МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение

высшего образования

**«КУБАНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»**

**(ФГБОУ ВО «КубГУ»)**

**Факультет математики и компьютерных наук**

**Кафедра вычислительной математики и информатики**

Допустить к защите

Заведующий кафедрой

д-р. тех. наук, доцент

Н. А. Наумова

(подпись)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_2025 г.

Руководитель ООП

д-р. тех. наук, профессор

Ю. М. Вишняков

(подпись)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_2025 г.

**ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА**

**(МАГИСТЕРСКАЯ ДИССЕРТАЦИЯ)**

**ИНСТРУМЕНТЫ ДЛЯ РАЗРАБОТКИ СПЕЦИАЛИЗИРОВАННЫХ МАТЕМАТИЧЕСКИХ ПАКЕТОВ**

Работу выполнил                          Д. В. Кочергин

(подпись)

Направление подготовки 02.04.01 Математика и компьютерные науки .

Направленность (профиль) Вычислительная математика

Научный руководитель

канд. физ-мат. доцент   В. А. Кирий

(подпись)

Нормоконтролер

преподаватель    А. А. Лахтина

(подпись)

Краснодар  
2025

**Содержание**

[Введение 3](#_Toc202232457)

[1. Анализ предметной области и существующих решений для автоматизации нормоконтроля 7](#_Toc202232458)

[1.1 Сущность и задачи нормоконтроля текстовых документов 7](#_Toc202232459)

[1.2 Обзор существующих методов и средств автоматизации нормоконтроля 9](#_Toc202232460)

[1.3 Анализ форматов электронных документов и средств их программной обработки 13](#_Toc202232461)

[1.3.1 Ключевые форматы 13](#_Toc202232462)

[1.3.2 Особенности формата DOCX (Office Open XML) 15](#_Toc202232463)

[1.3.4 Вывод 20](#_Toc202232464)

[1.4 Постановка задачи разработки системы нормоконтроля 20](#_Toc202232465)

[2. Проектирование и разработка системы 26](#_Toc202232466)

[2.1 Архитектура системы и выбор ключевых решений 26](#_Toc202232467)

[2.2 Разработка алгоритмов определения структурных элементов документа 28](#_Toc202232468)

[2.2.3 Каскадный метод определения типов элементов 29](#_Toc202232469)

[2.2.4 Реализация функций-детекторов (*is\_...)* 30](#_Toc202232470)

[2.3 Разработка алгоритмов проверки правил нормоконтроля с учетом стилей 31](#_Toc202232471)

[2.4 Реализация способа предоставления результатов проверки 32](#_Toc202232472)

[2.5 Разработка и развертывание веб-интерфейса 33](#_Toc202232473)

[3. Тестирование, анализ результатов и ограничений системы 38](#_Toc202232474)

[3.1 Методика тестирования 38](#_Toc202232475)

[3.2 Результаты тестирования и их анализ 41](#_Toc202232476)

[3.3 Обсуждение достигнутых результатов и нерешенных проблем 45](#_Toc202232477)

[3.4 Сравнение с альтернативными подходами 47](#_Toc202232478)

[3.5 Рекомендации по использованию и направления для дальнейшего развития 47](#_Toc202232479)

[Заключение 49](#_Toc202232480)

[Список использованных источников 51](#_Toc202232481)

[ПРИЛОЖЕНИЕ А 54](#_Toc202232482)

Введение

Сегодня – в период глобальной цифровизации – в профессиональной и академической среде ключевую роль всё ещё играет качество оформления текстовых документов. Процесс стандартизации представления научной или технической информации способствует не только эффективному восприятию или же обмену, но и в целом является важнейшим требованием для большинства видов публикаций или же отчётности. В таком ключе на первый план выходит один из важнейших этапов подготовки документов – нормоконтроль – процесс проверки соответствия оформления документов стандартам и правилам, заданным нормативными документами.

Как правило, нормоконтроль производится вручную. Очевидно, что при таком подходе возникает множество проблем. Во-первых, высокая трудоемкость: визуальный анализ и сверка реализованного оформления с требуемым требует крупных затрат времени и усилий как от авторов, так и от тех, кто работы проверяет. Вторым проблемным местом выступает субъективность и человеческий фактор. Они вносят погрешности, ведь специалисты могут различным образом интерпретировать нормативные требования. Усталость же и невнимательность приводят к пропуску ошибок. В-третьих, авторы документов, особенно студенты, часто допускают повторяющиеся, можно сказать типичные ошибки. И, наконец, современные стандарты оформления, такие как ГОСТы или методические указания многих учебных заведений, содержат большое количество детализированных и сложных для запоминания правил.

Вышеперечисленные факторы могут обусловить актуальность разработки автоматизированных систем нормоконтроля. Подобные системы могут повысить как объективность, скорость и эффективность процесса проверки, так и снизить нагрузку на специалистов нормоконтроля или научных руководителей. К тому же такая система могла бы предоставить авторам удобный инструмент для самостоятельной предварительность подготовки документов, минимизируя количество ошибок в оформлении при работе. Практический же интерес составляет разработка инструмент для автоматизированного анализа и проверки именно формата DOCX (Microsoft Word) ввиду его повсеместного использования на сегодняшний день.

Объектом исследования в работе выступает процесс нормоконтроля текстовых документов, которые создаются и сохраняются в формате DOCX, на предмет их соответствия заданным правилам и стандартам.

Предметом же исследования являются некоторые методы, алгоритмы и программные средства, которые позволяли бы автоматизировать анализ структуры документа. Автоматизировать так же его форматирование на уровне абзацев, символов и некоторых специфических элементов, таких как заголовки различных уровней, списки, таблицы, библиографические описания в файлах DOCX. Особое внимание в исследовании уделено разработке подхода для корректного учёта наследования стилей при анализе форматирования и обеспечение точной идентификации элементов документа для предоставления пользователю обратной связи о выявленных несоответствиях.

Цель работы – это разработка и реализация программно системы для автоматизации проведения нормоконтроля. Реализуемая система должна быть способна выявлять несоответствия по заложенным в неё правилам из методических указаний,корректно анализировать форматирование документа с учетом прямого форматирования и при влиянии пользовательских или встроенных стилей, а так же обеспечивать точную привязку обнаруженных ошибок к соответствующим элементам в документе для предоставления наглядной обратной связи.

Чтобы достичь поставленной цели, в работе решены следующие задачи поставленного исследования:

1. был проведен анализ предметной области нормоконтроля, а именно изучение целей и типовых этапов процесса и основных нормативных документов, которые регулируют оформление текстовых работ на примере учебно-методических указаний по структуре и оформлению магистерской диссертации, бакалаврской, дипломной и курсовой работ Кубанского Государственного Университета [1];
2. исследованы особенности формата электронных документов DOCX (на базе стандарта OOXML), а так же проведен анализ существующих библиотек и инструментов, позволяющих совершить программную обработку. Был сделан акцент на использование библиотеки python-docx, оценка её применимости для задач анализа форматирования и модификации документов для предоставления наглядной обратной связи пользователю;
3. разработаны и реализованы алгоритмы определения структурных элементов в документе на основе каскадного подхода, который сочетает анализ стилей, текстовых паттернов и наиболее вероятных атрибутов форматирования;
4. разработаны и реализованы алгоритмы проверки правил форматирования для идентифицированных элементов в документе;
5. проведено исследование и дальнейший выбор наиболее оптимального метода предоставления результатов проверки пользователю, поиск практических решений для точной привязки сообщений об ошибках к проверяемым элементам;
6. спроектирован и реализован прототип системы автоматизированного нормоконтроля с модулями анализа, проверки, отображения ошибок и минимальным пользовательским интерфейсом;
7. проведено тестирование реализованной системы на наборе подготовленных документов, некоторых работах автора, выявлены проблемы и ограничения подхода.

При учёте решенных проблем с корректной привязкой генерируемых комментариев к ошибкам, потенциальных смещений элементов DOCX-документов при работе с XML-структурой на низком уровне, обоснованием целесообразности использования высокоуровневого API библиотеки python-docx (1.2.0+) для минимизации конфликтов работа имеет научную значимость.

Возможность же использования прототипа студентами для предварительной самостоятельной проверки оформления работа, возможность использования предложенных алгоритмов и подходов к анализу DOCX-документов при разработке или усовершенствовании других программных продуктов и реализация системы в виде доступного веб-приложения без необходимости установки и настройки локально программного обеспечания отражает так же и прикладную значимость.

**1. Анализ предметной области и существующих решений для автоматизации нормоконтроля**

Чтобы поставить задачу разработки собственной системы необходимо было провести теоретический анализ изучаемой области (нормоконтроля) и в целом парсинга DOCX документов. Рассмотреть существующие подходы, проанализировать их для возможности программной обработки формата DOCX и принять дальнейшее решение для использования в создаваемом решении.

**1.1 Сущность и задачи нормоконтроля текстовых документов**

Нормоконтроль – это важный этап в жизенном цикле подготовки и выпуска разнообразных видов текстовых документов. В особенности в научной, технической сферах. Он применяется для унификации, повышения читаемости и восприятия документа, а так же для удобного последующего форматирования и дополнения. Для оценки решаемых задач был использован документ методических указаний КубГУ[1].

Нормоконтроль решает такие ключевые задачи как:

1. проверка структуры документа, в которой обязательно наличие и последовательность таких разделов как введение, основная часть, заключение, список литературы, приложения и т.п., требованиям методических указаний или стантартов;
2. контроль форматирования текста, в ходе которого проверяются параметры полей, ориентации страницы, оформление шрифтов (их размера, начертания, цвета), абзацных отступов, интервалов между строками, абзацами и параметры выравнивание текста;
3. контроль оформления заголовков, который предполагает соответствие уровней их нумерации, так же правильность форматирования шрифта, выравнивания, наличия или отсутствия точек в нумерации по требованиям методических указаний;
4. проверка правильности использования маркеров в списках (цифры, буквы), пунктуации в элементах списка;
5. проверка нумерации и наличия подписей к таблицам, их расположение на странице относительно таблицы;
6. контроль нумерации и наименования иллюстрацией, качество самих иллюстраций и их расположение, выравнивание и связанность с наименованиями;
7. контроль формул, проверяющий оформление и их расположение, корректность используемых символов и нумерации самих формул в документе;
8. проверка оформления ссылок на источники информации, списка литературы, соответствие форматов цитирования и библиографических описаний стандартам методических указаний и конкретно ГОСТ[2], [3].
9. опциональным, насколько позволяет понять анализ, но довольно важным является контроль за единообразием оформления, т.е. применение одинаковых правил форматирования к однотипным элементам в документе;
10. очевидно и выявление орфографических и пунктуационных ошибок (пусть это зачастую решается отдельным процессом корректуры) как часть базовых проверок.

За основу для нормоконтроля берутся нормативные документы. В Российской Федерации – это государственные стандарты (ГОСТы) и СИБИД (Система стандартов по информации библиотечному и издательскому делу), которые, впрочем, так же описаны в ГОСТ[4]. Существуют и отраслевые стандарты, а в академической среде упор делается на методические указания, которые разрабатываются ими же. Такие указания детализируют и адаптирую требования ГОСТов к специфике работ.

Собственно, указанные источники и выступали базой для выделения требований к правилам, которые в дальнейшем были использованы для проверок форматирования в документах.

**1.2 Обзор существующих методов и средств автоматизации нормоконтроля**

Несмотря на очевидную потребность, анализ показал, что в открытом доступе для интересующихся есть не так много автоматизированных полноценных (об этом позже) решений. Вообще существующие подходы можно разделить на несколько категорий.

Первой выступает полностью ручная проверка. При таком методе очевидно требуется визуальный анализ документа нормоконтролёром или научным руководителем. Его недостатки лежат на поверхности – это трудоёмкость и субъективность. Улучшением этого метода может выступать внедрением разнообразных чек-листов и шаблонов, которые через попытку составить список некоторых контрольных вопросов пытаются формализовать процесс проверки. Серьёзным улучшением же можно считать использование шаблона документа с предустановленными стилями, которые снижают вероятность пропуска правил, но, увы, не устраняют ручной труд, ведь некоторые требования нормативных документов затруднительно или вовсе невозможно реализовать на платформе текстовых процессоров таких как Word, Libre Office и т.д. К тому же в педагогической практике автора встречались трудности с обратной совместимостью стилей в двух указанных ранее программах.

Второй подход закрывает только часть задач. Проверка орфографии, например, грамматики и система стилей реализованы в большинстве текстовых процессоров, но их корректное использование и настройка требуют от пользователя некоторого опыта. И всё же соответствие самих стилей требованиям остаётся за рамками функциональности программ. Отсюда такой подход скорее является базой для первого. Эти инструменты не обеспечивают комплексной проверки специфических правил нормоконтроля (например, правильность оформления списка литературы по ГОСТ, нумерацию разделов, форматирование подписей к рисункам согласно методичке и т.п.

Третий подход – это использование специализированных программных решений и сервисов. Среди коммерческих продуктов отдельно стоит выделить NormaCS (Россия)[5]. Эта система, ориентированная на работу с нормативно-технической документацией, включает модули для проверки оформления документов на соответствие ГОСТ ЕСКД, СПДС и др. Она используется в проектных и конструкторских организациях. Анализ и ознакомление с документацией показали, что это мощное, но довольно узкоспециализированное решение, которое требует существенной начальной настройки (а главное опыта в таковой) для внедрения в него правил, рассматриваемых в этой работе.

А так же внимания заслуживает PerfectIt (Intelligent Editing, США/Великобритания) [6]. Это надстройка для MS Word, ориентированная на проверку стилей, сокращений, форматирования таблиц и списков. Согласно документации с официального сайта поддерживает создание пользовательских стилей проверки. Система намного универсальнее, но требует значительной настройки под российские ГОСТы и методические указания учебных заведений. Основные возможности отображены на Рисунке 1.

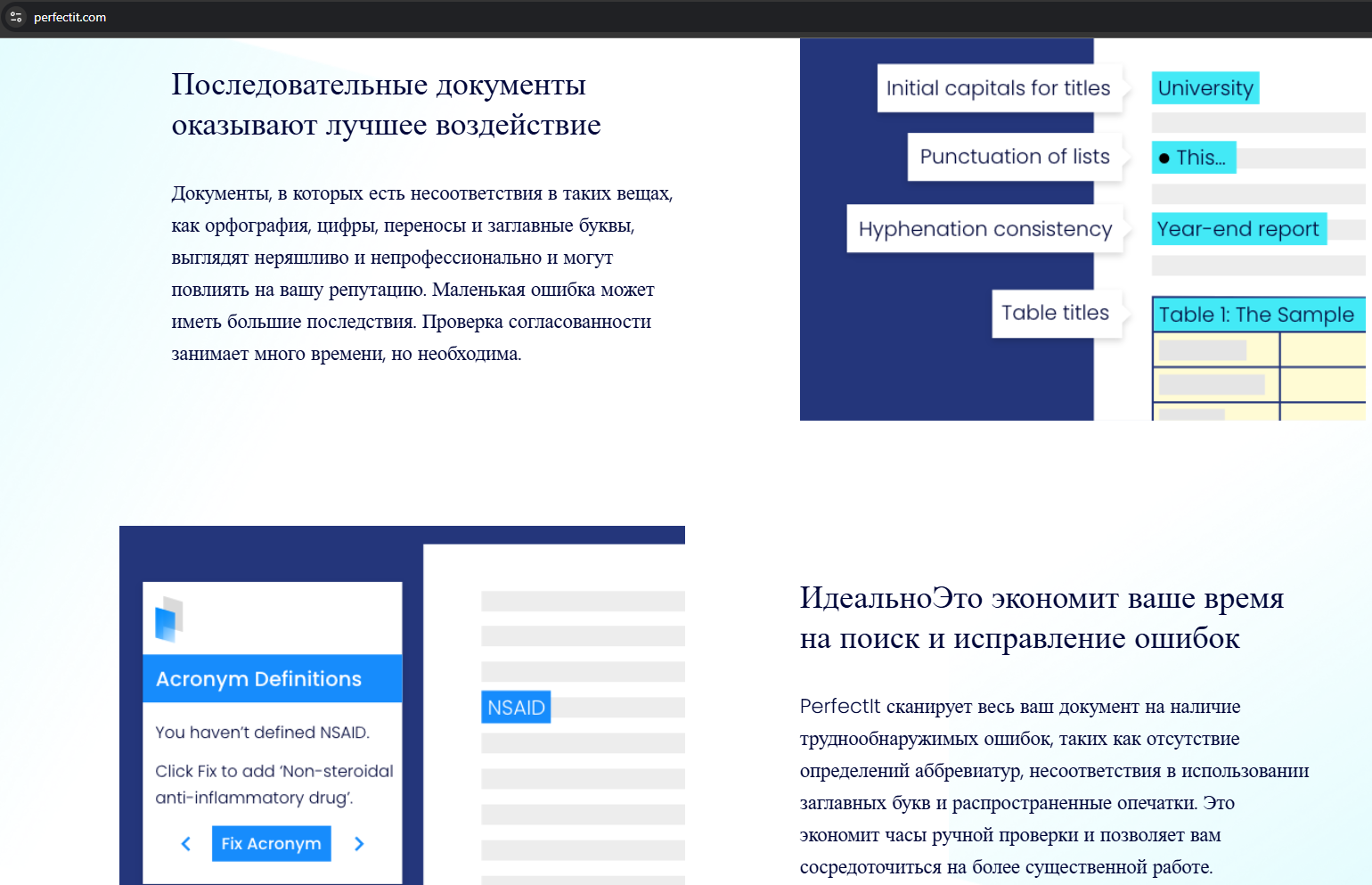


Рисунок 1 – Официальный сайт Perfectit с демонстрацией основной функциональности системы

Среди свободно распространяемых решений внимание надо уделить таким как Vale (США, GitHub)[7]. Инструмент является линтером (по сути анализатором для кода программ, прозы и т.п.) для проверки, работающий с текстовыми файлами (Markdown, reStructuredText, HTML, AsciiDoc). Можно настраивать правила с помощью YAML. Среди анализируемых решений это выступало наиболее интересным и применимым, но для DOCX потребовалась бы предварительная конвертация в поддерживаемый формат, что может привести к потере информации о форматировании

Другая система со свободным распространением – это LanguageTool (Германия, Open Source)[8]. Она выступает инструментом для проверки грамматики, стиля и орфографии во множестве языков. Есть плагины для текстовых редакторов. Но основной фокус – это лингвистическая корректность, а не детальное форматирование по ГОСТ.

В целом, на GitHub было обнаружено несколько проектов, которые предоставляли интерес и являлись решениями для проверки некоторых аспектов форматирования. Но проекты либо не обновлялись несколько лет и не покрывает всего спектра правил, необходимых для комплексного нормоконтроля либо были довольно сложны в установке и применении, а также настройки.

Отдельным подходом можно выделить некоторые сервисы проверки орфографии и грамматики и базовых правил стилистики, которые являются модулями крупных сервисов антиплагиата (например, "Антиплагиат.ВУЗ"). Так, например, на упоминаемом сервисе [9] заявлена возможность анализа структуры документа, что видно на Рисунке 2, а по моим предположениям так же позволяет и проверять соответствие правилам ГОСТ. Проверить гипотезу проблематично в силу закрытости платформы и её коммерческой направленности.

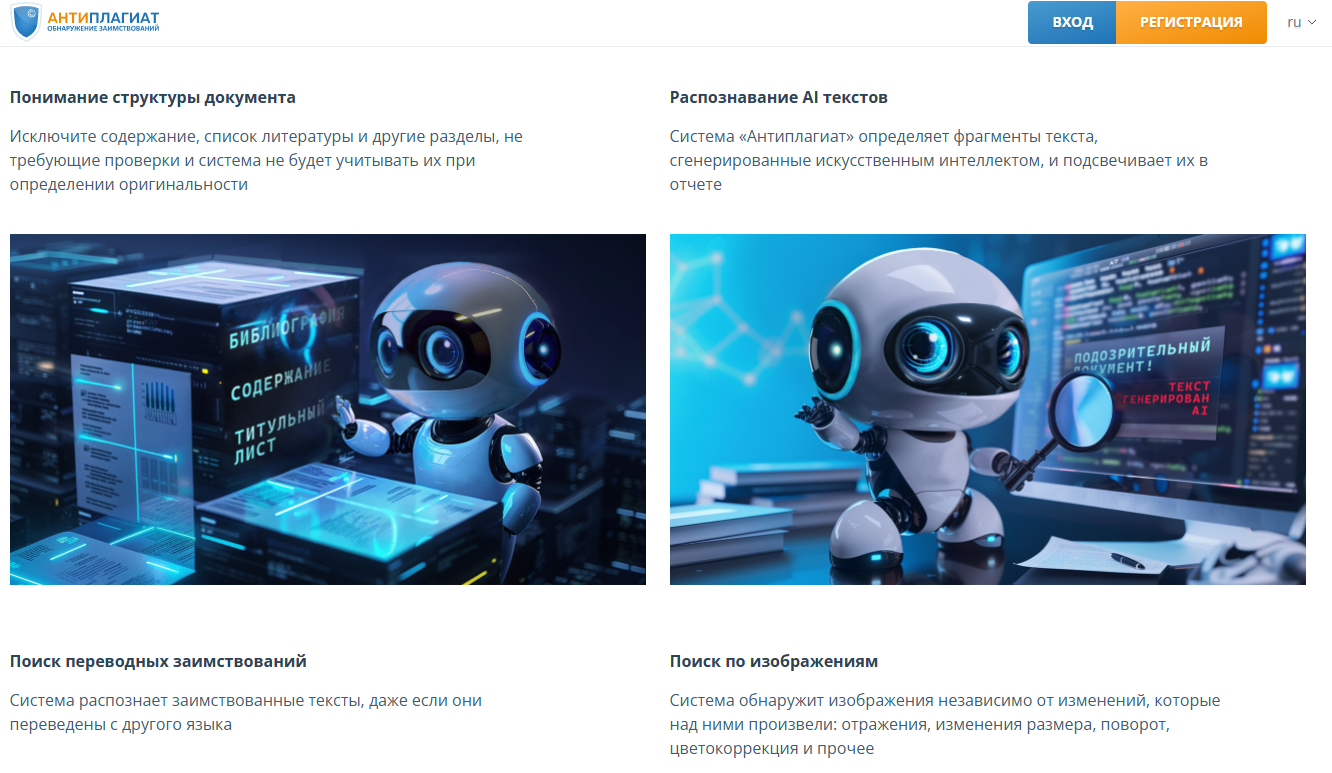


Рисунок 2 – Сервис Antiplagiat.ru реализует возможность анализа структуры документа

Перечисленные универсальные инструменты не предназначены для специфики российских ГОСТов и методических указаний конкретных вузов, а специализированные решения могут быть дорогостоящими или узкопрофильными. В свою очередь open source проекты часто решают лишь часть задач или требуют доработки. В них отсутсвует гибкость в настройке правил под конкретные требования пользователя или организации. И как показал анализ такие инструменты не предоставляют удобной обратной связи непосредственно в документе, что мне видится наиболее удобным и наглядным для пользователя.

Поэтому задача создания собственного прототипа системы осталась актуальной после проведенного анализа. Сам же анализ имеющихся решений и дальнейший анализ форматов документов и их средств обработки (об этом упоминается далее) позволил точнее определить границы применения и наиболее важные требования к системе. То есть работа с DOCX форматом, возможность задания правил из конкретных методических указаний ВУЗа на основе ГОСТ, наглядная обратная связь для пользователя, простота доступа к инструменту.

**1.3 Анализ форматов электронных документов и средств их программной обработки**

Для эффективного создания системы необходимо было провести выбор целевого формата документа и подходящего инструментария для его обработки. С этой целью был проведен сравнительный анализ библиотек и средств работы с форматом DOCX на языке Python.

**1.3.1 Ключевые форматы**

Современный документооборот включает множество форматов, каждый из которых обладает своими особенностями:

1. Первый формат - .doc. Это бинарный формат Microsoft Word (версии до 2007 года). Проблема в том, что его закрытая структура существенно затрудняет программный анализ и модификацию без специализированных инструментов, что делает его менее предпочтительным для задач автоматизации;
2. Вторым является современный стандартный формат Microsoft Word docx, основанный на Office Open XML (OOXML) [10]. Он представляет собой ZIP-архив, содержащий набор XML-файлов, описывающих структуру, содержимое и форматирование документа, а также другие ресурсы (например, изображения).
3. Существует и ODT (Open Document Text) стандарт формата документов, также базирующийся на XML и использующий ZIP-архивацию [11]. Применяется в офисных пакетах, таких как LibreOffice и OpenOffice. В целом, технически подходит для анализа, но уступает DOCX в распространенности в академической среде;
4. Следующий формат – PDF – Portable Document Format, который предназначен в основном для финального представления и обмена документами с сохранением их внешнего вида на различных платформах. Но анализ и модификация структуры и форматирования PDF-документов на уровне, который был необходим для нормоконтроля виделся мне крайне сложной задачей;
5. TeX/LaTeX – это мощная система верстки, широко используемая в научной среде для подготовки документов со сложным математическим и техническим контентом. Документы создаются путем написания кода на языке разметки, который затем компилируется. При этом автоматизация нормоконтроля таких документов потребовала бы анализа TeX-кода, что затруднительно, т.к. хотелось бы следовать концепции работы с WYSIWYG (What You See Is What You Get — Что видишь то и получишь) форматами документов [12].

Учитывая эти нюансы и доминирующее положение Microsoft Word при подготовке учебных, научных работ и открытую XML-основу формата, выбор DOCX в качестве целевого формата виделся наиболее удачным. Его внутренняя структура позволила бы наиболее удобно автоматизировать извлечение информации об оформлении и последующей модификации.

**1.3.2 Особенности формата DOCX (Office Open XML)**

Первым шагом к решению поставленных задач было понимание структуры DOCX [13]. От этого зависели дальнейшие алгоритмы анализа. Необходимо было принять во внимание основные его компоненты и концепции.

Во-первых, архитектура DOCX-файла по своей сути является ZIP-архивом. Распаковав его, можно получить доступ к иерархии папок и XML-файлов.

Во-вторых, выделяются следующие XML-компоненты:

1. word/document.xml содержит основное тело документа – это текст, абзацы (<w:p>), таблицы (<w:tbl>), ссылки на встроенные объекты и другие структурные элементы;
2. word/styles.xml определяет все стили, используемые в документе, включая встроенные ("Обычный", "Заголовок 1" и т.д.) и пользовательские;
3. word/numbering.xml содержит определения для нумерованных и маркированных списков;
4. word/footnotes.xml, word/endnotes.xml в них хранятся данные о сносках;
5. word/comments.xml как файл хранит тексты комментариев и информация об их авторах;
6. word/\_rels/document.xml.rels предстваляет собой файл связей, описывающий отношения между основным документом и другими его частями (изображениями, сносками, комментариями).

С точки зрения представления форматирования сущесвуют такие концепции:

1. Прямое форматирование при котором существуют атрибуты, примененные непосредственно к выделенному тексту или абзацу (например, изменение шрифта для одного слова). В XML это отражается в элементах <w:rPr> (свойства прогона текста) и <w:pPr> (свойства абзаца).
2. В форматировании через стили есть атрибуты, унаследованные от примененного стиля, определение которого хранится в styles.xml. Документ может содержать сложную иерархию стилей, где один стиль наследует свойства от другого.

На этом моменте стало понятно, что взаимодействие прямого форматирования и многоуровневого наследования стилей может являться основной трудностью при попытке программно определить видимое пользователю форматирование элемента. Т.к. свойства, видимые пользователю, на уровне атрибутов могут быть скрытыми за многими слоями наследования, например, при использовании стилей.

**1.3.3 Сравнительный анализ библиотек и инструментов для программной работы с DOCX файлами на языке Python**

Обоснование выбора языка Python для решения задач исследования приводить видится излишним. Достаточно упомянуть, что в процессе анализа и поиска существующих решений, подавляющее большинство использовало этот язык. Он наиболее удобен для работы со строковыми типами данных и лучше других известен мне как автору работы. К тому же в экосистеме Python существует несколько подходящих и доступных библиотек для реализации системы. Например:

1. Python-docx [14]. Эта библиотека является наиболее популярной и функциональной библиотекой для работы с DOCX. В неё есть возможность создавать, читать и модифицировать документы, включая добавление и изменение абзацев, таблиц и т.д. Библиотека предоставляет API для доступа к атрибутам форматирования (шрифт, размер, цвет, выравнивание, отступы и т.д.), можно работать со стилями (применение, создание, чтение некоторых свойств стиля), а так же, что выяснится на этапах реализации системы, библиотека, начиная с версии 1.2.0 и выше, позволяет добавлять комментарии в документ с использованием методом библиотеки. В дальнейшем это станет основной сложностью при попытке реализации удобного интерфейса обратной связи по найденным ошибкам для пользователя. Существенным преимуществом являлась кроссплатформенность, наличие подробной документации и широкого перечня обучающих материалов.

Среди недостатков, выявленных во время анализа документации и примеров решений было выявлено то, что, например, библиотека предоставляет доступ к атрибутам форматирования, которые заданы напрямую для элемента (абзаца) или к атрибутам его непосредственного стиля. А вот автоматическое вычисление «эффективного» (так буду называть форматирование, которое напрямую видит пользователь в документе) является результатом применения всей цепочки наследования стилей. К примеру, *Стиль по умолчанию* затем *Стиль абзаца* затем *Стиль символов* затем *Прямое форматирование.* Справится с этой проблемой в рамках данной работы в дальнейшем не получилось (подробно об этом будет рассказано в главах с реализацией системы). То есть API возвращает *None* для атрибута форматирования, даже если визуально к документе оно присутствует. Таким образом есть существенное ограничение при работе со множеством вложенных пользовательских стилей. Увы. Такое ограничение потребовало разработки дополнительных скриптов для глубокого анализа вложенности стилей, что оказалось практически нереализуемой в моих компетенциях задачей.

Ещё одним недостатком, который тем не менее получилось разрешить, было добавление комментариев (в версиях до появления мтеода document.add\_comment()) В таких версиях отсутствовал прямой API и приходилось бы работать напрямую с низкоуровневой структурой XML. И, наконец, доступ к некоторым метаданным документам ограничен, требует использования нижележащего lxml APU, на котором основана python-docx [15];

1. Работа с XML напрямую (например, с использованием библиотеки lxml.etree) являлась другим подходом к решению задачи.

Преимуществами являлись полный и неограниченный контроль над всей XML-структурой документа. Теоретически это позволяла реализовать любой сложный анализ и модификацию файла. Но сразу же обнаружились и недостатки. В их числе была высокая сложность понимания стандарта OOXML с его многочисленными схемами и взаимосвязями элементов. Написание кода для таких манипуляций значительно более трудоёмкий процесс, чем использование высокоуровневого API.

Так как анализ не заканчивался и в процессе реализации системы, то ещё одной проблемой, появившейся на этапе разработки, оказалась проблема обратной совместимости кода программы. Разные версии MS Word или документы, созданные в других программах, имели незначительные вариации внутренней структуры XML.

Наконец, наиболее существенной трудностью являлась попытка точной привязки модификаций. Так, например, если в качестве обратной связи для предоставления информации ошибок выбрать подход с добавлением комментариев в копию анализируемого документа, то возникает проблема стабильного позиционирования таких комментариев. Дело в том, что анализ документа происходит на основе первоначального присвоения индексов элементам (параграфам). На уровне python-docx даже пустые абзацы считались полноценными параграфами, которым присваивался индекс. В случае с более низкоуровневой структурой XML учет пустых абзацев или, например, элементов внутри таблиц приводил к некорректному расчёту индексов параграфов. Из-за этого после, например, таблицы (в ней могло быть множество параграфов с точки зрения XML-структуры, а в python-docx такой элемент распознавался как самостоятельный – таблица) добавленные комментарии могли попросту «съезжать». Это наглядно видно на Рисунке 3, на котором отображена промежуточная реализация системы при работе с XML напрямую на примере анализа документа моей бакалаврской дипломной работы;

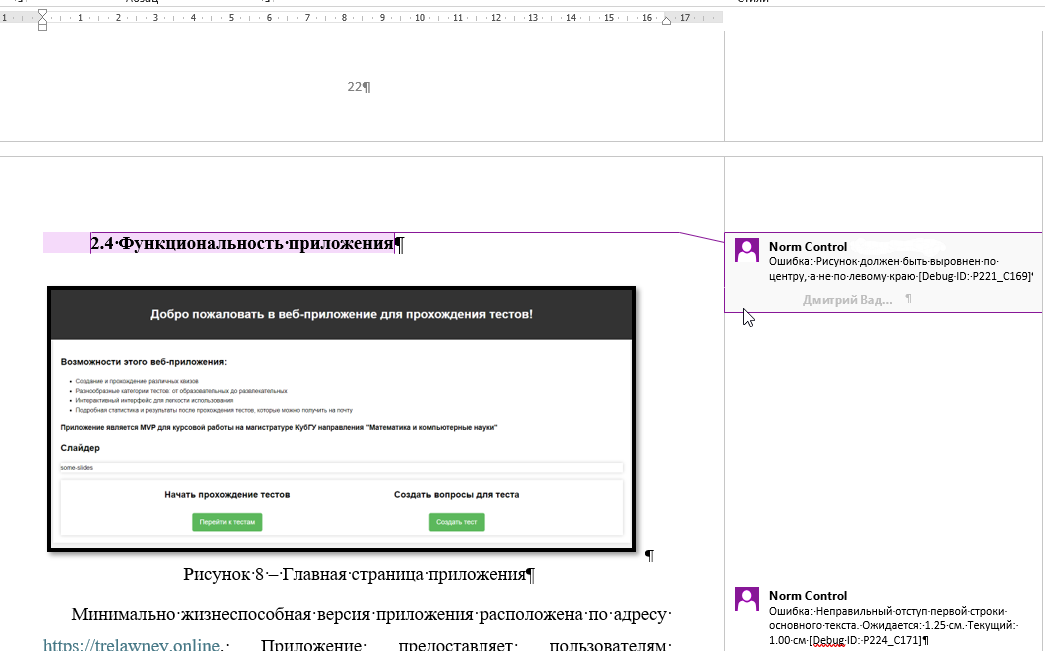


Рисунок 3 – Смещение добавляемых комментариев относительно целевых участков текста

1. Наконец, возможно и использование COM-интерфейса через библиотеку *py-win32* [16].

Такой подход позволяет Python-скрипту напрямую обращаться к приложению MS Word, используя его объектную модель. То есть можно получить доступ к функциональности Word, определять «эффективное» форматирование и использовать стандартные механизмы добавления комментариев [17]. Но сразу видны и недостатки. Такой механизм работает исключительно на Windows OS, он требует наличия установленного MS Word для выполнения кода. Это дольше с точки зрения времени выполнения, чем прямая работа с файлами. Получается, что интеграция COM-компонентов в серверную среду веб-приложения слишком трудна. Это при условии, что мы не берём в учет лицензионные ограничения. Да и кроссплатформенность не гарантируется;

1. Существуют и более простые библиотеки, как, например, docx2txt, которые позволяют быстро извлечь текстовое содержимое, но очевидно они не подходят для реализации анализа форматирования и структуры документа. То есть не подходят для целей нормоконтроля.

**1.3.4 Вывод**

Проведенный анализ позволил мне сделать осознанный выбор в пользу DOCX формата и библиотеки python-docx. На первых этапах разработки велась попутная разработка скриптов для работы с внутренней XML-структурой, однако в дальнейшем от неё было принято решение отказаться в виду проблем с добавлением комментариев. Ограничения, возникающие при работе с python-docx, были частично компенсированы обновлением версии и изучением актуальной документации. Но некоторые ограничения преодолеть так и не удалось. О них будет рассказано в следующих главах.

**1.4 Постановка задачи разработки системы нормоконтроля**

На основе анализа предметной области, существующих решений и инструментария, была сформулирована и задача диссертационной работы.

Несмотря на выявленные ограничения (по части анализа наследования стилей и ранних версий API), библиотека *python-docx* выбрана благодаря ее кроссплатформенности, отсутствию внешних зависимостей от проприетарного ПО, высокому уровню абстракции для работы с основными элементами документа и активному развитию

Были выделены две группы требований. Первая из них – это функциональные требования:

1. Загрузка и анализ документов формата DOCX;
2. Идентификация ключевых структурных элементов (основные заголовки, заголовки разделов/подразделов, основной текст, списки, таблицы, рисунки, список литературы, приложения);
3. Проверка форматирования этих элементов на соответствие заданному набору правил из методических указаний КубГУ[1];
4. Способность частично учитывать наследование стилей при анализе форматирования.
5. Предоставление пользователю отчета о выявленных несоответствиях с точной привязкой к элементам документа (например, путем вставки комментариев Word).

Вторая группа требований – нефункциональные требования, которые бы описывали как должна выглядеть и работать система:

1. Удобный пользовательский интерфейс в виде лаконичного и информативного веб-интерфейса;
2. Разумное время обработки документа;
3. Сам интерфейс должен предоставлять, во-первых, возможность загрузки документа, затем запуск процесса проверки и , наконец, скачивание документа с результатами проверки (с комментариями) или отображение отчета.

Первичный анализ документа методических указаний [1] позволил выявить ключевые требования(правила), о реализации которых имел бы смысл говорить.

В соответствии с общими требованиями, работа должна быть оформлена на листах формата А4 (210 x 297 мм) в книжной ориентации. Необходимо соблюдать следующие размеры полей: левое поле – 30 мм, правое – 15 мм, а верхнее и нижнее – по 20 мм.

Основной текст работы набирается шрифтом Times New Roman размером 14 пунктов. Допускается также использование 12-го кегля, поскольку стандарт требует, чтобы высота букв была не менее 1,8 мм. Цвет шрифта должен быть черным. Для всего основного текста устанавливается полуторный межстрочный интервал. Абзацный отступ, или «красная строка», должен быть одинаковым по всему документу и составлять 1,25 см. Выравнивание основного текста следует делать по ширине.

Использование полужирного начертания допускается только для заголовков структурных элементов, таких как разделы и подразделы. Курсив может применяться для выделения специальных терминов или обозначения объектов.

Особые требования предъявляются к заголовкам. Основные заголовки, такие как «РЕФЕРАТ», «СОДЕРЖАНИЕ», «ВВЕДЕНИЕ», «ЗАКЛЮЧЕНИЕ», «СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ» и «ПРИЛОЖЕНИЕ», пишутся прописными буквами и выделяются полужирным шрифтом и располагаются по центру строки. Точка в конце таких заголовков не ставится, и они не подчеркиваются.

Заголовки разделов и подразделов (например, «1. Название раздела») оформляются с абзацного отступа, нумеруются арабскими цифрами и пишутся с прописной буквы полужирным шрифтом. Точка в конце этих заголовков также не ставится, а переносы слов в них не допускаются. Каждый крупный раздел основной части должен начинаться с новой страницы. Между любым заголовком и последующим текстом необходимо оставлять одну пустую строку.

Нумерация страниц в работе должна быть сквозной и выполняться арабскими цифрами. Номер страницы размещается в центре нижней части листа без точки после цифры. Титульный лист включается в общую нумерацию, но сам номер на нем не проставляется. Нумерация разделов, подразделов и пунктов также использует арабские цифры и имеет иерархическую структуру (например, 1, 1.1, 1.1.1). Номера печатаются с абзацного отступа, и точка в конце номера не ставится.

Иллюстрации (рисунки) нумеруются арабскими цифрами сквозным образом по всей работе или в пределах каждого раздела. Подпись к иллюстрации имеет формат «Рисунок [номер] – Наименование рисунка», располагается под ней по центру, пишется с прописной буквы и не имеет точки в конце. Переносы слов в наименовании рисунка не допускаются.

Для таблиц также предусмотрена сквозная или пораздельная нумерация арабскими цифрами. Заголовок в формате «Таблица [номер] – Наименование таблицы» располагается над таблицей, выравнивается по левому краю без абзацного отступа и пишется с прописной буквы без точки в конце. Если таблица переносится на следующую страницу, над ее продолжением следует указать «Продолжение таблицы [номер]».

Формулы и уравнения размещаются на отдельных строках с отступом в одну пустую строку сверху и снизу от основного текста. Их нумерация (сквозная или по разделам) выполняется арабскими цифрами в круглых скобках, которые выравниваются по правому краю строки. Ссылки на формулы в тексте также даются в круглых скобках, например: «в формуле (2)».

Библиографические ссылки должны соответствовать ГОСТ Р 7.0.5–2008 [2]. Чаще всего используются затекстовые ссылки в квадратных скобках, где указывается номер источника из списка литературы и, при необходимости, номер страницы, например: [15] или [15, с. 5].

Список использованных источников озаглавливается «СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ», оформленным по общим правилам для основных заголовков. Источники в списке нумеруются арабскими цифрами с точкой и располагаются в алфавитном порядке или по мере упоминания. Оформление каждого источника должно строго соответствовать ГОСТ Р 7.0.100-2018 [3].

Каждое приложение необходимо начинать с новой страницы. Вверху по центру полужирным шрифтом пишется слово «ПРИЛОЖЕНИЕ» и его буквенное обозначение (например, «ПРИЛОЖЕНИЕ А»). Ниже, отдельной строкой по центру, располагается заголовок приложения, который пишется с прописной буквы, выделяется полужирным шрифтом и не имеет точки в конце.

Для каждого такого правила была проведена оценка их реализуемости при поддержке выбранной мной библиотеки. Оценка реализуемости основывалась на процессе разработки и рассмотрении документации. Множество трудностей появились лишь на этапе программирования системы.

Исходя из этого в первую группу можно выделить правила с *высоким уровнем реализуемости*. То есть правила, напрямую проверяемые через API python-docx. А это проверка конкретных значений шрифта, размера, выравнивания для элементов с прямым форматированием, проверка текстовых паттернов.

Во вторую группу выделены правила со *средним уровнем реализуемости*. Они требовали более сложной логики или обхода ограничений библиотеки. Такие правила были связаны с анализом отступов с учетом стилей, проверкой нумерации, базовая проверка структуры списка литературы.

*Низкий уровень реализуемости* или же не реализуемые в рамках работы правила, включали в себя те, что требуют семантического анализа и глубокого понимания контекста или же визуальной оценки. Такие как проверка "висячих строк" (не было возможности понять механизм, по которому без разрывов страниц *python-docx* понимает конец страницы), качества изображений, полного соответствия библиографии ГОСТ по содержанию и проверка сложных многоуровневых списков с нестандартным форматированием).

Весь процесс анализа файла условно должен был состоять из этапов идентификации элементов и затем проверки их форматирования. Идентификация проводилась бы каскадным образом. Анализ полей, количества рисунков, таблиц в работе проводился в первую очередь, т.к. такие операции не требуют специфических условий (они или есть и верные, или их нет). Вторым образом определялись наиболее вероятные и простые в определении элементы – заголовки структурных абзацев («ВВЕДЕНИЕ», «ЗАКЛЮЧЕНИЕ» и т.д.), т.к. они буквально содержат в абзаце единственные слова-маркеры. Затем процесс переходил к идентификации названий разделов и подразделов. Такие элементы обычно начинаются с номера или выделены жирным шрифтов (весь абзац), затем идентифицировались оставшиеся абзацы, начинающиеся с номеров (это элементы списков), наконец определялись изображения и таблицы, а после них или перед (для таблиц) определялись абзацы, которые идентифицировались как подписи к ним. После «СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ» (или другого допустимого названия) идентифицировался список литературы. Наконец, оставшийся текст считался основным текстом работы. Такой подход позволил бы справиться с проблемой идентификации элементов без семантического или контекстного анализа. К сожалению, он имеет ряд недостатков, о которых будет рассказано в главе с разработкой системы.

**2. Проектирование и разработка системы**

После того, как был проведен анализ предметной области и существующих средств решения задач нормоконтроля, который позволил выявить как высокую актуальность задачи, так некоторые технические сложности, связанные с анализом DOCX, был начат этап непосредственной разработки. Сам этап для меня состоял из процесса проектирования и программное реализации системы с использованием выбранных архитектурных решений. По ходу разработки появлялись новые ограничения, которые правились и устранялись, а затем процесс продолжался вплоть до этапа появления работоспособного MVP, который успешно отрабатывал на тестовом файле. Здесь и далее название методов будут ссылаться на листинги кода в Приложение А.

**2.1 Архитектура системы и выбор ключевых решений**

Когда проектировалась система, то основной упор был сделан на модульность и четкое разделение ответственности между компонентами. Подобное позволило упростить разработку и сопровождение всего кода. В результате процесса была сформирована трехуровневая архитектура, включающая ядро анализа, набор утилит для работы с форматированием и пользовательский интерфейс.

Саму архитектуру системы можно подробно рассмотреть вот так:

1. *Модуль анализа и проверки (formatting\_checker.py)* – это основной компонент системы. Его задача – загрузить DOCX-документ, последовательно итерировать по его элементам (абзацам, таблицам, сноскам), определять тип каждого элемента и применять к нему соответствующий набор правил проверки. В этом модуле инкапсулированы как функции-детекторы (*is\_..*.), так и функции-валидаторы (*check\_...\_format*). Результатом работы модуля является структурированный список выявленных несоответствий, содержащий информацию о месте ошибки и ее характере;
2. *Модуль утилит форматирования (formatting\_utils.py)*был выделен в отдельный компонент для решения одной из ключевых проблем, выявленных в первой главе, – сложности анализа форматирования при использовании стилей. Функции этого модуля (*get\_effective\_alignment, get\_first \_line\_indent\_cm* и др.) предоставляют вышестоящим модулям "эффективное" (напомню, что этим термином я подразумеваю то форматирование, которое пользователь наблюдает в Word непосредственно. Оно может быть скрыто за слоями насследования стилей или в специфических пользовательских стилях)значение атрибута форматирования, пытаясь учесть как прямое форматирование, так и наследование от стилей абзаца и его базовых стилей. Так я попытался отделить логику определения прямого форматирования от форматирования со стилями;
3. *Веб-интерфейс (app.py и связанные шаблоны .html файлов, .css правил стилей и .js скриптов).*Для взаимодействия с пользователем была выбрана реализация в виде веб-приложения на базе микрофреймворка Flask. Вообще выбор был обусловлен его легковесностью, гибкостью и простотой развертывания на предпочитаемом мной хостинге. Веб-интерфейс отвечает за прием файла от пользователя, передачу его на обработку ядру системы, получение результатов проверки и предоставление пользователю либо сообщения об успешной проверке, либо ссылки на скачивание исправленного документа с комментариями;
4. *Модуль обратной связи (comment\_utils.py)* является компонентом, который отвечает за финальный этап – предоставление результатов пользователю. На основе списка ошибок, сформированного модулем *formatting\_checker.py*, он программно модифицирует копию исходного документа, добавляя в него комментарии MS Word.

Общая файловая структура с учетом указанных модулей представлена на Рисунке 4.

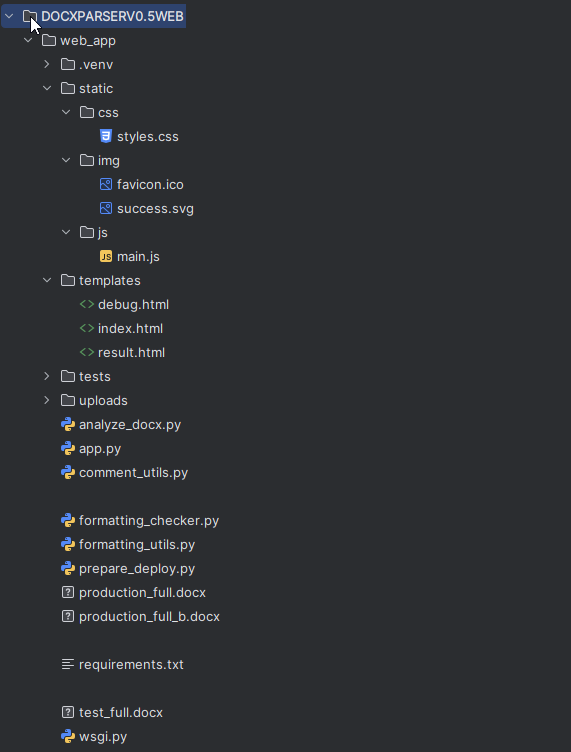


Рисунок 4 – Файловая структура проекта с ключевыми модулями

**2.2 Разработка алгоритмов определения структурных элементов документа**

Самая сложная задача при автоматизации нормоконтроля – это правильная идентификация типа каждого абзаца (то есть соотнесение его с типом того или иного элемента). Простой текст, заголовок, элемент списка, подпись к рисунку – все они представлены одним и тем же объектом *Paragraph* в *python-docx*. Для решения этой проблемы применялся *каскадный метод определения типов элементов*, который основан на иерархии правил и мтеоде исключений. В главе с анализом упоминалась его концепция. Пришлось немного её доработать в процессе реализации. Полагается, что и текущая реализация не идеальна.

**2.2.3 Каскадный метод определения типов элементов**

Его суть заключается в последовательной проверке абзаца на принадлежность к различным типам, двигаясь от наиболее специфичных и легко идентифицируемых к более общим. Так я попытался избежать ложные срабатывания при идентификации. Порядок проверки в основной функции-диспетчере (*check\_document\_formatting\_final*) выстроен следующим образом:

1. Элемент таблицы проверяется в первую очередь, так как его содержимое имеет совершенно иной контекст. В случае же *lxml* они и вовсе обрабатывались как отдельный абзац;
2. Заголовок приложения ("ПРИЛОЖЕНИЕ А") определялось по строгому текстовому паттерну, имеет высокий приоритет;
3. Структурный заголовок ("ВВЕДЕНИЕ", "ЗАКЛЮЧЕНИЕ" и т.д.) определяется по списку ключевых слов;
4. А Подпись к рисунку / Название таблицы определяются по характерным начальным словам ("Рисунок", "Таблица") и паттернам. Так, например, они очевидно следуют в следующем / предыдущем (для таблицы) абзаце;
5. Заголовок раздела («1. Заголовок») идентифицировался по стилю «Заголовок 1» или по комбинации текстового паттерна («N. Текст») и полужирного начертания;
6. Заголовок подраздела («1.1 Заголовок») аналогично, как и заголовок раздела, но с другим стилем и паттерном;
7. Элемент списка литературы попробовал определить по контексту (нахождение после элемента "СПИСОК ЛИТЕРАТУРЫ") и формату номера;
8. Элемент списка определяется по допустимым маркерам (-, а), 1) ).
9. Наконец основной текст определялся так: если абзац не прошел ни одну из вышеперечисленных проверок, он классифицируется как основной текст.

**2.2.4 Реализация функций-детекторов (*is\_...)***

Для того, чтобы реализовать этот метод был создан набор функций-детекторов, каждая из которых отвечает за определение одного типа элемента. Тут можно выделить комбинированный подход, в котором большинство функций (*is\_section\_heading, is\_subsection\_heading*) используют комбинацию нескольких признаков. Сперва *анализ стиля*, в котором проверяется имя стиля абзаца (*para.style.name*). Если оно соответствует ожидаемому (например, "Заголовок 1"), элемент классифицируется немедленно. Затем происходит *анализ текстового паттерна*. Если стиль не помог, применяется регулярное выражение для анализа текста абзаца (например, для поиска нумерации с помощью такого паттерна: ^\d+\.\s+.+). Наконец происходит *анализ форматирования.* В качестве дополнительного критерия используется проверка атрибутов форматирования, в первую очередь – полужирного начертания (*check\_all\_runs\_are\_bold*). Это позволяет, например, отличить заголовок раздела ("1. Мой заголовок") от элемента списка с таким же форматом номера. В некоторых детекторах (как например для подписей к рисункам и таблицах и для заголовков списка литературы) была попытка реализовать устойчивость к вариациям (например, "СПИСОК ЛИТЕРАТУРЫ", "СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ").

На тестовом документе такой подход показал свою высокую работоспособность даже при использовании с документами, где добавлялись пользовательские стили. Однако наблюдались проблемы при работе с сторонними документами, о которых будет сказано в главе с тестированием решения.

**2.3 Разработка алгоритмов проверки правил нормоконтроля с учетом стилей**

После определения типа элемента к нему применяются соответствующие правила проверки, реализованные в функциях *check\_...\_format*. Главной особенностью этих функций является использование утилит из модуля *formatting\_utils* для анализа форматирования.Как упоминалось ранее, значения атрибутов форматирования, полученные напрямую из *para.paragraph\_format* или *run.font*, могут быть *None*, если форматирование унаследовано от стиля. Мне для решения этой проблемы понадобилось, для каждого проверяемого атрибута (отступ, выравнивание, шрифт, размер, цвет, жирность) вызывать соответствующую функция из *formatting\_utils* (например, *get\_effective\_alignment(para)).* Эта функция инкапсулирует логику "каскадного" поиска значения: сначала в прямом форматировании, затем в стиле элемента, затем в его базовом стиле.

Примеры реализации проверки конкретных правил:

1. Проверка заголовка раздела происходит, например, с помощью функции *check\_section\_heading\_format*, которая проверяет, что *get\_effective\_alignment(para)* возвращает *WD\_ALIGN\_PARAGRAPH.LEFT*, а *get\_first\_line\_indent\_cm(para)* возвращает значение, близкое к 1.25 см. И далее уже, с помощью регулярных выражений проверяется, что текст заголовка начинается с номера, за которым следует точка;
2. Проверка заголовка подраздела с помощью функции *check\_subsection\_heading\_format* выполняет аналогичные проверки, но проверяет, что в формате номера (например, "1.1") точка после последней цифры отсутствует;
3. И, например, Проверка структурного заголовка при использовании функции *check\_structural\_or\_appendix\_heading\_format* проверяет выравнивание по центру и отсутствие отступа первой строки, а также требует, чтобы все текстовые *runs* внутри этого абзаца были полужирными.

**2.4 Реализация способа предоставления результатов проверки**

Для надежного способа информирования пользователя об ошибках, найденных в ходе проверки, требовалось как-то наглядно предоставить результаты. В главе с разбором анализа инструментария уже было сказано, что для этого был выбран подход с добавлением комментариев в документ.

После неудачных экспериментов с прямой модификацией XML-структуры документа, которые приводили к непредсказуемому "съезжанию" комментариев, и отказа от платформозависимого COM-интерфейса, было принято решение полностью перейти на использование высокоуровневого API библиотеки *python-docx* (важное уточнение в том, что это стало доступным только с версии 1.2.0 и выше). Основным инструментом стал метод *document.add\_comment(runs, text, author, initials*). Он позволяет программно создавать комментарии, аналогичные тем, что пользователи добавляют вручную в MS Word, и привязывать их к конкретным участкам текста (объектам *Run*). Работает корректно в отличии от библиотеки *lxml*, которая позволяла мне работать с xml структурой DOCX, верно привязывал комментарии к индексам определяемых абзацев. Использование *document.add\_comment()*полностью устранило проблему, с которой пришлось столкнуться при работе с XML. Поскольку *python-docx* теперь сам управлял созданием и привязкой комментариев на основе объектов *Run*, полученных из его же объектной модели документа, рассинхронизация индексов исключалась. Библиотека корректно вставляет XML-маркеры *commentRangeStart* и *commentRangeEn* вокруг указанных *runs* и создает соответствующие записи в *comments.xml* и файлах связей.

Для взаимодействия между модулем проверки и модулем добавления комментариев используется простая структура данных – список кортежей(*paragraph\_index, comment\_text, author*). Здесь *paragraph\_index* – это глобальный индекс абзаца в списке *doc.paragraphs*, полученный через *enumerate*, а *comment\_text* – текст сообщения об ошибке и *Author* –это имя автора комментария. Его возможно менять в интерфейсе приложения.

В функции *comment\_utils.py* (после уже перехода с *lxml* библиотеки на нативные методы добавления комментариев из *python-docx)* по *paragraph\_index* из этого списка находился нужный объект абзаца (*doc.paragraphs[paragraph\_index]*), и из него извлекаются все его runs (paragraph.runs), а к ним уже привязывается комментарий. Для общих ошибок (например, полей страницы) используется специальный индекс -1, который обрабатывается отдельно.

**2.5 Разработка и развертывание веб-интерфейса**

Очевидно, что для обеспечения кроссплатформенности и доступности широкому кругу интересующихся лиц требовалась веб-версия приложения. Сперва приложение в своём наиболее раннем виде предоставляло набор скриптов и интерфейс на *Tkinter* библиотеке, но после закрытия вопросов с реализацией механизмов идентификации и проверки элементов в тексте, было принято решение перейти на какой-нибудь веб-фреймворк. И в качестве основы был выбран микрофреймворк *Flask* для Python. Довольно гибкий и минималистичный набор основных методов и компонентов легко позволил перенести функциональность на хостинг.

У созданного веб-приложения получилось реализовать то, что планировалось на этапе анализа, а именно:

1. Главная страница с формой для загрузки файла (index.html), интерфейс которой предоставлен на Рисунке 5;

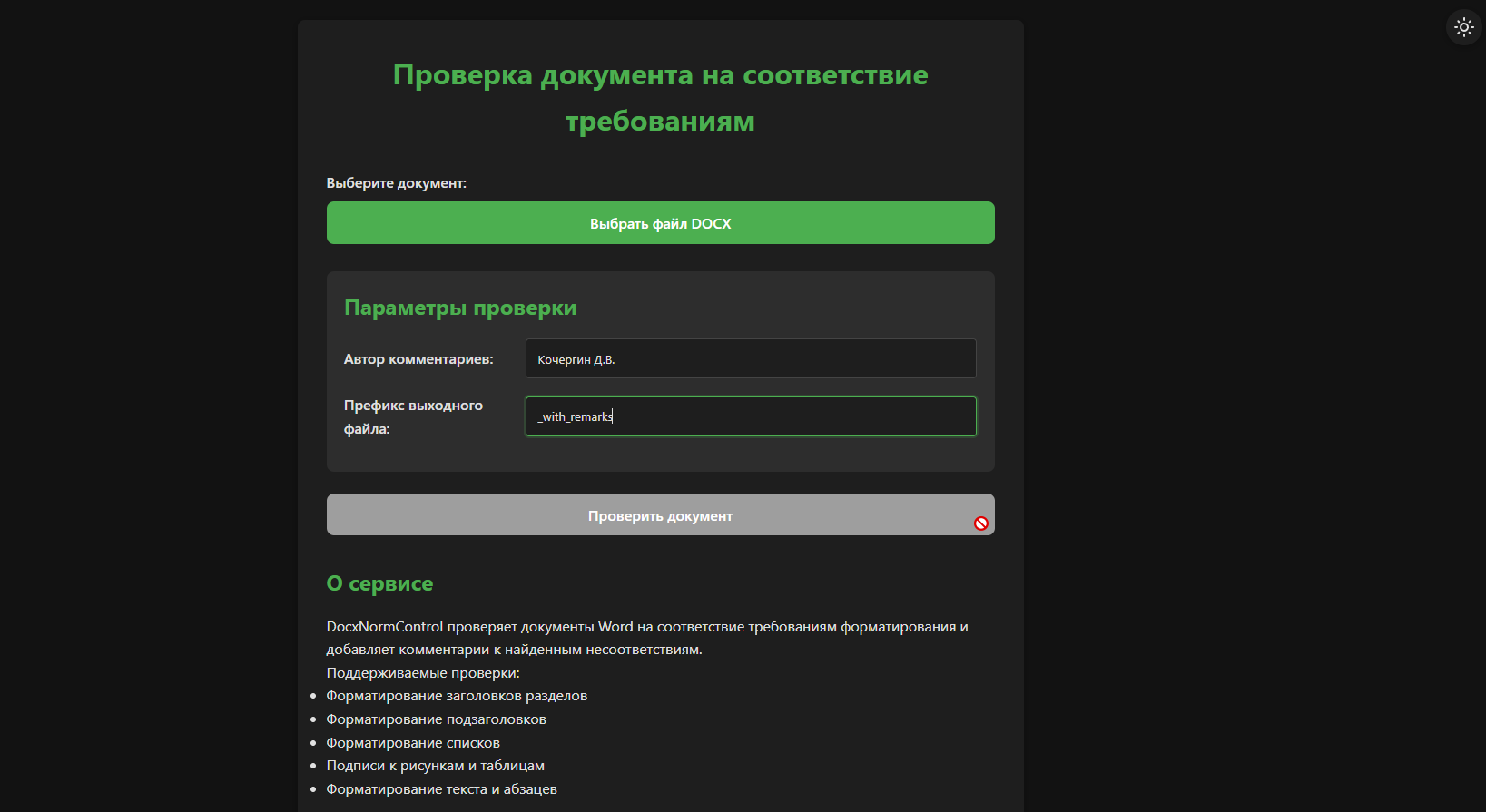


Рисунок 5 – Внешний вид главной страницы приложения

1. Возможность выбора файла формата DOCX и задания некоторых параметров проверки (например, имя автора комментариев, а так же приписку к имени выходного файла), что видно на Рисунке 6;

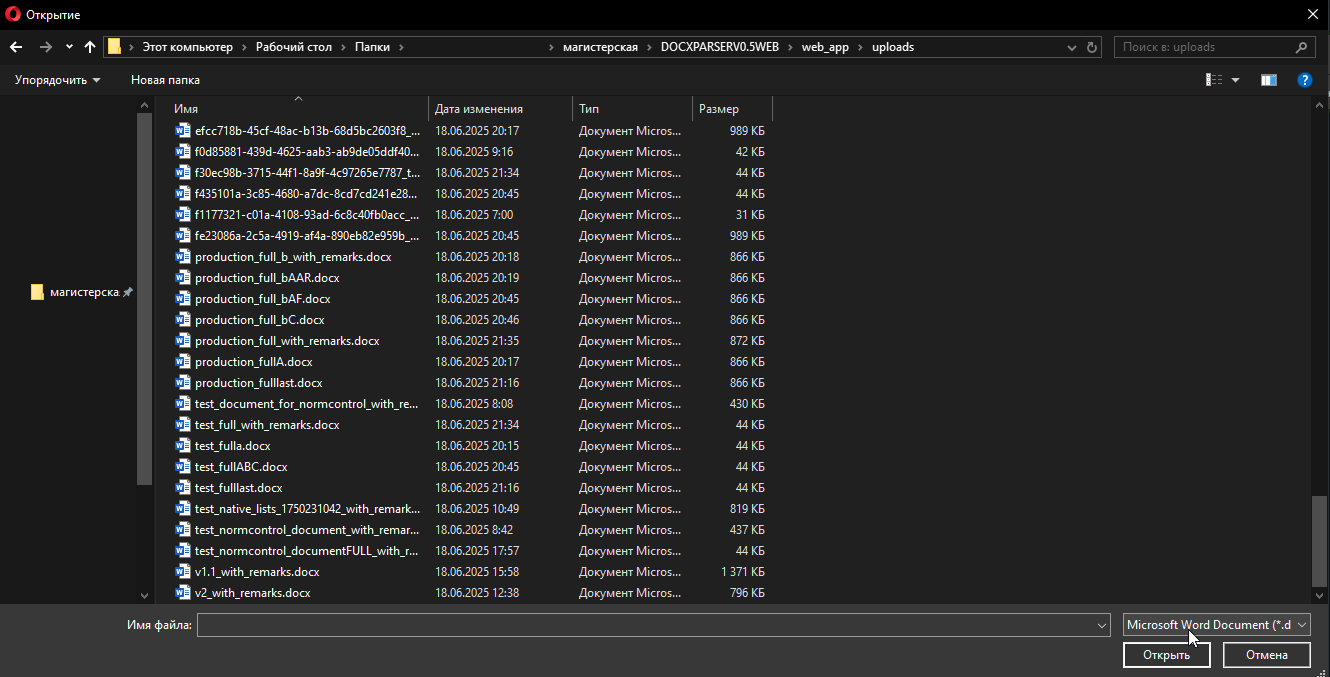


Рисунок 6 – Загрузка файла строго заданного формата

1. Маршрут для обработки загруженного файла, который вызывает ядро системы нормоконтроля (*check\_document\_formatting\_final*);
2. И страница результатов (*result.html*), которая либо сообщает об отсутствии ошибок, либо предоставляет пользователю ссылку для скачивания обработанного документа с комментариями, а также выводит краткую статистику проверки, как на Рисунке 7 и Рисунке 8.

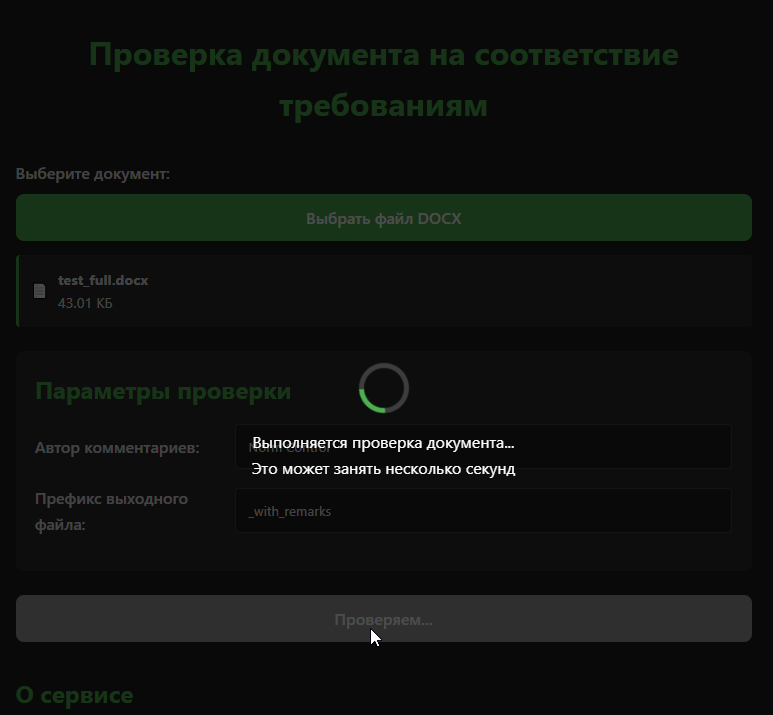


Рисунок 7 – Процесс обработки файла со стороны пользователя

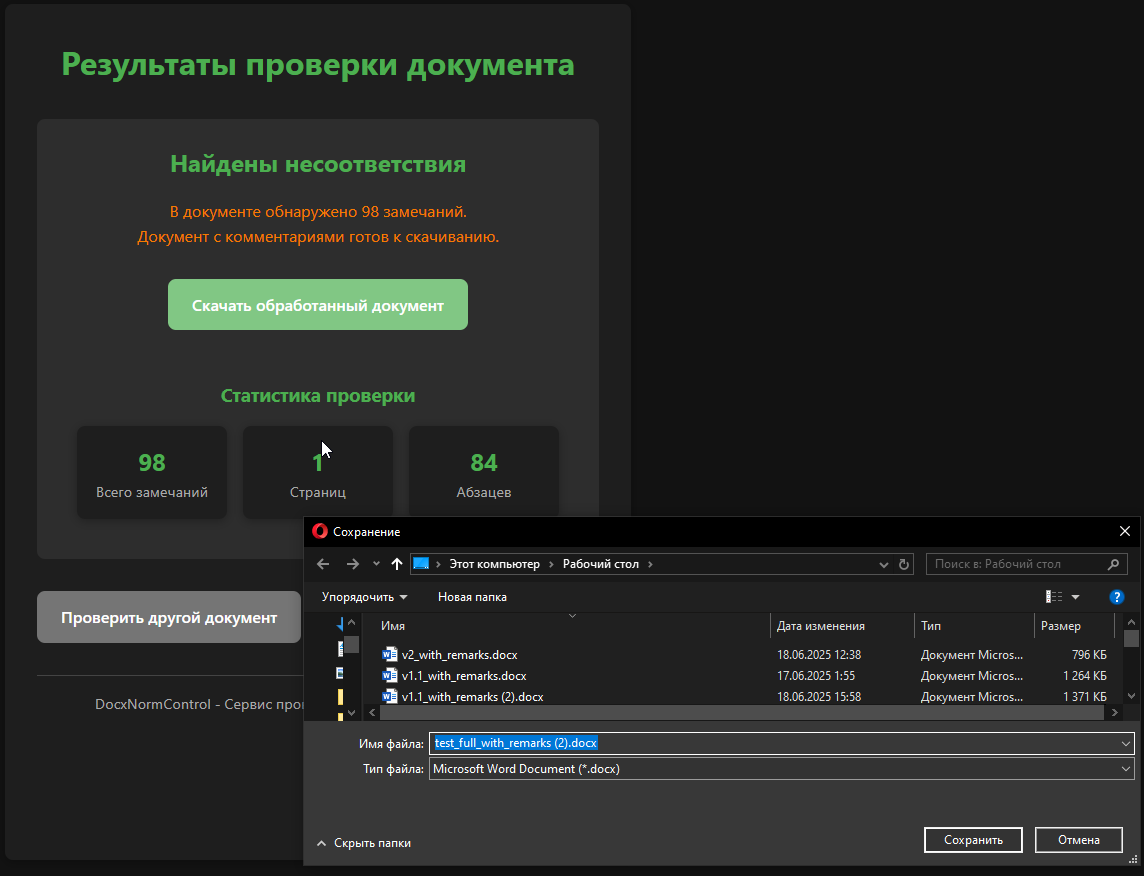


Рисунок 8 – Экран с результатами проверки и возможностью скачивания файла с внесёнными комментариями

Для публикации приложения в сети рассматривалось много вариантов, но была выбрана платформа*PythonAnywhere* [18] как первая, с которой возникло меньше всего трудностей в процессе переноса и настройки клиента приложения. На самом хостинге довольно просто настроить Flask, есть бесплатный тарифного плана, достаточный для прототипа, и отсутствует необходимости в сложном администрировании сервера. Сам процесс развертывания включал загрузку кода, настройку WSGI-файла для связи веб-сервера с Flask-приложением и конфигурирование статических файлов CSS и JS.

**3. Тестирование, анализ результатов и ограничений системы**

После завершения этапов проектирования и разработки ключевых модулей системы автоматизированного нормоконтроля нужно было провести комплекс мероприятий по ее тестированию и анализу результатов для того, чтобы выявить ограничения разработанной системы на реальных DOCX файлах, предоставленных другими пользователями.

**3.1 Методика тестирования**

Сперва для о проверки функциональности системы создавались синтетические тестовых данных и автоматизация процесса проверки.Для этого был создан набор специализированных DOCX-документов. Вместо того чтобы полагаться исключительно на случайные документы, был разработан программный скрипт (*generate\_test\_docx.py*) с использованием библиотеки *python-docx,* который генерировал тестовые файлы с преднамеренными ошибками оформления. Получилось целенаправленно проверить реакцию системы на каждое конкретное правило нормоконтроля. Пример содержания файла предоставлен на Рисунке 9 и Рисунке 10.

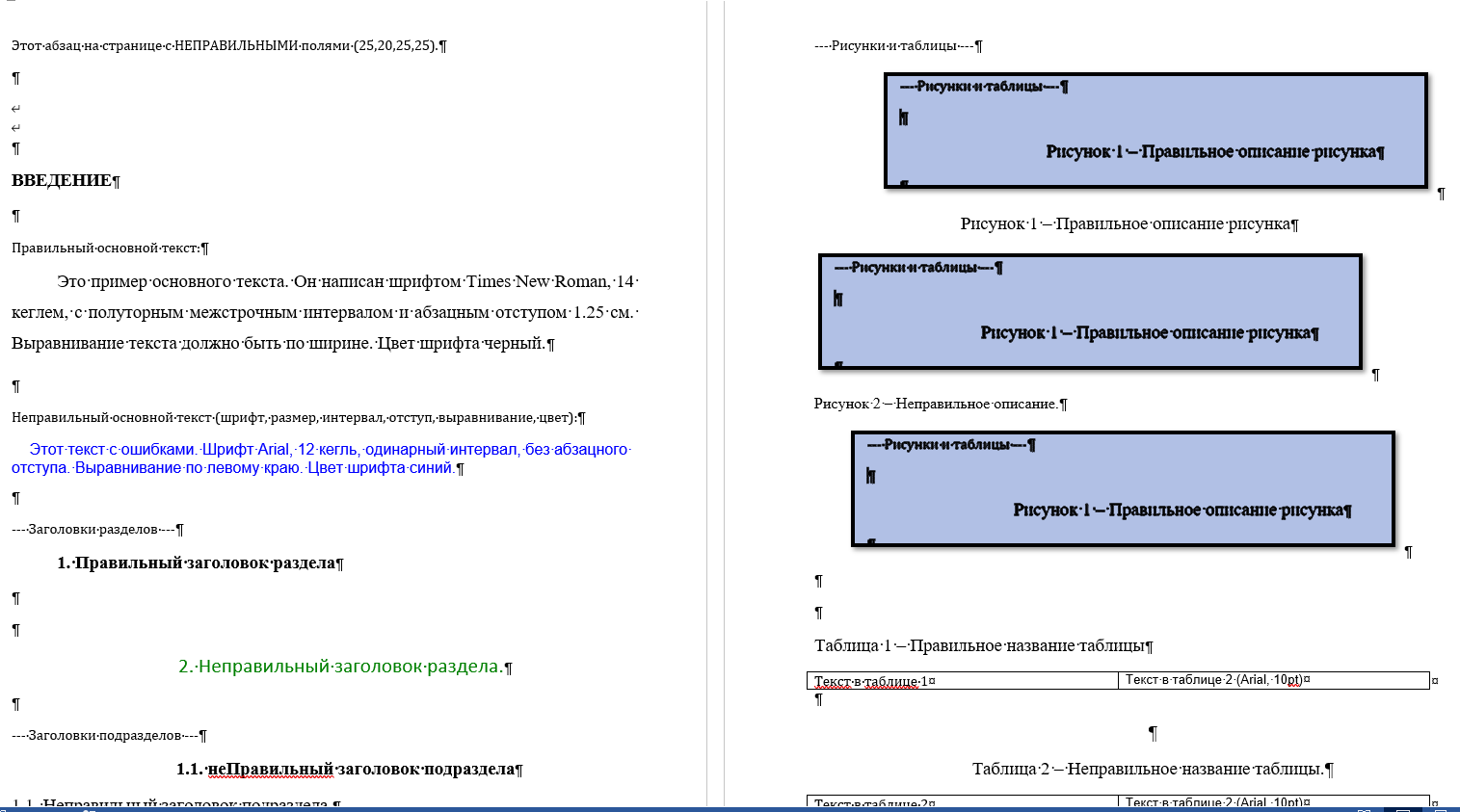


Рисунок 9 – Содержимое тестового .docx файла

Тестовые документы были спроектированы таким образом, чтобы охватить множество проблемных случаев, выявленных на этапе анализа.

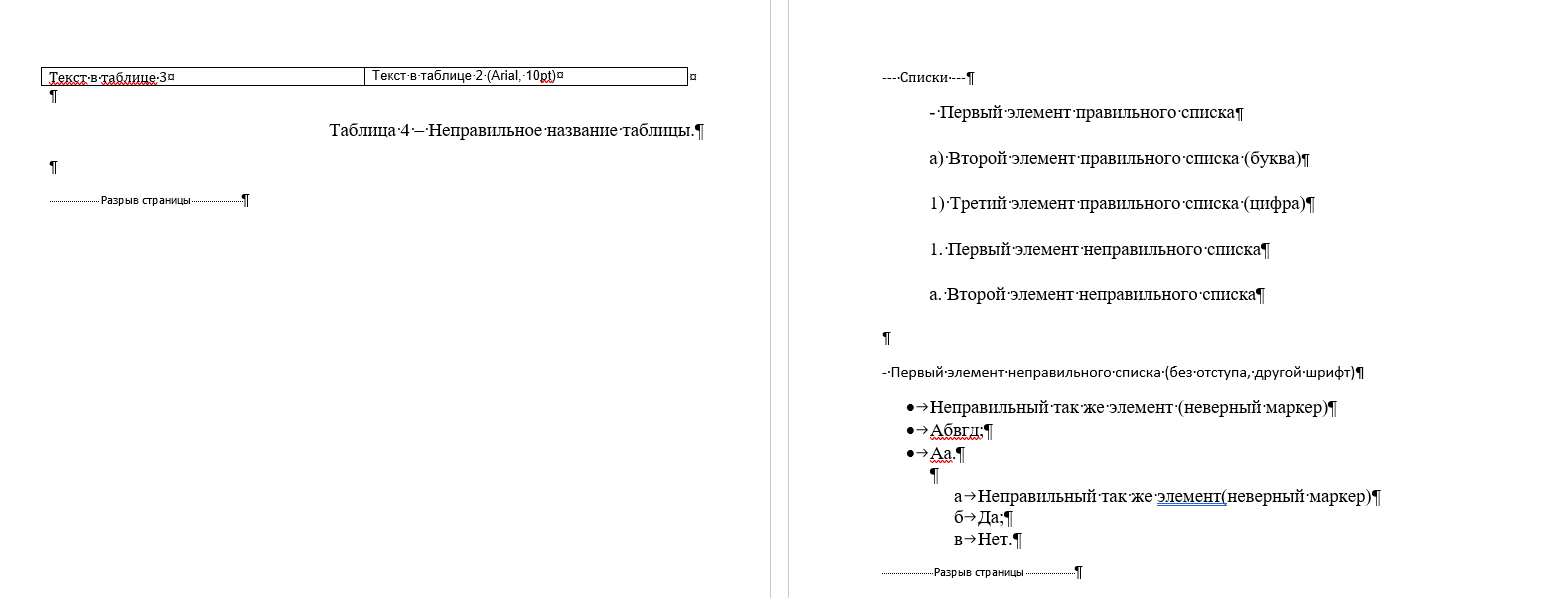


Рисунок 10 – Содержимое тестового .docx файла с генерированными списками

Такие документы содержали абзацы с неправильным шрифтом, размером, цветом, межстрочным интервалом, выравниванием и отступами. В них использовались элементы, где форматирование задавалось как напрямую, так и через встроенные или пользовательские стили, для проверки корректности работы модуля *formatting\_utils.py* по определению "эффективного" форматирования.

Для проверки форматирования структурных заголовков включались заголовки разделов и подразделов с неправильной нумерацией (наличие/отсутствие точек), без полужирного начертания, с некорректным регистром или выравниванием. Ещё документы содержали различные способы создания пустых строк (через пустые абзацы или через настройки интервалов *space\_after/space\_before*), таблицы, списки, подписи к рисункам, что позволяло проверить устойчивость каскадного алгоритма определения типов элементов.

А в целом для каждого правила также были созданы правильно оформленные элементы, чтобы убедиться в отсутствии ложноположительных срабатываний системы.

В процессе разработки стало понятно, что нужна ещё и систематическая проверка разработанных функций. Для этого были созданы модульные тесты. Как видно из структуры проекта на Рисунке 11 для каждого ключевого компонента или группы правил был подготовлен отдельный тестовый скрипт.

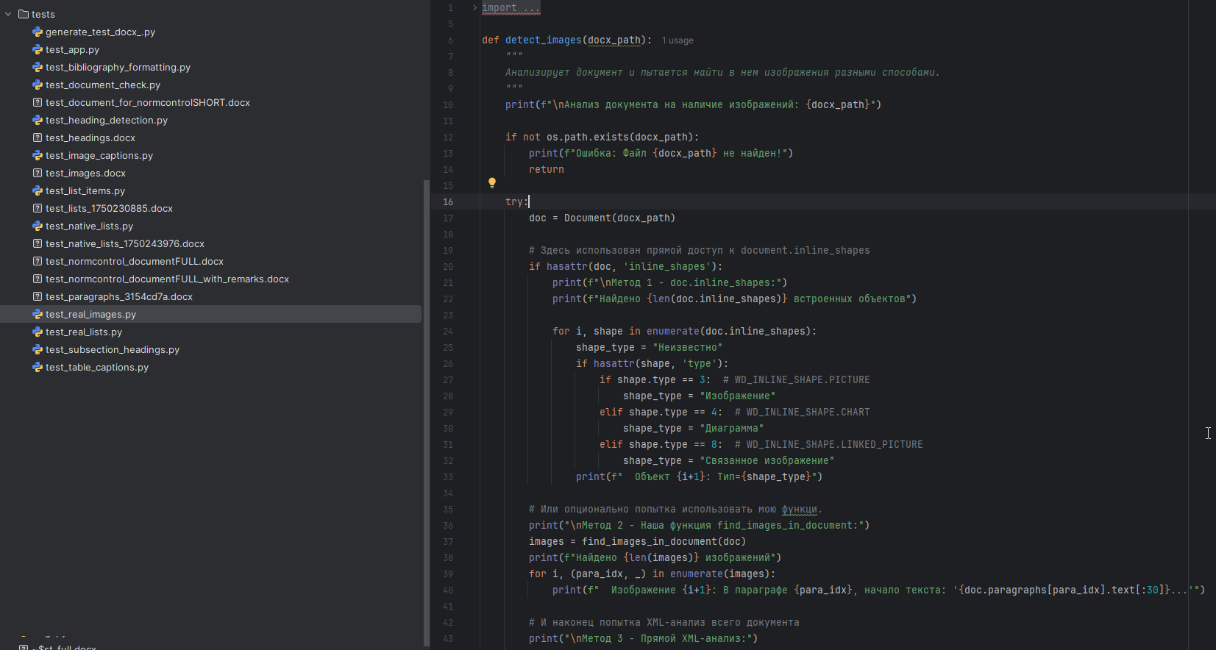


Рисунок 11 – Отдельные скрипты для тестирования изменений функций и их корректной работы

Так получилось изолированно проверять правильность работы каждой функции-детектора (*is\_*...) и функции-валидатора (*check\_*...*\_format*) и автоматически запускать проверки после внесения изменений в основной код. Получилась эдакая практика непрерывной интеграции, которая позволила избежать наслоения ошибок при непрерывном изменении одной и той же функции без промежуточной проверки её работоспособности.

Основным критерием оценки работы системы в ходе тестирования являлось сравнение списка ошибок, сгенерированных системой (*comments\_to\_add*), с ожидаемым списком ошибок для каждого тестового документа. Ожидаемый список ошибок формировался вручную самим автором работы.

**3.2 Результаты тестирования и их анализ**

Тестирование системы на наборе подготовленных документов позволило получить важные данные о ее точности, полноте и понять особенности работы. Пример выходного файла после проверки предоставлен на Рисунке 12, Рисунке 13, Рисунке 14, Рисунке 15 и Рисунке 16.

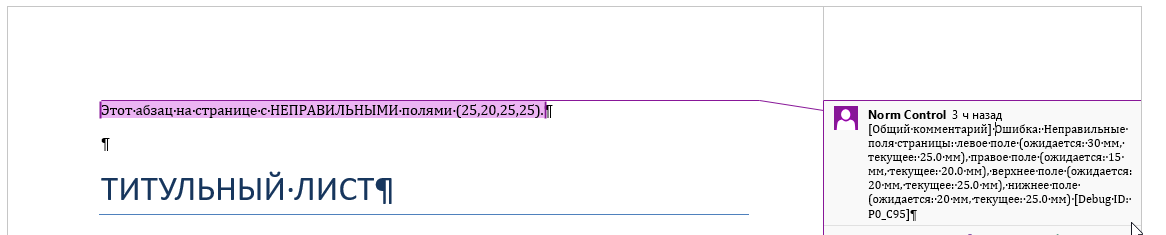


Рисунок 12 – Отображение неверного форматирования полей документа

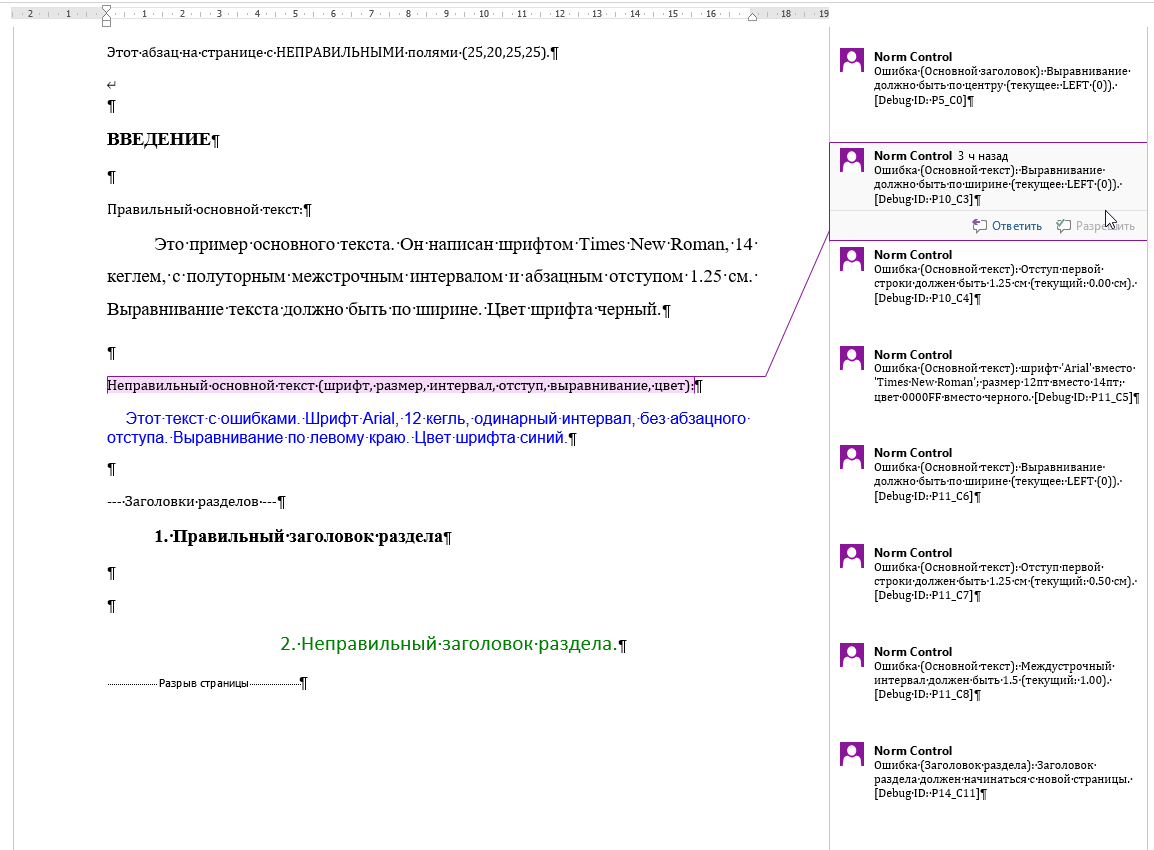


Рисунок 13 – Множественные ошибки неверного форматирования отступов, интервалов, шрифтов и положений заголовков

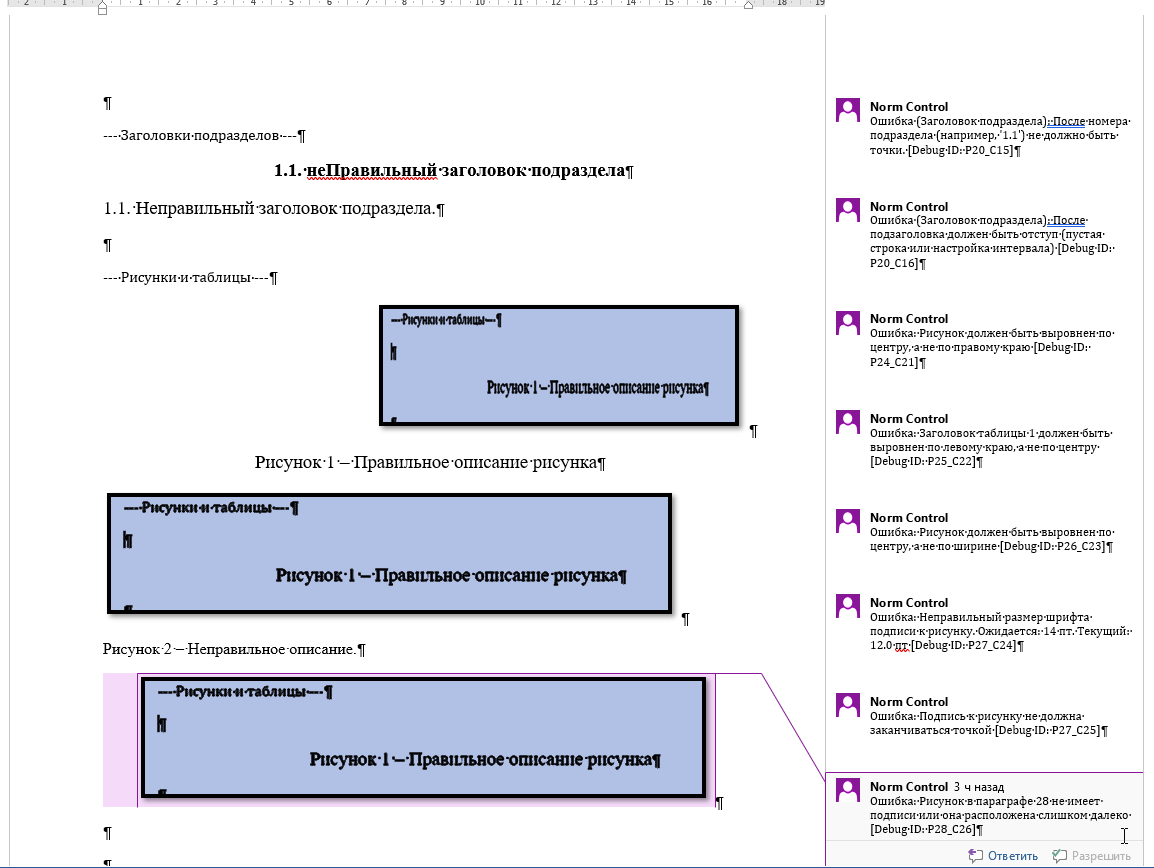


Рисунок 14 – Множественные ошибки с неверным положение рисунков, подписей к ним и ошибки нумерации разделов

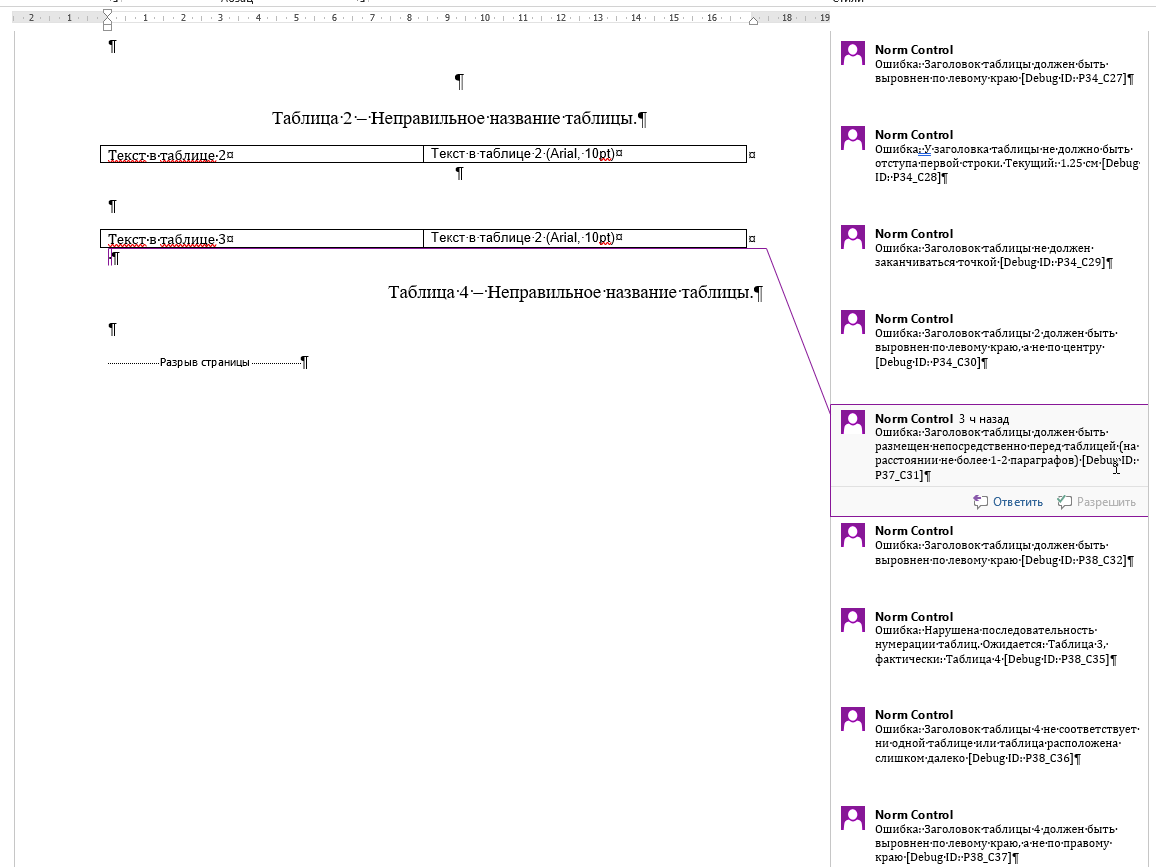


Рисунок 15 – Множественные ошибки расположения подписей к таблицам, отсутствия подписей и форматирования самих подписей

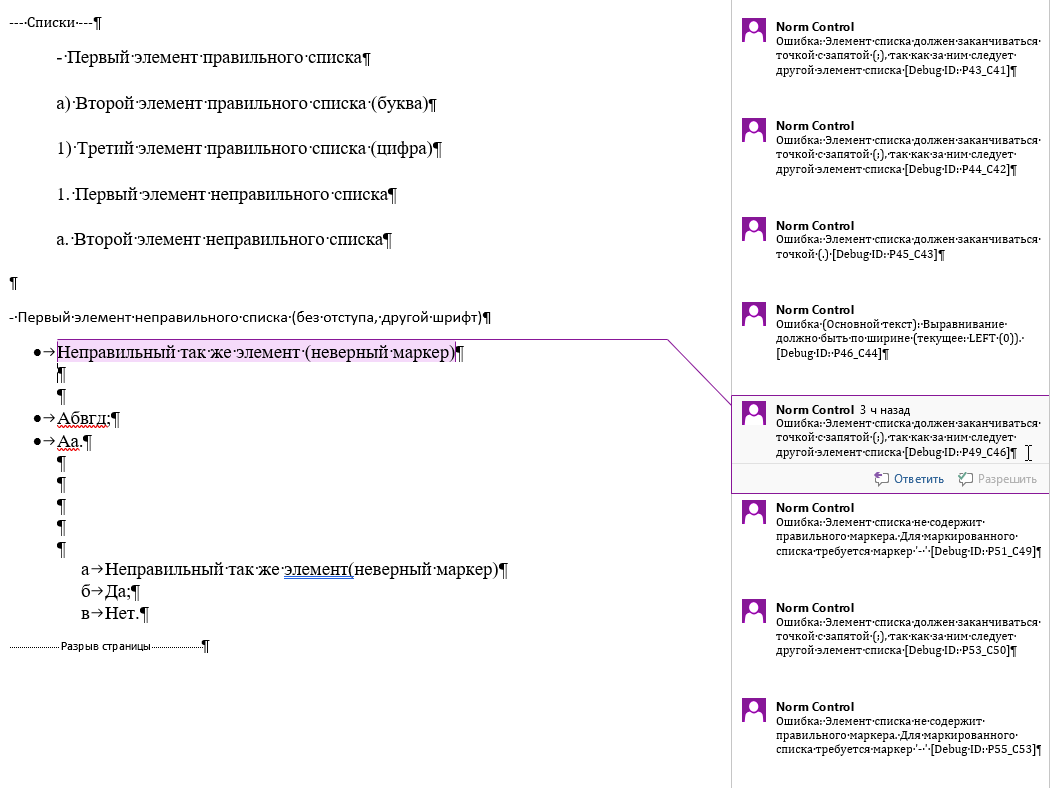


Рисунок 16 – Ошибки в форматировании нумерованных и маркированных списков (некоторые удалены для удобства отображения)

В ходе тестирования было установлено, что переход на использование высокоуровневого API *document.add\_comment()* библиотеки *python-docx* (версии 1.2.0 и выше) позволил решить основную проблему "съезжания" комментариев, которая наблюдалась при попытках ручной модификации XML. В документах со стандартной структурой, где абзацы не содержат сложных вложенных элементов, комментарии к ошибкам форматирования (например, неправильный отступ или шрифт в абзаце основного текста) корректно привязывались к соответствующим абзацам.Но несмотря на общее улучшение, были выявлены сценарии, где точность привязки может быть неидеальной. Например, если ошибка относится к форматированию пустого абзаца (который не содержит текста и, следовательно, объектов *Run*), API *document.add\_comment()* не может привязать комментарий, так как ему требуется хотя бы один *Run*. В текущей реализации такие ошибки игнорируются или требуют обходных путей (например, вставка временного пробела, от чего было решено отказаться в пользу стабильности). Также, привязка комментариев к элементам, не являющимся текстом (например, общие ошибки полей страницы), была реализована условно – путем прикрепления комментария к первому абзацу документа.

Когда была обнаружена проблема работы с сложной системой насследования пользовательских стилей, созданный модуль *formatting\_utils.py* позволил частично разрешить конфликты. В случаях, когда форматирование (например, отступ или выравнивание) было задано через стиль абзаца, а не напрямую, система в большинстве случаев успешно извлекала "эффективное" значение, обращаясь к свойствам объекта *para.style*. Это позволило избежать большого количества ложных срабатываний, когда атрибут прямого форматирования возвращал *None*.

Тем не менее проблемы возникали в документах со сложной иерархией вложенных стилей (когда один пользовательский стиль основан на другом, который, в свою очередь, основан на встроенном). Текущая реализация *formatting\_utils.py*, проверяющая прямое форматирование и один уровень родительского стиля, не всегда может вычислить итоговое значение, что иногда приводит к пропуску ошибок или, наоборот, к ложным срабатываниям. К сожалению, в ходе работы так и не получилось прийти к адекватному разрешению этой ситуации.

Можно подвести итог того, что система продемонстрировала высокую точность в определении и проверке *структурных заголовков и заголовков разделов/подразделов*, где правила определения (комбинация стиля, паттерна и жирности) были надежными (т.е. уже были корректно заданы некоторые условия идентификации этих элементов, и пользователь хотя бы писал правильно названия этих заголовков, а в разделах добавлял нумерацию).

Проверка основного текста на соответствие базовым требованиям (шрифт, размер, отступ, интервал, выравнивание) также показала хорошие результаты на документах с простым и "чистым" форматированием. Т.е. без использования множества насследуемых друг от друга стилей.

Точность проверки более сложных элементов, таких как многоуровневые списки и библиографические записи, оказалась ниже из-за большого разнообразия способов их оформления пользователями и сложности семантического анализа. Он не был реализован в процессе этой работы.

**3.3 Обсуждение достигнутых результатов и нерешенных проблем**

Разработан и успешно применен каскадный метод определения типов элементов, который показал свою эффективность в идентификации основных структурных единиц документа.

По мнению автора успешно решена проблема анализа базового форматирования в документах со стилями путем создания утилитарных функций для получения "эффективного" форматирования, что является значительным шагом вперед по сравнению с наивным анализом только прямых атрибутов, которые я реализовывал в первых версиях приложения.

Найдено и реализовано надежное решение для добавления комментариев через высокоуровневое API *python-docx*, что устранило критический недостаток "съезжания" ошибок, присутствовавший в первоначальных прототипах.

Создана рабочая архитектура, состоящая из независимых модулей, что упрощает дальнейшее расширение и поддержку системы.

С другой стороны, существуют и ограничения *python-docx* в полном анализе каскада стилей. Как и предполагалось, библиотека не предоставляет инструментов для полного разрешения сложной иерархии наследования стилей. Разработанные утилиты решают проблему лишь частично, что оставляет пространство для ошибок в сложных документах. Собственно, так и происходит. Велик процент ложноположительных срабатываний при анализе таких элементов.

Существуют так же и проблемы с «невидимыми" или нестандартно структурированными элементами DOCX. Проблемы с индексацией все еще могут возникать на этапе анализа, если, например, *doc.paragraphs* и *doc.tables* возвращают элементы в порядке, не строго соответствующем их визуальному расположению в документе со сложной версткой.

Считаю нужным выделить так же то, что совершенно не получилось реализовать в рамках работы. Это:

1. Прежде всего семантический анализ списка использованных источников. Текущая проверка ограничивается форматом (*N. Текст*) и наличием ключевых признаков (год, разделители). Полная проверка на соответствие ГОСТ по содержанию и правильности всех элементов описания требует гораздо более сложных лингвистических и эвристических алгоритмов;
2. Сейчас осуществляется только проверка простых одноуровневых списков, а вот анализ корректности вложенности и форматирования многоуровневых списков не реализован. Полагается, что основной трудностью являлось отсутствие в *python-docx* прозрачного и понятного механизма анализа нумерованного или маркированного списка, вложенного в другой список.
3. Не получилось и воплотить в жизнь и такие требования, как "не начинать страницу с последней строки абзаца" (висячие строки) или "иллюстрация должна быть качественной". Автору видится, что они в принципе не могут быть проверены инструментами, анализирующими структуру, а не визуальное представление документа. Здесь понадобились бы алгоритмы OCR (оптического распознавания текста) и, возможно, использование современных технологий нейросетевых моделей.

В целом стоит выделить следующий перечень нереализованных проверок:

1. Проверка "висячих строк";
2. Проверка корректности переносов слов в заголовках;
3. Полный семантический анализ библиографических записей;
4. Детальный анализ форматирования содержимого таблиц.

**3.4 Сравнение с альтернативными подходами**

Вообще, оглядываясь на анализ, который проводился ещё в первой главе, хотелось бы заключить, что выбранный подход, несмотря на свои ограничения, является оптимальным в поставленной задаче.

Из доступных автору инструментов, например, в сравнении с работой с XML напрямую, итоговое решение значительно проще в разработке и поддержке, а главное – лишено критической проблемы некорректной привязки комментариев.

В сравнении с использованием COM-интерфейса, разработанная система является кроссплатформенной, не требует наличия установленного MS Word и легко развертывается как веб-сервис, что вообще и было для меня ключевым требованием.

А вот в сравнении с существующими универсальными линтерами (Vale, LanguageTool), данная система сфокусирована именно на специфических правилах нормоконтроля оформления по российским стандартам, а не на общей грамматике и стилистике.

**3.5 Рекомендации по использованию и направления для дальнейшего развития**

По мнению автора систему рекомендуется использовать как инструмент для предварительной самопроверки документов. Пользователям следует помнить об ограничениях системы и не рассматривать ее как замену финальной проверке человеком, особенно по части семантических и сложных визуальных правил.

Для дальнейших улучшений стоило бы рассмотреть такие направления:

1. Улучшение алгоритмов определения, видимого пользователю, а не «зашитого» в каскад насследуемых стилей форматирования. Возможно исследование более глубокого анализа XML-файла *styles.xml* для построения полного дерева наследования стилей.
2. Постепенное добавление проверок для более сложных случаев (многоуровневые списки, содержимое таблиц).
3. Наконец развитие модуля для проверки списка использованных источников. Тут можно рассмотреть интеграцию с внешними сервисами или разработка более сложных эвристик для проверки соответствия ГОСТ.
4. Наиболее очевидным, кстати, является использование машинного обучения. В перспективе, для определения типов элементов в очень неоднозначных случаях можно было бы исследовать применение моделей машинного обучения, обученных на большом корпусе правильно оформленных документов. Для автора эта задача видится довольно сложной в силу отсутствия релевантного опыта.
5. Так же можно развить и веб-интерфейс для предоставления более детальных отчетов и, возможно, предложений по автоматическому исправлению некоторых ошибок, раз *python-docx* отлично справляется с генерацией тестовых документов, то смог бы и автоматически исправлять некоторые проблемные места.

Заключение

Объективно оценивая результаты работы, видится, что поставленная цель – разработка системы автоматизированного нормоконтроля – была достигнута частично. Путь к ней оказался сложнее и интереснее, чем предполагалось изначально. В ходе исследования и разработки удалось создать работающий прототип [19], глубоко погрузиться в проблемы программной обработки и анализа современных текстовых документов.

Выбор инструментария играет решающую роль. Высокоуровневые библиотеки, такие как *python-docx*, значительно ускоряют разработку, но их ограничения (например, в части анализа наследования стилей) требуют создания собственных "обходных путей" и утилит. При этом, низкоуровневые подходы, как прямая работа с XML, хоть и кажутся всемогущими, на практике оказываются слишком хрупкими и трудоемкими, а также порождают собственные, не менее сложные проблемы, вроде синхронизации элементов документа.

Для надежного определения типа элемента (заголовок, основной текст, список и т.д.) наиболее эффективным оказался каскадный метод, сочетающий анализ нескольких признаков: стиля, текстового паттерна и атрибутов форматирования. Полагаться только на один из них в "живых" документах, созданных разными людьми, оказалось недостаточно.

В целом, удалось разработать и реализовать систему [19], которая анализирует DOCX-документы и предоставляет обратную связь пользователю в виде комментариев. Главные трудности, которые пришлось преодолеть, – это корректное получение "эффективного" форматирования из стилей и обеспечение точной привязки комментариев к проблемным местам в тексте. Первая проблема была частично решена созданием вспомогательных функций, вторая потребовала полного отказа от первоначального подхода с ручной модификацией XML. Это дало некоторую стабильность результатов.

С практической точки зрения созданный прототип – это готовый инструмент, который может помочь студентам и авторам научных работ значительно сократить время на самопроверку и улучшить качество оформления своих документов. Реализация системы в виде веб-приложениях [19] делает ее доступной для любого пользователя без необходимости установки какого-либо программного обеспечения. А главное, что разработанные алгоритмы и модули могут быть использованы и другими разработчиками для создания или улучшения собственных систем анализа текстовых документов.

Работа закладывает основу для дальнейших исследований и разработок в области автоматизированного анализа и улучшения качества текстовых документов.

Список использованных источников

1. Структура и оформление магистерской диссертации, бакалаврской, дипломной и курсовой работ : учебно-методические указания / М.Б. Астапов, Ж.О. Карапетян, О.А. Бондаренко, В.В. Бондаренко. – Краснодар : Кубанский государственный университет, 2021. – 58 с.
2. ГОСТ Р 7.0.5–2008. Система стандартов по информации, библиотечному и издательскому делу. Библиографическая ссылка. Общие требования и правила составления : национальный стандарт Российской Федерации : издание официальное : утвержден и введен в действие Приказом Федерального агентства по техническому регулированию и метрологии от 28 апреля 2008 г. № 95-ст : введен впервые : дата введения 2009-01-01. – Москва : Стандартинформ, 2008. – 21 с.
3. ГОСТ Р 7.0.100–2018. Система стандартов по информации, библиотечному и издательскому делу. Библиографическая запись. Библиографическое описание. Общие требования и правила составления : национальный стандарт Российской Федерации : издание официальное : утвержден и введен в действие Приказом Федерального агентства по техническому регулированию и метрологии от 3 декабря 2018 г. № 1050-ст : введен впервые : дата введения 2019-07-01. – Москва : Стандартинформ, 2018. – IV, 65 с.
4. ГОСТ 7.32–2017. Система стандартов по информации, библиотечному и издательскому делу. Отчет о научно-исследовательской работе. Структура и правила оформления : межгосударственный стандарт : издание официальное : введен в действие Приказом Федерального агентства по техническому регулированию и метрологии от 24 октября 2017 г. № 1494-ст : дата введения 2018-07-01. – Москва : Стандартинформ, 2017. – IV, 27 с.
5. Система NormaCS : [сайт]. – URL: <https://www.normacs.ru> (дата обращения: 25.04.2025).
6. PerfectIt – The Professional Proofreading Software for Professionals : [сайт]. – URL: [https://www.perfectit.com](https://www.google.com/url?sa=E&q=https%3A%2F%2Fwww.perfectit.com) (дата обращения: 25.04.2025).
7. Vale: A syntax-aware linter for prose : [Электронный ресурс] // GitHub. – URL: <https://github.com/errata-ai/vale> (дата обращения: 12.05.2025).
8. LanguageTool – Style and Grammar Checker for 25+ Languages : [Электронный ресурс] // GitHub. – URL: <https://github.com/languagetool-org/languagetool> (дата обращения:13.05.2024).
9. Антиплагиат : [сайт]. – URL: <https://antiplagiat.ru>(дата обращения: 12.05.2025).
10. Файл DOCX - Что такое файл .docx и как его открыть? : [Электронный ресурс] // FileFormat.com. – URL: <https://docs.fileformat.com/ru/word-processing/docx/> (дата обращения: 14.05.2025).
11. Файл ODT - Что такое файл .odt и как его открыть? : [Электронный ресурс] // FileFormat.com. – URL: <https://docs.fileformat.com/ru/word-processing/odt/> (дата обращения: 14.05.2025).
12. Новый редактор : [Электронный ресурс] // Хабр. – URL: <https://habr.com/ru/docs/help/wysiwyg/> (дата обращения: 14.05.2025).
13. Кузнецов, А. Работа с Microsoft Word из Python / А. Кузнецов // Хабр. – 2017. – 10 марта. – URL: <https://habr.com/ru/companies/intersystems/articles/321044/> (дата обращения: 20.05.2025).
14. python-docx Documentation : [Электронный ресурс] // Read the Docs. – URL: <https://python-docx.readthedocs.io/en/latest/> (дата обращения: 20.05.2025).
15. lxml - XML and HTML with Python : [сайт]. – URL: <https://lxml.de> (дата обращения: 13.04.2025).
16. pywin32 Documentation : [Электронный ресурс]. – URL: <https://brian3johnson.github.io/pywin32/index.html> (дата обращения: 18.05.2025).
17. Антонов, А. Самые полезные модули для Python по версии DataArt / А. Антонов // Хабр. – 2017. – 29 ноября. – URL: <https://habr.com/ru/articles/344086/> (дата обращения: 13.04.2025).
18. PythonAnywhere Help : [сайт]. – URL: <https://help.pythonanywhere.com/pages/> (дата обращения: 25.05.2025).
19. DocxNormControl : система автоматизированного нормоконтроля : [сайт] – URL:<https://trelawney.pythonanywhere.com> (дата обращения: 25.05.2025).

ПРИЛОЖЕНИЕ А

Код модуля добавления комментариев *comment\_utils.py*

import uuid

from datetime import datetime

from lxml import etree

from docx import Document

import zipfile

import os

import shutil

from pathlib import Path

def qn(tag):

    """

    Я сделал эту функцию для квалификации имен XML тегов в docx

    """

    namespace = {'w': 'http://schemas.openxmlformats.org/wordprocessingml/2006/main'}

    prefix, tagroot = tag.split(':')

    return '{{{}}}{}'.format(namespace[prefix], tagroot)

def is\_paragraph\_in\_table(paragraph\_element, namespaces):

    """

    Проверяет, находится ли параграф внутри таблицы.

    Args:

        paragraph\_element: XML-элемент параграфа

        namespaces: словарь namespaces для xpath

    Returns:

        bool: True, если параграф находится внутри таблицы

    """

    parent = paragraph\_element.getparent()

    while parent is not None:

        if parent.tag == '{http://schemas.openxmlformats.org/wordprocessingml/2006/main}tc':

            return True

        parent = parent.getparent()

    return False

def add\_comments\_to\_docx(input\_path, output\_path, comments\_info):

    """

    Добавляет комментарии в DOCX документ

    Args:

        input\_path: путь к исходному документу

        output\_path: путь для сохранения документа с комментариями

        comments\_info: список кортежей (paragraph\_index, comment\_text, author)

    """

    doc = Document(input\_path)

    debug\_info = []

    debug\_info.append(f"Всего параграфов в документе: {len(doc.paragraphs)}")

    # Сортируем комментарии по возрастанию индекса параграфа

    sorted\_comments = sorted(comments\_info, key=lambda x: x[0] if x[0] >= 0 else float('inf'))

    # А это - словарь для хранения отображений специальных индексов на реальные параграфы

    special\_index\_mapping = {

        -1: 0,  # Индекс -1 (сноски) -> первый параграф

    }

    # Фильтруем параграфы, исключая те, которые находятся в таблицах

    body\_paragraphs = []

    for i, para in enumerate(doc.paragraphs):

        # Проверяем, не находится ли параграф в таблице

        in\_table = False

        for table in doc.tables:

            for row in table.rows:

                for cell in row.cells:

                    if para in cell.paragraphs:

                        in\_table = True

                        break

                if in\_table:

                    break

            if in\_table:

                break

        if not in\_table:

            body\_paragraphs.append(para)

    debug\_info.append(f"Параграфов основного тела (не в таблицах): {len(body\_paragraphs)}")

    # Добавляем комментарии

    for comment\_index, (paragraph\_index, comment\_text, author) in enumerate(sorted\_comments):

        if paragraph\_index < 0:

            if paragraph\_index in special\_index\_mapping:

                paragraph\_index = special\_index\_mapping[paragraph\_index]

            else:

                # Для неизвестных отрицательных индексов используем первый параграф

                paragraph\_index = 0

            # Добавляем префикс к тексту комментария для общих комментариев

            if "[Общий комментарий] " not in comment\_text:

                comment\_text = f"[Общий комментарий] {comment\_text}"

            target\_para = doc.paragraphs[paragraph\_index]

        else:

            # Проверка на валидность индекса

            if paragraph\_index >= len(body\_paragraphs):

                debug\_info.append(f"Ошибка: Индекс {paragraph\_index} вне диапазона основного тела (всего {len(body\_paragraphs)} параграфов)")

                # Используем последний параграф основного тела, если индекс вне диапазона

                paragraph\_index = len(body\_paragraphs) - 1

            target\_para = body\_paragraphs[paragraph\_index]

        # Добавляю ID параграфа для отладки

        debug\_id = f"P{paragraph\_index}\_C{comment\_index}"

        # Добавляею информацию об отладке в текст комментария

        full\_comment\_text = f"{comment\_text} [Debug ID: {debug\_id}]"

        # Создаю здесь комментарий с использованием метода add\_comment

        try:

            # Получаем все runs в параграфе

            runs = target\_para.runs

            if runs:

                comment = doc.add\_comment(runs, full\_comment\_text, author)

                debug\_info.append(f"Комментарий добавлен: {debug\_id} к параграфу {paragraph\_index} через существующие runs")

            else:

                # Пришлось сделать так: если runs нет, создаем новый run

                run = target\_para.add\_run(" ")

                comment = doc.add\_comment(run, full\_comment\_text, author)

                debug\_info.append(f"Комментарий добавлен: {debug\_id} к параграфу {paragraph\_index} через новый run")

        except Exception as e:

            # Если что-то пошло не так, добавляем информацию об ошибке в отладочный отчет

            debug\_info.append(f"Ошибка при добавлении комментария {debug\_id}: {str(e)}")

    doc.save(output\_path)

    # Для создания отладочного файл

    debug\_path = output\_path + '.debug.txt'

    with open(debug\_path, 'w', encoding='utf-8') as debug\_file:

        debug\_file.write('\n'.join(debug\_info))

    return output\_path

Код модуля идентификации и проверки элементов в документе с учетом наследования стилей *formatting\_utils.py* :

"""

Утилиты для работы с форматированием документов DOCX.

"""

from docx.shared import Pt, Cm, RGBColor

from docx.enum.text import WD\_ALIGN\_PARAGRAPH

def \_get\_style\_attr(style\_obj, attr\_path):

    """Вспомогательная функция для безопасного получения атрибута из цепочки стилей."""

    current\_style = style\_obj

    while current\_style:

        obj = current\_style

        found = True

        for attr\_name in attr\_path.split('.'):

            if hasattr(obj, attr\_name):

                obj = getattr(obj, attr\_name)

                if obj is None and attr\_name != attr\_path.split('.')[-1]:

                    found = False

                    break

            else:

                found = False

                break

        if found and obj is not None:

            return obj

        current\_style = current\_style.base\_style if hasattr(current\_style, 'base\_style') else None

    return None

def get\_effective\_first\_line\_indent\_obj(para):

    """

    Функция, чтобы получать отступ первой строки с учетом наследования стилей.

    """

    # 1. Прямое форматирование параграфа

    if para.paragraph\_format and para.paragraph\_format.first\_line\_indent is not None:

        return para.paragraph\_format.first\_line\_indent

    # 2. Стиль параграфа и его базовые стили

    if para.style:

        return \_get\_style\_attr(para.style, 'paragraph\_format.first\_line\_indent')

    return None

def get\_first\_line\_indent\_cm(para):

    """

    Получает отступ первой строки в сантиметрах, с учетом стилей.

    """

    indent\_obj = get\_effective\_first\_line\_indent\_obj(para)

    if indent\_obj and hasattr(indent\_obj, 'cm'):

        try:

            return indent\_obj.cm

        except:

            if hasattr(indent\_obj, 'pt') and indent\_obj.pt is not None:

                return indent\_obj.pt \* 0.0352778

            return 0.0

    return 0.0

def get\_effective\_alignment(para):

    """

    Получает "эффективное" (т.е. видимое пользователю в Word самом) выравнивание с учетом наследования стилей.

    """

    # 1. Прямое форматирование параграфа

    if para.paragraph\_format and para.paragraph\_format.alignment is not None:

        return para.paragraph\_format.alignment

    # 2. Стиль параграфа и его базовые стили

    if para.style:

        alignment = \_get\_style\_attr(para.style, 'paragraph\_format.alignment')

        if alignment is not None:

            return alignment

    return WD\_ALIGN\_PARAGRAPH.LEFT

def get\_run\_font\_name(run, para\_style=None):

    """Решил добавить функцию получения имени шрифта для фрагмента текста."""

    if run.font.name:

        return run.font.name

    if run.style and run.style.font and run.style.font.name:

        return run.style.font.name

    if para\_style:

        font\_name = \_get\_style\_attr(para\_style, 'font.name')

        if font\_name:

            return font\_name

    return None

def get\_run\_font\_size\_pt(run, para\_style=None):

    """Получает размер шрифта для run в пунктах, учитывая стили."""

    if run.font.size is not None and hasattr(run.font.size, 'pt'):

        return run.font.size.pt

    if run.style and run.style.font and run.style.font.size is not None and hasattr(run.style.font.size, 'pt'):

        return run.style.font.size.pt

    if para\_style:

        size = \_get\_style\_attr(para\_style, 'font.size')

        if size is not None and hasattr(size, 'pt'):

            return size.pt

    return None

def get\_run\_font\_color\_rgb(run, para\_style=None):

    """Здесь получаю цвет шрифта для проверки черного текста."""

    if run.font.color and run.font.color.rgb is not None:

        return run.font.color.rgb

    if run.style and run.style.font and run.style.font.color and run.style.font.color.rgb is not None:

        return run.style.font.color.rgb

    if para\_style:

        color = \_get\_style\_attr(para\_style, 'font.color.rgb')

        if color is not None:

            return color

    return None

def get\_run\_bold\_status(run, para\_style=None):

    """тут получаю статус полужирности для run, учитывая стили."""

    if run.bold is not None:

        return run.bold

    if run.style and run.style.font and run.style.font.bold is not None:

        return run.style.font.bold

    if para\_style:

        bold\_status = \_get\_style\_attr(para\_style, 'font.bold')

        if bold\_status is not None:

            return bold\_status

    return None

Код скрипта веб-приложения *app.py*:

#!/usr/bin/env python3.13

# -\*- coding: utf-8 -\*-

"""

Приложение для проверки форматирования документов DOCX и добавления комментариев.

Требуется Python 3.13+ и python-docx 1.2.0+, которые поддерживают API для комментариев.

"""

from flask import Flask, render\_template, request, send\_from\_directory, url\_for, redirect, flash, jsonify

import os

import uuid

import time

import sys

import platform

from datetime import datetime

from werkzeug.utils import secure\_filename

from pathlib import Path

import docx

# Импортируем существующие модули

from formatting\_checker import check\_document\_formatting

from comment\_utils import add\_comments\_to\_docx

# Определяем базовую директорию приложения (для корректной работы абсолютных путей)

BASE\_DIR = os.path.dirname(os.path.abspath(\_\_file\_\_))

app = Flask(\_\_name\_\_)

app.secret\_key = os.urandom(24)

# Настройки для загрузки файлов

UPLOAD\_FOLDER = os.path.join(BASE\_DIR, 'uploads')

ALLOWED\_EXTENSIONS = {'docx'}

app.config['UPLOAD\_FOLDER'] = UPLOAD\_FOLDER

app.config['MAX\_CONTENT\_LENGTH'] = 16 \* 1024 \* 1024  # Ограничение размера файла 16MB

# Создаем директорию для загрузок, если она не существует

os.makedirs(UPLOAD\_FOLDER, exist\_ok=True)

def allowed\_file(filename):

    """Проверяет допустимое расширение файла"""

    return '.' in filename and \

           filename.rsplit('.', 1)[1].lower() in ALLOWED\_EXTENSIONS

def get\_document\_stats(file\_path):

    """Получает статистику документа"""

    try:

        doc = docx.Document(file\_path)

        stats = {

            'paragraphs': len(doc.paragraphs),

            'tables': len(doc.tables),

            'sections': len(doc.sections),

            'pages': estimate\_pages(doc)

        }

        return stats

    except Exception as e:

        print(f"Ошибка при получении статистики документа: {e}")

        return None

def estimate\_pages(doc):

    """Здесь описана примерная оценка количества страниц в документе"""

    total\_chars = sum(len(p.text) for p in doc.paragraphs)

    # Добавляем символы из таблиц

    for table in doc.tables:

        for row in table.rows:

            for cell in row.cells:

                total\_chars += sum(len(p.text) for p in cell.paragraphs)

    # Грубая эмпирическая оценка: ~3000 символов на страницу

    estimated\_pages = max(1, round(total\_chars / 3000))

    return estimated\_pages

@app.route('/')

def index():

    """Главная страница с формой загрузки"""

    return render\_template('index.html')

@app.route('/upload', methods=['POST'])

def upload():

    """Обработка загрузки файла"""

    if 'docx\_file' not in request.files:

        flash('Файл не выбран')

        return redirect(url\_for('index'))

    file = request.files['docx\_file']

    if file.filename == '':

        flash('Файл не выбран')

        return redirect(url\_for('index'))

    if file and allowed\_file(file.filename):

        # Безопасное сохранение файла

        filename = secure\_filename(file.filename)

        unique\_id = str(uuid.uuid4())

        temp\_filename = f"{unique\_id}\_{filename}"

        file\_path = os.path.join(app.config['UPLOAD\_FOLDER'], temp\_filename)

        file.save(file\_path)

        author = request.form.get('author', 'Norm Control')

        output\_prefix = request.form.get('output\_prefix', '\_with\_remarks')

        try:

            document\_stats = get\_document\_stats(file\_path)

            comments = check\_document\_formatting(file\_path, author)

            # Если есть комментарии, добавляем их в документ

            if comments:

                base\_name = Path(filename).stem

                output\_filename = f"{base\_name}{output\_prefix}.docx"

                output\_path = os.path.join(app.config['UPLOAD\_FOLDER'], output\_filename)

                result\_file = add\_comments\_to\_docx(file\_path, output\_path, comments)

                return render\_template('result.html',

                                      filename=output\_filename,

                                      comment\_count=len(comments),

                                      document\_stats=document\_stats)

            else:

                return render\_template('result.html',

                                      no\_comments=True,

                                      filename=filename,

                                      document\_stats=document\_stats)

        except Exception as e:

            flash(f"Ошибка при обработке файла: {e}")

            return redirect(url\_for('index'))

    else:

        flash('Разрешены только файлы с расширением .docx')

        return redirect(url\_for('index'))

@app.route('/download/<filename>')

def download(filename):

    """Скачивание обработанного файла"""

    return send\_from\_directory(app.config['UPLOAD\_FOLDER'], filename)

@app.route('/debug/paths')

def debug\_paths():

    """Добавил этот маршрут для диагностики проблем на хостинге"""

    paths\_info = {

        "BASE\_DIR": BASE\_DIR,

        "UPLOAD\_FOLDER": UPLOAD\_FOLDER,

        "upload\_folder\_exists": os.path.exists(UPLOAD\_FOLDER),

        "upload\_folder\_is\_dir": os.path.isdir(UPLOAD\_FOLDER),

        "upload\_folder\_writable": os.access(UPLOAD\_FOLDER, os.W\_OK),

        "uploaded\_files": os.listdir(UPLOAD\_FOLDER) if os.path.exists(UPLOAD\_FOLDER) else []

    }

    return jsonify(paths\_info)

@app.route('/debug/test-upload', methods=['GET', 'POST'])

def test\_upload():

    """Тестовый маршрут для проверки загрузки файлов"""

    if request.method == 'GET':

        return '''

        <!doctype html>

        <title>Тест загрузки файлов</title>

        <h1>Загрузите тестовый файл</h1>

        <form method=post enctype=multipart/form-data>

          <input type=file name=file>

          <input type=submit value=Загрузить>

        </form>

        '''

    if 'file' not in request.files:

        return 'Файл не выбран'

    file = request.files['file']

    if file.filename == '':

        return 'Файл не выбран'

    filename = secure\_filename(file.filename)

    file\_path = os.path.join(app.config['UPLOAD\_FOLDER'], filename)

    file.save(file\_path)

    return f'''

    <h1>Файл успешно загружен</h1>

    <p>Путь: {file\_path}</p>

    <p>Размер: {os.path.getsize(file\_path)} байт</p>

    <p>Скачать: <a href="/download/{filename}">{filename}</a></p>

    '''

@app.route('/debug')

def debug():

    """Страница диагностики приложения"""

    # Информация о системе

    try:

        import flask

        flask\_version = flask.\_\_version\_\_

    except ImportError:

        flask\_version = "Не установлен"

    system\_info = {

        "working\_directory": os.getcwd(),

        "upload\_folder": UPLOAD\_FOLDER,

        "python\_version": platform.python\_version(),

        "flask\_version": flask\_version

    }

    # Проверка директории загрузок

    upload\_exists = os.path.exists(UPLOAD\_FOLDER)

    upload\_writable = os.access(UPLOAD\_FOLDER, os.W\_OK) if upload\_exists else False

    files = []

    file\_count = 0

    if upload\_exists:

        try:

            file\_list = os.listdir(UPLOAD\_FOLDER)

            file\_count = len(file\_list)

            for file\_name in file\_list[:10]:  # Показываем только первые 10 файлов

                file\_path = os.path.join(UPLOAD\_FOLDER, file\_name)

                if os.path.isfile(file\_path):

                    size\_kb = round(os.path.getsize(file\_path) / 1024, 2)

                    mod\_time = datetime.fromtimestamp(os.path.getmtime(file\_path)).strftime('%Y-%m-%d %H:%M:%S')

                    files.append({"name": file\_name, "size": size\_kb, "modified": mod\_time})

        except Exception as e:

            files = [{"name": f"Ошибка чтения директории: {str(e)}", "size": 0, "modified": ""}]

    upload\_check = {

        "exists": upload\_exists,

        "writable": upload\_writable,

        "file\_count": file\_count,

        "files": files

    }

    # Проверка наличия модулей

    modules = [

        {"name": "formatting\_checker.py", "exists": os.path.exists(os.path.join(BASE\_DIR, "formatting\_checker.py"))},

        {"name": "comment\_utils.py", "exists": os.path.exists(os.path.join(BASE\_DIR, "comment\_utils.py"))}

    ]

    return render\_template('debug.html',

                          system\_info=system\_info,

                          upload\_check=upload\_check,

                          modules\_check=modules)

def cleanup\_old\_files(max\_age\_hours=24):

    """Сделал функцию для очистки старых файлов, чтобы не захламлять сервер"""

    current\_time = time.time()

    for filename in os.listdir(app.config['UPLOAD\_FOLDER']):

        file\_path = os.path.join(app.config['UPLOAD\_FOLDER'], filename)

        if os.path.isfile(file\_path) and os.path.exists(file\_path):

            file\_age\_hours = (current\_time - os.path.getmtime(file\_path)) / 3600

            if file\_age\_hours > max\_age\_hours:

                try:

                    os.remove(file\_path)

                except Exception:

                    pass

if \_\_name\_\_ == '\_\_main\_\_':

    # При запуске приложения очищаем старые файлы

    cleanup\_old\_files()

    # Для локального запуска используем debug=True

    app.run(debug=True)

# На PythonAnywhere будет использоваться application = app

application = app

Код модуля *formatting\_checker.py* содержит основную логику системы: функции-детекторы для определения типов элементов, функции-валидаторы для их проверки на соответствие правилам и главную функцию-диспетчер, управляющую всем процессом. Сокращен для удобства чтения:

from docx import Document

from docx.shared import Pt, Cm, RGBColor

from docx.enum.text import WD\_ALIGN\_PARAGRAPH

import re

from formatting\_utils import (

    get\_first\_line\_indent\_cm,

    get\_effective\_alignment,

    # ... и другие импорты из formatting\_utils

)

# --- Константы ---

STRUCTURAL\_HEADINGS\_KEYWORDS = ["ВВЕДЕНИЕ", "ЗАКЛЮЧЕНИЕ", "СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ", ...]

HEADING\_1\_STYLE\_NAMES = ["heading 1", "заголовок 1", ...]

HEADING\_2\_STYLE\_NAMES = ["heading 2", "заголовок 2", ...]

# --- Функции-детекторы (`is\_...`) ---

def check\_all\_runs\_are\_bold(para):

    """Проверяет, что весь текст в абзаце является полужирным."""

    # ... (реализация, итерирующая по para.runs и проверяющая get\_run\_bold\_status) ...

    return True # или False

def is\_structural\_heading\_type(para):

    """Определяет, является ли абзац структурным заголовком (ВВЕДЕНИЕ, и т.д.)."""

    return para.text.strip().upper() in STRUCTURAL\_HEADINGS\_KEYWORDS

def is\_section\_heading(para):

    """Определяет заголовок раздела (1. Заголовок)."""

    style\_name = get\_paragraph\_style\_name(para)

    if any(h\_style in style\_name for h\_style in HEADING\_1\_STYLE\_NAMES):

        return True

    # Проверка по паттерну "N. Текст" + жирность

    if re.fullmatch(r"\d{1,2}\.\s+.+", para.text.strip()):

        if check\_all\_runs\_are\_bold(para):

            return True

    return False

def is\_subsection\_heading(para):

    """Определяет заголовок подраздела (1.1 Заголовок)."""

    style\_name = get\_paragraph\_style\_name(para)

    if any(h\_style in style\_name for h\_style in HEADING\_2\_STYLE\_NAMES):

        return True

    # Проверка по паттерну "N.M Текст" (без точки) + жирность

    if re.fullmatch(r"\d+(\.\d+)+\s+.+", para.text.strip()):

         if check\_all\_runs\_are\_bold(para):

            return True

    return False

# ... (пропущены другие функции-детекторы: is\_figure\_caption, is\_table\_title и т.д.) ...

# --- Функции-валидаторы (`check\_...\_format`) ---

def check\_structural\_or\_appendix\_heading\_format(para, para\_idx, comments\_list, author, element\_name):

    """Проверяет форматирование структурных заголовков и приложений."""

    # Правила: 14 пт, Times New Roman, черный, полужирный, по центру, без отступа.

    # Пример проверки выравнивания:

    alignment = get\_effective\_alignment(para)

    if alignment != WD\_ALIGN\_PARAGRAPH.CENTER:

        comments\_list.append((para\_idx, f"Ошибка ({element\_name}): Выравнивание должно быть по центру.", author))

    # Пример проверки отступа:

    first\_line\_indent\_cm = get\_first\_line\_indent\_cm(para)

    if abs(first\_line\_indent\_cm) > 0.01:

        comments\_list.append((para\_idx, f"Ошибка ({element\_name}): Не должно быть отступа первой строки.", author))

    # ... (проверка шрифта, размера, цвета, жирности) ...

def check\_section\_heading\_format(para, para\_idx, comments\_list, author):

    """Проверяет форматирование заголовка раздела."""

    element\_name = "Заголовок раздела"

    # Правила: 14 пт, TNR, черный, полужирный, по левому краю, отступ 1.25 см.

    # ... (проверки шрифта, выравнивания, отступа) ...

    # Пример проверки формата номера:

    if not re.match(r"^\d{1,2}\.\s+", para.text.strip()):

        comments\_list.append((para\_idx, f"Ошибка ({element\_name}): После номера раздела должна быть точка.", author))

# ... (пропущены другие функции проверки: check\_subsection\_heading\_format и т.д.) ...

# --- Основная функция-диспетчер ---

def check\_document\_formatting\_final(doc\_path, author="Norm Control"):

    """

    Главная функция, управляющая процессом проверки документа.

    """

    try:

        doc = Document(doc\_path)

    except Exception as e:

        return [(-1, f"Критическая ошибка: Не удалось открыть документ: {e}", author)]

    comments\_to\_add = []

    # 1. Глобальная проверка полей страницы (вынесена отдельно)

    # check\_page\_margins(doc.sections[0], comments\_to\_add, author)

    # 2. Итерация по абзацам с применением каскадного метода

    processing\_active = False

    all\_doc\_paragraphs = list(doc.paragraphs)

    for i, para in enumerate(all\_doc\_paragraphs):

        if not para.text.strip():

            continue

        # Активация обработки после заголовка "ВВЕДЕНИЕ"

        if not processing\_active and para.text.strip().upper() == "ВВЕДЕНИЕ":

            processing\_active = True

        if not processing\_active:

            continue

        # 3. Каскадное определение типа элемента

        # Порядок важен: от более специфичных к более общим

        para\_text\_upper\_stripped = para.text.strip().upper()

        if is\_appendix\_heading\_type(para\_text\_upper\_stripped):

            check\_structural\_or\_appendix\_heading\_format(para, i, comments\_to\_add, author, "Заголовок приложения")

        elif is\_structural\_heading\_type(para\_text\_upper\_stripped):

            check\_structural\_or\_appendix\_heading\_format(para, i, comments\_to\_add, author, "Структурный заголовок")

        elif is\_section\_heading(para):

            check\_section\_heading\_format(para, i, comments\_to\_add, author)

        elif is\_subsection\_heading(para):

            check\_subsection\_heading\_format(para, i, comments\_to\_add, author)

        # ... (здесь следуют вызовы is\_figure\_caption, is\_table\_title и т.д.) ...

        else:

            # check\_main\_text\_format(para, i, comments\_to\_add, author)

            pass # Другие типы элементов обрабатываются далее

    # 4. Вызов функций для общих проверок всего документа

    # ... (например, проверка сносок, согласованности ссылок) ...

    return comments\_to\_add