



МИНОБРНАУКИ РОССИИ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«МИРЭА – Российский технологический университет»
РТУ МИРЭА

Институт информационных технологий
Кафедра корпоративных информационных систем

КУРСОВАЯ РАБОТА

по дисциплине

Структура и алгоритмы обработки данных

Тема курсовой работы: «Создание декларативного языка для описания GUI с автоматической генерацией кода для GUI-библиотек»

Студент группы ИКБО-07-18

Басыров Сергей Амирович

(подпись студента)

Руководитель курсовой работы

Советов Пётр Николаевич

(подпись руководителя)

Работа представлена к защите

«20» декабря 2019 г

Допущен к защите

«23» декабря 2019 г.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ	3
ОСНОВНАЯ ЧАСТЬ.....	5
1 Декларативный подход и существующее решение	5
2 GUI-библиотека Tkinter	7
2.1 Пример графического интерфейса	8
2.2 Описание компонентов.....	9
2.3 Менеджеры размещения компонентов	9
3 Декларативный язык описания графического интерфейса.....	11
3.1 Синтаксис языка.....	11
3.2 Структура кода	12
3.3 Ключевые слова	12
3.4 Расширенные формы Бэкуса-Наура.....	14
4 Проектирование и реализация транслятора	16
4.1 Препроцессор	17
4.2 Лексический анализатор.....	18
4.3 Синтаксический анализатор.....	19
4.4 Генератор	22
4.5 Постпроцессор.....	24
5 Проектирование и реализация приложения	25
6 Тестирование	26
7 Пример использования языка	268
ЗАКЛЮЧЕНИЕ	29
СПИСОК ИСПОЛЬЗУЕМЫХ ИСТОЧНИКОВ	30
ПРИЛОЖЕНИЕ А Описание свойств компонентов.....	32
ПРИЛОЖЕНИЕ Б Исходный код unit-тестов	34
ПРИЛОЖЕНИЕ В Исходный код приложения	36

ВВЕДЕНИЕ

Цель курсового проекта – создание декларативного языка для описания графического интерфейса с автоматической генерацией кода под GUI-библиотеку.

Актуальность проекта заключается в том, что для научных сотрудников выбор больших и сложных библиотек позволяющие создавать графический интерфейс является не практичным. Так как для их задач нет необходимости в создании сложных графических интерфейсов с множеством форм и различных компонентов.

Создаваемый язык позволит создавать графические интерфейсы для скриптовых программ, написанных на языке Python, в декларативном стиле и получать сгенерированный код под GUI библиотеку, который сразу можно будет запустить. Такой подход обеспечит понятность написанного кода и скорость реализации графического интерфейса.

Объектом исследования является процесс создания графического интерфейса для скриптовых программ, написанных на языке Python.

Предметом исследования является декларативный подход в создании графического интерфейса.

Результатом курсового проекта будет декларативный язык, позволяющий быстро создавать графический интерфейс для скриптовых программ на языке Python.

Основными преимуществами языка будут:

- 1) простота синтаксиса;
- 2) поддержка стандартных компонентов графического интерфейса;
- 3) возможность вывода пользовательских данных;
- 4) наличие документации;
- 5) кроссплатформенность.

Основными задачами курсового проекта являются:

- 1) исследование декларативного подхода и существующего аналога;
- 2) создание декларативного языка;

3) разработка приложения.

Структура курсового проекта отражает поставленные задачи, исследование предметной области, процесс создания языка и полученные результаты.

ОСНОВНАЯ ЧАСТЬ

1 Декларативный подход и существующее решение

Декларативный подход в программировании подразумевает, что решение поставленной задачи происходит путём её описания, а не чёткого указания последовательности действий. Соответственно языки программирования, использующие декларативный подход, называются декларативными языками программирования.

Одним из примеров декларативного языка, с помощью которого описывается графический интерфейс является QML (Qt Modeling Language). Данный язык используется в рамках библиотеки Qt и технологии Qt Quick, для создания динамичных приложений. Данный язык предоставляет разработчику следующие возможности:

- декларативное описание интерфейса с помощью привычных компонентов;
- обработка событий;
- вставка JavaScript-кода;
- интеграция с C++ логикой.

Достоинствами языка являются:

- простота удобочитаемость кода;
- интегрируемость с JavaScript;
- интегрируемость с Qt/C++.

Пример кода на QML приведён в таблице 1

Таблица 1 – Примеры кода на QML

Пример кода на QML
<pre>import QtQuick 2.12 import QtQuick.Window 2.12 Window { visible: true width: 640 height: 480 title: qsTr("Window app") Text { anchors.centerIn: parent text: "Hello World" } }</pre>

При запуске вышеприведённого кода мы увидим (Рисунок 1) созданное приложение.

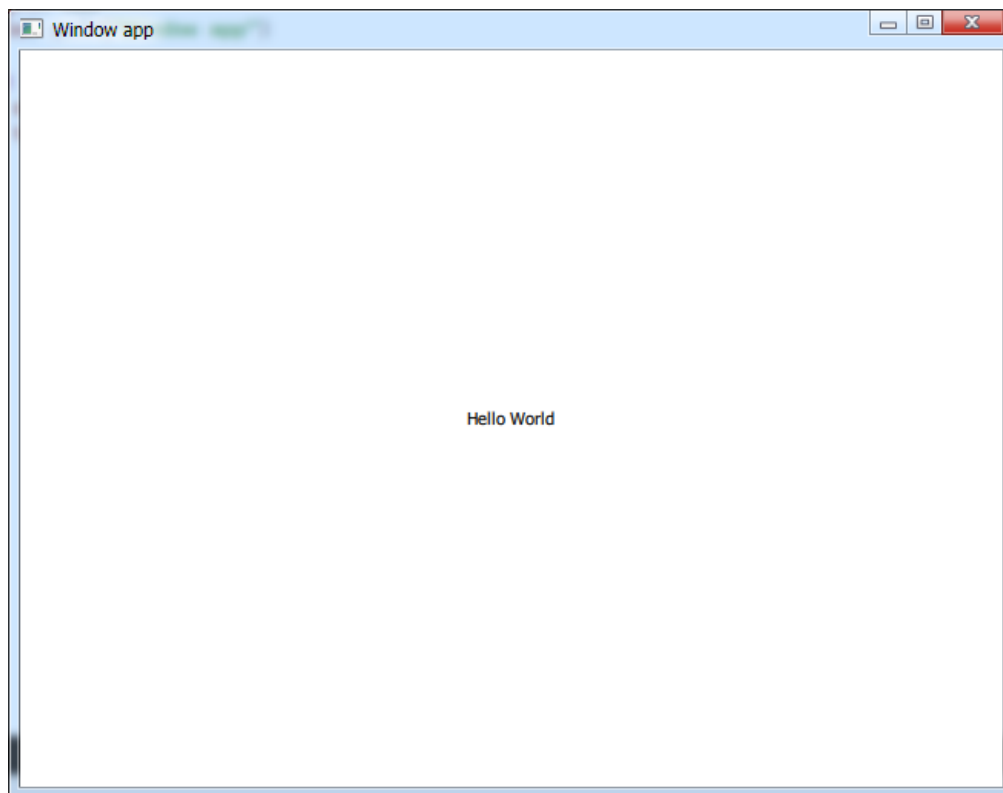


Рисунок 1 – Пример приложения на QML

Приведённый пример показывает, как можно описать графический интерфейс и быстро получить приложение. К тому же написанный код является понятным и не должен вызывать затруднений в его понимании.

Перейдём к рассмотрению того, как работает QML. Предварительно разработчик создаёт файл и пишет QML-код, после чего он его сохраняет. Затем разработчик запускает компиляцию приложения и в работу включается QQmlEngine, который отвечает за создание всех описанных компонентов уже в виде C++ объектов, для последующей компиляции и запуска приложения.

Схема работы изображена на рисунке 2.

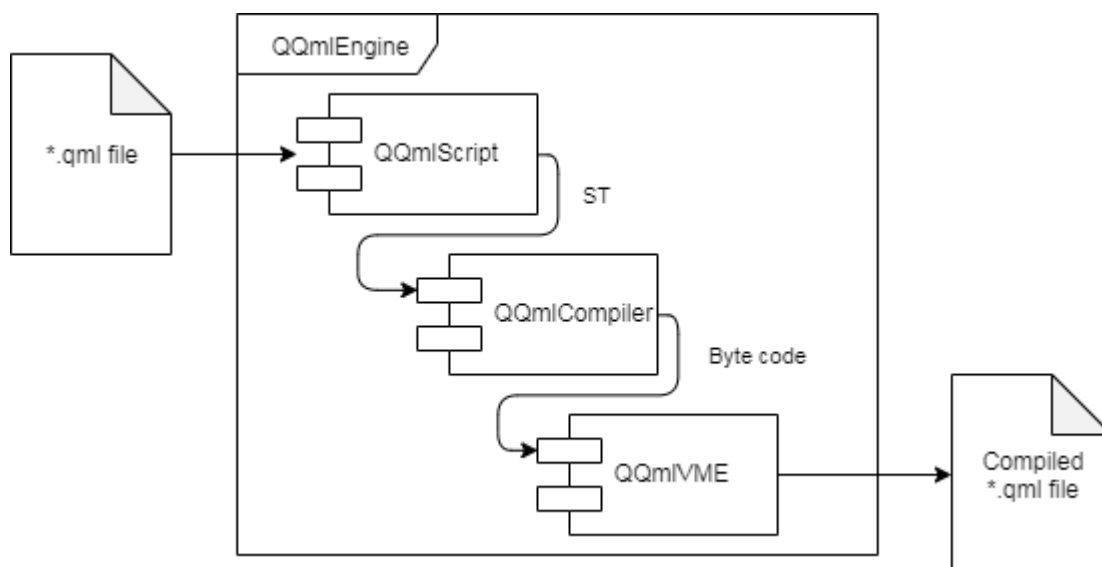


Рисунок 2 – Схема работы QML

Файл QML проходит следующие этапы:

- 1) разбор;
- 2) компиляция;
- 3) создание скомпилированного файла.

В результате разбора строится синтаксическое дерево, которое отражает компонент, связанные свойства, связанные события.

В результате компиляции из полученного дерева, получается байт-код, который в последствии будет выполняться виртуальной машиной.

В результате создания C++ объектов, они записываются в скомпилированный файл, после чего идёт происходить запуск приложения.

2 GUI-библиотека Tkinter

Tkinter – это библиотека для создания графического интерфейса на языке Python. Рассмотрим основные её достоинства и недостатки.

К достоинствам относятся:

- кроссплатформенность;
- является частью стандартной библиотеки Python;
- позволяет создавать исполняемые приложения;
- отсутствие зависимостей с посторонними библиотеками;
- простое API для создания интерфейса.

К недостаткам относятся:

- отсутствие дизайнера форм;
- отсутствие различных компонентов, например вкладок.

2.1 Пример графического интерфейса

Рассмотрим пример кода, написанный с помощью Tkinter для того чтобы понять, каким образом создаётся графический интерфейс (Таблица 2).

Таблица 2 – Пример кода на Tkinter

```
from Tkinter import *

class App:
    def __init__(self, master):
        frame = Frame(master)
        frame.pack()

        self.button = Button(
            frame, text="QUIT", fg="red", command=frame.quit
        )
        self.button.pack(side=LEFT)

        self.hi_there = Button(frame, text="Hello", command=self.say_hi)
        self.hi_there.pack(side=LEFT)

    def say_hi(self):
        print "hi there, everyone!"

root = Tk()
app = App(root)
root.mainloop()
```

При запуске мы увидим окно, изображённое на рисунке 3.



Рисунок 3 – Пример приложения на Tkinter

Проанализировав структуру кода можно выделить определённую последовательность действий при создании графического интерфейса:

- 1) создание главного компонента формы (root);
- 2) создание дочернего/дочерних компонента/компонентов (frame, button);
- 3) задание свойств компонентам;
- 4) задание способа размещения.

Приведённая последовательность одновременно отображает и структуру приложения. Данную структуру необходимо учесть при проектировании генератора и получении целевого кода.

2.2 Описание компонентов

Tkinter включает в себя стандартный набор компонентов для создания графического интерфейса. Их описание приведено в таблице 3.

Таблица 3 – Описание компонентов в Tkinter

Компонент	Описание
Button	Кнопка
Checkbutton	Флажок
Entry	Однострочное поле ввода
Frame	Виджет для группировки
Label	Текстовая надпись
Listbox	Список
Radiobutton	Радиокнопка
Spinbox	Выпадающий список со значениями
Text	Многострочный текст
Combobox	Выпадающий список с элементами

2.3 Менеджеры размещения компонентов

Главным образом стоит рассмотреть менеджеры размещения компонентов. И учесть их возможности при создании синтаксиса языка.

В Tkinter имеются следующие менеджеры для размещения:

- 1) Grid geometry manager;
- 2) Pack geometry manager;
- 3) Place geometry manager.

Их особенностью является то, что они вызываются одноимённым методом и получают определённые параметры, отвечающие за размещение на компонента на форме.

2.3.1 Grid geometry manager

Менеджер, используемый для размещения компонентов на сетке формы. Для расположения элемента используется метод `grid`, вызываемый у любого

компонента, в котором указываются номер строки и колонки. Аргументы, которые принимает метод `grid` перечислены в таблице 4.

Таблица 4 – Описание параметров метода `grid`

Свойство	Описание
<code>row</code>	Строка
<code>column</code>	Колонка
<code>rowspan</code>	Объединяет указанное кол-во строк
<code>colspan</code>	Объединяет указанное кол-во колонок
<code>padx</code>	Внешний по <i>у</i>
<code>pady</code>	Внешний отступ по <i>х</i>
<code>ipadx</code>	Внутренний отступ по <i>х</i>
<code>ipady</code>	Внутренний отступ по <i>у</i>
<code>in</code>	Родительский компонент

2.3.2 Pack geometry manager

Менеджер, используемый по умолчанию для размещения компонентов.

Для расположения используется метод `pack`, в котором указывается сторона, к которой будет примыкать компонент. Аргументы метода `pack` перечислены в таблице 5.

Таблица 5 – Описание параметров метода `grid`

Свойство	Описание
<code>anchor</code>	Расположение на форме, принимает значения: <code>n</code> , <code>ne</code> , <code>e</code> , <code>se</code> , <code>s</code> , <code>sw</code> , <code>w</code> , <code>nw</code> и <code>center</code>
<code>side</code>	Расположение по краю формы, принимает значения: <code>LEFT</code> , <code>RIGHT</code> , <code>TOP</code> , <code>BOTTOM</code>
<code>fill</code>	Заполнение по указанному пространству, принимает значения: <code>X</code> , <code>Y</code> , <code>BOTH</code> , <code>NONE</code>
<code>expand</code>	Свойство позволяющее включить заполнение виджета по всему пространству, принимает значения: <code>True</code> или <code>False</code>
<code>padx</code>	Внешний по <i>у</i>
<code>pady</code>	Внешний отступ по <i>х</i>
<code>ipadx</code>	Внутренний отступ по <i>х</i>
<code>ipady</code>	Внутренний отступ по <i>у</i>

2.3.3 Place geometry manager

Менеджер, используемый для размещения компонентов с помощью явного указания его положения и размеров.

Для расположения используется метод `place`, в котором указываются координаты. Аргументы метода `place` перечислены в таблице 6.

Таблица 6 – Описание аргументов метода `place`

Свойство	Описание
<code>anchor</code>	Расположение на форме, принимает значения: <code>n</code> , <code>ne</code> , <code>e</code> , <code>se</code> , <code>s</code> , <code>sw</code> , <code>w</code> , <code>nw</code> и <code>center</code>
<code>x</code>	Координата <code>x</code>
<code>y</code>	Координата <code>y</code>
<code>width</code>	Ширина
<code>height</code>	Высота
<code>in</code>	Родительский виджет

3 Декларативный язык описания графического интерфейса

Создание декларативного языка включает в себя следующие этапы:

- 1) описание синтаксиса;
- 2) описание структуры кода;
- 3) описание ключевых слов;
- 4) описание РБНФ.

3.1 Синтаксис языка

Синтаксис представляет собой последовательное описание вложенных друг в друга компонентов, которые впоследствии будут отражены в графическом интерфейсе.

Описание интерфейса начинается с главного оконного компонента. При этом название каждого из компонентов начинается с заглавной буквы. После чего описываются свойства и вложенные компоненты.

Для наглядности приведём пример кода в таблице 7. В нём описывается компонент `Window`, который является основным окном приложения, и указание его свойств:

- 1) `title` – заголовок окна;
- 2) `width` – ширина окна;
- 3) `height` – высота окна.

После чего, внутри описывается компонент Label, являющийся надписью, и для которого указывается свойство caption, содержащее текст надписи.

Таблица 7 – Пример кода

```
Window {  
    title: "Window app"  
    width: 200  
    height: 200  
  
    Label {  
        caption: "It's a label caption"  
    }  
}
```

3.2 Структура кода

Из приведённого выше примера, можно выделить общую структуру кода самого языка, приведённой в таблице 8. Такая структура удобна для восприятия и не должна вызывать трудности в понимании. В свою очередь это также позволяет структурировать код.

Таблица 8 – Структура кода языка

```
Window {  
    [property: value]  
    [...]  
  
    [Component: { [property: value] }]  
    [Component: { ... }]  
}
```

3.3 Ключевые слова

В качестве ключевых слов будут выступать имена компонентов и их свойства. Все компоненты делятся на четыре категории:

- 1) компоненты форм;
- 2) компоненты слоёв;
- 3) компоненты ввода;
- 4) компоненты вывода.

Стоит отметить, что для всех компонентов имеется общий набор свойств, к ним относятся свойства позиционирования и внешнего вида. Данные свойства приведены в таблице А.1-А.2.

3.3.1 Компоненты форм

Компонентами форм являются такие компоненты, которые описывают собой формы приложения. К примеру, это могут быть обычные окна, пользовательские диалоговые окна, стандартные диалоговые окна.

В разрабатываем языке пока будет один единственный компоненты окна – это Window. Он будет представлять собой обычное окно приложения. Именно с него начинается описание любого графического интерфейса. В таблице А.3 приведены свойства компонента Window.

3.3.2 Компоненты слоёв

Компонентами слоёв являются такие компоненты, которые отвечают за расположение других компонентов, вложенных в них. К таким компонентам относятся:

- 1) Frame – невидимый компонент для группировки;
- 2) Grid – компонент для сеточной группировки.

В таблице А.4 приведены свойства компонентов слоёв.

3.3.3 Компоненты ввода

Компонентами ввода являются такие компоненты, в которые пользователь может вводить данные. К таким компонентам относятся:

- 1) Button – кнопка;
- 2) Radio – радиокнопка;
- 3) Checkbox – флажок;
- 4) Combobox – выпадающий список;
- 5) Spinbox – выпадающий список значений;
- 6) Line – однострочное поле ввода;
- 7) Text – многострочное поле ввода.

В таблице А.5 приведены свойства компонентов ввода.

3.3.4 Компоненты вывода

Компонентами вывода являются такие компоненты, которые отвечают за вывод каких-либо данных. К таким компонентам относятся:

- 1) Label;

- 2) List;
- 3) Table;
- 4) Tree.

В таблице А.6 приведены свойства компонентов вывода.

3.4 Расширенные формы Бэкуса-Наура

РБНФ (Расширенные формы Бэкуса-Наура) – это форма записи грамматики контекстно-свободных языков. Для разрабатываемого языка составлены следующие правила, приведённые в таблице 9.

Таблица 9 – РБНФ языка

№	Синтаксическое уравнение
1	<code>syntax = mainComponent "{" {property} {component} "}"</code>
2	<code>component = componentName "{" {property} {component} "}"</code>
3	<code>property = propertyName ":" string number</code>
4	<code>componentName = string {string}</code>
5	<code>propertyName = string {string}</code>
6	<code>mainComponent = "Window"</code>
7	<code>string = char {char}</code>
8	<code>number = digit {digit}</code>
9	<code>char = "A" ... "Z"</code>
10	<code>digit = "0" ... "9"</code>

Как правило РБНФ описывают рекурсивную структуру, следовательно для реализации синтаксического анализатора, распознающего вышеописанные правила, необходимо использовать метод рекурсивного спуска.

При реализации метода рекурсивного спуска необходимо для каждого правила создать процедуру, которая будет строить текущую часть дерева. Такими процедурами будут:

- syntax;
- component;
- property.

В таблице 10 приведён псевдокод реализации метода рекурсивного спуска.

Таблица 10 – Реализация МРС с помощью псевдокода

Реализация правила syntax
<pre> procedure syntax() { token = tokens.next() if (token != MAIN_COMPONENT) error() token = tokens.next() if (token != OBRACE) error() parseProperty(st) // Add properties to root component token = tokens.next() while (token == COMPONENT) { component = ComponentNode(token) parseComponent(component) // Add components to root component st.append(component) token = tokens.next() } token = tokens.next() if (token != CBRACE) error() return st } </pre>
Реализация правила component
<pre> procedure parseComponent(component) { token = tokens.next() if (token != OBRACE) error() parseProperty(component) // Add properties to component token = tokens.next() while (token == COMPONENT) { subComponent = ComponentNode(token) // Create component node parseComponent(subComponent) // Add components to component component.append(subComponent) token = tokens.next() } token = tokens.next() if (token != CBRACE) error() } </pre>
Реализация правила property
<pre> procedure parseProperty(component) { token = tokens.next() while (token.type == PROPERTY_NAME) { if (token != component.getProperty(token)) { error() } property = token token = tokens.next() } } </pre>

```

        if (token!= PROPERTY_STR_VALUE or token != PROPERTY_INT_VALUE) {
            error()
        }

        component.properties[property] = token

        token = tokens.next()
    }

    tokens.append(token)
}

```

Для правил: `componentName`, `propertyName`, `string` и `number` нет необходимости в реализации МРС, а лучше регулярные выражения, проверяющие лексемы на корректность (Таблица 11).

Таблица 11 – Регулярные выражения для проверки правил

Правило	Регулярное выражение
<code>componentName</code>	<code>^[A-Z][a-z]+\$</code>
<code>propertyName</code>	<code>^[a-z]*[a-z-]+:\$</code>
<code>string</code>	<code>\d+</code>
<code>number</code>	<code>^\ "[\S\w]+\ "\$</code>

4 Проектирование и реализация транслятора

Транслятор представляет собой программу, которая осуществляет процесс трансляции из одного языка в другой.

В нашем случае транслятор осуществляет трансляцию из исходного языка в сгенерированный Tkinter-код.

Основными модулями разрабатываемого транслятора являются:

- 1) препроцессор;
- 2) лексический анализатор;
- 3) синтаксический анализатор;
- 4) генератор;
- 5) постпроцессор.

Схема работы транслятора изображена на рисунке 4.

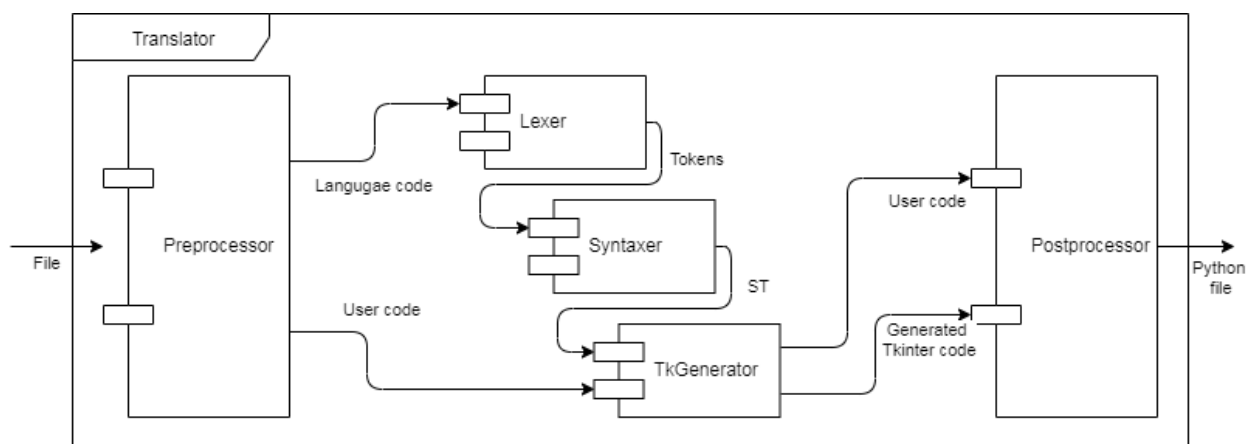


Рисунок 4 – Схема работы транслятора

4.1 Препроцессор

Препроцессор отвечает за разделение пользовательского кода и кода, описывающий графический интерфейс. Также препроцессор отвечает за исполнение пользовательского кода с целью получения переменных и их значений, которых затем подставляются в графические компоненты.

Для решения задачи разделения и исполнения кода используются макросы. Макросы позволяют разделить код, затем отдельно выполнить пользовательский код и получить список переменных, а также получить код, описывающий графический интерфейс, который затем будет передан лексическому анализатору.

Макросы, которые будут использоваться приведены в таблице 12.

Таблица 12 – Описание макросов препроцессора

Макрос	Описание
#LUI	Разделяет пользовательский код от кода, описывающего графический интерфейс
#FILENAME	Указывает имя целевого Python файла
#VESRION	Указывает версию приложения

Ниже, на рисунке 5, приведена UML-диаграмма, на которой изображён класс Preprocessor.

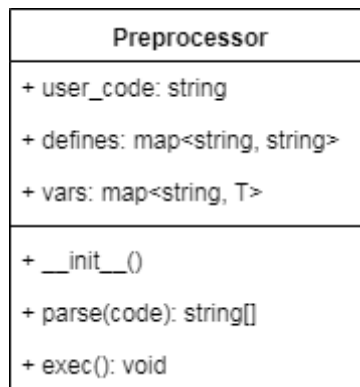


Рисунок 5 – UML-диаграмма класса Preprocessor

Дадим краткое описание методам класса Preprocessor в нижеприведённой таблице 13.

Метод parse разделяет код и записывает распознанные макросы в словарь.

Метод exec выполняет пользовательский код и записывает полученные переменные в словарь

4.2 Лексический анализатор

Лексический анализатор разбирает код на лексемы и по ним он создаёт список токенов. Токен – это единица лексического разбора, содержащая лексему и её тип.

В программе токен представляется как класс, который имеет поля:

- data (лексема);
- type (тип токена).

Соответственно типами токенов будут являться:

- 1) COMPONENT;
- 2) PROPERTY_NAME;
- 3) PROPERTY_STRING_VALUE;
- 4) PROPERTY_NUMBER_VALUE;
- 5) PROPERTY_VAR_VALUE;
- 6) OBRACE;
- 7) CBRACE.

При разборе кода, для каждой лексемы присваивается вышеперечисленный тип, в зависимости от содержания лексемы. К примеру,

если встречено имя компонента, то будет создан токен с типом COMPONENT, а в качестве лексемы будет выступать само имя компонента.

Ниже, на рисунке 6, приведена UML-диаграмма классов: Lexer, Token и TokenType.

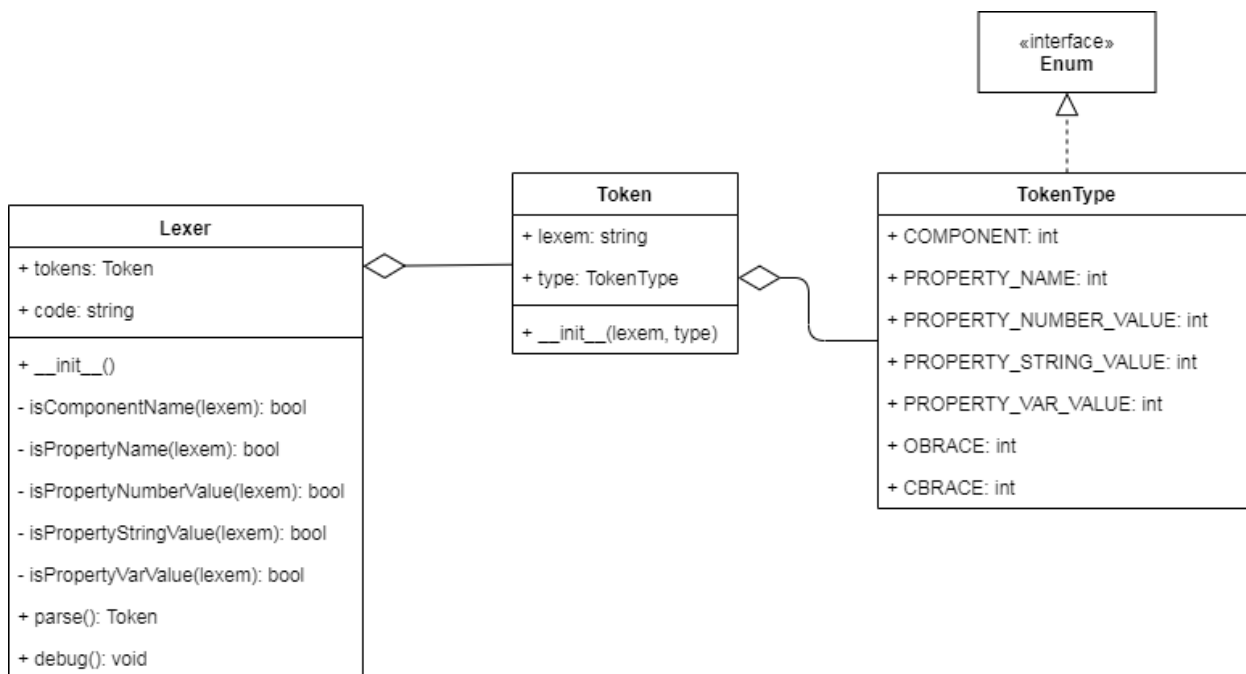


Рисунок 6 – UML-диаграмма классов Lexer, Token и TokenType

Методы с приставкой is проверяют полученную лексему на соответствие какому-либо правилу регулярного выражения, описанного в пункте 3.4.

Метод parse осуществляет разбор кода на лексемы создаёт список токенов.

4.3 Синтаксический анализатор

Синтаксический анализатор по полученному списку токенов строит синтаксическое дерево.

Абстрактное синтаксическое дерево (АСД) – это структура данных, используемая для представления структуры синтаксических уравнений в виде дерева.

Как и любое дерево оно имеет следующие определения:

- корнем дерева называется самый верхний узел, который не имеет родительского узла;

- узлом дерева называется элемент, который имеет родительский элемент и дочерний/дочерние элементы;
- листом дерева называется элемент, которые не имеет дочерних узлов.

По описанным правилам в разделе 3.4 построим общее синтаксическое дерево. Корнем такого дерева будет один из оконных компонентов, а листьями корня будут:

- список свойств в формате словаря;
- список вложенных компонентов.

При этом каждый элемент списка компонентов будет иметь такую же структуру. Пример синтаксического дерева изображён на рисунке 7.

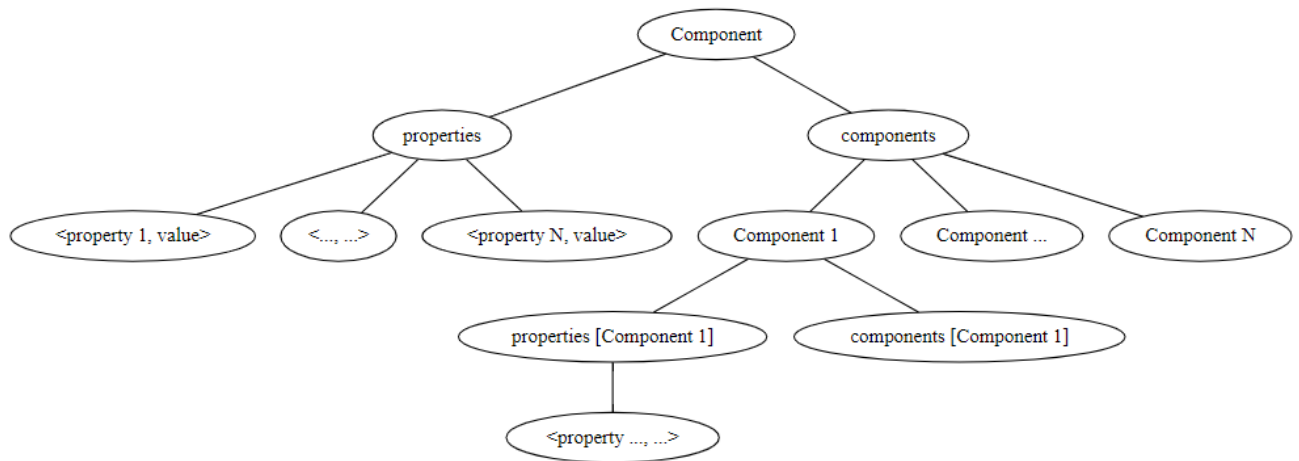


Рисунок 7 – Общее синтаксическое дерево

Для примера приведём код, а затем построим по нему синтаксическое дерево (таблица 13).

Таблица 13 – Пример кода

```

Window {
    title: "Window app"
    width: 150
    height: 200
    Button {
        caption: "Click me"
    }
    Grid {
        Label {
            caption: "It's mine label"
        }
        List {
            data: lst
        }
    }
}
  
```

Полученное дерево, изображённое на рисунке 8, должно быть получено в результате окончания работы синтаксического анализатора, как если бы он получил соответствующий список токенов.

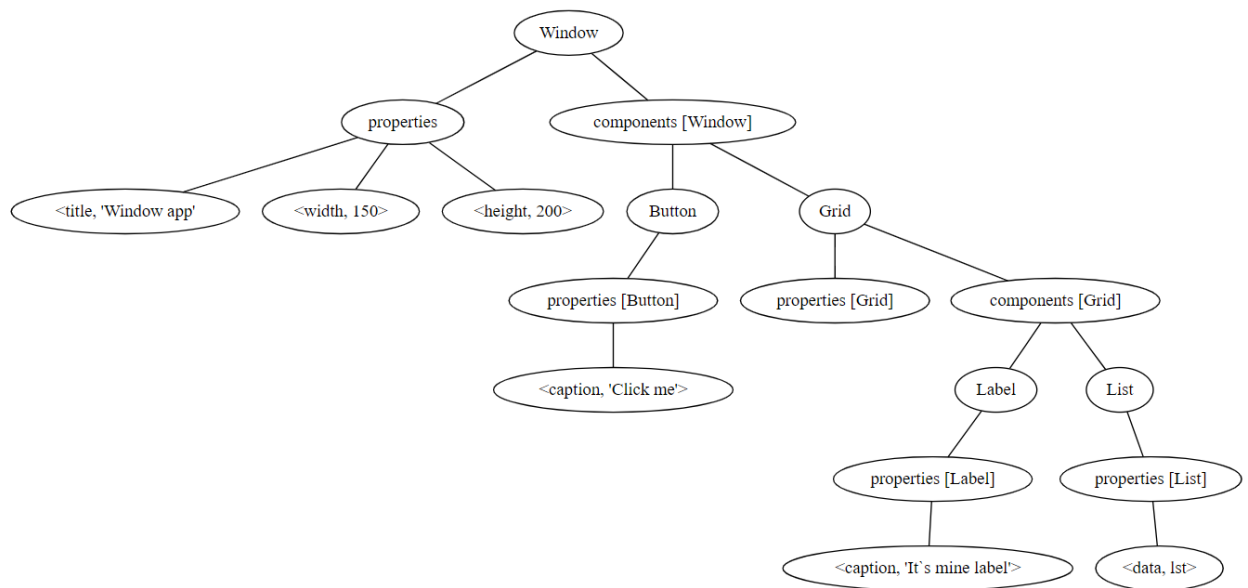


Рисунок 8 – Построенное по коду синтаксического дерево

Класс, описывающий узел дерева должен содержать соответствующие поля:

- название компонента;
- свойства (в качестве словаря);
- список компонентов (в качестве списка, содержащего такие же узлы).

Ниже, на рисунке 9, приведена UML-диаграмма классов Syntaxer и ComponentNode.

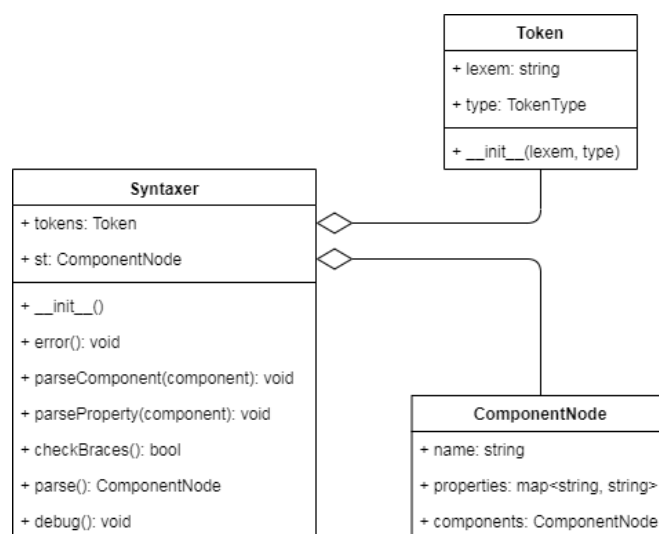


Рисунок 9 – UML-диаграмма класса Syntaxer и ComponentNode

Методы `parse`, `parseComponent` и `parseProperty` являются процедурами, реализующий метод рекурсивного спуска.

Метод `checkBraces` осуществляет проверку кол-ва открытых и закрытых фигурных скобок.

4.4 Генератор

Генератор отвечает за создание целевого Tkinter-кода по полученному синтаксическому дереву.

Процесс генерации включает в себя три этапа:

- 1) генерацию компонента;
- 2) генерацию свойств;
- 3) генерацию дочерних компонентов.

Данный процесс продолжается до тех пор, пока не будет закончен весь обход дерева.

Каждый компонент представляется в виде класса и наследуется от общего класса всех компонентов, который содержит общие для всех свойства. Это позволит в дальнейшем расширять их, добавлять новые свойства, которые присутствуют только у них.

На рисунке 10 изображена UML-диаграмма классов.

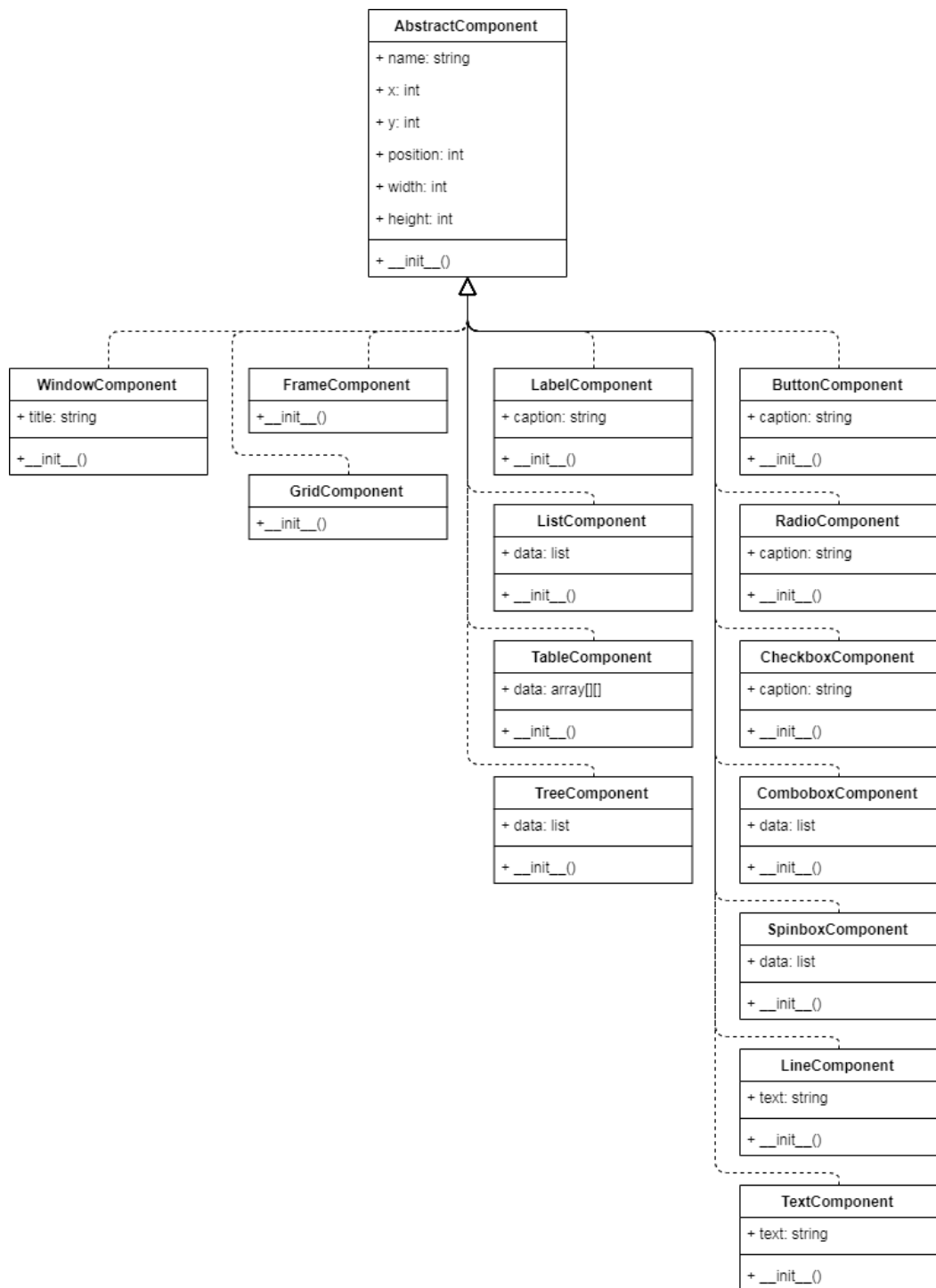


Рисунок 10 – UML-диаграмма классов компонентов

Алгоритм генерации заключается в обходе синтаксического дерева и использовании классов для генерации кода:

- TkComponentFactory, возвращает компонент в виде класса;
- TkComponentGenerator, генерирует компонент.

Псевдокод приведён в таблице 14, а ниже, на рисунке 11, представлена UML-диаграмма классов TkComponentFactory и TkComponentGenerator.

Таблица 14 – Псевдокод алгоритма генерации

Реализация правила syntax

```
void generate(AbstractComponent component, AbstractComponent parent) {

    generatedComponent = TkComponentFactory::get(component)
    generatedComponent.component = component.name

    for (property in component.properties)
        generatedComponent.property = component.properties[property]

    tkGenerator = TkComponentGenerator(generatedComponent, parent)
    code += tkGenerator.generate()

    for (subComponent in component.components)
        parentComponent = TkComponentFactory::get(component.name)

        generatedComponent(subComponent, parent)

}
```

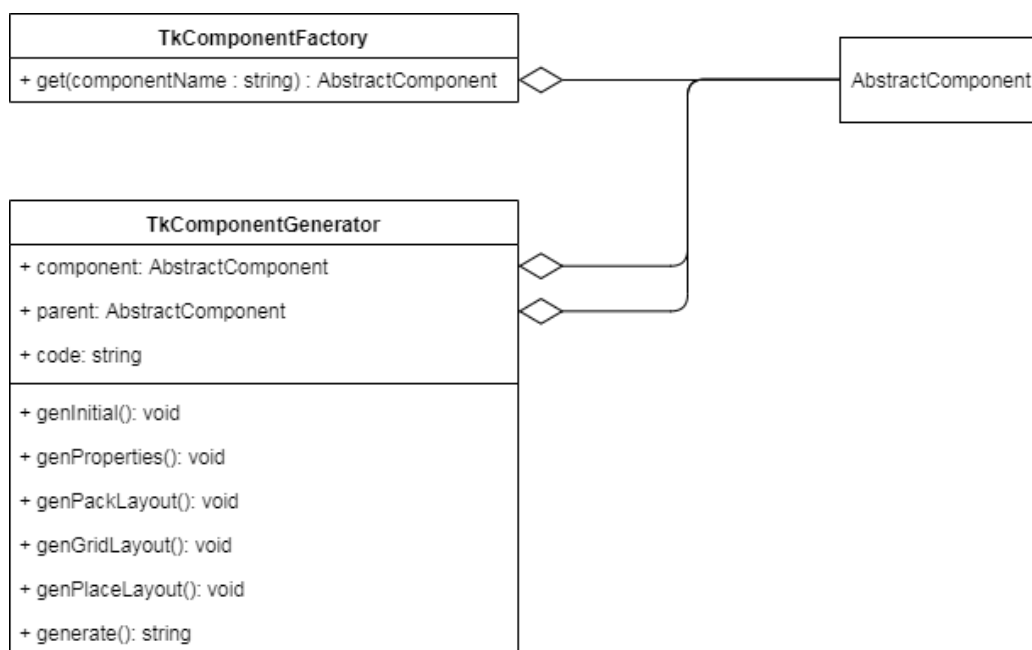


Рисунок 11 – UML-диаграмма классов TkComponentFactory и TkComponentGenerator

4.5 Постпроцессор

Постпроцессор отвечает за создание файла и обработку макросов полученных на этапе работы препроцессора.

Ниже, на рисунке 12, приведена UML-диаграмма класса Preprocessor.

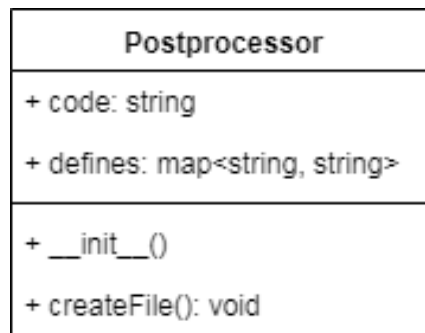


Рисунок 12 – UML-диаграмма класса Postprocessor

5 Проектирование и реализация приложения

В качестве приложения будет выступать консольное приложение, которое принимает на вход исходный файл и команды, а в результате выдаёт Python файл со сгенерированным кодом.

Список принимаемых команд приведен в таблице 15.

Таблица 15 – Команды консольного приложения

Команда	Описание
--help	Выводит справку по доступным командам
--file=[filename]	Путь до файла
--debug	Включает режим отладки, выводя содержимое в командной строке
--version	Выводит текущую версию приложения

Схема работы с приложением изображена на рисунке 13:

- 1) файл передаётся на вход консольного приложения и также предаются дополнительные команды;
- 2) приложение передаёт код из файла транслятору;
- 3) происходит трансляция;
- 4) если во время трансляции ошибок нет, то создаётся Python файл со сгенерированным кодом;
- 5) иначе инициируется исключение и выводится сообщение об ошибке.

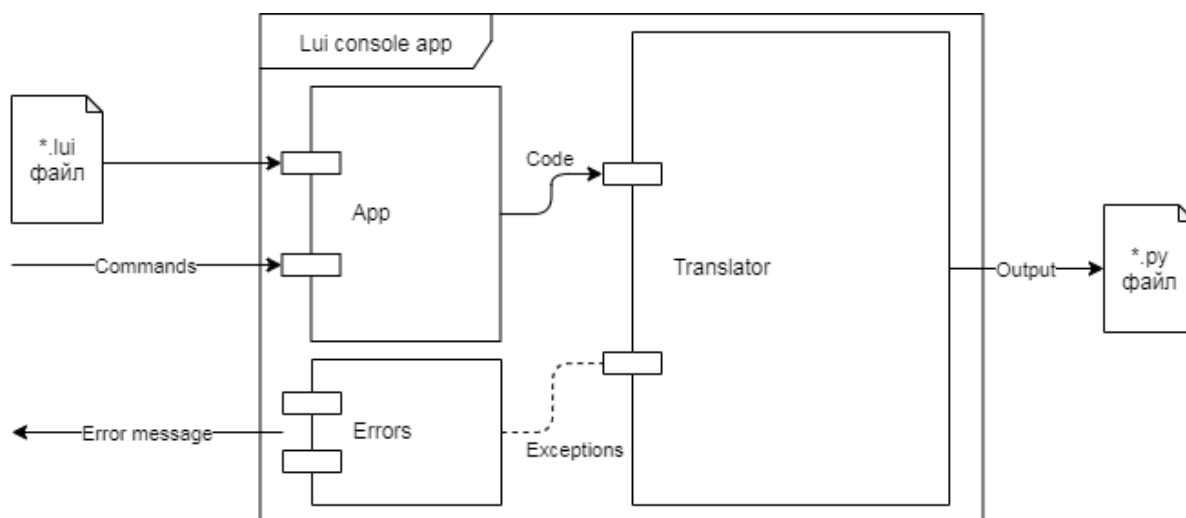


Рисунок 13 – Схема работы консольного приложения

6 Тестирование

Тестирование отдельных компонентов, в частности компонентов транслятора производилось путём написания unit-тестов. Исходный код unit-тестов приведён в приложении Б.

На рисунках 14-15 приведены результаты прохождения тестов.

```

G:\Programs\Python\Python37-32\python.exe G:/Projects/mirea/lui/tests/test_lexer.py
..
-----
Ran 2 tests in 0.004s

OK

Process finished with exit code 0

```

Рисунок 14 – Результаты тестирования класса Lexer

```

G:\Programs\Python\Python37-32\python.exe G:/Projects/mirea/lui/tests/test_syntaxer.py
..
-----
Ran 2 tests in 0.000s

OK

Process finished with exit code 0

```

Рисунок 15 – Результаты тестирования класса Syntaxer

7 Пример использования языка

В качестве демонстрации результата приведём пример для программы которая разделяет числа на положительные и все остальные из исходного линейного односвязного списка и создаёт два новых (Таблица 16).

Таблица 16 – Исходный код приложения

```
from list_class import *
import random

lst = List()
lstPositive = List()
lstOther = List()

for i in range(15):
    lst.append(random.randint(-100, 100))

for index in range(lst.size):
    if lst.at(index) > 0:
        lstPositive.append(lst.at(index))
    else:
        lstOther.append(lst.at(index))

lstPositive = lstPositive.toList()
lstOther = lstOther.toList()
```

Теперь создадим графический интерфейс для существующей программы, добавив код, описывающий интерфейс уже к существующему (Таблица 17).

Таблица 17 – Код описывающий графический интерфейс

```
#LUI
Window {
    title: "Test"
    width: 300
    height: 300

    Frame {
        position: LEFT

        Label {
            caption: "Положительные числа"
        }

        List {
            data: lstPositive
        }
    }
}
```

```

    }

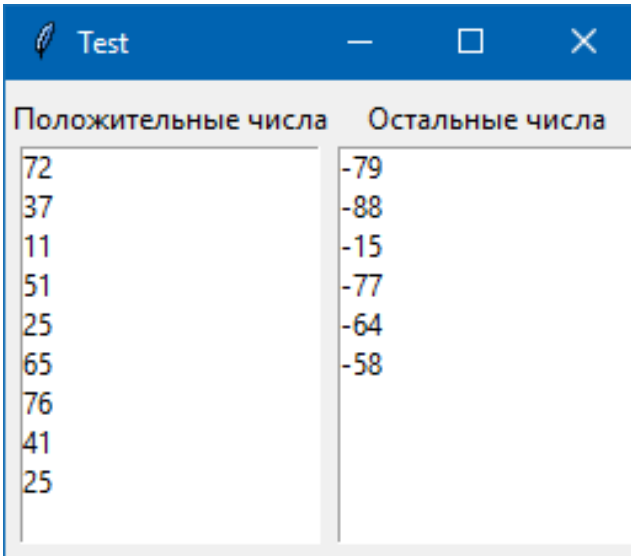
    Frame {
        position: RIGHT

        Label {
            caption: "Остальные числа"
        }

        List {
            data: lstOther
        }
    }
}

```

После чего в командной строке вызовем приложение транслятора с указанием пути к файлу. В результате мы получаем файл со следующим сгенерированным кодом (Рисунок 18).



Положительные числа	Остальные числа
72	-79
37	-88
11	-15
51	-77
25	-64
65	-58
76	
41	
25	

Рисунок 16 – Полученное

ЗАКЛЮЧЕНИЕ

Результатом выполненного курсового проекта является созданный декларативный язык для описания графического интерфейса, названный Lui.

Достоинства разработанного языка:

- простой синтаксис;
- наличие документации;
- открытый исходный код транслятора;
- кроссплатформенность транслятора.

Основным недостатком разработанного языка является полное отсутствие поддержки обработки событий, что в свою очередь ограничивает разработчика, позволяя ему создавать графические интерфейсы только выводящие информацию.

В процессе выполнения были выполнены следующие этапы:

- 1) анализ декларативного подхода и существующего решения;
- 2) изучение GUI-библиотеки Tkinter;
- 3) создание декларативного языка;
- 4) проектирование и реализация транслятора;
- 5) проектирование и реализация приложения;
- 6) тестирование.

СПИСОК ИСПОЛЬЗУЕМЫХ ИСТОЧНИКОВ

Нормативные документы:

1. ГОСТ 7.32-2017 Система стандартов по информации, библиотечному и издательскому делу. Отчет о научно-исследовательской работе. Структура и правила оформления. – М.: Стандартинформ, 2017. – 32 с.;
2. Методические указания по выполнению курсового проекта. – М.: РТУ МИРЭА, 2019. – 45 с.;
3. О введении в действие Инструкции по организации и проведению курсового проектирования. – М.: РТУ МИРЭА, Приказ №1325 от 05.10.2018. – 17 с..

Книги:

1. Построение компиляторов [Текст]: / Н. Вирт – М.: ДМК Пресс, 2016. – 272 с.;
2. Алгоритмы и структуры данных [Текст]: / Н. Вирт – М.: ДМК Пресс, 2014. – 192 с..

Электронные ресурсы:

1. Graphical widget [Электронный ресурс]. – URL: [https://en.wikipedia.org/wiki/Widget_\(GUI\)](https://en.wikipedia.org/wiki/Widget_(GUI));
2. Writing a C Compiler, Part 1 [Электронный ресурс]. – URL: <https://norasandler.com/2017/11/29/Write-a-Compiler.html>;
3. An Introduction to Tkinter (Work in progress) [Электронный ресурс]. – URL: <http://effbot.org/tkinterbook/>;
4. tkinter – Python interface to Tcl/Tk [Электронный ресурс]. – URL: <https://docs.python.org/3.7/library/tkinter.html>;
5. unittest – Unit testing framework [Электронный ресурс]. – URL: <https://docs.python.org/3.7/library/unittest.html>;
6. pydoc – Documentation generator and online help system [Электронный ресурс]. – URL: <https://docs.python.org/3.7/library/pydoc.html>;
7. Python GUI, PyQt vs Tkinter [Электронный ресурс]. – URL: <https://dev.to/amigosmaker/python-gui-pyqt-vs-tkinter-5hdd>;

8. Built-in Functions [Электронный ресурс]. – URL: <https://docs.python.org/3/library/functions.html#exec>;
9. QML Syntax Basics [Электронный ресурс]. – URL: <https://doc.qt.io/qt-5/qtqml-syntax-basics.html>;
10. QML Engine Internals, Part I [Электронный ресурс]. – URL: <https://www.kdab.com/qml-engine-internals-part-1-qml-file-loading/>;
11. Webgraphviz [Электронный ресурс]. – URL: <http://webgraphviz.com/>.

ПРИЛОЖЕНИЕ А

Описание свойства компонентов

Таблица А.1 – Свойства позиционирования компонентов

Свойство	Описание	Значения
x	Значение x	int
y	Значение y	int
position	Позиция элемента	LEFT, RIGHT, TOP, BOTTOM
padding-x	Отступ относительно x	int
padding-y	Отступ относительно y	int

Таблица А.2 – Свойства оформления компонентов

Свойство	Описание	Значения
background-color	Цвет заднего фона	string

Таблица А.3 – Свойства компонента Window

Свойство	Описание	Значение
title	Заголовок окна	string
width	Ширина	int
height	Высота	int

Таблица А.4 – Свойства компонентов слоёв

Компонент		
Свойство	Описание	Значение
Frame		
width	Ширина	int
height	Высота	int
Grid		
rows	Кол-во строк	int
columns	Кол-во колонок	int

Таблица А.5 – Свойства компонентов ввода

Компонент		
Свойство	Описание	Значение
Button		
caption	Текстовая надпись	string
Radio		
caption	Текстовая надпись	string
Checkbox		
caption	Текстовая надпись	string
Combobox		
data	Список элементов	list
Spinbox		
data	Список значений	list
Line		
text	Текст	string
Text		
text	Текст	string

Таблица А.6 – Свойства компонентов вывода

Компонент		
Свойство	Описание	Значение
Label		
caption	Текст	string
List		
data	Список элементов	list
Table		
data	Двумерный массив	two dim array
Tree		
data	Древовидная структура	list

ПРИЛОЖЕНИЕ Б

Исходный код unit-тестов

Таблица Б.1 – Исходный код unit-теста класса Lexer18

test_lexer.py
<pre>import unittest from translator.lexer.lexer import * class TestLexer(unittest.TestCase): def getCode(self, file): lui_code = "" with open(file) as f: code = "" for line in f.readlines(): code += line if code.find("#LUI") != -1: user_code = code.split("#LUI")[0] lui_code = code.split("#LUI")[1] else: lui_code = code lui_code = re.sub("\n", " ", lui_code) lui_code = re.sub("\t", " ", lui_code) lui_code = re.sub(" +", " ", lui_code) lui_code = lui_code.strip() return lui_code def testParse(self): lexer = Lexer() lexer.lui_code = self.getCode("../examples/basic_1.lui") tokens = lexer.parse() base_tokens = [Token(TokenType.COMPONENT, "Window"), Token(TokenType.OBRACE, "{"), Token(TokenType.PROPERTY_NAME, "title"), Token(TokenType.PROPERTY_STRING_VALUE, '"Window title"'), Token(TokenType.PROPERTY_NAME, "width"), Token(TokenType.PROPERTY_NUMBER_VALUE, 150), Token(TokenType.PROPERTY_NAME, "height"), Token(TokenType.PROPERTY_NUMBER_VALUE, 150), Token(TokenType.CBRACE, "}"),] base_tokens.reverse() for i in range(len(tokens)): self.assertEqual(tokens[i].type, base_tokens[i].type) self.assertEqual(tokens[i].data, base_tokens[i].data) def testGetTokens(self): lexer = Lexer() lexer.lui_code = self.getCode("../examples/basic_2.lui") tokens = lexer.parse() self.assertEqual(len(tokens), 14) unittest.main()</pre>

Таблица Б.2 – Исходный код unit-теста класса Syntaxer19

	test_syntaxer.py
	<pre> import unittest from translator.token import * from translator.syntaxer.syntaxer import * class TestSyntaxer(unittest.TestCase): def testGetST(self): base_tokens = [Token(TokenType.COMPONENT, "Window"), Token(TokenType.OBRACE, "{"), Token(TokenType.PROPERTY_NAME, "title"), Token(TokenType.PROPERTY_STRING_VALUE, "Window title"), Token(TokenType.PROPERTY_NAME, "width"), Token(TokenType.PROPERTY_NUMBER_VALUE, 150), Token(TokenType.PROPERTY_NAME, "height"), Token(TokenType.PROPERTY_NUMBER_VALUE, 150), Token(TokenType.CBRACE, "}"),] base_tokens.reverse() base_st = ComponentNode("Window") base_st.properties["title"] = "Window title" base_st.properties["width"] = 150 base_st.properties["height"] = 150 syntaxer = Syntaxer() syntaxer.tokens = base_tokens st = syntaxer.parse() self.assertEqual(str(st), str(base_st)) def testErrorBraces(self): base_tokens = [Token(TokenType.COMPONENT, "Window"), Token(TokenType.PROPERTY_NAME, "title"), Token(TokenType.PROPERTY_STRING_VALUE, "Window title"), Token(TokenType.PROPERTY_NAME, "width"), Token(TokenType.PROPERTY_NUMBER_VALUE, 150), Token(TokenType.PROPERTY_NAME, "height"), Token(TokenType.PROPERTY_NUMBER_VALUE, 150), Token(TokenType.CBRACE, "}"),] base_tokens.reverse() syntaxer = Syntaxer() syntaxer.tokens = base_tokens try: st = syntaxer.parse() except Exception as e: self.assertEqual(str(e), "Syntax error: unclosed brace") unittest.main() </pre>

ПРИЛОЖЕНИЕ В

Исходный код приложения

Таблица В.1 – Исходный код приложения

app.py	
<pre>import sys from translator.translator import * class LuiApp: def __init__(self): self.translator = LuiTranslator() self.code = "" self.is_debug = False def helpCommand(self): print("List of lui cmd args:") print("\t --help - view list of lui cmd args") print("\t --file=[path] - select path to *.lui file") print("\t --debug - turn on debug mode") print("\t --version - view Lui version") self.exit() def fileCommand(self, path): filename = path.split("=") if filename is not None: with open(filename[1]) as f: self.code = f.read() def debugCommand(self): self.is_debug = not self.is_debug if True else False def versionCommand(self): print("Lui version: 1.1") self.exit() def run(self): self.translator.code = self.code self.translator.is_debug = self.is_debug #try: self.translator.run() #except Exception as e: # print(e) def exit(self): sys.exit(0)</pre>	

Таблица В.2 – Исходный код транслятора

translator.py	
<pre>from translator.preprocessor.preprocessor import * from translator.lexer.lexer import * from translator.syntaxer.syntaxer import * from translator.generator.generator import * from translator.postprocessor.postprocessor import * class LuiTranslator: def __init__(self): self.code = "" self.preprocessor = Preprocessor() self.lexer = Lexer() self.syntaxer = Syntaxer() self.generator = TkGenerator() self.postprocessor = Postprocessor() self.is_debug = False</pre>	

Продолжение таблицы В.2

```
def debugLexer(self):
    print("Tokens:")
    self.lexer.debug()
    print("\n")

def debugSyntaxer(self):
    print("Syntax tree:")
    self.syntaxer.debug()
    print("\n")

def debugTranslator(self):
    self.generator.debug()

def run(self):
    self.generator.user_code, self.lexer.lui_code = self.preprocessor.parse(self.code)
    self.preprocessor.exec()

    tokens = self.lexer.parse()
    if self.is_debug:
        self.debugLexer()

    self.syntaxer.tokens = tokens
    st = self.syntaxer.parse()
    if self.is_debug:
        self.debugSyntaxer()

    self.generator.locals = self.preprocessor.vars
    self.generator.st = st
    code = self.generator.generate()

    #if self.is_debug:
    #    self.generator.debug()

    self.postprocessor.code = code
    self.postprocessor.defines = self.preprocessor.defines
    self.postprocessor.createFile()
```

preprocessor.py

```
import re

class Preprocessor:
    def __init__(self):
        self.user_code = ""
        self.defines = {}
        self.vars = {}

    def parse(self, code):
        lui_code = ""

        filename_def = re.search("#FILENAME=\S+", code)
        if filename_def is not None:
            bpos, epos = filename_def.span()
            self.defines["filename"] = code[bpos:epos].split("=")[1]

        version_def = re.search("#VERSION=\S+", code)
        if version_def is not None:
            bpos, epos = version_def.span()
            self.defines["version"] = code[bpos:epos].split("=")[1]

        lui_def = re.search("#LUI", code)
        if lui_def is not None:
            tmp_code = code.split("#LUI")
            if len(tmp_code) > 0:
                self.user_code = tmp_code[0]
                lui_code = tmp_code[1]
            else:
                lui_code = tmp_code[0]
        else:
            lui_code = code

        lui_code = re.sub("#FILENAME=\S+", "", lui_code)
        lui_code = re.sub("#VERSION=\S+", "", lui_code)

        lui_code = re.sub("\n", " ", lui_code)
        lui_code = re.sub("\t", " ", lui_code)
        lui_code = re.sub(" +", " ", lui_code)
        lui_code = lui_code.strip()
```

Продолжение таблицы В.2

<pre> return self.user_code, lui_code def exec(self): if len(self.user_code) != 0: code = "import sys" code += '\n' code += "sys.path.append(\".\")" code += '\n' code += self.user_code exec(code) self.vars = locals()</pre>	
<div>lexer.py</div> <pre># @package lui Lexer from translator.token import * import re class Lexer: def __init__(self): self.tokens = [] self.lui_code = "" def isComponentName(self, componentName): return re.match("^[A-Z][a-z]+\$", componentName) is not None def isPropertyName(self, propertyName): return re.match("^[a-z]*[a-z-]+:\$", propertyName) is not None def isPropertyStringValue(self, propertyValue): return re.match("\d+", propertyValue) is not None def isPropertyNumberValue(self, propertyValue): return re.match("[^\"'\\S\\w]+\"", propertyValue) is not None def isPropertyVarValue(self, propertyValue): return re.match("[a-zA-z][a-zA-Z0-9_]*\$", propertyValue) is not None def parse(self): token = "" isQuotes = False for c in self.lui_code: if c is "{": self.tokens.append(Token(TokenType.OBRACE, "{")) token = "" if c is "}": self.tokens.append(Token(TokenType.CBRACE, "}")) token = "" if c is "\"": isQuotes = not isQuotes if True else False if not isQuotes and c is " ": token = token.lstrip(" ") if self.isComponentName(token): self.tokens.append(Token(TokenType.COMPONENT, token)) elif self.isPropertyName(token): self.tokens.append(Token(TokenType.PROPERTY_NAME, token[:-1])) elif self.isPropertyStringValue(token): self.tokens.append(Token(TokenType.PROPERTY_STRING_VALUE, token)) elif self.isPropertyNumberValue(token): self.tokens.append(Token(TokenType.PROPERTY_NUMBER_VALUE, int(token))) elif self.isPropertyVarValue(token): self.tokens.append(Token(TokenType.PROPERTY_VAR_VALUE, token)) token = "" token += c self.tokens.reverse() return self.tokens</pre>	

Продолжение таблицы В.2

<pre> def debug(self): tokens = self.tokens.copy() tokens.reverse() for token in tokens: print(token) </pre>	
<div style="text-align: right;">syntaxer.py</div> <pre> from translator.syntaxer.syntax_error import * from translator.token import TokenType from translator.lui_definition import * class ComponentNode: def __init__(self, name): self.name = name self.properties = {} self.components = [] def toString(self, i=1): string = self.name for property in self.properties: string += "\n" for t in range(i): string += "\t" string += "Property[" + str(self.properties[property]) + "]" for component in self.components: string += "\n" for t in range(i): string += "\t" string += component.toString(i + 1) return string def __str__(self): return self.toString() class Syntaxer: def __init__(self): self.tokens = [] self.st = ComponentNode("") def error(self, code=None, data=None): message = "Syntax error: " if code is SyntaxerError.UNCLOSED_BRACE: message += "unclosed brace" elif code is SyntaxerError.COMPONENT_NOT_EXISTS: message += "component '" + data + "' is not exists" elif code is SyntaxerError.PROPERTY_NOT_EXISTS: message += "component '" + data[0] + "' doesn't have property '" + data[1] + "'" raise Exception(message) def parseComponent(self, component): token = self.tokens.pop() if token.type is not TokenType.OBRACE: self.error() self.parseProperty(component) token = self.tokens.pop() while token.type is TokenType.COMPONENT: if token.data not in components.keys(): self.error(SyntaxerError.COMPONENT_NOT_EXISTS, token.data) subComponent = ComponentNode(token.data) self.parseComponent(subComponent) component.components.append(subComponent) token = self.tokens.pop() self.tokens.append(token) </pre>	

Продолжение таблицы В.2

syntaxer.py

```
token = self.tokens.pop()
if token.type is not TokenType.CBRACE:
    self.error()

def parseProperty(self, component):
    token = self.tokens.pop()
    while token.type is TokenType.PROPERTY_NAME:
        if token.data not in components.get(component.name) and token.data not in
positions:
            self.error(SyntaxerError.PROPERTY_NOT_EXISTS, [component.name, token.data])

        property = token.data

        token = self.tokens.pop()
        if token.type in [TokenType.PROPERTY_NUMBER_VALUE, TokenType.PROPERTY_STRING_VALUE,
                        TokenType.PROPERTY_VAR_VALUE]:
            component.properties[property] = token.data

        token = self.tokens.pop()

    self.tokens.append(token)

def checkBraces(self):
    countOBraces = 0
    countCBraces = 0
    tokens = self.tokens.copy()

    for token in tokens:
        if token.type is TokenType.OBRACE:
            countOBraces += 1

        if token.type is TokenType.CBRACE:
            countCBraces += 1

    return countOBraces == countCBraces

def parse(self):
    if not self.checkBraces():
        self.error(SyntaxerError.UNCLOSED_BRACE)

    token = self.tokens.pop()
    if token.type is not TokenType.COMPONENT or token.data not in components.keys():
        self.error(SyntaxerError.COMPONENT_NOT_EXISTS, token.data)

    self.st.name = token.data

    token = self.tokens.pop()
    if token.type is not TokenType.OBRACE:
        self.error()

    self.parseProperty(self.st)

    token = self.tokens.pop()
    while token.type is TokenType.COMPONENT:
        if token.data not in components.keys():
            self.error(SyntaxerError.COMPONENT_NOT_EXISTS, token.data)

        component = ComponentNode(token.data)

        self.parseComponent(component)

        self.st.components.append(component)
        token = self.tokens.pop()

    if token.type is not TokenType.CBRACE:
        self.error()

    return self.st

def debug(self):
    print(self.st)
```


Продолжение таблицы В.2

generator.py

```
# Generator

from translator.generator.ComponentGenerator.TkComponentGenerator import *
from components.OutputComponent import *

dicOfMatching = {
    "x": "x",
    "y": "y",
    "width": "width",
    "height": "height",
    "caption": "text",
    "background-color": "bg",
    "position": "side"
}

class TkGenerator:
    def __init__(self, ):
        self.st = None
        self.code = []
        self.locals = {}
        self.user_code = ""
        self.components_counter = {}
        self.tkGenerator = TkComponentGenerator()

    def debug(self):
        print("Locals:")
        print(self.locals)
        print("User code:")
        print(self.user_code)

    def getList(self, component):
        dataProperty = None
        for property in component.properties.keys():
            if property == "data":
                dataProperty = self.locals[component.properties[property]]

        self.code.append("listbox = Listbox(window)")

        if isinstance(dataProperty, list):
            self.code.append("for item in " + component.properties[property] + ":")
            self.code.append("\t\tlistbox.insert(END, item)")

        self.code.append("listbox.pack()")

    def generateComponent(self, component, parent=None):
        self.components_counter[component.name] = self.components_counter.get(component.name,
0) + 1
        generatedComponent = TkComponentFactory.get(component.name)

        generatedComponent.__dict__["component"] = component.name
        generatedComponent.__dict__["name"] = component.name.lower() + "_" +
str(self.components_counter[component.name])

        for property in component.properties.keys():
            generatedComponent.__dict__[property] = component.properties.get(property)

        if not isinstance(generatedComponent, WindowComponent):
            tkGenerator = TkComponentGenerator(generatedComponent, parent)
            self.code.append(tkGenerator.generate())

        for subComponent in component.components:
            parentComponent = TkComponentFactory.get(component.name)
            parentComponent.name = component.name.lower() + "_" +
str(self.components_counter[component.name])

            self.generateComponent(subComponent, parentComponent)

    def generate(self):
        if self.st is not None:
            self.code.append("from tkinter import *")
            self.code.append(self.user_code)
            self.code.append("window_1 = Tk()")

            if self.st.properties.get("title"):
```

Продолжение таблицы В.2

<pre> self.code.append("window_1.title(" + self.st.properties.get("title") + ")") if len(self.st.components) > 0: self.generateComponent(self.st) self.code.append("window_1.mainloop()") return self.code else: raise Exception("Generate code error")</pre>	
postprocessor.py	
<pre>class Postprocessor(): def __init__(self): self.code = "" self.defines = {} def createFile(self): if "filename" in self.defines.keys(): filename = self.defines["filename"] else: filename = "tmp" with open(filename + ".py", "w") as f: f.write("\n".join(self.code))</pre>	