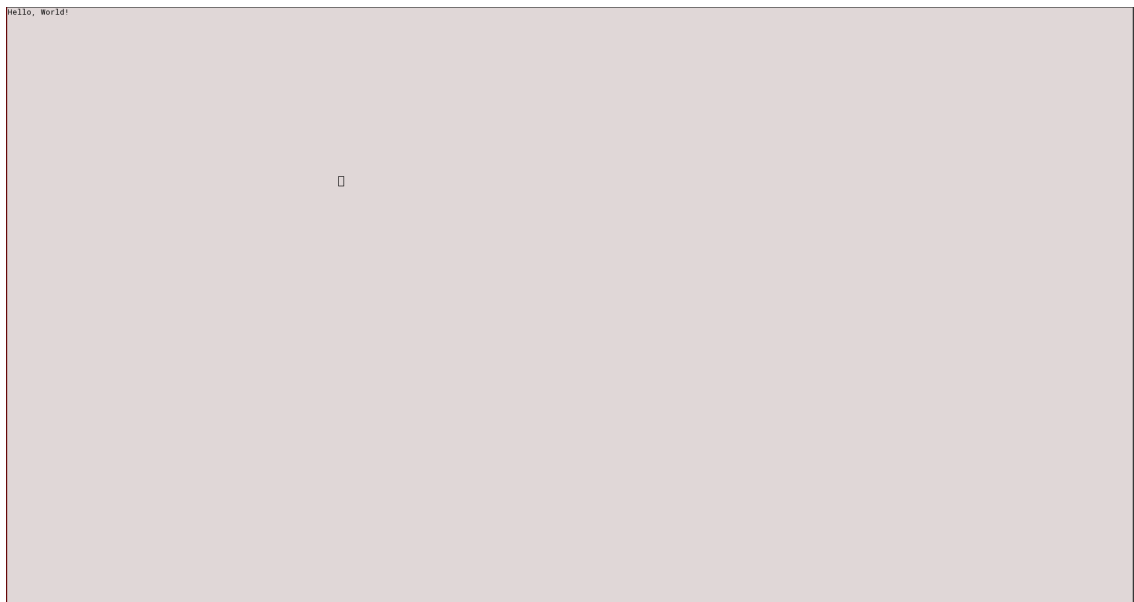


```

1 from tui.app import App
2 from tui.components.label import Label
3
4 app = App()
5 app.root.append_child(
6     Label("Hello, World!", style="rows=1, columns=13")
7 )
8 app.run()

```

Както виждаме, за да определим размера на компонента **Label**, трябва да укажем неговия размер. Това важи за всички компоненти в текущата версия на проекта (1.0.0).



Фигура 4.3:

За да преместим текста (фиг. 4.3), в средата на екрана, трябва да разгледаме подпакета **styles**. Най-лесният начин е да променим размера на надписа и центрираме текста в средата. Друга възможност е да използваме **padding**:

```

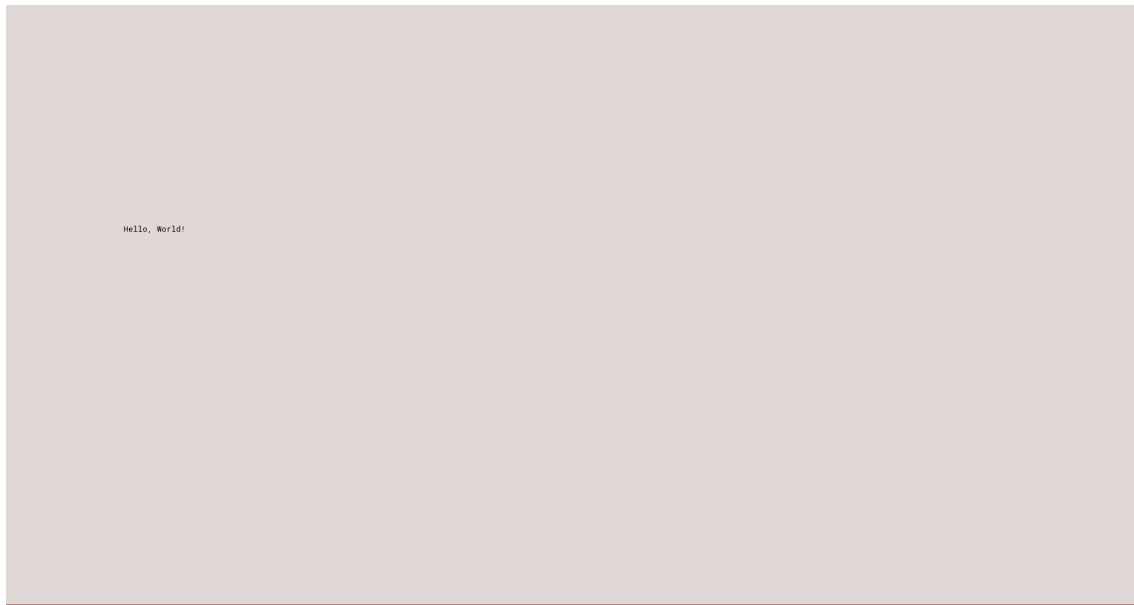
1 from tui.app import App
2 from tui.components.division import Division
3 from tui.components.label import Label
4
5 app = App()
6
7 app.root.set_style("padding", 25)

```

```

8  app.root.append_child(
9      Label("Hello, World!", style="rows=1, columns=13")
10 )
11
12 app.run()

```



Фигура 4.4:

Може да използваме метода **add\_border**, за да добавим рамка към КОМПОНЕНТИТЕ:

```

1  app = App()
2  root = app.root
3
4  root.set_style("margin", 2)
5
6  root.append_child(
7      Label("Hello, World!", identifier="lbl", style=f"\
8          rows={root.area.rows - 2},\
9          columns={root.area.columns - 2},\
10         text_align=center,\
11         vertical_align=center,\
12         margin_top={25},\
13         margin_bottom={25},\
14         margin_left={43},\
15         margin_right={43}")
16 )
17
18 root.children.get_by_id("lbl").add_border(DefaultBorder)
19 root.add_border(DefaultBorder)
20 app.run()

```



Фигура 4.5:

Вече надписът е центриран и заедно с **root** компонентът има рамка.

Библиотеката има добре описана документация и може да научите много ако разгледате някои от файловете като **components/widget.py**, **components/container.py** и **app.py**. Същевременно, може да разгледате и директорията **demo**, в която са събрани други примери.

## Глава 5

# Заклучение

Настоящата дипломна работа реализира базов вариант на библиотека за създаване на текстов потребителски интерфейс. Библиотеката бе написана на Python, като е достъпна за платформата Linux. Тя представя абстрактен интерфейс, посредством който могат да се изразят и опишат логически компоненти, което позволява изграждането на приложения с по-богата функционалност. Библиотеката успешно разпознава и категоризира събития от клавиатура и мишка, предоставя механизъм за стилизиране на елементи и оптимизира принтирането в терминал.

Възможностите за бъдещо развитие на проекта са многобройни. Едни от най-важните функционалности са динамично уразмеряване и стилизиране на компоненти. Други полезни насоки са поддръжка на цветове, скролбар за компоненти, припокриване на компоненти (3-то измерение), диагностични записи, поддръжка на повече терминални емулятори, markdown език, който описва компонентното дърво, повече вградени компоненти и други.

# Библиография

- [1] Linus Åkesson. *The TTY demystified*. URL: <http://www.linusakesson.net/programming/tty/index.php?tty>.
- [2] Erich Gamma и др. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley, 1994.
- [3] Paul Jansen. *TIOBE Programming Community index*. URL: <https://www.tiobe.com/tiobe-index/>.
- [4] Tom Preston-Werner. *Semantic Versioning 2.0.0*. URL: <https://semver.org>.
- [5] Guido Van Rossum. *Comparing Python to Other Languages*. URL: <https://www.python.org/doc/essays/comparisons/>.