

Министерство науки и высшего образования Российской Федерации
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО»
(Университет ИТМО)

Факультет Инфокоммуникационных технологий

Образовательная программа **Мобильные и облачные технологии**

Направление подготовки (специальность) 09.04.03 Прикладная информатика

О Т Ч Е Т

о производственной, технологической (проектно-технологической) практике

Тема задания: Реализация подмодуля парсинга ODT документов как часть системы автоматизированного нормоконтроля.

Обучающийся Терещенко Владислав Витальевич, группа K42402

Руководитель практики от университета: Горлушкина Н.Н., доцент, к.т.н..

Практика пройдена с оценкой _____

Дата _____

Санкт-Петербург, 2023

РЕФЕРАТ

Отчет 25 с., 11 рис., 10 источн.

ПАРСИНГ, ODT, СТРУКТУРНЫЕ ЭЛЕМЕНТЫ, XML, PYTHON, ODFPY.

Цель работы – реализовать подмодуль парсинга ODT документов как часть системы автоматизированного нормоконтроля.

В процессе работы была спроектирована и реализована архитектура подмодуля, оптимизированы алгоритмы парсинга ODT документов и модернизированы классы структурных элементов для связи с алгоритмами, извлекающими характеристики структурных элементов.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ.....	4
Актуальность	4
Постановка проблемы.....	4
Цели и задачи	5
1 Разработка архитектуры подмодуля парсинга ODT документов.....	6
2 Оптимизация алгоритмов парсинга ODT документов	12
3 Модернизация классов структурных элементов и связка их с парсингом ODT документов	18
ЗАКЛЮЧЕНИЕ	24
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ	25

ВВЕДЕНИЕ

Актуальность

В процессе написания научных и выпускных квалификационных работ студенты, а иногда и научные сотрудники, допускают ошибки при оформлении. В результате текст работы может не соответствовать ГОСТам или другим различным стандартам. Вместе с тем процесс проверки работ научными руководителями и иными уполномоченными лицами проводится преимущественно вручную, что требует больших затрат человеческих усилий. Именно поэтому автоматизация описанного процесса является достаточно актуальной.

В рамках данной практической работы было принято решение сосредоточиться на доработке функционала разрабатываемого сервиса автоматизированного нормоконтроля документов.

Постановка проблемы

В ходе предыдущих этапов работ была подробно исследована структура электронных документов форматов DOCX, PDF, ODT и возможность парсинга таких файлов с использованием высокоуровневых языков Python и Java [1]. Также были исследованы особенности хранения таблиц, списков, изображений и их характеристик. Была выявлена необходимость для объектов проверять содержимое файла content.xml на наличие в нем связанных с ними дополнительных элементов. Полученные данные позволили реализовать алгоритмы, извлекающие описания этих объектов и их характеристики как по названию объекта, так и по названию характеристики.

Были проанализированы и описаны особенности разметки в файлах формата ODT, конвертированных из редактора текста Pages для устройств на платформе iOS и macOS. Полученные данные позволят либо унифицировать алгоритмы парсинга, либо выявить какие изменения необходимо внести, учитывая при этом особенности как можно большего количества различных текстовых редакторов.

Были реализованы унифицированные классы основных структурных элементов и другого содержимого документа, которые затем были интегрированы в алгоритмы, извлекающие данные из объектов документа.

Цели и задачи

Цель производственной, технологической (проектно-технологической) практики заключается в реализации подмодуля парсинга ODT [2] документов как часть системы автоматизированного нормоконтроля.

В качестве задач для работы были выделены следующие этапы:

- разработать и реализовать архитектуру подмодуля парсинга документов формата ODT,
- провести оптимизацию и модернизацию существующих алгоритмов для извлечения характеристик объектов электронных документов формата ODT,
- модернизировать классы, хранящие необходимые характеристики, получаемые из структурных элементов документа с помощью обновленных алгоритмов парсинга ODT документов.

1 Разработка архитектуры подмодуля парсинга ODT документов

Разрабатываемая система автоматизированного нормоконтроля документов, схема которой изображена на рисунке 1, состоит из множества взаимосвязанных модулей и сервисов. Основными среди них являются следующие:

— пакет NLP API – позволяет анализировать отчет документа, а именно определить содержит ли он оглавление и введение, проверяет корреляцию текста введению, предсказывает наличие необходимого содержимого и к какому департаменту университета относится документ,

— классификатор – определяет и прогнозирует виды структурных элементов и контента документа,

— пакет Transport API – приложение, реализованное с помощью фреймворка Flask [3], реализующее передачу данных между модулями сервиса,

— библиотека PyOPWParse – библиотека парсинга и унифицированных структурных элементов для электронных документов форматов ODT, PDF и Microsoft Word, структура которой изображена на рисунке 2,

— веб-приложение normcontrol.ru – клиент-серверное приложение, реализованное с помощью фреймворка Django [4], позволяющее пользователям проходить нормоконтроль.

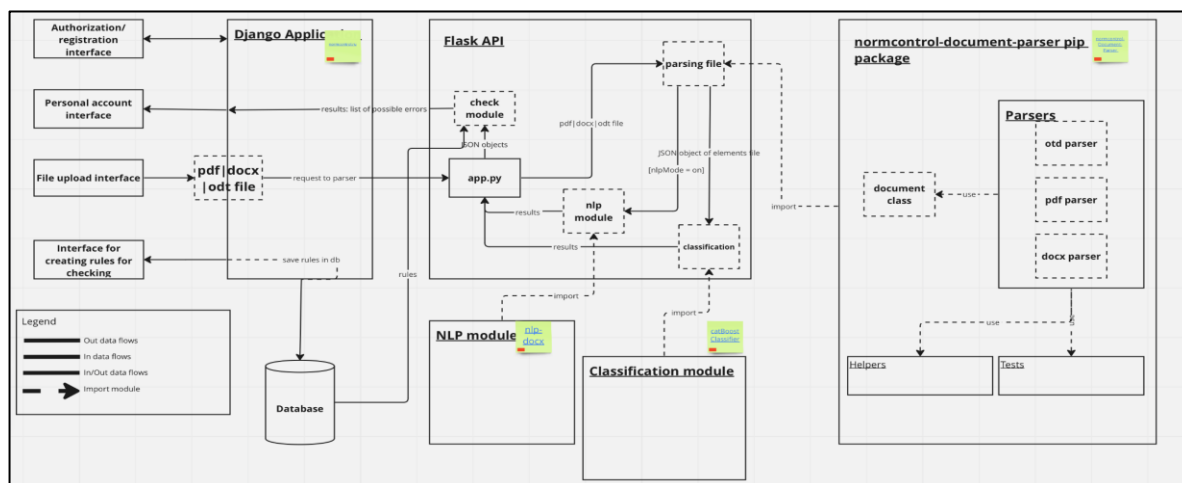


Рисунок 1 – Схема системы автоматизированного нормоконтроля

Модуль парсинга документов, структура которого изображена на рисунке 3, состоит из трех основных подмодулей, которые реализуют парсинг документов форматов ODT, PDF и Word, подмодуля, содержащего классы унифицированных структурных элементов и другого контента, и дополнительные модули с документацией, вспомогательными функциями и тестами.










 .github/workflows	[FIX] Moved tests directory to root folder
 docs	[feat]: updated elements page in ODT documentation
 examples/pdf_example	[Fix] Tests and examples are highlighted in the root of the project
 src	[fix]: modified Table class & list classes are aggregated in a single...
 tests	[FEAT] Интеграция с классом src/Class/Paragraph.py (#41)
 .gitignore	Initial commit
 LICENSE.md	Create LICENSE.md
 README.md	[FEAT] Update README.MD of project
 requirements.txt	Merge branch 'PDF_Pictures'

Рисунок 2 – Структура библиотеки PyOPWParse










 PDF	[Fix] Tests and examples are highlighted in the root of the project
 classes	[fix]: modified Table class & list classes are aggregated in a single...
 doc	[FEAT] Интеграция с классом src/Class/Paragraph.py (#41)
 helpers	[FEAT] Интеграция с классом src/Class/Paragraph.py (#41)
 odt	[fix]: fixed errors in the ODT submodule according to the review
 pdf/pdfclasses	[Fix] Changed documentation on classes and methods
 Flask.py	[Feat] Redirect Flask.py file
 __init__.py	[feat]: added string method to classes and initialized packages
 settings.ini	[Fix] Changed documentation on classes and methods

Рисунок 3 – Структура парсеров библиотеки PyOPWParse

В процессе разработки подмодуля, реализующего парсинг документов формата ODT, было принято решение выделить отдельно все методы парсинга автоматических, регулярных стилей и стилей по умолчанию. Методы парсинга данных структурных элементов реализовывались в отдельных скриптах. Однако, ввиду работы постоянной работы со стилями, часть методов парсинга структурных элементов располагалась в скриптах парсинга стилей, что вызывало сложности при внесении изменений. Дополнительно для большего

удобства каждый скрипт парсинга структурных элементов был исполняемым, что позволяло в реальном времени проверять работоспособность кода. Подобная организация подмодуля была удобна, но совершенно непонятна сторонним разработчикам и ошибочна с точки зрения проектирования и паттернов разработки программного обеспечения. Отдельно следует отметить, что в результате подобной организации кодовая база некоторых пакетов получилась маленькой, других же наоборот большой. Дополнительно возникали сложности с расширением созданных программных пакетов.

При проектировании новой архитектуры, ввиду оптимизации и переориентации модуля на библиотеку с открытым исходным кодом, была проведена более четкая грань в разделении методов и зон ответственности пакетов. Требовалось спроектировать новую систему с максимальным потенциалом для расширения и возможностей поддержки. Было решено разбить подмодуль на пять программных пакетов взаимосвязанных между собой. Пакет — это каталог или директория с файлами модулей и имеющая специальный файл “__init__.py”. Наличие этого специального файла указывает интерпретатору языка, что данный каталог необходимо воспринимать как пакет, и дополнительно помогает импортировать реализованные модули в другие пакеты и модули. Также было важно добавить типизацию в аргументы функций и обеспечить возможность тестирования методов парсинга без особых затрат, для упрощения реализации и поддержки тестов. В результате были созданы следующие группы классов, изображенные на рисунке 4:

- класс `ODTDocument` – содержит путь к документу и его загруженный в память образ,

- класс `ODTParser` – специальный объект, являющийся оберткой над модулями парсинга объектов и предоставляющий доступ к их функционалу,

- пакет `Helpers` – содержит вспомогательные методы для работы с документами:

1) `converters` – скрипт, содержащий методы конвертации данных для создания экземпляров унифицированных классов;

2) `path` – скрипт, содержащий функционал по восстановлению абсолютного пути к файлу для устройств под управлением разных операционных систем;

3) `timing` – специальный модуль для определения времени работы программы и затраченной памяти;

4) `consts` – файл с глобальными переменными, хранящими значения по умолчанию для характеристик различных структурных элементов;

— `Elements` – пакет, содержащий классы-парсеры структурных элементов и контента документов:

1) класс `AutomaticStylesParser` – содержит методы для работы с автоматическими стилями ODT документа;

2) класс `RegularStylesParser` – содержит методы для работы с регулярными стилями ODT документа;

3) класс `DefaultStylesParser` – содержит методы для работы со стилями по умолчанию ODT документа;

4) класс `ImagesParser` – содержит методы для работы с изображениями ODT документа;

5) класс `ListsParser` – содержит методы для работы со списками ODT документа;

6) класс `NodesParser` – содержит методы для работы с узлами стилей ODT документа;

7) класс `TablesParser` – содержит методы для работы с таблицами, колонками, столбцами и ячейками ODT документа;

8) класс `ParagraphsParser` – содержит методы для работы с параграфами ODT документа.

— пакет `Examples` – содержит примеры использования парсера для извлечения данных из документа.

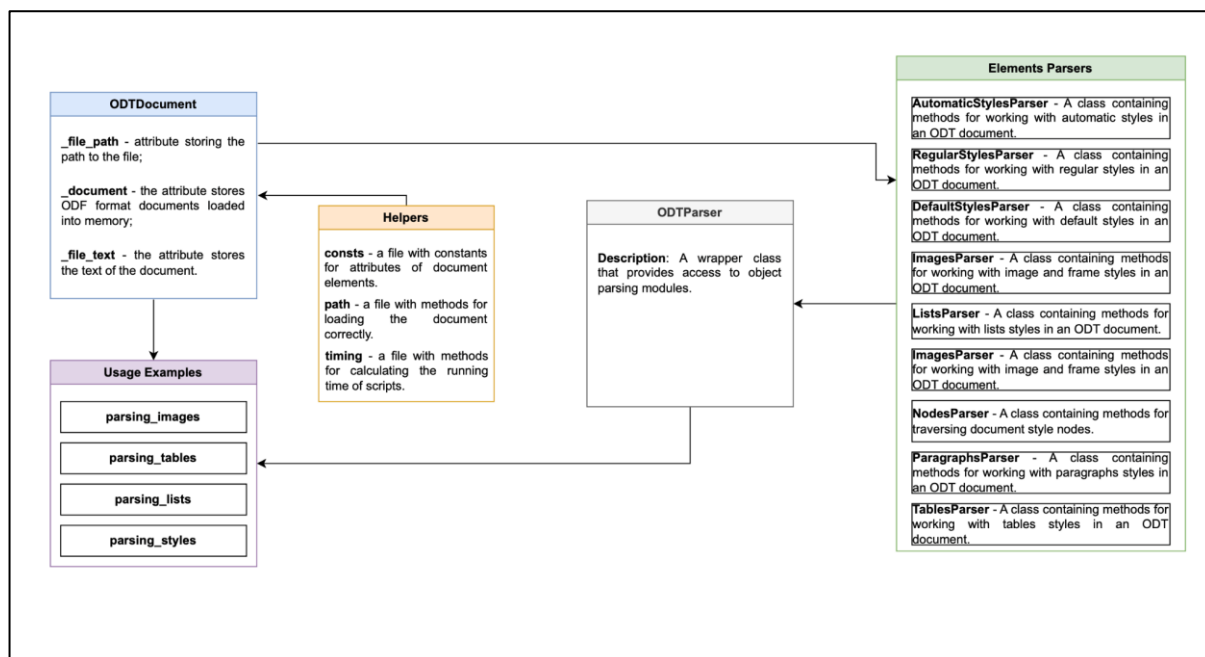


Рисунок 4 – Диаграмма созданной архитектуры подмодуля парсинга ODT документов

Во все пакеты и классы подмодуля была добавлена подробная документация в виде Docstring в соответствии со стандартами языка Python PEP 526 [5]. Docstring – это специальная строковая переменная, идущая сразу за объявлением модуля, функции, класса или метода. Данный функционал предоставляется стандартной библиотекой языка Python и является удобным способом добавления документации. Данный объект похож на комментарии, но он обернут в тройные кавычки. В соответствии с конвенциями языка все функции и сущности должны иметь docstring, содержащий общую информацию и описание работы или использования комментируемого функционала. Python также предоставляет возможность извлекать реализованную документацию, путем обращения к атрибуту `__doc__` у соответствующего объекта. В рамках дальнейшей работы планируется подключить вспомогательных ботов для проверки качества добавляемого кода в репозиторий и средства автоматической генерации документации. Документация репозитория и проекта в целом была реализована при помощи инструмента Docsify [6]. Он позволяет генерировать документацию на лету благодаря анализу и загрузке файлов, реализованных с помощью языка

разметки Markdown. Затем эти файлы отображаются как веб-сайт без генерации html-страниц.

Обновленная архитектура подмодуля имеет преимущества перед своей предыдущей версией. Так, за счет разделения зон ответственности, удалось увеличить скорость работы реализованного функционала, что заметно для сложных и тяжеловесных методов, ориентированных на обработку всего документа. Например, время обхода всех узлов документа с пяти – десяти минут сократилось до пятнадцати секунд. Дополнительно, поскольку библиотека разрабатывается как open-source решение, повышается устойчивость системы к расширению сторонними разработчиками.

2 Оптимизация алгоритмов парсинга ODT документов

Процесс оптимизации неразрывно связан с изменением архитектуры всего подмодуля в целом. Важным обновлением является изменение логики работы с документом. До этого каждый метод принимал на вход путь к документу, что приводило к множественной загрузке одного и того же документа в память и дублированию логики загрузки для каждого метода. Эта проблема была решена добавлением специального класса документа, в который при инициализации единожды загружается документ, а использовать его можно везде, где необходимо. Дополнительно часть методов в процессе разделения логики на пакеты была удалена из проекта по причине устаревания или дублирования уже реализованного функционала. В результате были получены следующие классы с реализованным функционалом:

— класс `AutomaticStylesParser`:

1) `get_automatic_styles(doc: ODTDocument)` – Возвращает список всех автоматических стилей документа с их атрибутами;

2) `get_text_styles_from_automatic_styles(doc: ODTDocument)` – Возвращает список всех стилей текста с их атрибутами из автоматических стилей документа;

3) `get_automatic_style_by_name(doc: ODTDocument, style_name: str)` – Выполняет поиск стиля по имени среди автоматических стилей и возвращает его атрибуты;

4) `get_text_style_by_name(doc: ODTDocument, text_name: str)` – Выполняет поиск текстового стиля по имени среди автоматических стилей и возвращает его атрибуты;

5) `get_automatic_style_object_by_name(doc: ODTDocument, style_name: str)` – Выполняет поиск стиля по имени среди автоматических стилей и возвращает его как объект для работы;

6) `has_automatic_style_parameter(doc: ODTDocument, style, param_name: str, default: str, property_type: str)` – Проверяет содержится ли параметр в автоматических стилях;

7) `is_automatic_style(self, doc: ODTDocument, style_name: str, param_name: str, property_type: str, default: str)` – Определяет является ли указанный стиль автоматическим или встроенным в редактор;

— класс `RegularStylesParser`:

1) `get_regular_styles(doc: ODTDocument)` – Возвращает список всех обычных стилей документа с их атрибутами;

2) `get_text_styles_from_regular_styles(doc: ODTDocument)` – Возвращает список всех стилей текста с их атрибутами из обычных стилей документа;

3) `get_regular_style(doc: ODTDocument, style_name: str)` – Возвращает список атрибутов заданного обычного стиля;

4) `get_regular_style_object(doc: ODTDocument, style_name: str)` – Выполняет поиск стиля по семейству стилей среди стилей по умолчанию и возвращает его как объект для работы;

5) `get_parameter_from_regular_style_fast(doc: ODTDocument, default: str, style_name: str, param_name: str, property_type: str)` – Проверяет наличие указанного стиля в стилях редактора без рекурсии;

6) `get_parameter_from_regular_style(doc: ODTDocument, default: str, style_name: str, param_name: str, property_type: str)` – Проверяет наличие указанного стиля в стилях редактора с использованием рекурсии;

7) `get_paragraph_alignment(doc: ODTDocument, style_name: str)` – Возвращает значение выравнивания параметра абзаца;

— класс `DefaultStylesParser`:

1) `get_default_styles(doc: ODTDocument)` – Возвращает список всех стилей по умолчанию документа с их атрибутами;

2) `get_default_style_by_name(doc: ODTDocument, style_family: str)` – Выполняет поиск стиля по семейству стилей среди стилей по умолчанию и возвращает его атрибуты;

3) `get_default_style_object_by_family(doc: ODTDocument, style_family: str)` – Выполняет поиск стиля по семейству стилей среди стилей по умолчанию и возвращает его как объект для работы;

4) `get_default_style_parameters(style, parameter_name: str, property_type: str)` – Получает параметр стиля по имени и атрибуту среди стилей по умолчанию;

5) `has_default_parameter(self, doc: ODTDocument, style_parameter: str, family: str, parameter_name: str, property_type: str)` – Проверяет содержится ли параметр в стилях по умолчанию;

— класс `ImagesParser`:

1) `get_frame_styles(doc: ODTDocument)` – Возвращает список всех стилей рамок документа с их атрибутами;

2) `get_image_styles(doc: ODTDocument)` – Возвращает список всех стилей изображений документа с их атрибутами;

3) `get_frame_parameter(doc: ODTDocument, style_name: str, parameter_name: str)` – Получает параметр стиля по имени и атрибуту среди стилей рамок;

4) `get_image_parameter(doc: ODTDocument, style_name: str, parameter_name: str)` – Получает параметр стиля по имени и атрибуту среди стилей изображений;

— класс `ListsParser`:

1) `get_list_styles(doc: ODTDocument)` – Возвращает список всех стилей списков документа с их атрибутами;

2) `get_lists_text_styles(doc: ODTDocument)` – Возвращает список всех текстовых стилей с их атрибутами из списка стилей документа;

3) `get_list_styles_from_automatic_styles(doc: ODTDocument)` – Возвращает список всех стилей списков из автоматических стилей документа с их атрибутами;

4) `get_list_parameter(style, parameter_name: str)` – Возвращает параметр стиля по атрибуту среди стилей списков;

— класс TablesParser:

1) `get_table_styles(doc: ODTDocument)` – Возвращает список всех стилей таблиц документа с их атрибутами;

2) `get_automatic_table_styles(doc: ODTDocument)` – Возвращает список всех стилей таблиц из автоматических стилей документа с их атрибутами;

3) `get_table_parameter(style, parameter_name: str)` – Возвращает параметр стиля по атрибуту среди стилей таблиц;

4) `get_table_row_parameter(style, parameter_name: str)` – Возвращает параметр стиля по атрибуту среди стилей строк таблиц;

5) `get_table_column_parameter(style, parameter_name: str)` – Возвращает параметр стиля по атрибуту среди стилей столбцов таблиц;

6) `get_table_cell_parameter(style, parameter_name: str)` – Возвращает параметр стиля по атрибуту среди стилей ячеек таблиц;

7) `get_all_odt_tables_text(doc: ODTDocument)` – Возвращает текст всех таблиц документа;

— класс NodesParser:

1) `print_all_document_nodes(start_node, level=0)` – Выводит каждый узел документа и его атрибуты;

2) `print_all_document_nodes_with_style(start_node, global_style_name, doc: ODTDocument, level=0)` – Выводит каждый текстовый узел документа и его атрибуты с прохождением по всем стилям, выполняя поиск на одном уровне вложенности;

3) `print_all_document_nodes_with_higher_style_data((start_node, attributes_list, doc: ODTDocument, level=0)` – Выводит каждый текстовый узел документа и его атрибуты с прохождением по всем стилям, выполняя поиск, в случае отсутствия, в узлах уровнем выше;

— класс ParagraphsParser:

1) `get_paragraph_parameters(style, parameter_name: str, property_type: str)` – Возвращает значение искомого параметра абзаца;

2) `get_paragraph_properties_from_automatic_styles(doc: ODTDocument, style_name: str)` – Возвращает значение параметра абзаца из автоматических стилей;

3) `get_paragraph_styles_from_regular_styles(doc: ODTDocument)` – Возвращает параметры текста из обычных стилей.

Для тестирования реализованного функционала использовался модуль `unittest` [7, 8], встроенный в стандартную библиотеку языка Python. По формату написания тестов данная библиотека схожа с библиотекой JUnit для написания тестов на языке Java:

- 1) тесты должны быть реализованы в классе,
- 2) класс должен наследовать базовый класс `unittest.TestCase`,
- 3) имена всех функций, являющихся тестами, должны начинаться с ключевого слова `test`,
- 4) внутри функций вызываются операторы сравнения, начинающиеся с ключевого слова `assert`, где происходит проверка на соответствие полученных значений заявленным.

Сами классы представляют собой юнит-тесты. Такие тесты являются обязательной частью жизненного цикла программного обеспечения и формируют основу для проведения регрессионного тестирования. Иными словами, они гарантируют, что продукт ведет себя согласно определенному сценарию даже после изменения или добавления новых функциональных возможностей. Сам тест имеет следующую структуру:

- 1) `setUp()` – стандартный метод, вызываемый для подготовки тестового окружения. Вызывается непосредственно перед запуском тестирования, реализация по умолчанию ничего не делает,
- 2) далее идут сами тесты для функционала,
- 3) `tearDown()` – стандартный метод, вызывающийся сразу после окончания тестирования и записи результата. Вызов происходит даже если тестовый метод вызвал исключение, поэтому необходимо проявлять особую осторожность при проверке внутреннего состояния. Любое исключение.

отличное от `AssertionError` или `SkipTest`, вызванное этим методом, будет рассматриваться как дополнительная ошибка, а не сбой теста. Таким образом общее количество зарегистрированных ошибок увеличится.

Методы `setUp()` и `tearDown()` поставляются фреймворком `unittest` и в зависимости от конкретного тестового случая можно либо переопределять, либо не переопределять два этих метода по умолчанию. В дальнейшем весь функционал разрабатываемой библиотеки будет покрыт тестами.

3 Модернизация классов структурных элементов и связка их с парсингом ODT документов

Поскольку унифицированные классы объектов документа являются важнейшим аспектом всего проекта в целом, они постоянно модифицируются и дорабатываются. В частности, для удобства работы с данными был произведен переход на классы данных, или *dataclasses*. *Dataclasses* [9] — это специальные классы, предназначенные для хранения данных и призванные автоматизировать генерацию кода для классов. Основная идея данного подхода заключается в том, что, когда разработчики используют классы, которые используются как контейнеры данных, тратится значительное количество времени на написание шаблонного кода с большим количеством аргументов и сопутствующими неопределенными функциями. Классы данных решают эту проблему, предоставляя ряд дополнительных полезных методов.

Единый объект для изображений и фреймов был разделен на два разных, которыми они фактически и являются. Также это связано с тем, что в формате ODT существует понятие фрейма для картинок и других графических объектов, но в остальных форматах его может не быть.

Объекты, отражающие маркированные и нумерованные списки, были объединены в единую сущность с добавлением дополнительного атрибута, ответственного за тип списка. Изменения также коснулись классов таблиц. Так для класса *Table* была добавлена возможность хранения связанных с таблицей столбцов, строк и ячеек.

Отдельно следует добавить, что для всех свойств классов были добавлены значения по умолчанию, ввиду того что классы создаются унифицированными, и характеристики объектов могут зависеть от формата документа. В дальнейшем планируется связать эти значения с данными из нормативных документов.

Обновленные объекты также интегрированы в обновленные алгоритмы, получающие данные из структурных элементов. Так в алгоритме, разработанном для таблиц и изображенном на рисунке 5, выполняется обход

всех элементов в автоматических стилях. Для каждого элемента, который содержит любой из табличных объектов, происходит извлечение всех характеристик как родительского, так и всех дочерних узлов. Затем определяется тип исследуемого объекта и добавляется в результирующую коллекцию объектов. Результат работы продемонстрирован на рисунке 6.

```
def get_automatic_table_styles(self, doc: ODTDocument):
    """Returns a list of all tables styles from automatic document styles with their attributes.

    Keyword arguments:
        doc - an instance of the ODTDocument class containing the data of the document under study.
    """
    Возвращает список всех стилей таблиц из автоматических стилей документа с их атрибутами.

    Аргументы:
        doc - экземпляр класса ODTDocument, содержащий данные исследуемого документа.
    """
    styles_dict = {}
    table_objs = []
    for ast in doc.document.automaticstyles.childNodes:
        name = ast.getAttribute('name')
        if name.count('able') > 0:
            style = {}
            styles_dict[name] = style
            for key in ast.attributes.keys():
                style[ast.qname[1] + "/" + key[1]] = ast.attributes[key]
            for node in ast.childNodes:
                for key in node.attributes.keys():
                    style[node.qname[1] + "/" + key[1]] = node.attributes[key]
    for cur_style in styles_dict:
        if cur_style.count('Column') > 0:
            table_objs.append(from_dict(data_class=TableColumn,
                                       data=convert_to_table_column(cur_style, styles_dict[cur_style])))
        elif cur_style.count('Row') > 0:
            table_objs.append(from_dict(data_class=TableRow,
                                       data=convert_to_table_row(cur_style, styles_dict[cur_style])))
        elif cur_style.count('Cell') > 0:
            table_objs.append(from_dict(data_class=TableCell,
                                       data=convert_to_table_cell(cur_style, styles_dict[cur_style])))
        else:
            table_objs.append(from_dict(data_class=Table,
                                       data=convert_to_table(cur_style, styles_dict[cur_style])))
    return table_objs
```

Рисунок 5 – Алгоритм обработки элементов таблиц

```
[TableColumn(_column_name='TableColumn2', _column_family='table-column', _column_properties_column_width=1.277,
(_column_name='TableColumn3', _column_family='table-column', _column_properties_column_width=1.5534, _column_pr
_column_family='table-column', _column_properties_column_width=1.5229, _column_properties_use_optimal_column_wi
_column_properties_column_width=1.3534, _column_properties_use_optimal_column_width=False), TableColumn(_column
_column_properties_column_width=1.1368, _column_properties_use_optimal_column_width=False), Table(_table_name='
_table_properties_width=6.8437, _table_properties_margin_left=0.0, _table_properties_align='right', _table_cell
_row_family='table-row', _row_properties_min_row_height=0.6729, _row_properties_use_optimal_row_height=False),
_cell_properties_border=0.0069, _cell_properties_writing_mode='lr-tb', _cell_properties_padding_top=0.0, _cell
_cell_properties_padding_right=0.0), TableCell(_cell_name='TableCell15', _cell_family='table-cell', _cell_prope
_cell_properties_padding_top=0.0, _cell_properties_padding_left=0.0, _cell_properties_padding_bottom=0.0, _cell
_cell_family='table-cell', _cell_properties_border=0.0069, _cell_properties_writing_mode='lr-tb', _cell_proper
_cell_properties_padding_bottom=0.0, _cell_properties_padding_right=0.0), TableCell(_cell_name='TableCell23', _
```

Рисунок 6 – Результат работы алгоритма создания объектов, связанных с таблицами

Для подготовки и конвертации извлеченных характеристик структурных элементов из документов в унифицированные классы объектов был разработан вспомогательный модуль, содержащий специальные конверторы, изображенные на рисунках 7–11. Данные методы принимают название объекта и список его характеристик. Затем происходит поиск среди списка необходимых атрибутов для класса данных, и, если такой атрибут найден, происходит добавление в результирующий словарь записи вида [название характеристики: ее значение], и в конце метод возвращает словарь с подготовленными данными.

```
def convert_to_table(table_name: str, table_data: dict):
    converted_data = {}
    data_keys = list(table_data.keys())
    converted_data.update({'_table_name': table_name})
    for element in data_keys:
        cur_element = element.split('/')
        match cur_element[1]:
            case 'family':
                converted_data.update({'_table_family': table_data[element]})
            case 'master-page-name':
                converted_data.update({'_table_master_page_name': table_data[element]})
            case 'width':
                converted_data.update({'_table_properties_width': float(table_data[element].split('i')[0])})
            case 'margin-left':
                converted_data.update({'_table_properties_margin_left': float(table_data[element].split('i')[0])})
            case 'align':
                converted_data.update({'_table_properties_align': table_data[element]})
    return converted_data
```

Рисунок 7 – Конвертер данных для таблиц

```
def convert_to_table_column(column_name: str, column_data: dict):
    converted_data = {}
    data_keys = list(column_data.keys())
    converted_data.update({'_column_name': column_name})
    for element in data_keys:
        cur_element = element.split('/')
        match cur_element[1]:
            case 'family':
                converted_data.update({'_column_family': column_data[element]})
            case 'column-width':
                converted_data.update({'_column_properties_column_width': float(column_data[element].split('i')[0])})
            case 'use-optimal-column-width':
                converted_data.update({'_column_properties_use_optimal_column_width': column_data[element] == 'true'})
    return converted_data
```

Рисунок 8 – Конвертер данных для столбцов таблиц

```
def convert_to_table_row(row_name: str, row_data: dict):
    converted_data = {}
    data_keys = list(row_data.keys())
    converted_data.update({'_row_name': row_name})
    for element in data_keys:
        cur_element = element.split('/')
        match cur_element[1]:
            case 'family':
                converted_data.update({'_row_family': row_data[element]})
            case 'min-row-height':
                converted_data.update({'_row_properties_min_row_height': float(row_data[element].split('i')[0])})
            case 'use-optimal-row-height':
                converted_data.update({'_row_properties_use_optimal_row_height': row_data[element] == 'true'})
    return converted_data
```

Рисунок 9 – Конвертер данных для строк таблиц

```
def convert_to_table_cell(cell_name: str, cell_data: dict):
    converted_data = {}
    data_keys = list(cell_data.keys())
    converted_data.update({'_cell_name': cell_name})
    for element in data_keys:
        cur_element = element.split('/')
        match cur_element[1]:
            case 'family':
                converted_data.update({'_cell_family': cell_data[element]})
            case 'border':
                converted_data.update({'_cell_properties_border': float(cell_data[element].split('i')[0])})
            case 'writing-mode':
                converted_data.update({'_cell_properties_writing_mode': cell_data[element]})
            case 'padding-top':
                converted_data.update({'_cell_properties_padding_top': float(cell_data[element].split('i')[0])})
            case 'padding-left':
                converted_data.update({'_cell_properties_padding_left': float(cell_data[element].split('i')[0])})
            case 'padding-bottom':
                converted_data.update({'_cell_properties_padding_right': float(cell_data[element].split('i')[0])})
            case 'padding-right':
                converted_data.update({'_cell_properties_padding_bottom': float(cell_data[element].split('i')[0])})
    return converted_data
```

Рисунок 10 – Конвертер данных для ячеек таблиц

```

def convert_to_list(list_name: str, list_data: dict):
    converted_data = {}
    data_keys = list(list_data.keys())
    if 'number' in data_keys[0]:
        converted_data.update({'_list_type': 'numbered'})
    else:
        converted_data.update({'_list_type': 'bulleted'})
    converted_data.update({'_list_name': list_name})
    for element in data_keys:
        cur_element = element.split('/')
        match cur_element[1]:
            case 'level':
                converted_data.update({'_list_level': list_data[element]})
            case 'start-value':
                converted_data.update({'_list_start_value': list_data[element]})
            case 'bullet-char':
                converted_data.update({'_list_style_char': list_data[element]})
            case 'style-name':
                converted_data.update({'_list_style_name': list_data[element]})
    return converted_data

```

Рисунок 11 – Конвертер данных для списков

Подготовленные данные передаются в метод `from_dict` предоставляемый модулем `dacite`. `Dacite` [10] – специальный сторонний модуль для упрощенного создания экземпляров классов данных из словарей с требуемыми для инициализации атрибутами. Данное решение является удобным, поскольку позволяет объединить логику классов данных и атрибутов для них, которые могут быть получены из базы данных или их других сервисов автоматизированного нормоконтроля из запросов. Важно отметить, что данная библиотека не содержит встроенной проверки данных. Именно для этой цели и подготовки характеристик структурных элементов для инициализации были реализованы продемонстрированные ранее конверторы.

Необходимо отметить, что унифицированные классы находятся на этапе разработки, поэтому они постоянно модернизируются для решения задач разрабатываемого сервиса. Но разработанный функционал является расширяемым и не потребует больших затрат ресурсов для внесения изменений. Также в модернизации нуждается инициализация атрибутов

классов данных значениями, поскольку на данный момент атрибуты, которые могут отсутствовать в иных форматах электронных документов, имеют значение по умолчанию None. Для этого планируется реализовать связь с вспомогательным модулем `consts`, хранящим значения характеристик структурных элементов документа в соответствии с нормативными документами.

ЗАКЛЮЧЕНИЕ

В ходе выполненной работы была разработана, реализована и описана архитектура подмодуля парсинга ODT документов как часть библиотеки парсинга документов PyOPWParse, разрабатываемой в рамках сервиса автоматизированного нормоконтроля.

Была проведена оптимизация и модернизация существующих алгоритмов для извлечения характеристик объектов электронных документов формата ODT. Также были модернизированы классы структурных элементов таблиц, списков и изображений, хранящих характеристики объектов, извлекаемых из документов.

Обновленная архитектура подмодуля и проведенная оптимизация алгоритмов позволили сократить время выполнения демонстрационных скриптов, структурировать весь реализованный функционал и адаптировать библиотеку к критериям open-source проекта.

На следующем этапе исследований планируется агрегация всех алгоритмов и иных разработок в виде веб-сервиса. Дополнительно необходимо добавить в модуль парсинга функциональные возможности по замене значений отсутствующих атрибутов у элементов документа через глобальные атрибуты, хранящие значения из нормативных документов.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Марцинкевич В.И., Ларионова Г.С., Терещенко В.В., Ситникова К.А., Горлушкина Н.Н. Анализ возможностей парсинга электронных текстовых документов для автоматизации нормоконтроля // Экономика. Право. Инновации – 2022. – No 3. – С. 39-49
2. ISO 26300-1:2015. Open Document Format for Office Applications (OpenDocument) v1.2 – Part1: Open Document Schema [Электронный ресурс]. – URL: <https://www.iso.org/standard/66363.html> (дата обращения: 23.03.2023).
3. Miguel G. Flask Web Development: Developing Web Applications with Python – A: O'Reilly Media, 2014 – 258 с.
4. Rubio D. Beginning Django: Web Application Development and Development with Python – A: Apress, 2017 – 620 с.
5. PEP 525 – Syntax for Variable Annotations [Электронный ресурс]. – 2016. – URL: <https://peps.python.org/pep-0526/> (дата обращения: 15.03.2023).
6. Docsify Documentation [Электронный ресурс]. – 2022. – URL: <https://docsify.js.org/#/?id=docsify> (дата обращения: 20.03.2023).
7. Pajankar A. Python Unit Test Automation: Automate, Organize and Execute Unit Tests in Python – A: Apress, 2021 – 232 с.
8. unittest – Unit testing framework Documentation [Электронный ресурс]. – 2023. – URL: <https://docs.python.org/3/library/unittest.html> (дата обращения: 23.03.2023).
9. Dataclasses Documentation [Электронный ресурс]. – 2023. – URL: <https://docs.python.org/3/library/dataclasses.html> (дата обращения: 28.03.2023).
10. Dacite Documentation [Электронный ресурс]. – 2023. – URL: <https://pypi.org/project/dacite/> (дата обращения: 28.03.2023).