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 2

ТЕХНОЛОГИЧНО УЧИЛИЩЕ ЕЛЕКТРОННИ СИСТЕМИ
към ТЕХНИЧЕСКИ УНИВЕРСИТЕТ - СОФИЯ

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/

Съдържание

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

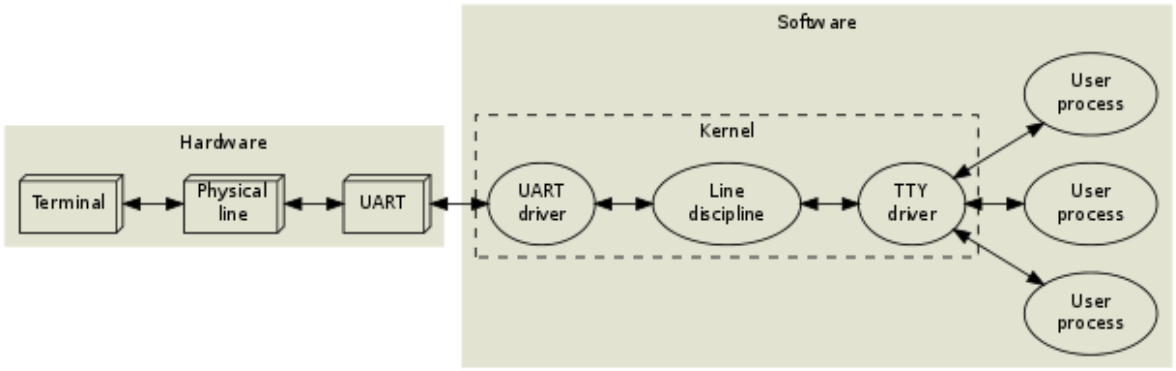
Проучване

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

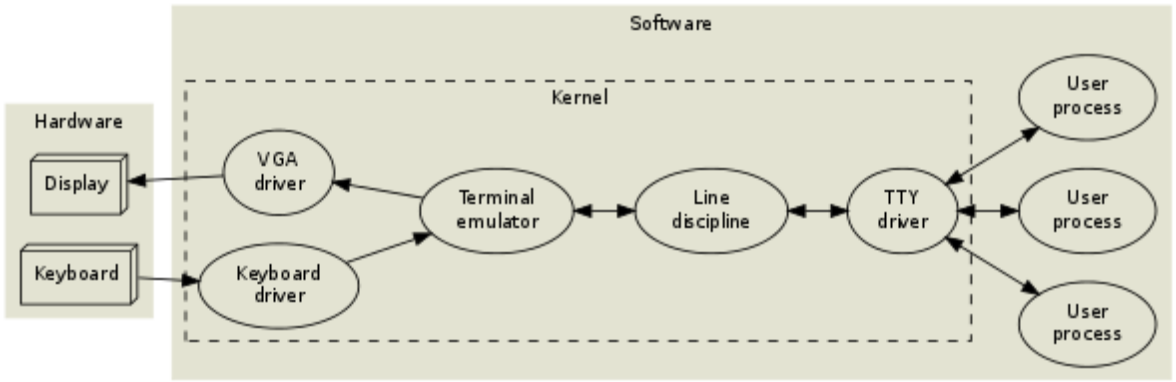
Този вид терминал се нарича така, поради сходността си с телеграфния апарат. На фигура 1.1 е показана блокова схема на принцип на работа на TTY терминал. Потребителят пише на хардуерния терминал, който е свързан към UART (универсален асинхронен приемник и предавател) на компютъра. Операционната система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



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 курсура, 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 разпознавча 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

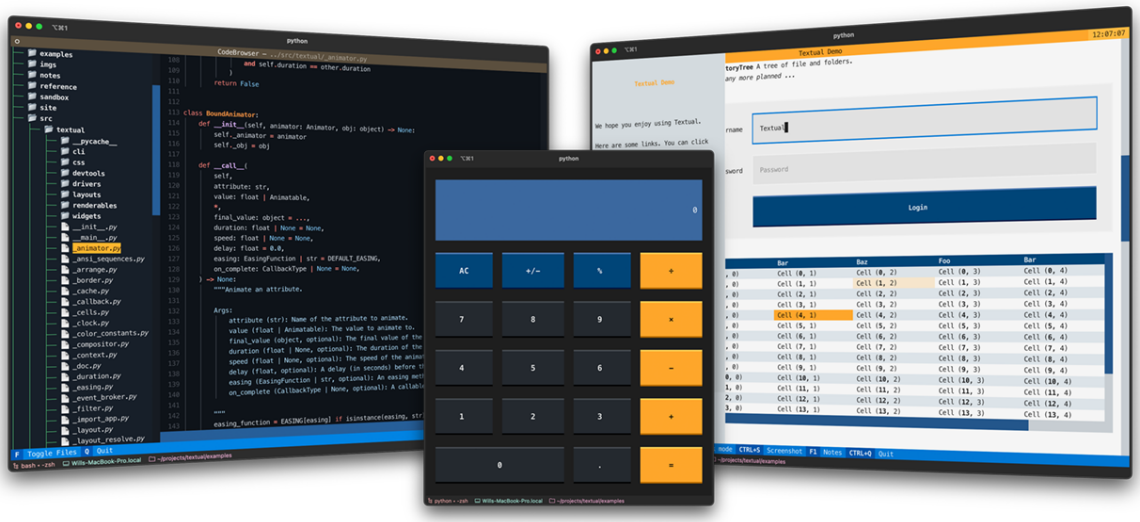
Фигура 1.3: Midnight Commander - файлов мениджър написани с ncurses

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 потреителски инерфейс
  - 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 C. Поради синтактичната си близкост до английския език и гореизброените причини Python бързо набира популярност.

Python е обектно ориентиран и строго типзиран език от високо ниво, като типовете на данните се определят по време на изпълнението. Поради скриптовия си характер, липсата на необходмиост 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 помалко. Те са 3-5 пъти по-кратки от еквивалентите на Java и 5-10 пъти от тези на C++.

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.

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 комютри.

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 допълните практики при писане на код, които се различават.

2.2.3 Компонентно тестове

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 подпоага сътрудничеството на големи екипи от хроа.

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 елеметн 'въвеждане'
- compositor/label - разработка на елемент 'надпис'
- compositor - разработка на алгоритъм за коммпозиране на елементи
- events - работа със събития от периферни устройства
- styles - разработка на настройки за персонализиране на елементи

2.2.5 Непрекъсната интерграция

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

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 платфомата GitHub. Настоящата документация бе написана с помощта на LaTeX.

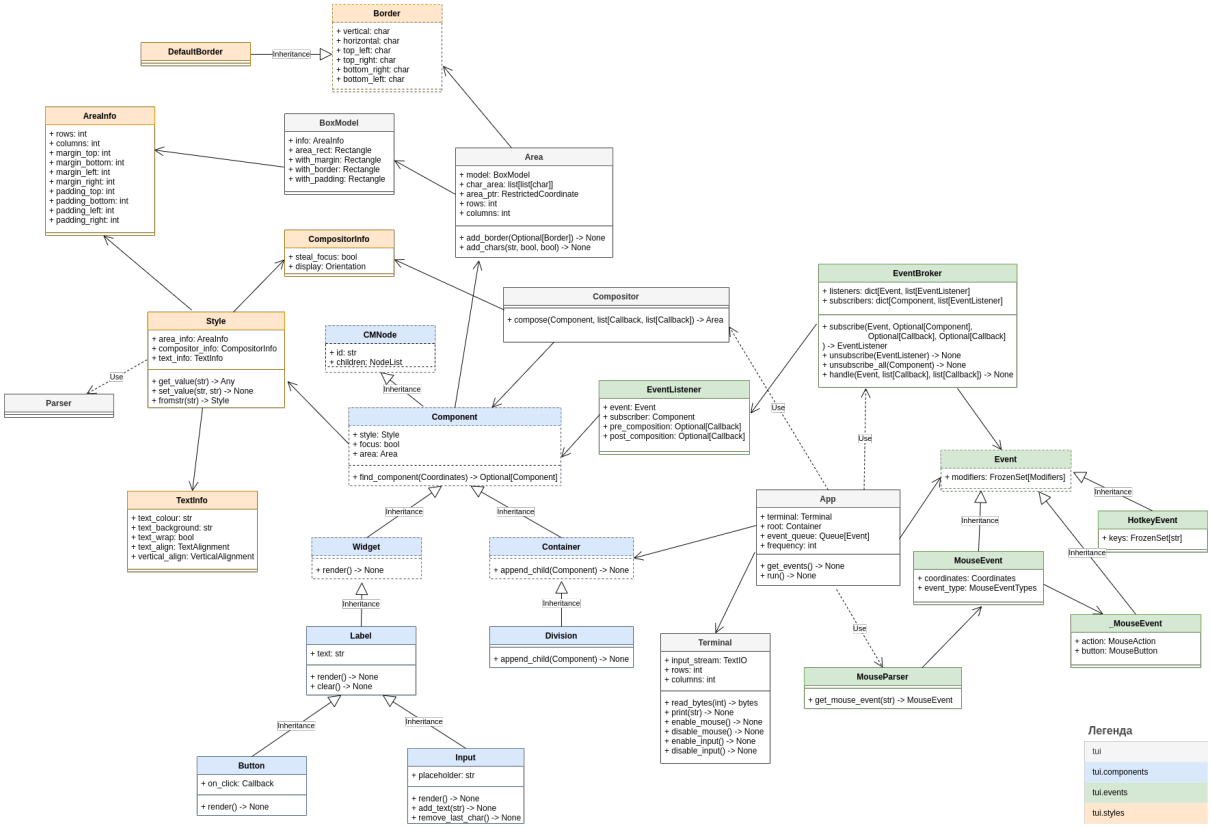
Глава 3

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

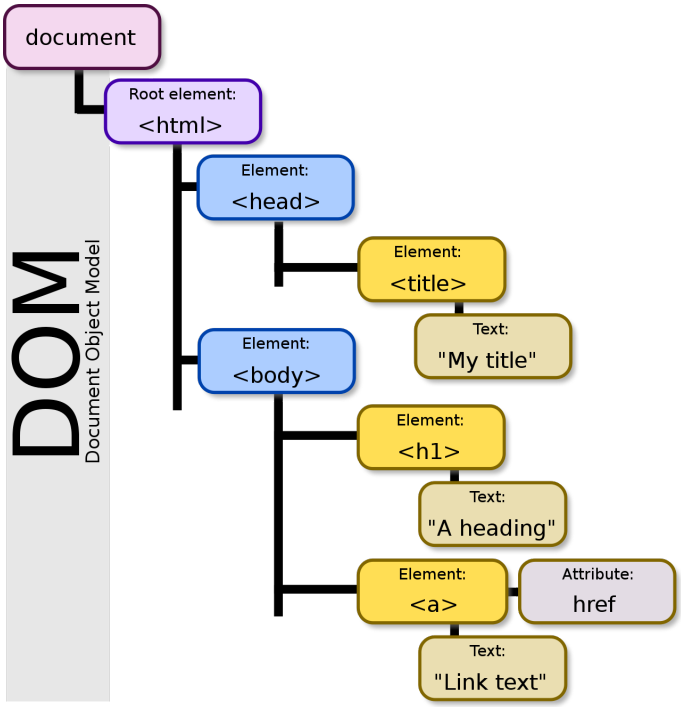
Фаловете **_coordinates.py** и **_parser.py** са с общо предназначение.

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.

```python
class NodeList(Sequence):
    """Responsible for providing a structure for accessing an ordered component list"""

    def __init__(self) -> None:
        self._nodes_list: list[Component] = []  # preserve order
        # components with no id aren't in the dict
        self._nodes_dict: dict[str, Component] = {}  # fast search by index

    def __len__(self) -> int:
        return len(self._nodes_list)

    def __getitem__(self, index: int | slice) -> Component | list[Component]:
        return self._nodes_list[index]

    def __setitem__(self, index: int, new_component: Component) -> None:
        prev_component = self._nodes_list[index]
        self._nodes_list[index] = new_component

        if prev_component.id is not None:
            self._nodes_dict.pop(prev_component.id)
        if new_component.id is not None:
            self._nodes_dict[new_component.id] = new_component

    def __contains__(self, component: Component) -> bool:
        return component in self._nodes_list

    def get_by_id(self, identifier: str) -> Optional[Component]:
        """Get component (search by id)"""
        return self._nodes_dict.get(identifier)

    def append(self, component: Component) -> None:
        """Add component at the end of the list"""
        if component in self._nodes_list:
            raise ValueError("Component already exists")

        if component.id in self._nodes_dict:
            raise KeyError("Id already exists")

        self._nodes_list.append(component)

        if component.id is not None:
            self._nodes_dict[component.id] = component

    def pop(self, index: int) -> Component
        """Remove component (search by id)"""
        component = self._nodes_list.pop(index)

        if component.id is not None:
            self._nodes_dict.pop(component.id)

        return component

    def remove(self, component: Component) -> None:
        """Remove component"""
        self._nodes_list.remove(component)
        if component.id is not None:
            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.

```python
class Area:
    """
    Responsible for rendering how a component should look.
    This includes the box model and any dynamic changes that may happen within it
    """
    def __init__(self, area_info: AreaInfo, border: Optional[Border] = None) -> None:
        self.model = BoxModel(info=area_info)
        # char_area is where every change is reflected
        # rows x columns
        self.char_area: list[list[str]] = [
                [' ' for _ in range(self.columns)]
                for _ in range(self.rows)
            ]

        # the area pointer is used for easier navigating and writing to
        # char_area as it automatically keeps track of a lot of variables
        self.area_ptr = RestrictedCoordinates(
                    _row=0,
                    _column=0,
                    _restriction=self.model.with_padding,
                    relative=True
                )

        self.add_border(border)

    def __str__(self):
        return "\n".join(["".join(list(row)) for row in self.char_area])

    def add_border(self, border: Optional[Border]) -> None:
        """Apply border to the area. This will shrink the space the component
        can use depending on the border's size"""

        ... # code omitted

    def add_chars(
            self,
            string: str,  # string to be added to the area
            column_preserve: bool = False,  # should new line start at the current area pointer column
            ptr_preserve: bool = True  # reset area pointer to initial position
    ) -> None:
        """Write to char_area starting from area_ptr coordinates.
        Any '\n' in the string translate to a row increment.
        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)

```python
class Container(Component):
    """
    Abstract component that's responsible for containing and organizing other components.
    `@property def children(self)` should be implemented in child classes
    """

    def __init__(
            self,
            *children: Component,  # Child components
            style: str | Style = Style(),  # Style properties for the component
            identifier: Optional[str] = None,  # Unique identifier
    ) -> None:
        super().__init__(identifier=identifier, style=style)

        for child in children:
            # Components that inherit Container could pass *children to
            # super().__init__() which is seen as an empty tuple here
            if isinstance(child, tuple):
                continue

            self.append_child(child)

    def append_child(self, component: Component):
        """Add a child at the end of the component list of the container.
        You can manipulate this by adding other components for positioning or composition"""
        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 изъплняването на конкретна функционалност. Примери за уиджети са класовете **Label** (надпис) и **Button** (бутон).

```python
class Widget(Component):
    """An abstract component that can't hold other components and can override event behaviour."""
    def __init__(
            self,
            style: str | Style = Style(),  # Style properties for the component
            identifier: Optional[str] = None,  # Unique identifier
    ) -> None:
        super().__init__(identifier=identifier, style=style)

    @abstractmethod
    def _render_to_area(self) -> None:
        """Render the component's content to its area"""

    @property
    def children(self) -> list:
        """Widgets can't have child components.
        Return an empty list for compatibility"""
        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

```
class Compositor:
    """Responsible for compositing components into a single area"""

    @staticmethod
    def compose(
            root: Component,  # the component which's area is being composed
            pre_composit: list[Callback],
            post_composit: list[Callback]
    ) -> Area:
        """Compose a component with its children components recursively. Run
        pre-compose and post-compose hooks"""
        for callback in pre_composit:
            callback()

        # set root rect mapping to itself
        root._rect_mapping = root.area.model.area_rect
        new_area = Compositor._compose(root=root)

        for callback in post_composit:
            callback()

        return new_area

    @staticmethod
    def _compose(root: Component) -> Area:
        """Compose a component with its children components recursively

        root: the component's area that's being composed
        next_component: rectangular area which the next component should occupy
        """

        new_area = copy.deepcopy(root.area)

        # the rectangle the previous child was in
        prev_rect: Optional[Rectangle] = None

        for child in root.children:
            try:
                prev_rect = Compositor._get_next_rectangle(
                        parent=root,
                        prev_rect=prev_rect,
                        component=child
                    )
                child._row_mapper = prev_rect
            except CoordinateError as exc:
                raise InsufficientAreaError(
                        "Component area isn't large enough"
                    ) from exc

            # recursion ends when there are no more children
            child_area = Compositor._compose(child)
            new_area.area_ptr.row = (
                prev_rect.top_left.row
                + new_area.model.with_padding.top_left.row
            )
            new_area.area_ptr.column = (
                prev_rect.top_left.column
```

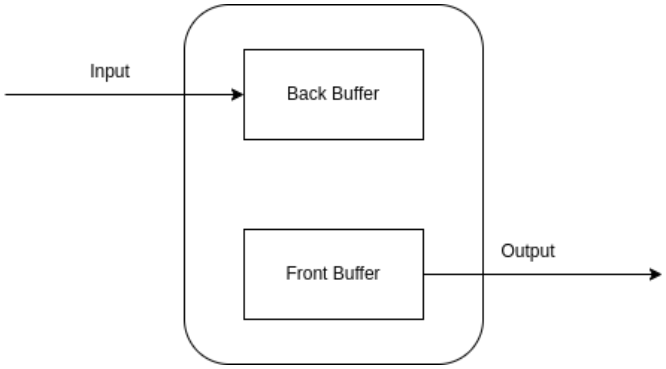
```
                + new_area.model.with_padding.top_left.column
            )

            # draw child component
            try:
                new_area.add_chars(str(child_area), column_preserve=True)
            except IndexError as exc:
                raise InsufficientAreaError(
                        "Component area isn't large enough"
                    ) from exc

        return new_area

    @staticmethod
    def _get_next_rectangle(
            parent: Component,  # the parent component this one will reside in
            component: Component,  # the component calculations are done for
            prev_rect: Optional[Rectangle] = None
    ) -> Rectangle:
        """Helper function that decides where components are placed when
        compositing"""
        if parent.style.compositor_info.display == Orientation.INLINE:
            return Compositor.__get_next_rectangle_inline(
                    parent=parent,
                    component=component,
                    prev_rect=prev_rect
                )

        return Compositor.__get_next_rectangle_block(
                parent=parent,
                component=component,
                prev_rect=prev_rect
            )

    @staticmethod
    def __get_next_rectangle_inline(
            parent: Component,  # the parent component this one will reside in
            component: Component,  # the component calculations are done for
            # rectangle for the previous component
            prev_rect: Optional[Rectangle] = None
    ) -> Rectangle:
        """Helper function for inline compositing. Returns the area the next
        component should be placed in

        Default prev_rectangle is a rectangle outside of the parent's area
        that's derived to place the component in the top left of the parent
        component's area:
            top_left:  row = 0 && column <= -1
            bottom_right:  row >= 0 && column = -1
        """
        if prev_rect is None:
            prev_rect = Rectangle(
                    top_left=Coordinates(_row=0, _column=-1),
                    bottom_right=Coordinates(_row=0, _column=-1)
                )

        return Rectangle(
                top_left=Coordinates(
                        _row=prev_rect.top_left.row,
```

```
                        _column=prev_rect.bottom_right.column + 1
                    ),
                bottom_right=Coordinates(
                        _row=parent.area.rows - 1,
                        _column=prev_rect.bottom_right.column +
                        component.area.columns
                    )
            )

    @staticmethod
    def __get_next_rectangle_block(
            parent: Component,  # the parent component this one will reside in
            component: Component,  # the component calculations are done for
            # rectangle for the previous component
            prev_rect: Optional[Rectangle] = None
    ) -> Rectangle:
        """Helper function for block compositing. Returns the area the next
        component should be placed in

        Default prev_rectangle is a rectangle outside of the parent's area
        that's derived to place the component in the top left of the parent
        component's area:
            top_left:  row = -1 && column <= 0
            bottom_right:  row >= -1 && column = 0
        """
        if prev_rect is None:
            prev_rect = Rectangle(
                    top_left=Coordinates(_row=-1, _column=0),
                    bottom_right=Coordinates(_row=-1, _column=0)
                )

        return Rectangle(
                top_left=Coordinates(
                        _row=prev_rect.bottom_right.row + 1,
                        _column=prev_rect.top_left.column
                    ),
                bottom_right=Coordinates(
                        _row=prev_rect.bottom_right.row + component.area.rows,
                        _column=parent.area.columns - 1
                    )
            )
```

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.

```
class Terminal:
    """Responsible for providing an interface to the terminal the program is being ran in"""

    def __init__(self, input_stream: TextIO = stdin) -> None:
        ... # code omitted
        self._prev_state: Optional[str] = None
        ... # code omitted

    def print(self, string: str) -> None:
        """Print a string to the terminal. Overwrites content in the terminal
        for higher response time and no stuttering."""
        # handle first print
        if self._prev_state is None:
            print(string, end="", flush=True)
            self._prev_state = string
            return

        print(self._mutate_on_diff(self._prev_state, string), end="", flush=True)
        self._prev_state = string

    def _mutate_on_diff(self, str1: str, str2: str) -> str:
        """Find the difference between 2 strings and return a string which
        describes how the first one should change to become the second. Assume
        that both strings have the same dimensions. Expected strings are
        unnecessarily bloated for this sole reason (a lot of whitespaces can be
        followed by '\n') """
        mutate_str = ""
        # -1 since it will be incremented for the first char and we want
        # (0,0) to be top left
```

```
        column = -1
        row = 0

        # are the char differences consecutive
        streak = False

        for count, char1 in enumerate(str1):
            char2 = str2[count]

            if char2 == '\n':
                column = 0
                row += 1
            else:
                column += 1

            if char1 != char2:
                if streak is False:
                    # move cursor
                    mutate_str += self._move_cursor(row=row, column=column)
                    streak = True

                mutate_str += char2
            else:
                streak = False

        return mutate_str

    def _move_cursor(self, row: int, column: int) -> str:
        """Return an ANSI code which moves the string to the specified coordinates."""
        return f"\x1b[{row};{column}f"

    ... # 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

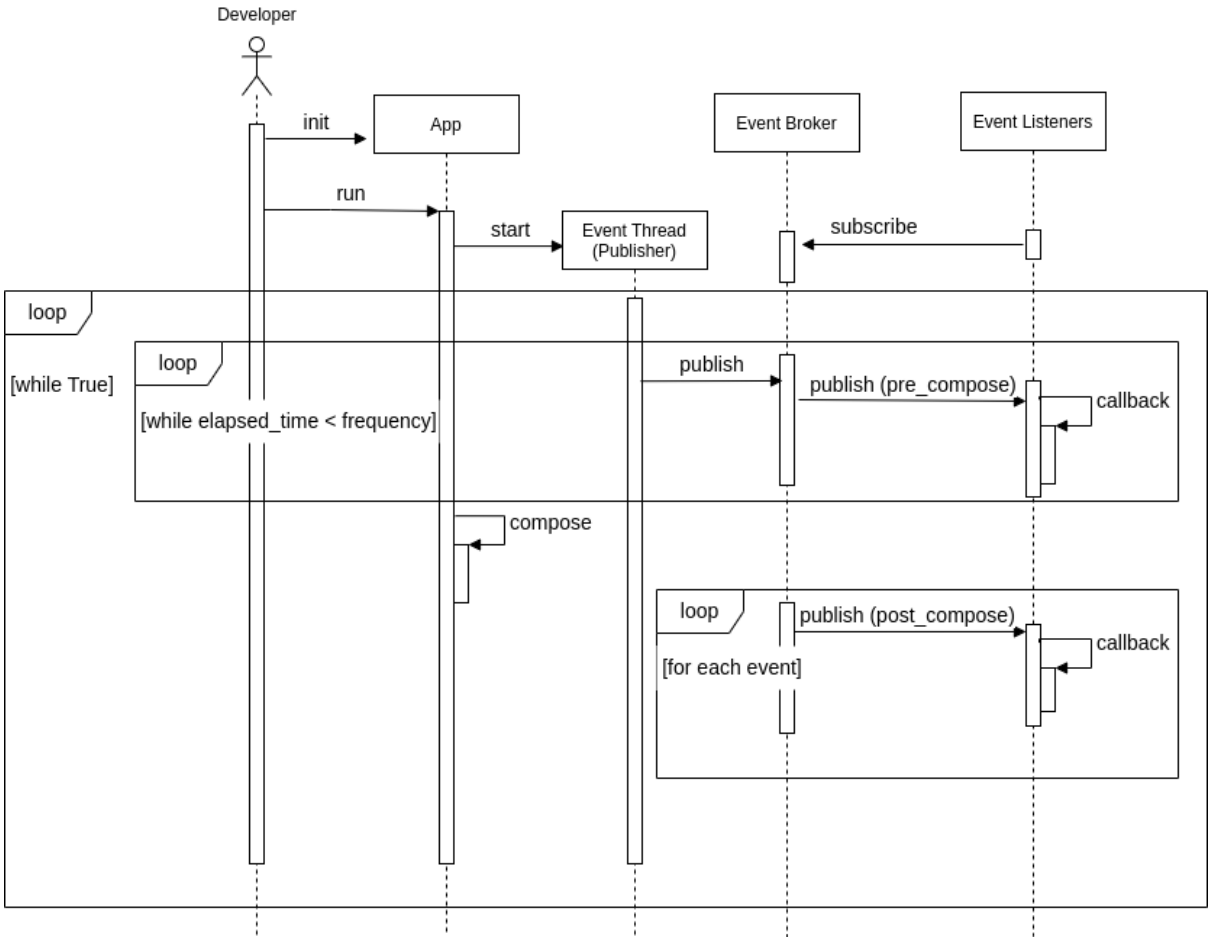
Текушият проект разпознава натискането на единични клавиши, комбинация от клавиши, когато е поддържана емулация на VT100 терминал, и събития от мишка в xterm терминални емулатори.

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:

```python
class App:
    """Contains everything necessary for running the application."""

    ... # methods omitted

    def get_events(self) -> None:
        """Continuously fetch events and put them in the event queue."""
        from tui.events._mouse_parser import MouseParser
        from tui.events.key_event import HotkeyEvent
        from tui.events.keys import ANSI_SEQUENCES_KEYS

        while True:
            # read from stdin
            control_code = self.__terminal.read_bytes(_bytes=16).decode()
            # check if it's a control sequence or a simple key press
            if len(control_code) == 1:
                self.event_queue.put(HotkeyEvent(control_code))
                continue

            # check if it's a mouse control code and add to the event queue
            try:
                self.event_queue.put(HotkeyEvent(ANSI_SEQUENCES_KEYS[control_code]))
            except KeyError:
                self.event_queue.put(MouseParser.get_mouse_event(control_code))

    ... # 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:

```
class App:
    """Contains everything necessary for running the application."""
    def __init__(
            self,
            fps: int = 60,  # frames per second
            # static row and col amount (in case terminal fails)
            rows: Optional[int] = None,
            columns: Optional[int] = None,
    ) -> None:

        ... # code omitted

        def set_focus(event: MouseEvent) -> None:
            """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_compostion.

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.

```
class EventBroker:
    """Responsible for managing event subscription and event callbacks"""

    # a 'subscriber' with id ==  -1 is used to signify a global event
    __global_component = Division(identifier='-1')

    listeners: dict[Event, list[EventListener]] = {}
    subscribers: dict[Component, list[EventListener]] = {}

    ... # methods omitted

    @staticmethod
    def handle(
            event: Event,
            pre_composit_hook: list[Callback],
            post_composit_hook: list[Callback]
    ) -> None:
        """Find the corresponding listeners for an event and prepare their
        callbacks. The callbacks are appended to the pre/post composite hooks.
        """

        def handle_listener(listener: EventListener, event: Event) -> None:
            if listener.pre_composition is not None:
                pre_composit_hook.append(
                        lambda: listener.pre_composition(event)
                    )
            if listener.post_composition is not None:
                post_composit_hook.append(
                        lambda: listener.post_composition(event)
                    )

        # Convert MouseEvent to _MouseEvent, since that is what listeners are
        # subscribed to
        if isinstance(event, MouseEvent):
            _event = event.event.value
        else:
            _event = event

        try:
            for listener in EventBroker.listeners[_event]:
                handle_listener(listener, event)

        except KeyError:
            # Do nothing if event isn't listened to
            return

        # handle listeners to all keys
        try:
            if not isinstance(event, HotkeyEvent):
                return
```

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

```

class App:
    """Contains everything necessary for running the application."""

    ... # methods omitted

    def run(self) -> None:
        """Start and constantly update the app."""
        input_thread = threading.Thread(
                target=self.get_events,
                args=(),
                daemon=True
            )

        input_thread.start()

        try:
            pre_composit_hook = []
            post_composit_hook = []
            timer_start = perf_counter()
            while True:
                while self.event_queue.qsize() > 0:
                    event = self.event_queue.get()
                    EventBroker.handle(
                            event=event,
                            pre_composit_hook=pre_composit_hook,
                            post_composit_hook=post_composit_hook
                    )

                if perf_counter() - timer_start >= self.frequency:
                    self.__terminal.print(
                            str(Compositor.compose(
                                    self.root,
                                    pre_composit=pre_composit_hook,
                                    post_composit=post_composit_hook))
                        )
                    timer_start = perf_counter()
                    pre_composit_hook = []
                    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:

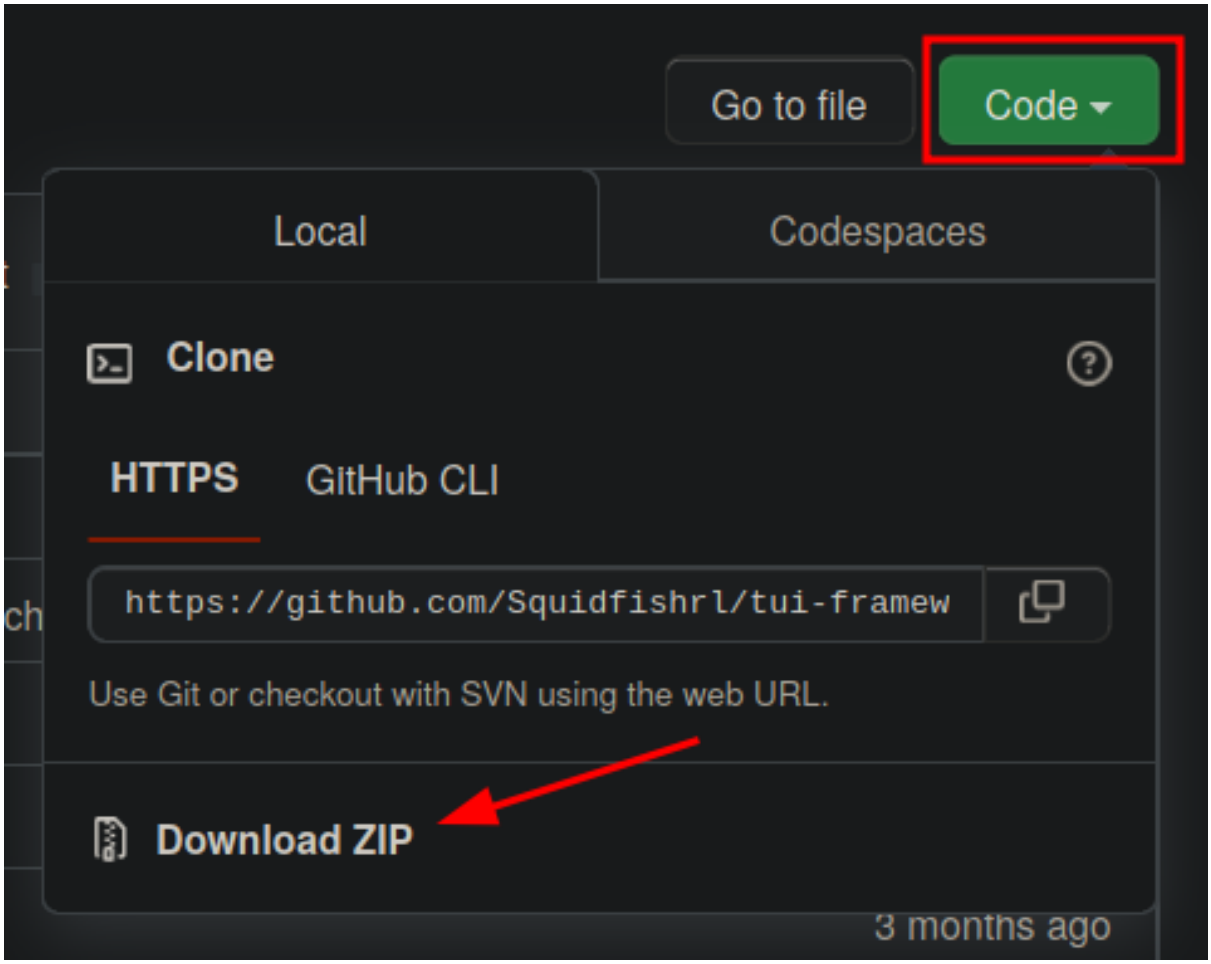
- git
- Python интерпретатор (версия 3.10 или 3.11)

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. Влезте в главната директория на проекта

```
$ cd tui-framework
```

2. (по желание) Създайте виртуална среда, за да изолирате пакета

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

WIP

Глава 5

Заключение

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 представлява 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 стилиране на компоненти. Други полезни наскои са поддръжка на цветове, скролбар за компоненти, припокриване на компоненти (3-то измерение), диагностични записи, поддръжка на повече терминални емулатори, markdown език, които описва компонентото дърво, повече вградени компоненти и други.