|  |  |
| --- | --- |
|  | |
| 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

Предметной областью данного курсового проекта является графические интерфейсы пользователя и их создание с помощью GUI библиотек.

Основные задачи, возникающие при создании GUI как правило, являются: изучение соответствующей библиотеки и его реализация. Первая задача как правило реализуема в относительно короткие сроки, однако, вторая задача может потребовать значительных временных затрат, ввиду как специфики языка программирования, так и библиотеки.

Если такое сравнение уместно, то, например, реализация сложного графического интерфейса на базе Qt в С++ будет быстрее и менее трудоёмкой задачей, нежели на базе Tkinter в Python.

Исходя из темы курсовой работы поставим вопрос, возможно ли создать декларативный язык, для описания графического интерфейса, который одновременной решал бы и первую задачу и вторую, за короткие сроки? Чтобы время на его изучение было минимальным и при этом реализация графического интерфейса не уступала бы аналогам.

Одним из вариантов решения является создание языка на базе существующей GUI-библиотеки. Следовательно, создаваемый предметно-ориентированный язык декларативного стиля будет решать задачу source-to-source генерации. Для решения такой задачи необходимо:

1. создать языка;
2. разработать транслятор;
3. разработать генератор кода;
4. создать приложение.

Для реализации транслятора, генератора и приложения, в рамках курсового проекта был выбран язык Python, по требованию технического задания. В качестве GUI-библиотеки, для которой будет генерироваться код, был выбран Tkinter, так как данная библиотека является «визитной карточкой» Python для создания графических приложений и входит в стандартную библиотеку языка.

Целью курсового проекта является автоматизация создания графического интерфейса с помощью декларативного языка.

Актуальность курсового проекта заключается в том, что разработчик, не знающий GUI-библиотеку Tkinter сможет для своего приложения без графического интерфейса реализовать графическую часть, в короткие сроки.

Основными этапами во время выполнения курсовой работы являются:

1. изучение декларативного подхода и существующих решений;

Данный раздел посвящён описанию декларативного подхода в программирования и изучению существующих решений для создания графического интерфейса.

1. проектирование языка;

Данный раздел посвящён описанию нового декларативного языка, в частности: синтаксис, РБНФ, компоненты.

1. изучение структуры данных – абстрактное синтаксическое дерево;

Данный раздел посвящён описанию структуры данных – абстрактного синтаксического дерева.

1. проектирование и реализация транслятора;

Данный раздел посвящён описанию внутренних компонентов транслятора, в частности: препроцессор, лексический и синтаксический анализаторы.

1. изучение GUI-библиотеки Tkinter;

Данный раздел посвящён краткому описанию библиотеки, в частности: работа с библиотекой и компонентам.

1. проектирование и реализация генератора;

Данный раздел посвящён описанию компонентов генератора кода, а также алгоритму генерации.

1. проектирование и реализация приложения.

Данный раздел посвящён созданию консольного приложения, которое представляет собой инструментальное средство.

# основная часть

## 1 Описание декларативного подхода и существующих решений

Декларативный язык программирования – это язык, с помощью которого решение задачи происходит посредством описания решаемой задачи, а не указания конкретной последовательности действий, приводящей к выполнению задачи.

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

* описание GUI с помощью привычных компонентов;
* создание адаптивного GUI и создание анимации;
* обработка событий;
* вставка JavaScript-кода;
* интеграция с С++ логикой.

Достоинствами языка являются:

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

Основным недостатком можно выделить более долгую компиляцию, по сравнению с обычными виджетами.

Пример кода на QML приведён в таблице 1

Таблица 1 – Примеры кода на QML

|  |
| --- |
| **Пример кода на QML** |
| 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"  }  } |

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



Рисунок 1 – Пример приложения на QML

Приведённый пример показывает, как можно быстро описать графический интерфейс и получить приложение.

Рассмотрим как работает QML. Предварительно разработчик создаёт файл и пишет в нём QML-код, после чего он его сохраняет. После этого разработчик запускает компиляцию приложения и в работу включается QQmlEngine, который отвечает за создание всех описанных компонентов уже в виде С++ объектов, для последующей компиляции и запуска самого 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, для создания C++ объектов, в соответствии таблицей типов.

В результате создания С++ объектов, они записываются в скомпилированный файл, а после чего идёт уже запуск приложения.

## 2 Проектировании языка

Проектирование языка включается в себя следующие задачи:

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

## 2.1 Описание синтаксиса

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

Описание компонентов начинается с названия компонента, с заглавной буквы. После чего описываются свойства и вложенные компоненты.

Для наглядности приведём пример кода, который выглядит следующим образом (Таблица 2).

Таблица 2 – Пример кода

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

В приведённом коде идёт описание компонента Window, который является основным окном приложения, и указание свойств:

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

После чего, внутри компонента Window, описывается компонент Label, который является надписью, и для которого указывается свойство caption, что является самим текстом надписи.

## 2.3 Структура кода

Из приведённого выше примера, опишем общую структуру кода языка (Таблица 3).

Таблица 3 – Структура кода языка

|  |
| --- |
| Window {  [property: value]  […]    [Component: { [property: value] }]  [Component: { … }]  } |

Такая структура удобна для восприятия и не должна вызывать трудности в понимании. В свою очередь это также позволяет структурировать код.

## 2.3 Описание ключевых слов

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.

## 2.3.1 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.

## 2.3.2 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.

## 2.3.3 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.

## 2.3.4 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.

## 2.4 Описание РБНФ

РБНФ (Расширенные формы Бэкуса-Наура) – это форма записи грамматики контекстно-свободных языков. Для разрабатываемого языка они будут иметь вид, приведённый в таблице 4.

Таблица 4 – РБНФ языка

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

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

При реализации метода рекурсивного спуска необходимо учесть, что для каждого правила надо написать собственную процедуру.

Приведём реализацию по вышеописанным правилам с помощью псевдокода (Таблица 5).

Таблица 5 – Реализация МРС с помощью псевдокода

|  |
| --- |
| **Реализация правила syntax** |
| 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  } |
| **Реализация правила component** |
| 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()  } |
| **Реализация правила property** |
| procedure parseProperty(component) {  token = tokens.next()  while (token.type == PROPERTY\_NAME) {  if (token != component.getProperty(token)) {  error()  }    property = token    token = tokens.next()  if (token!= PROPERTY\_STR\_VALUE or token != PROPERTY\_INT\_VALUE) {  error()  }    component.properties[property] = token    token = tokens.next()  }    tokens.append(token)  } |

Для проверки правил: componentName, propertyName, string и number использовались функции для проверки правильности регулярных выражений. Соответственно, данные регулярные выражения приведены в таблице 6.

Таблица 6 – Регулярные выражения для проверки правил

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

## 3 Изучение абстрактного синтаксического дерева

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

АСД строится на этапе синтаксического анализа и отражает структуру написанного кода. По описанным синтаксическим уравнениям в разделе 2.4 построим синтаксическое дерево. Корнем такого дерева будет является один из оконных компонентов. Листьями корня будут:

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

При этом каждый элемент списка компонентов будет иметь такую же структуру. То есть вершиной будет компонент, а листьями свойства и список компонентов. Пример синтаксического дерева изображён на рисунке 3.



Рисунок 3 – Общее синтаксическое дерево

В качестве примера, приведём кода, в котором будут описанные вложенные компоненты (Таблица 7).

Таблица 7 – Пример кода

|  |
| --- |
| Window {  title: "Window app"  width: 150  height: 200    Button {  caption: "Click!"  }    Grid {  Row {  Label {  caption: "It's mine Label"  }    List {  data: lst  }  }  }  } |

По приведённому коду построим синтаксическое дерева, которое должно сформироваться после синтаксического анализе. На рисунке 4 изображено построенное дерево.

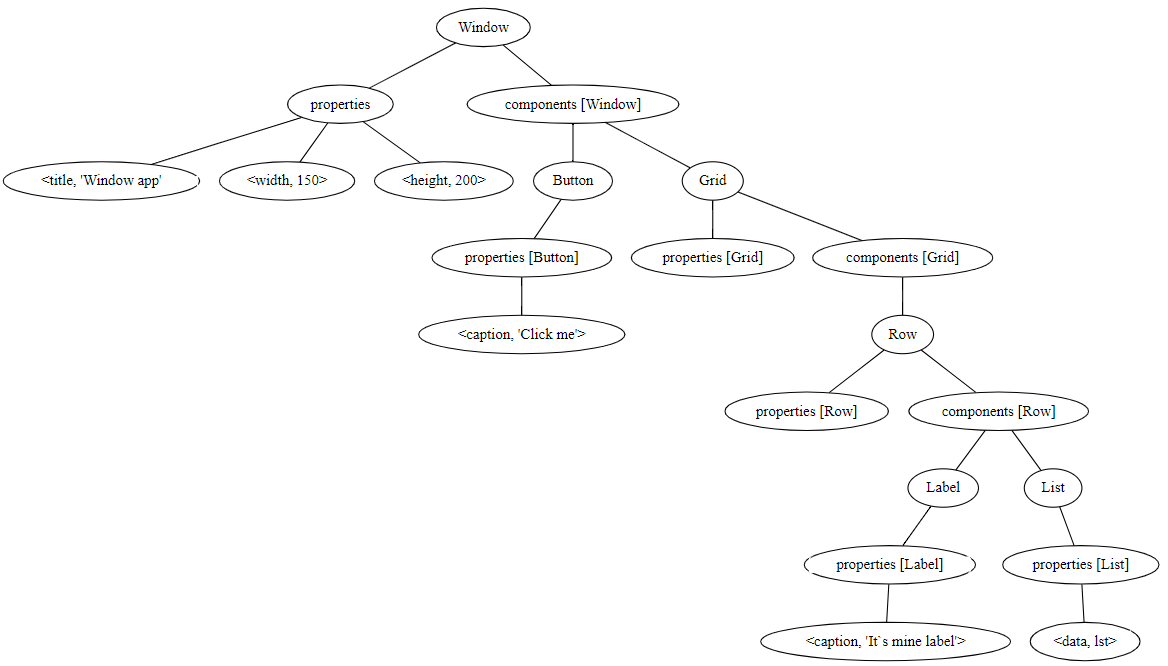


Рисунок 4 – Построенное по коду АСД

## 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 будут:

1. препроцессор;

Разделяет пользовательский код (затем передаёт его в генератор) и код описания графического интерфейса (затем передаёт его в лексический анализатор).

1. лексический анализатор;

По полученному код составляет список токенов, классифицируя их по типу и значению. Затем передаёт список токенов в синтаксический анализатор.

1. синтаксический анализатор;

По полученному списку токенов строит синтаксическое дерево. Затем передаёт синтаксическое дерево в генератор.

1. генератор;

По полученному синтаксическому дереву генерирует готовый Tkinter-код. Затем передаёт сгенерированный код в препроцессор.

1. постпроцессор.

Завершает работу транслятора, создавая файл и записывая в него пользовательский и сгенерированный код.

Схема работы транслятора изображена на рисунке 5.

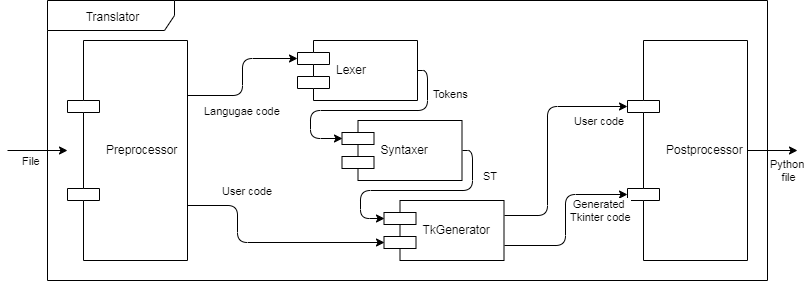


Рисунок 5 – Схема работы транслятора

## 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, значения которых затем подставляются в графические компоненты.

Для решения задачи разделения и исполнения кода решено использовать макросы. Макросы позволят 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 8.

Таблица 8 – 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 приведена UML-диаграмма (Рисунок 6), на которой изображён класс Preprocessor.

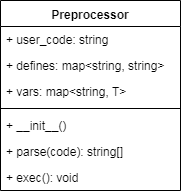


Рисунок 6 – UML-диаграмма класса Preprocessor

Дадим краткое описание методам класса Lexer в нижеприведённой таблице 9.

Таблица 9 – Описание методов класса Lexer

|  |  |
| --- | --- |
| **Метод** | **Описание** |
| \_\_init\_\_() | Конструктор класса |
| parse(code) | Разделяет пользовательский код от кода, описывающий графический интерфейс и записывает распознанные макросы в словарь |
| exec() | Выполняет пользовательский код и записывает полученные переменные в словарь |

## 4.2 Лексический анализатор

Лексический анализатор разбирает код на лексемы и по ним он создаётся список токенов. В свою очередь, токен – это структурная единица, которая содержит лексему и 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 приведена UML-диаграмма (Рисунок 7), на которой изображены классы: Lexer, Token и TokenType.

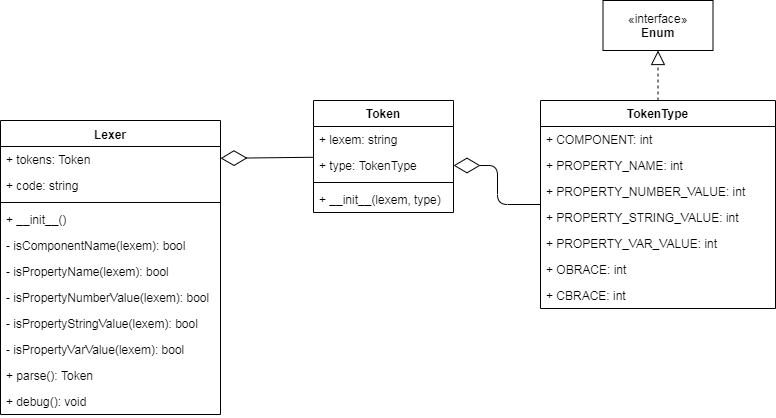


Рисунок 7 – UML-диаграмма классов Lexer, Token и TokenType

Дадим краткое описание методам класса Lexer в нижеприведённой таблице 10.

Таблица 10 – Описание методов класса Lexer

|  |  |
| --- | --- |
| **Метод** | **Описание** |
| \_\_init\_\_() | Конструктор класса. Создаёт пустой список токенов |
| isComponentName(lexem) | Проверяет полученную лексему на соответствие имени компонента (по регулярному выражению, приведённому в пункте 2.4, таблица 6) |
| isPropertyName(lexem) | Проверяет полученную лексему на соответствие имени свойства (по регулярному выражению, приведённому в пункте 2.4, таблица 6) |
| isPropertyNumberValue(lexem) | Проверяет полученную лексему на соответствие числовому значению (по регулярному выражению, приведённому в пункте 2.4, таблица 6) |
| isPropertyStringValue(lexem) | Проверяет полученную лексему на соответствие строковому значению (по регулярному выражению, приведённому в пункте 2.4, таблица 6) |
| isPropertyVarValue(lexem) | Проверяет полученную лексему на соответствие имени переменной |
| parse() | Осуществляет разбор кода на лексемы и формирует список токенов |
| debug() | Выводит список tokens по окончанию работы |

## 4.3 Синтаксический анализатор

Синтаксический анализатор по полученному списку лексем создаёт синтаксическое дерево. Исходя из раздела 3 и описанного в нём синтаксического дерева, нам необходимо создать класс, который описывал бы узел дерева и содержал соответствующие поля:

* название компонента;
* свойства (в качестве словаря);
* список компонентов (такие же узлы).

Ниже приведена UML-диаграмма (Рисунок 8), на которой приведены классы: Syntaxer и ComponentNode.

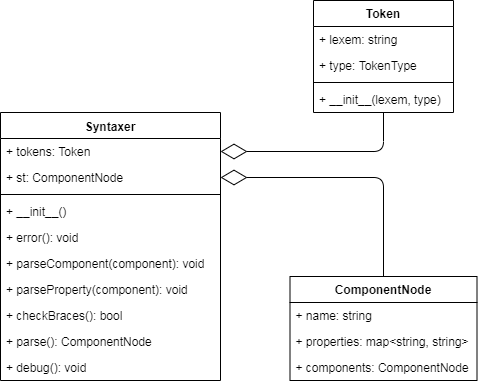


Рисунок 8 – UML-диаграмма класса Syntaxer и ComponentNode

Дадим краткое описание методам класса Syntaxer в нижеприведённой таблице 11.

Таблица 11 – Описание методов класса Syntaxer

|  |  |
| --- | --- |
| **Метод** | **Описание** |
| \_\_init\_\_() | Конструктор класса. Создаёт корневой элемент синтаксического дерева |
| error() | Формирует сообщение об ошибке и инициирует исключение |
| parseComponent(component) | Метод реализует рекурсивный спуск для разбора компонента (по правилу component из пункта 2.4, таблицы 5) |
| parseProperty(component) | Метод реализует рекурсивный спуск для разбора свойств (по правилу property из пункта 2.4, таблицы 5) |
| checkBraces() | Проверка открытых и закрытых фигурных скобок. Возвращает True или False |
| parse() | Метод реализует рекурсивный спуск для разбора основного компонента (по правилу syntax из пункта 2.4, таблицы 5) |
| debug() | Вывод синтаксическое дерево по окончанию работы |

## 4.4 Генератор

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

Процесс генерации включает себя три этапа:

1. генерация компонента;
2. генерация свойств;
3. генерация дочерних компонентов.

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

Ниже приведена UML-диаграмма класса TkGenerator (Рисунок 9), на которой приведены классы TkGenerator и ComponentNode.

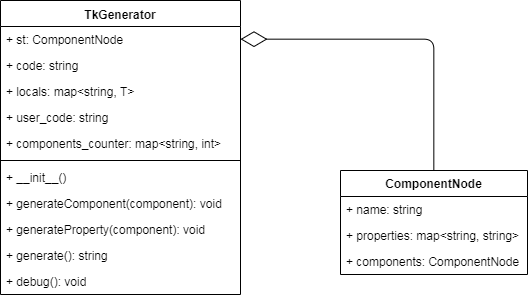


Рисунок 9 – UML-диаграмма класса TkGenerator

Дадим краткое описание методам класса TkGenerator в нижеприведённой таблице 12.

Таблица 12 – Описание методов класса Syntaxer

|  |  |
| --- | --- |
| **Метод** | **Описание** |
| \_\_init\_\_() | Конструктор класса |
| generateComponent(component) | Генерация переданного компонента |
| generateProperty(component) | Генерация свойств переданного компонента |
| generate() | Генерация основного компонента |
| debug() | Вывод сгенерированного кода, списка пользовательских переменных после окончания работы |

## 4.5 Постпроцессор

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

Ниже приведена UML-диаграмма класса Preprocessor (Рисунок 10).

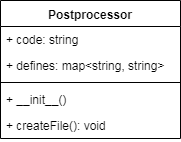


Рисунок 10 – UML-диаграмма класса Postprocessor

Дадим краткое описание методам класса Syntaxer в нижеприведённой таблице 13.

Таблица 13 – Описание методов класса Syntaxer

|  |  |
| --- | --- |
| **Метод** | **Описание** |
| \_\_init\_\_() | Конструктор класса |
| createFile() | Создаёт целевой Python файл |

## 5 Изучение GUI-библиотеки Tkinter

Tkinter – это библиотека для создания графического интерфейса для языка Python. Рассмотрим основные возможности, достоинства и недостатки.

Основными её особенностями являются:

1. кроссплатформенность;
2. наличие обширной документации.

В составе пакета Tkinter входя модули, такие как:

* scrolledtext;

Модуль предоставляющий scrollbar виджет.

* colorchooser;

Модуль предоставляющий диалоговое окно для выбора цвета.

* commondialog;

Модуль предоставляющий стандартные диалоговые окна, такие как сохранение и открытие файла.

* font;

Модуль предоставляющий инструменты для работы со шрифтами.

* messagebox;

Модуль предоставляющий окна для вывода сообщений.

* simpledialog;

Модуль предоставляющий средства для создания собственных диалоговых окон.

* dnd.

Модуль предоставляющий средства для поддержки Drag&Drop.

К достоинствам библиотеки можно отнести следующее:

* является частью стандартной библиотеки Python;
* позволяет создавать исполняемые приложения;
* отсутствие зависимостей с другими библиотеками;
* позволяет быстро создавать GUI за счёт простого API.

К недостаткам библиотеки относятся:

* отсутствие дизайнера форм;
* отсутствие дополнительных виджетов, например вкладок.

## 5.1 Компоненты

Tkinter позволяет создавать различные компоненты, от обычных кнопок и надписей до контекстных меню. Для краткости приведём основные компоненты (Таблица 14).

Таблица 14 – 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 |

## 5.2 Пример кода

Рассмотрим пример кода, написанный с помощью Tkinter, чтобы понять, каким образом создаётся на графический интерфейс (Таблица 15).

Таблица 15 – 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 11.

http://effbot.org/media/cache/76734d5c44455920d98197bd2958fec7.gif

Рисунок 11 – Пример приложения на Tkinter

Проанализировав структуру кода можно выделить определённую последовательность действий в создании приложения:

1. создание главного компонента (root);
2. создание дочернего/дочерних компонента/компонентов с указанием родительского (frame, button);
3. задание свойств дочернему компоненту;
4. задание способа размещения;
5. запуск цикла приложения.

Стоит отметить, что свойства компонента можно задавать, обращаясь к нему как к словарю. В дальнейшем, при разработке генератора, будет использоваться вышеприведённое последовательность действий для генерации кода.

## 5.3 Средства размещения

Стоит рассмотреть средства размещения виджетов. В Tkinter присутствуют три средства для размещения виджетов, называемых менеджерами:

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

Каждый менеджер вызывается одноимённым методом.

## 5.3.1 Grid geometry manager

Менеджер, используемый для размещения виджетов внутри родительского компонента по сетке.

Для расположения элемента используется метод grid, вызываемый у любого элемента, в котором указываются номер строки и колонки. Аргументы, которые принимает метод grid перечислены в таблице 16.

Таблица 16 – Описание аргументов метода grid

|  |  |
| --- | --- |
| **Свойство** | **Описание** |
| row | Строка |
| column | Колонка |
| rowspan | Объединяет указанное кол-во строк |
| colspan | Объединяет указанное кол-во колонок |
| padx | Внешний по y |
| pady | Внешний отступ по x |
| ipadx | Внутренний отступ по x |
| ipady | Внутренний отступ по y |
| in | Указывает виджет, в котором будет размещён данный виджет |

## 5.3.2 Pack geometry manager

Менеджер, используемый по умолчанию для размещения виджетов внутри родительского компонента.

Для расположения используется метод pack, в котором главным образом указывается сторона, относительно которой будет выровнен виджет. Аргументы метода pack перечислены в таблице 17.

Таблица 17 – Описание аргументов метода grid

|  |  |
| --- | --- |
| **Свойство** | **Описание** |
| anchor | Расположение виджета на форме, принимает значения: n, ne, e, se, s, sw, w, nw и center |
| side | Расположение виджета внутри родительского виджета, принимает значения: LEFT, RIGHT, TOP, BOTTOM |
| fill | Заполнение виджета по всему пространству, принимает значения: X, Y, BOTH, NONE |
| expand | Свойство позволяющее включить заполнение виджета по всему пространству, принимает значения: True или False |
| padx | Внешний по y |
| pady | Внешний отступ по x |
| ipadx | Внутренний отступ по x |
| ipady | Внутренний отступ по y |

## 5.3.3 Place geometry manager

Менеджер, используемый для размещения виджетов внутри родительского компонента, однако в отличии от других позволяет задать напрямую координаты x, y, а также width, height.

Для расположения используется метод place, в котором главным образом указываются координаты. Аргументы метода place перечислены в таблице 18.

Таблица 18 – Описание аргументов метода place

|  |  |
| --- | --- |
| **Свойство** | **Описание** |
| anchor | Расположение виджета на форме, принимает значения: n, ne, e, se, s, sw, w, nw и center |
| x | Координата x |
| y | Координата y |
| width | Ширина |
| height | Высота |
| in | Указывает виджет, в котором будет размещён данный виджет |

## 6 Проектирование и реализация генератора

Описание генератора было дано в пункте 4.4, однако следует описать то как генерируются сами компоненты. Каждый компонент представляет собой класс и реализует базовые методы родительского класса, то есть абстрактного компонента.

Все перечисленные компоненты в пункте 2.3 должны представляться в видео отдельных классов. Это позволит в дальнейшем расширять их, добавляя новый свойства, которые присутствуют только у них.

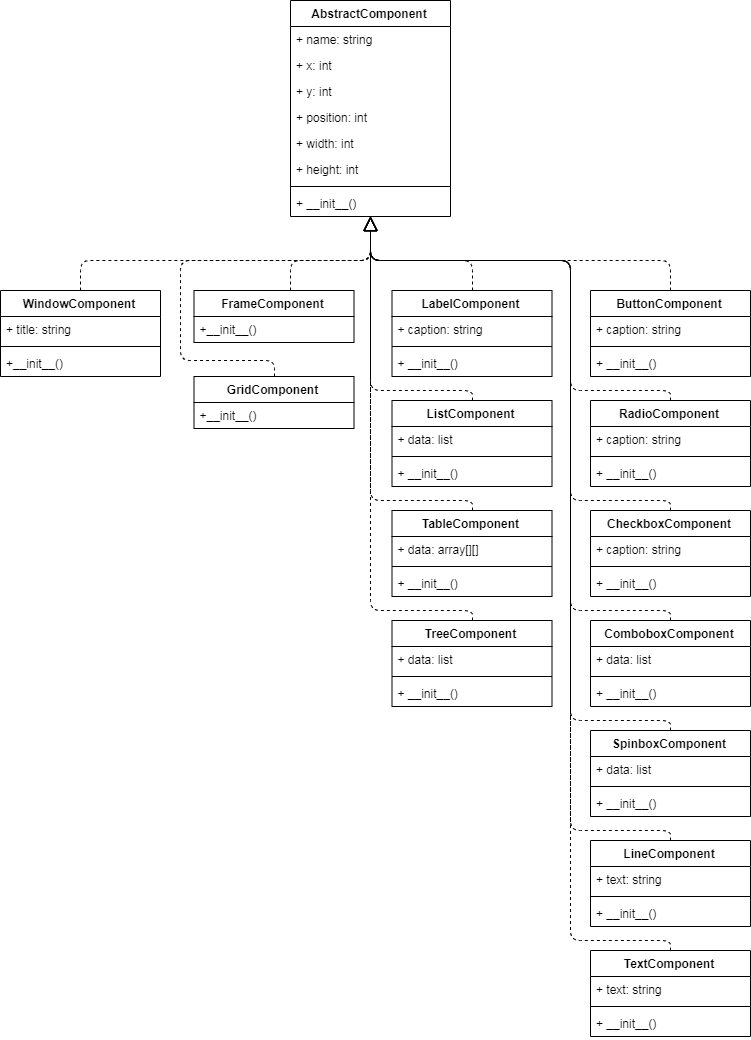


Рисунок 12 – UML-диаграмма классов компонентов

## 7 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.

Такое приложение удобно для использования в среде Unix и Windows. Список принимаемых команд приведен в таблице 16.

Таблица 19 – Команды консольного приложения

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

Схема работы с приложением изображена на рисунке …:

1. создаётся файл;
2. в него записывается код, описывающий графический интерфейс;
3. файл передаётся на вход консольного приложения и также предаются дополнительные команды;
4. само приложение передаёт код из файла транслятору;
5. идёт процесс трансляции;
6. если во время трансляции ошибок нет, то получает Python файл;
7. иначе инициируется исключение и выводится сообщение об ошибке.

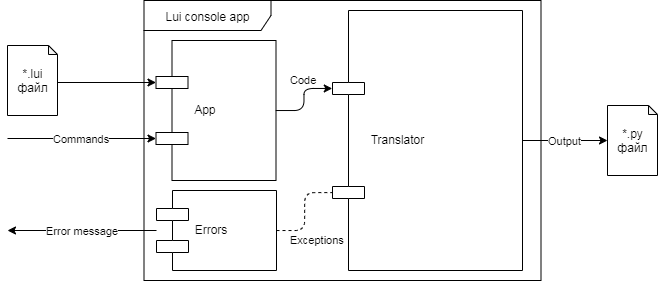


Рисунок 13 – Схема работы консольного приложения Lui

## 8 Тестирование

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

На рисунках … приведены результаты прохождения тестов.

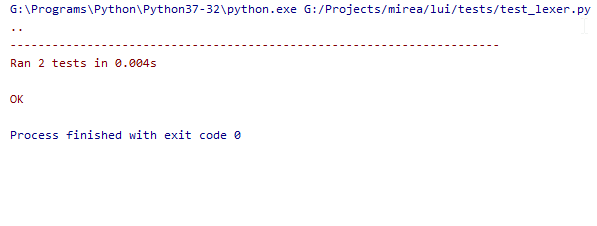


Рисунок 14 – Результаты тестирования класса Lexer

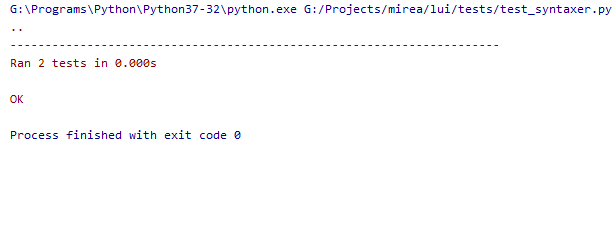


Рисунок 15 – Результаты тестирования класса Syntaxer

# заключение

В результате выполнения курсового проекта был спроектирован и разработан языка декларативного описания GUI Lui, позволяющий автоматически сгенерировать код под GUI-библиотеку Tkinter.

В процесс выполнения были выполненные следующие этапы:

1. анализ декларативного подхода и существующих решений
2. проектирование декларативного языка;
3. проектирование и реализация транслятора;
4. изучение GUI-библиотеки Tkinter;
5. проектирование и реализация генератора кода;
6. проектирование и реализация консольного приложения
7. тестирование.

Как и любое программное решение, данный язык имеет свои достоинства и недостатки.

Из достоинств стоит отметить:

* простоту декларативного языка;
* наличие документации;
* открытый исходный код языка;
* кроссплатформенность.

Из недостатков стоит отметить:

* отсутствие поддержки обработки событий;
* поддерживает только библиотеку Tkinter.

# 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);

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/.

# приложение а

**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 Lexer

|  |
| --- |
| **test\_lexer.py** |
| 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() |

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

|  |
| --- |
| **test\_syntaxer.py** |
| 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["heoght"] = 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() |