 Министерство образования и науки Российской Федерации

**Федеральное государственное бюджетное образовательное учреждение**

**высшего образования**

**«Московский государственный технический университет**

**имени Н.Э. Баумана**

**(национальный исследовательский университет)»**

**(МГТУ им. Н.Э. Баумана)**

ФАКУЛЬТЕТ \_\_\_\_\_\_Информатика и системы управления\_\_\_\_\_\_

КАФЕДРА \_\_\_Программное обеспечение ЭВМ и информационные технологии\_\_\_

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

К ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЕ

НА ТЕМУ:

«Метод обработки тестовых артефактов тестирования индикации с использованием регулярных выражений»

Студент ИУ7-83 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ С.А. Кукуев

(Подпись, дата)

Руководитель ВКР \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ А.А. Павельев

(Подпись, дата)

Нормоконтролер \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Ю.В. Строганов

(Подпись, дата)

2017 г.

# РЕФЕРАТ

Расчетно-пояснительная записка 67 с., 4ч., 28 рис., 3 табл., 11 источников.

ТЕСТИРОВАНИЕ БОРТОВОГО АВИАЦИОННОГО ОБОРУДОВАНИЯ, ОБРАБОТКА ТЕСТОВЫХ АРТЕФАКТОВ, МЕТОД ОБРАБОТКИ С ИСПОЛЬЗОВАНИЕМ РЕГУЛЯРНЫХ ВЫРАЖЕНИЙ

Объектом исследования является задача обработки тестовых артефактов, передаваемых на тестовый автомат для тестирования корректного отображения многофункционального индикатора бортового авиационного оборудования. Объектом разработки является метод обработки тестовых артефактов тестирования индикации с использованием регулярных выражений.

Цель работы – разработка метода обработки тестовых артефактов тестирования индикации с использованием регулярных выражений. Задачи, решаемые в работе:

* Анализ предметной области для решения поставленной задачи;
* Разработка метода обработки тестовых артефактов тестирования индикации с использованием регулярных выражений;
* Проектирование, реализация и тестирование программного обеспечения, реализующего разработанный метод;
* Экспериментальное исследование разработанного метода.

Область применения – тестирование бортового авиационного оборудование на специально разработанном тестовом автомате.

В первой части работы дается описание предметной области, а также анализируются побочные алгоритмы и методы, необходимые для решения поставленной задачи. Во второй части описываются основные параметры тестовых артефактов и этапы работы алгоритма. В третьей части описываются технологии, примененные при реализации алгоритма, основные моменты реализации и разработанные структуры данных. В четвертой части проведены эксперименты для проверки корректности работы метода.

Предлагаемые направления развития:

* Добавление обработки тестовых артефактов с линией передачи Arinc;
* Распараллеливание реализованного алгоритма;
* Оптимизация алгоритма чтения тестовых артефактов при помощи использования преобразования в xml-классы.

СОДЕРЖАНИЕ

[РЕФЕРАТ 2](#_Toc484782798)

[ВВЕДЕНИЕ 6](#_Toc484782799)

[1. Аналитический раздел 8](#_Toc484782800)

[1.1 Описание предметной области 8](#_Toc484782801)

[1.2 Формализация постановки задачи 12](#_Toc484782802)

[1.3 Методы доступа к данным 13](#_Toc484782803)

[*1.3.1* Open Database Connectivity (ODBC) 13](#_Toc484782804)

[*1.3.2* Object Linking and Embedding (OLE DB) 15](#_Toc484782805)

[*1.3.3* Microsoft ActiveX Data Objects (ADO) 18](#_Toc484782806)

[1.4 Метод обработки данных 18](#_Toc484782807)

[1.5 Алгоритмы поиска информации 21](#_Toc484782808)

[*1.5.1* Алгоритм Кнута-Морриса-Пратта 21](#_Toc484782809)

[*1.5.2* Алгоритм Бойера-Мура 22](#_Toc484782810)

[*1.5.3* Двоичный алгоритм поиска подстроки 24](#_Toc484782811)

[Выводы 26](#_Toc484782812)

[2 Конструкторский раздел 27](#_Toc484782813)

[2.1 Основные параметры тестовых артефактов 27](#_Toc484782814)

[2.2 Этапы работы алгоритма 29](#_Toc484782815)

[Выводы 33](#_Toc484782816)

[3 Технологический раздел 34](#_Toc484782817)

[3.1 Выбор средств разработки 34](#_Toc484782818)

[3.2 Основные моменты программной реализации 35](#_Toc484782819)

[3.3 Требования к запуску ПО 35](#_Toc484782820)

[3.4 Форматы входных и выходных данных 36](#_Toc484782821)

[3.5 Структура разработанного ПО 39](#_Toc484782822)

[3.6 Тестирование 44](#_Toc484782823)

[3.7 Интерфейс пользователя 45](#_Toc484782824)

[Выводы 46](#_Toc484782825)

[4 Экспериментальный раздел 47](#_Toc484782826)

[4.1 Критерии оценки 47](#_Toc484782827)

[4.2 Оценка времени работы алгоритмов 48](#_Toc484782828)

[*4.2.1* Чтение протоколов 48](#_Toc484782829)

[*4.2.2* Обработка протоколов 50](#_Toc484782830)

[Выводы 52](#_Toc484782831)

[ЗАКЛЮЧЕНИЕ 53](#_Toc484782832)

[СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 54](#_Toc484782833)

[ПРИЛОЖЕНИЕ А 56](#_Toc484782834)

[ПРИЛОЖЕНИЕ Б 60](#_Toc484782835)

[ПРИЛОЖЕНИЕ В 64](#_Toc484782836)

# ВВЕДЕНИЕ

Бортовое радиоэлектронное оборудование (БРЭО) [1] современного боевого летательного аппарата (ЛА) выполняет огромное количество многочисленных задач по обеспечению пилотирования, навигации, коммуникации и безопасности. С каждым годом, все больше и больше БРЭО на борту воздушных судн оснащаются встроенными вычислительными системами, начиная от поисковых прожекторов и заканчивая современными радарами. Объем программного обеспечения таких систем постоянно увеличивается и усложняется, в следствии чего растет потребность в изменениях и доработках.

Каждые доработки нуждаются в тщательном тестировании, поскольку любая неисправность БРЭО при испытаниях может повлечь за собой гибель летчика-испытателя. Затраты на корректировку таких систем могут достигать нескольких лет, что существенно замедляет процесс внедрения в эксплуатацию, поэтому поднимаются вопросы частичной автоматизации проверки работы оборудования. Создаются различные программные обеспечения и разрабатываются новые технологии тестирования, которые значительно сокращают время и минимизируют человеческий труд.

В связи с этим, актуальным является задача автоматизации методов и процессов верификации программного обеспечения. Полная ручная проверка достоверности системных комплексов для ЛА слишком трудоемкий процесс, который требует огромное количество человеческого ресурса и времени. Внедрение вспомогательных инструментальных средств позволит его сэкономить и частично исключить человеческий фактор, что приведет к резкому сокращению ошибок.

Целью работы является разработка и реализация унифицированного программного обеспечения для обработки тестовых артефактов тестирования индикации. В рамках работы решаются следующие задачи:

* анализ задачи обработки информации из тестовых артефактов;
* тестирование программного обеспечения;
* анализ методов интеграции в комплекс программного обеспечения тестового автомата;

Разработка данного программного обеспечения позволит частично автоматизировать процесс подготовки тестовых артефактов для тестового автомата, что даст возможность уделять больше времени самому тестированию комплексов БРЭО.

# Аналитический раздел

В этом разделе производится анализ предметной области, рассматриваются различные подходы к обработке тестовых артефактов и побочные алгоритмы, необходимые для работы с тестовыми артефактами. Также в этом разделе производится формализация поставленной задачи.

## Описание предметной области

Основным источником информации для летчика современного боевого ЛА являются многофункциональные индикаторы (МФИ) [1], оснащенные встроенной вычислительной системой. *МФИ* – средство отображения различной графической или телеинформации, поступающей от бортовых информационных систем, о количественном или качественном значении параметров. Для внутренней автоматизации тестирования функционального программного обеспечения индикаторов были разработаны специальный язык тестового сценария и тестовый автомат.

*Test Script Language (TSL)* [2] – инструмент написания тестовых сценариев, призванный облегчить написание скриптов и уменьшить вероятность появления ошибок при написании тестовых процедур. Отличительной особенностью является возможность использовать основные конструкции языка С++: функции, циклы, структуры и массивы. Основные команды сценария:

* ITEM\_EX – начало тестового сценария.
* END\_ITEM\_EX – окончание тестового сценария.
* SET – установка значения сигнала.
* CHECK – запрос у оператора результата сравнения ожидаемого и реального результата.
* DISPLAY – вывод в окно тестового автомата информации для оператора.
* SPEC – описание проверяемых требований в тесте.
* DisplaySpecMessage – описание и отображение данных спецификаций с созданием матрицы трассировки проверяемых требований в отчете.
* SetAndDisplay – задание значения сигналов вместе с описанием заданных значений в инструкции оператору.
* DoTesting – выполнение тестового объекта.

Основным элементом тестового сценария, служащим проверить достоверность отображения информации, является тестовый объект. *Тестовый объект* [2]– структура, заполненная ожидаемыми значениями отображения в соответствии со спецификацией требований программного обеспечения (СТПО). Структура содержит наименование теста, количество принимаемых индикатором параметров, их тестовые результаты и значения. На рисунке 1.1 приведен пример заполнения тестового объекта. Окончание ввода тестовых результатов и значений характеризуются соответствующими терминаторами *WTR\_TERM* и *OTD\_TERM*.

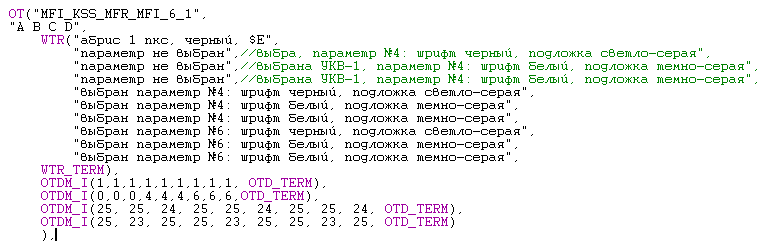
****

Рис. 1.1 – Пример заполнения тестового объекта.

*Тестовый автомат («Тестер»)* [2]– утилита, позволяющая проводить тестирование МФИ в соответствии со сценарием. Компьютер, оснащенный тестовым автоматом, подключается через штатные каналы информационного взаимодействия к многофункциональным индикаторам. В соответствии со сценарием проверки, программа подает на вход индикатора набор сигналов для каждого теста, и МФИ производит отображение информации. Тестировщик сравнивает ожидаемый результат с полученным и принимает решение о ее корректности. По окончанию всех тестов программное обеспечение выдает отчет.

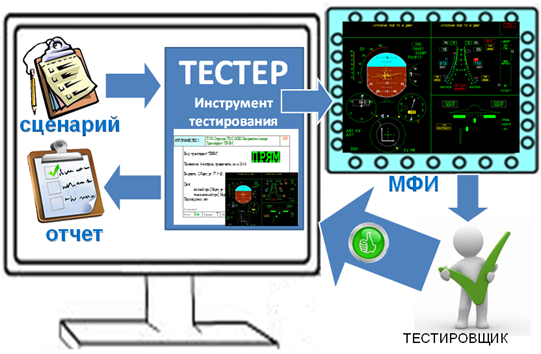


Рис. 1.2 – Работа тестового автомата («Тестера»).

Для корректного отображения информации на МФИ «Тестер» должен содержать в себе подробную информацию о существующих сигналах индикатора, чтобы при поступлении набора от тестового сценария изменить соответствующие сигналы на индикаторе. Данные характера содержат протоколы информационного взаимодействия (ПИВ). ПИВ – файл, формата Microsoft Office Excel, содержащий таблицы параметров информационного взаимодействия, выполненные в формате специальной электронной формы (СЭФ). Обработанные протоколы являются основными тестовыми артефактами.

Протокол информационного взаимодействия содержит заголовок в виде таких параметров, как:

* Источник – устройство, передающее сигналы;
* Режим – способ передачи данных (симплексный, полудуплексный, дуплексный, мультиплексный);
* Приемник – устройство, принимающее сигналы;
* Линия – соединение отдельных функциональных блоков между собой. Состав и количество линий определяет возможные способы передачи данных.

Таблица содержит в себе необходимые характеристики сигналов, от наименования сигнала до номера набора параметров, в котором он передается. Для дополнительной информации присутствует поле «Примечание», в котором содержатся различные пояснения к сигналу, вплоть до описания логики. Правила составления ПИВ в формате СЭФ определены стандартом предприятия [2] (СТО 730.0162-2012).

Подготовка протоколов информационного взаимодействия для тестового автомата производится в несколько этапов:

* Проверка оформления ПИВ (на соответствие стандартам предприятия);
* Оперативная корректировка параметров сигналов, если имеется несоответствие данных;
* Преобразование информации о каждом сигнале в формат, необходимый тестовому автомату, в виде текстового документа (\*.txt).

Для работы с информацией о сигналах, тестовому автомату необходимо принимать каждый сигнал отдельно. Поэтому протоколы информационного взаимодействия в виде таблицы преобразовываются в формат текстового документа, где каждый сигнал имеет специальную структуру. В дальнейшем тестовый автомат преобразовывает сигналы в двоичный код.

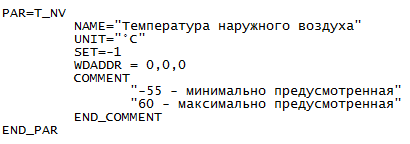


Рис. 1.3 – Формат сигнала, требуемый тестовым автоматом.

На рисунке 1.3 приведен формата сигнала, необходимый тестовому автомату, где *PAR, END\_PAR* – начало и окончание блока параметра, содержащего всю доступную информацию о сигнале, *NAME* – наименование сигнала, *UNIT* - размерность, *SET* – номер подкадра (-1, если в кадре отсутствуют подкадры), *WADDR* – адрес параметра в наборе, *COMMENT, END\_COMMENT* – начало и окончание блока примечания.

Данные этапы, на сегодняшний день, производятся вручную. На работу с одним протоколом уходит порядка двух месяцев у одного сотрудника, поэтому был поставлен вопрос о создании программного обеспечения для улучшения условий работы с протоколами информационного взаимодействия.

## Формализация постановки задачи

Основываясь на проведенном анализе предметной области, постановка решаемой в данной работе задачи может быть сформулирована следующим образом.

Необходимо реализовать программное обеспечение для работы с протоколами информационного взаимодействия. Входными данными являются файлы формата Microsoft Excel, содержащие в себе таблицы значений. Данные таблицы должны быть проверены на соответствие стандартам предприятия. При несоответствии параметров по окончании проверки выдается отчет. Выходными данными являются текстовые документы, содержащие в себе информацию о сигналах в формате, требуемом тестовым автоматом.

Также необходимо рассмотреть методы интеграции данного ПО в программный комплекс отдела тестирования.

## Методы доступа к данным

Таблицы протоколов информационного взаимодействия содержатся в файлах Microsoft Excel. Для работы с данными у компании Microsoft имеется несколько собственных API. В данном подразделе будет выполнен обзор этих программных интерфейсов, с целью последующего выбора оптимального подхода.

### Open Database Connectivity (ODBC)

Стандарт открытого доступа к базам данных является типовым программным интерфейсом для построения приложений управления БД в среде Microsoft Windows. Основная идея состоит в использовании специализированных драйверов баз данных подобно тому, как Windows применяет драйвера для работы различных устройств. Разработчики могут создавать свои собственные драйвера ODBC или же использовать разработанные другими фирмами. Список баз данных, для которых Microsoft уже разработало специализированные драйвера, можно посмотреть в[3].

Технология ODBC позволяет разработчику создавать и распространять клиент-серверные программы, не зависящие от конкретной системы управления базами данных. Всю привязку к определенной СУБД выполняет диспетчер драйверов. *Диспетчер драйверов* [4]– библиотека динамической загрузки, которая осуществляет доступ к БД. Основной функцией диспетчера является загрузка необходимого драйвера. Также он выполняет обработку некоторых инициализированных и информационных вызовов ODBC, передачу вызовов функций ODBC от приложения драйверу и контроль состояния. При необходимости, диспетчер может регистрировать в журнале все вызовы функций ODBC приложением. В журнале записывается имя каждой свободной от ошибок функции вместе со значениями входных аргументов и именами выходных. Прежде чем передать вызов драйверу, отвечающем за подключение к базе данных, диспетчер проверяет аргументы функций и корректность изменения состояния, а также другие условия отсутствия ошибок. Это дает возможность освободить БД от обработки большинства ошибок.

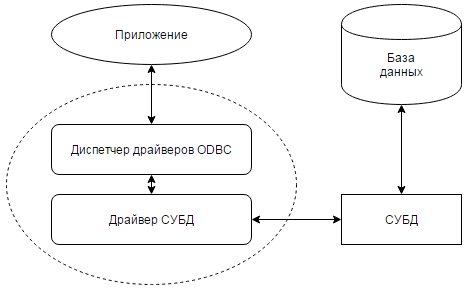


Рис. 1.4 – Архитектура ODBC.

Для установления соединения и управления драйвером СУБД, диспетчеру требуется информация о его местонахождении и типе драйвера. Данную информацию предоставляет именованный ресурс ODBC – DSN (Data Source Name). Существуют три типа именованного ресурса DSN: пользовательский, системный и файловый. Различия заключается в методе хранения требуемой информации и ее доступе. Информация для пользовательского DSN содержится в реестре Windows того компьютера, на котором создано соединение с данными. Системные DNS относятся к компьютеру, а не к пользователю. Таким образом, система или пользователь, обладающий соответствующими привилегиями, необходимыми для доступа к источнику данных, имеют право работать с БД. Информация для файлового DNS хранится в файле, доступном для чтения любому приложению.

Основная задача драйвера СУБД обрабатывать запросы, поступающие от приложения и преобразовывать их в набор команд API СУБД и, таким образом, производить какие-либо действия с базой данных. Также он отвечает за то, чтобы стандартные команды ODBC выполнялись корректно, поскольку в некоторых случаях источник данных не поддерживает некоторые функции. В таких ситуациях их выполняет драйвер СУБД. Также на драйвере лежит функция приведения кодов ошибок, поступающих от источника, к стандартным в ODBC.

Определяют два типа драйверов – одноуровневые и многоуровневые. Одноуровневые обрабатывают вызовы ODBC и операторы SQL. Многоуровневые обрабатывают только вызовы ODBC, оставляя право за СУБД осуществлять обработку SQL-запросов.

### Object Linking and Embedding (OLE DB)

Технология OLE DB представляет из себя низкоуровневый интерфейс для доступа к базам данным, разработанный компанией Microsoft. Отличительно особенностью является наличие доступа к нереляционным данным, таким как почтовые сообщения, электронные таблицы, текстовые документы и многие другие. В спецификации OLE DB определен набором COM-интерфейсов (*Component Object Model* [5]– технологический стандарт компании Microsoft, предназначенный для создания программного обеспечения на основе взаимодействующих компонентов), инкапсулирующих различные сервисы управления данными и предоставляющих однотипный доступ.

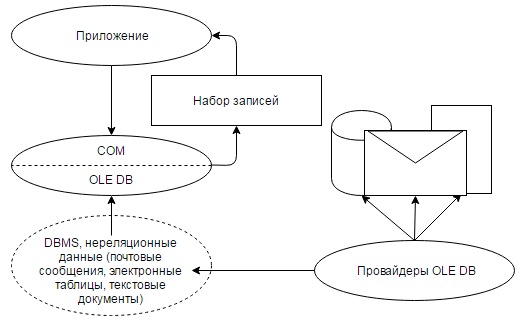


Рис. 1.5 – Архитектура OLE DB.

Рассматривая работу интерфейса OLE DB, выделяют две сущности: потребители и поставщики. В роли потребителей служат приложения, которым необходимо получить данные из долговременного источника. Для доступа нужно выбрать поставщика, соответствующего типу базы данных, к которой требуется доступ, и использовать базовые интерфейсы для взаимодействия, описанные в [6].

*Поставщик (провайдер)* – набор COM-объектов, обслуживающих вызовы интерфейса из пользовательского объекта. Основная задача заключается в передаче информации в табличном формате из хранилища данных приложению-получателю. Каждый из COM-объектов содержит несколько интерфейсов, которые подразделяются на обязательные и необязательные. С помощью реализации обязательных провайдер гарантирует минимальный уровень функциональности. В OLE DB интерфейсы определяются для нескольких типов объектов:

* Источник данных;
* Сеанс;
* Набор строк;
* Команда;
* Транзакция.

Все объекты, кроме строк и хранилищ, реализуются с помощью шаблонов поставщика. *Шаблоны* – предварительно упакованные реализации для всех необходимых интерфейсов.

Когда приложению-потребителю требуются данные, оно создает объект источника данных с целью запуска поставщика. Он содержит сведения о проверке подлинности авторизации, которые используются для предоставления разрешений потребителю. Используя интерфейс *IDBCreateSession*, объект источника данных создает один или несколько сеансов, с помощью которых потребитель подключается к источнику данных (рисунок 1.6). Каждый сеанс управляет своими взаимодействиями с базой данных независимо от других существующих сеансов и для управления инициализирует объекты команд (или таблиц) и набора строк.



Рис. 1.6 – шаблон поставщика OLE DB.

Наборы строк не предоставляют доступа к хранилищу данных, а лишь содержат информацию о нем – метаданные. При представлении информации используются три вида наборов: одиночный набор, групповой набор и массивный набор. Доступ к строкам осуществляется через класс команд (или таблиц). С помощью него выполняются базовые операции для работы с наборами строк (обновление, удаление, вставка). Запись и извлечение данных происходит через объект метода доступа, который содержит набор привязок между полями набора строк и элементами данных, объявленные в потребителе.

### Microsoft ActiveX Data Objects (ADO)

*Microsoft ActiveX Data Objects (ADO)* – набор программных объектов, построенных по технологиям ActiveX (COM), позволяющий получать данные и управлять ими на самых разных источниках. ADO является высокоуровневым интерфейсом, и для работы с источником использует разные подключения к базам данных, в том числе OLE DB и ODBC.

Объектная модель ADO [7]достаточно проста, и содержит всего три главных объекта:

* Объект *Connection* – позволяет установить соединение с источником данных и управлять им. Ошибки, которые возникают при попытке подключения помещаются в сопутствующую коллекцию *Errors*.
* Объект *Command* – представляет набор команд, с помощью которых производится выполнение определенных операций на хранилище данных (выполнение запроса, создание или изменение объекта, изменение данных и т.п.). Если источник данных является SQL-совместимым, то объект *Command* представляет SQL-команды. Данному объекту сопутствует коллекция *Parameters* – наборы параметров, передающихся запросу или хранимой процедуре.
* Объект *Recordset* – представляет набор строк, полученных с источника или сгенерированный другим способом. Ему сопутствует коллекция *Fields*, которая представляет информацию о столбцах в этом набор строк (имя, тип, размерность и т.п.), а также сами данные.

Для каждого из этих трех объектов предусмотрена коллекция *Properties,* которая определяет соответственно свойства соединения, команды или набора строк.

## Метод обработки данных

Обработка таблиц протоколов информационного взаимодействия основывается на проверке соответствия спецификациям требований к изложению, построению и оформлению. Решение задачи напрямую, а именно посимвольное сравнение строк и дальнейшего разбора на ошибки значений сигналов, является неэффективным. Исходя из этого поставлена задача разработки метода обработки данных. Для его создания требовалось изучить основные положения в спецификациях к построению и оформлению. При анализе была выявлена четкая постановка инструкций по построению таблиц, поэтому основная идея будущего алгоритма лежит в использовании шаблонов.

Конструкции шаблонов строятся на основе отношений – неких универсальных связей, установленных инструкциями или разработчиком. Связи являются бинарными, т.е. переходы осуществляются от одного узла к другому, а в качестве узлов подразумеваются неделимые единицы. Если существуют два различных узла *А* и *В*, и между ними присутствует универсальная бинарная связь, то такая связь является зависимостью. В совокупности, множество таких зависимостей должно покрывать все возможные связи между значениями.

*Базовый шаблон* – установленное правило, определяющее зависимость в анализируемых ячейках таблицы. Конструкция шаблона формируется из последовательности неделимых единиц, слов или знаков. Характеризуется уникальным идентификатором. Для каждого базового шаблона определяется набор альтернативных. *Альтернативный шаблон* (формула 1.1) – установленное правило, противоречащее базовому, которое позволяет определять причину нарушения зависимости. Отличительной особенностью является наличие производимой операции при обнаружении в ячейке таблицы. Кроме того, данные шаблоны обладают приоритетом.

*Приоритет* – формальное свойство, влияющее на очередность расположения конструкций в наборе. При одинаковом приоритете шаблоны располагаются исходя из алфавитной сортировки по идентификатору или усмотрению разработчика. Установка очередности альтернативных конструкций зависит от значимости причины нарушения зависимости.

*Операция* – действие, выполняемое при соответствии конструкции ячейки таблицы одному из шаблонов. Как правило, этим действием является запись ошибки в определенную структуру, с помощью которой потом формируется отчет об обработке тестового артефакта. Если отличия базового шаблона и альтернативного незначительно, то действием возможно автоматическое исправление значения.

Обработка значений таблицы в анализируемом тестовом артефакте строится на основе поиска схожей конструкции в множестве шаблонов. Пусть существует набор значений, множество   
базовых шаблонов. Тогда для каждого отдельного базового шаблона , где определено множество альтернативных шаблонов.

Все значения из набора *P* последовательно сопоставляются с выделенными отдельными базовыми шаблонами. При соответствии конструкций производится переход к следующему значению таблицы. В противном случае происходит последовательный поиск альтернативного шаблона из соответствующего множества . После обнаружения совпадения с альтернативной конструкцией происходит выполнение операции, закрепленной для шаблона . В случае, если совпадений с альтернативным шаблоном из множества не обнаружено, выдается сообщение о присутствии неизвестной ошибки.

Таких совпадений может быть несколько, поэтому сопоставление происходит до тех пор, пока не будут проверены все шаблоны. Для окончания работы всего алгоритма необходимо обработать последовательно все значения таблицы.

## Алгоритмы поиска информации

В тестовых артефактах содержится огромное количество значений, с которыми необходимо работать, поэтому поиск подстроки в строке является одной из важнейших задач. Примитивный алгоритм основан на переборе всех подстрок, длина которых равна длине шаблона поиска, и посимвольном сравнении таких подстрок с шаблонами поиска. Данный способ не является быстродействующим и в лучшем случае будет произведено *n-m+1* сравнений, где *n* - длинна строки, *m* – длинна необходимой подстроки, а в худшем и вовсе снижается скорость до *O(n\*m)*, что не подходит для решения поставленной задачи.

Алгоритмов существует огромное разнообразие, поэтому необходимо отталкиваться исходя от данных, на которых будет производится поиск. После изучения таблиц данных были выявлены следующие особенности:

* Огромное количество строк, отличающихся одним или несколькими символами (плохая выборка данных);
* Размер каждой строки не превышает длину алфавита.

Для рассмотрения, на основе выявленных особенностей таблиц данных, были выбраны алгоритм Кнута-Морриса-Пратта [8], алгоритм Бойера-Мура [10] и двоичный алгоритм поиска подстроки [11].

### Алгоритм Кнута-Морриса-Пратта

Идея алгоритма строится на использовании префикс-функции. Для более понятного объяснения введем определения префикса и суффикса. *Префикс­* – подстрока, начинающаяся с начала строки. *Суффикс* – подстрока, заканчивающаяся в конце строки. Тогда *префикс-функцией* [9]является функцией, которая для каждого элемента строки показывает длину наибольшего префикса строки, исключая вырожденные случаи, где префикс совпадает с этой строкой и одновременно является ее суффиксом.

Рассмотрим сам алгоритм Кнута-Морриса-Пратта [8]. Как и в примитивном алгоритме поиска подстроки в строке, шаблон сдвигается по строке слева направо с целью обнаружения совпадения. Однако ключевым отличием является то, что при помощи вычисленной префикс-функции избегаются бесполезные сдвиги. Пусть нам нужно найти подстроку *t* в строке *S*. Тогда необходимо найти префикс-функцию от строки , где # - символ, гарантировано не встречающийся ни в одной из строк. Если эта префикс-функция содержит значения равные длине *t*, значит *t* входит в *S*, а – последний символ вхождения *t* в *S*.

Время работы алгоритма линейно зависит от объема входных данных, поэтому разработать асимптотически эффективный алгоритм невозможно.

### Алгоритм Бойера-Мура

Алгоритм, разработанный двумя учеными, считается наиболее быстрым среди алгоритмов общего назначения. Отличительной чертой считается предобработка искомой строки, благодаря которой сравнение производится не во всех позициях.

Сравнение шаблона *x* справа налево, начиная с самого правого, один за другим с символами исходной строки *y.* При последовательном совпадении всех символов шаблона с наложенными символами строки, подстрока считается найденной, и поиск заканчивается. В случае несовпадения какого-либо символа он использует две предварительно вычисляемых эвристических функций, чтобы сдвинуть позицию для начала сравнения вправо. Таким образом алгоритм Бойера-Мура [10] для сдвига позиции выбирает между двумя функциями, называемыми эвристиками хорошего суффикса и плохого символа.

При совпадении в шаблоне одного или больше символов с текстом, шаблон сдвигается по правилу хорошего суффикса в зависимости от того, какой суффикс совпал. Различают два варианта сдвига хорошего суффикса.

Если существуют такие подстроки равные u, что они полностью входят в x и идут справа от символов, отличных от , то сдвиг происходит к самой правой из них, отличной от *u*. Понятно, что таким образом мы не пропустим никакую строку, так как сдвиг происходит на следующую слева подстроку *u* от суффикса. После выравнивания шаблона по этой подстроке сравнение шаблона опять начнется с его последнего символа. На новом шаге алгоритма можно строку *u*, по которой был произведён cдвиг, не сравнивать с текстом.

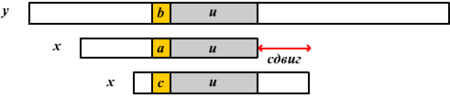


Рис. 1.7 – Сдвиг хорошего суффикса. Встречается вся подстрока.

Если не существует таких подстрок, то смещение состоит в выравнивании самого длинного суффикса *u* подстроки   
с соответствующим префиксом *x*. Поскольку такой подстроки нет, то единственным вариантом остается попадание в эту подстроку префикса.

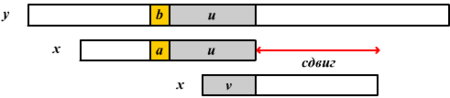


Рис. 1.8 – Сдвиг хорошего суффикса. Суффикс встречается повторно.

В таблице плохих символов указывается последняя позиция в шаблоне (исключая последнюю букву) каждого из символов алфавита. Для всех символов, не вошедших в шаблон, пишем *m*. Предположим, что у нас не совпал символ *c* из текста на очередном шаге с символом из шаблона. Очевидно, что в таком случае мы можем сдвинуть шаблон до первого вхождения этого символа *c* в шаблоне, потому что совпадений других символов точно не может быть. Если в шаблоне такого символа нет, то можно сдвинуть весь шаблон полностью.

Если символ исходного текста встречается в шаблоне *x*, то происходит его выравнивание с его самым правым появлением в подстроке

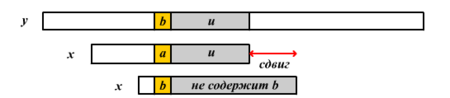


Рис. 1.9 – Сдвиг плохого символа. Символ «а» входит в «x».

Если не встречается в шаблоне x, то ни одно вхождение x в y не может включать в себя и левый конец окна сравнения совмещен с символом, непосредственно идущим после , то есть символ

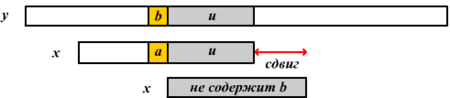


Рис. 1.10 – Сдвиг плохого символа. Символ «b» не входит в «x».

Необходимо учитывать, что сдвиг плохого символа может быть отрицательным, поэтому исходя из ранее приведенных свойств этих функций берется значение равное максимуму между сдвигом хорошего суффикса и сдвигом плохого символа.

### Двоичный алгоритм поиска подстроки

Основная идея алгоритма строится на понимании того, что в современных компьютерах битовый сдвиг и побитовое ИЛИ - атомарные операции. Используя данные операции, можно произвести оптимизацию примитивного алгоритма, благодаря которой за одну операцию производится до 32(64) сравнений одновременно, в зависимости от разрядности машины.

Рассмотрим подробнее работу алгоритма [11] на примере. Пусть *subStr –* искомая строка, а *str –* строка, в которой осуществляется поиск. Тогда *n* и *N* – длины строк соответственно. Построим матрицу *M* размером и заполним ее по следующему принципу: элементтогда и только тогда, когда первые *i* символов *subStr* точно совпадают с *i* символами *str*, заканчивая на позиции *j.* В других случаях элемент равняется нулю. В таком случае, искомая строка считается найденной, если в последней строке матрицы существует хотя бы одна единица. Пример заполнения показан в таблице 1.1.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Str  subStr | T | E | M | P | \_ | N | P | T | E | T |
| T | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 |
| E | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| M | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Таблица 1.1 – Пример работы алгоритма двоичного поиска

Для построения матрицы необходимо ввести понятие двоичного вектора. *Двоичный вектор* – строка, длина которой равна длине искомой подстроки, и в которой стоит единица в тех позициях, где в подстроке стоит буква *X.* Тогда столбцы формируются на основании предыдущего столбца по формуле 1.2:

Где *M[j] – j-*ыйстолбец матрицы *M, j –* индекс столбца*, BitShift() –* функция побитового смещения*, BinVector() –* функция формирования двоичного вектора*, str[j] – j-*ый символ строки, в которой производится поиск*.*

Данный алгоритм является очень гибким и при небольшой модификации приспосабливается под приблизительный поиск. Эффективен с подстроками, длина которых не превышает длину машинного слова.

## Выводы

В аналитическом разделе был проведен анализ предметной области, в ходе которого была сформулирована задача обработки тестовых артефактов, и предложен вариант метода обработки тестовых артефактов с использованием шаблонных конструкций. Также были проанализированы методы поиска и получения информации из тестовых артефактов.

# Конструкторский раздел

Данный раздел содержит описание основных параметров тестовых артефактов для тестового автомата и блок схемы этапов работы основных алгоритмов программного обеспечения.

## Основные параметры тестовых артефактов

* Линия – соединение отдельных функциональных блоков между собой. Состав и количество линий определяет возможные способы передачи данных.
* № слова – индекс слова, в которое будут упакованы сигналы с одинаковым номером словом. Под каждое слово выделяется по 32 используемых разряда, при этом, если набор параметров передается по линии передачи МКИО, то под слово выделяются с 4 по 19 разряд, а остальные остаются в резерве для тестового автомата.
* Наименование сигнала – предложение, характеризующее физический смысл сигнала (пример: Достоверность характеристического времени падения на первом участке).
* Условное обозначение сигнала – идентификатор сигнала для тестового автомата. Должен быть уникальным для каждого параметра.
* Минимальное значение – минимально допустимое значение данного сигнала. Отрицательное значение допускается, только если об этом описано в примечании.
* Максимальное значение – максимально допустимое значение данного сигнала.
* Цена старшего разряда – определяет веса каждого из выделенных разрядов для слова.
* Используемые разряды – диапазон выделенных битов для данного сигнала в слове, в которое он упаковывается.
* Примечание – параметр, в котором описывается логика работы данного сигнала.

## Этапы работы алгоритма

Основные этапы работы алгоритма можно представить на блок схеме, представленной на рисунке 2.1:

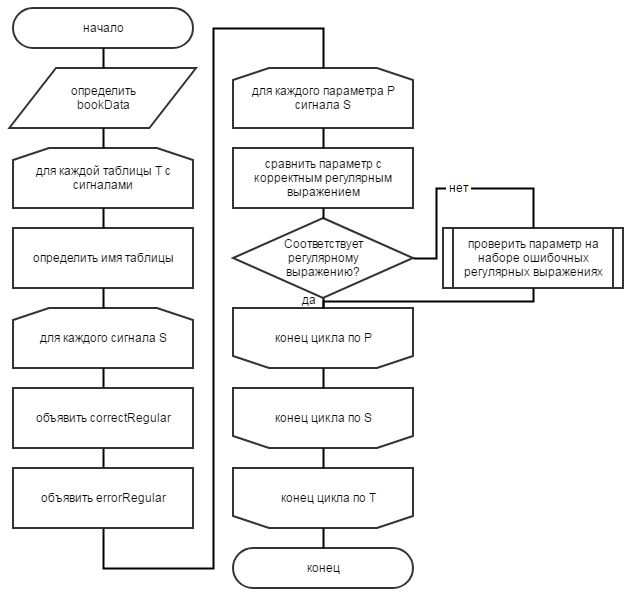


Рисунок 2.1 – Этапы работы разработанного алгоритма.

bookData – структура хранение данных тестового артефакта. Данные в этой структуре определяются в классе ReaderExcel с помощью алгоритма, представленного на рисунке 2.3.

correctRegular – набор регулярных выражений, требуемых для проверки корректных обозначений параметров. Определяется вручную в классе ErrorBase.

errorRegular – структура, содержащая набор регулярных выражений, требуемых для проверки параметров, не прошедших проверку на корректность, и описания ошибки к каждому регулярному выражению. Определяется вручную в классе ErrorBase. При появлении нового вида ошибок, необходимо добавить соответствующее регулярное выражение и пояснения ошибки.

Алгоритм проверки параметров на присутствие ошибок с помощью регулярных выражений представлен на рисунке 2.2:

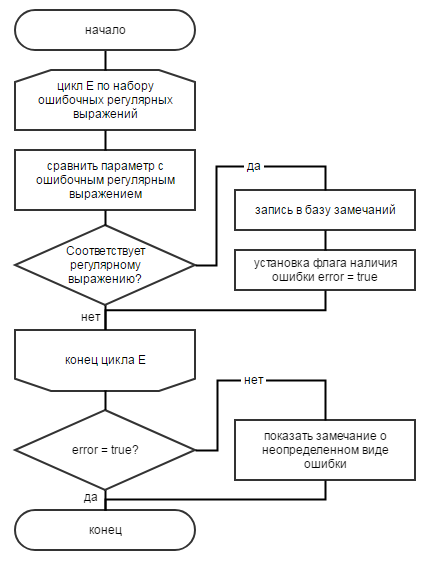


Рис. 2.2 – Алгоритм проверки параметров на присутствие ошибок.

Алгоритм чтения данных тестового артефакта в структуру bookData представлен на рисунке 2.3:

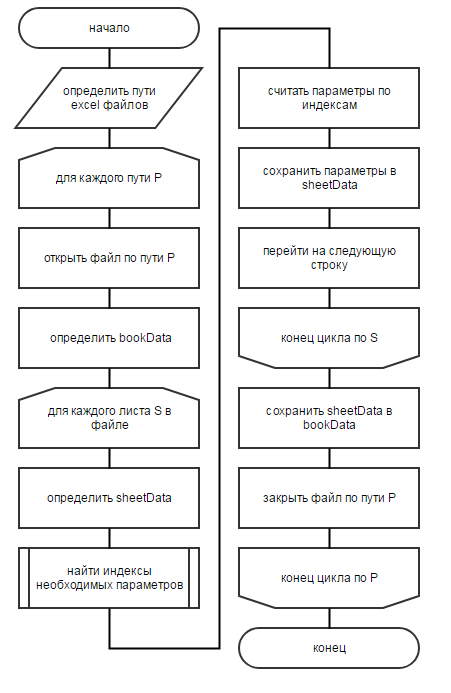


Рис. 2.3 – Этапы чтения протоколов информационного взаимодействия.

sheetData – структура хранения набора параметров на одном листе в excel файле.

signalData – структура хранения информации о наборе параметров сигнала.

## Выводы

В данном разделе было дано описание всех параметров сигналов тестового артефакта, используемых при обработке и генерации текстовых документов для тестового автомата. Также описаны основные этапы работы разрабатываемого алгоритма по обработке протоколов с использованием набора регулярных выражений.

# Технологический раздел

Данный раздел содержит обоснование выбора средств разработки программного обеспечения и описание программной реализации разрабатываемого метода. Также приводится описание форматов данных, структура разработанного ПО и основной интерфейс приложения.

## Выбор средств разработки

На сегодняшний день для работы с excel файлами, помимо библиотек для различных языков, компания Microsoft имеет собственный язык программирования VBA (Visual Basic for Application). Большим преимуществом является очень функциональное API, позволяющее автоматизировать практически любые задачи. Однако имеется сложность в установке подобных решений, а также в дальнейшей интеграции в глобальное приложение отдела, поэтому рассмотрены другие варианты. При анализе, были выделены следующие инструменты:

* Библиотека классов MFC (C++);
* VSTO (Visual Studio Tools for Office) (C#);

Для написания программного обеспечения была выбрана библиотека Microsoft Foundation Classes. С использованием API-функций, пакет MFC позволяет значительно повысить производительность процесса программирования. Также, при тяжелых задачах, связанных с обработкой большого количества данных, С++ имеет более высокую производительность кода, в отличии от C#.

## Основные моменты программной реализации

Для работы с тестовыми артефактами формата Microsoft Office Excel при помощи библиотеки MFC, необходимо добавить следующий набор классов:

* **CApplication.** Основной класс для работы с excel файлами, который позволяет выполнять базовые операции. Содержит параметры на уровне приложений и свойства, возвращающие объекты верхнего уровня.
* **CRange.** Класс, позволяющий работать с ячейками.
* **CWorkbook** и **CWorkbooks** используются для работы с наборами excel книг.
* **CWorksheet** и **CWorksheets** используются для выполнения различных операций с листами excel файлов.

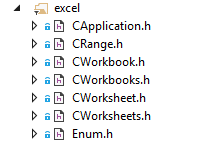


Рис. 3.1 – Используемые классы для работы с Excel.

Для использования всех необходимых методов этих классов был написан класс WorkExcel. Программная реализация методов класса приведена в приложении А.

Программные реализации основных алгоритмов, таких как чтение и обработка тестовых артефактов приведены в приложения Б и В соответственно.

## Требования к запуску ПО

Для запуска разработанного программного обеспечения необходимы следующие инструменты:

* Microsoft Visual Studio – среда разработки;
* Git – система управления версиями (или авторизация в Team Services);

При отсутствии необходимых библиотек, данная среда разработки предложит их автоматическую загрузку и последующую установку.

## Форматы входных и выходных данных

В качестве входных данных для программного обеспечения используются протоколы информационного взаимодействия допустимого формата excel: \*.xls, \*xlsx. (рисунок 3.2).

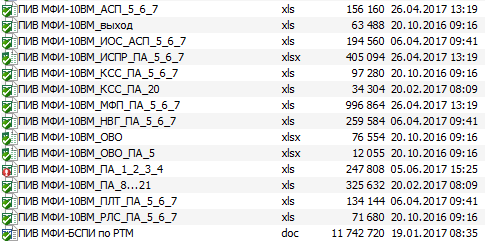


Рис.3.2 – Входные данные для ПО.

Требования для входных данных:

* На одном листе должна быть только одна таблица сигналов.
* Необходимая линия соединения: МКИО (рисунок 3.3).
* После таблицы не должно быть лишних данных (кроме примечаний) (рисунок 3.4).

При несоблюдении требований во время анализа данных могут возникнуть лишние замечания, поскольку возникает некорректное считывание сигналов из протоколов информационного взаимодействия.

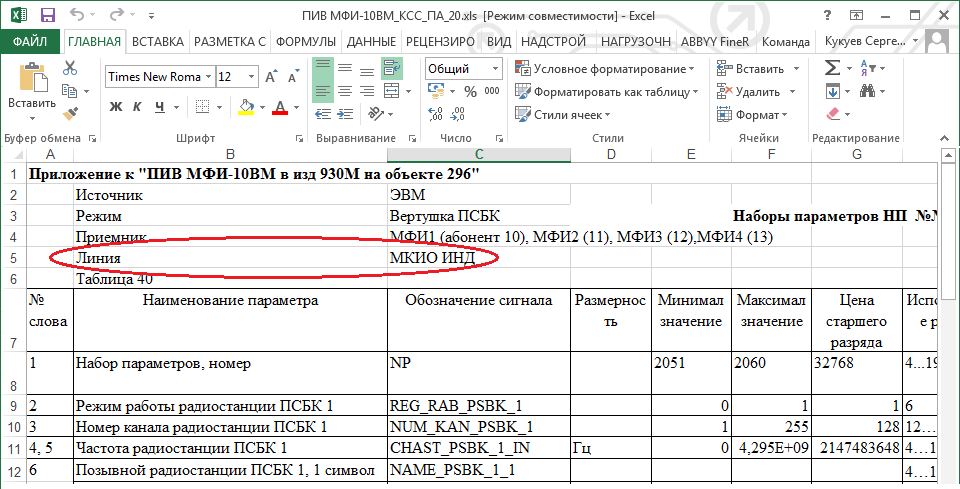


Рис. 3.3 – Обозначение линии соединения.

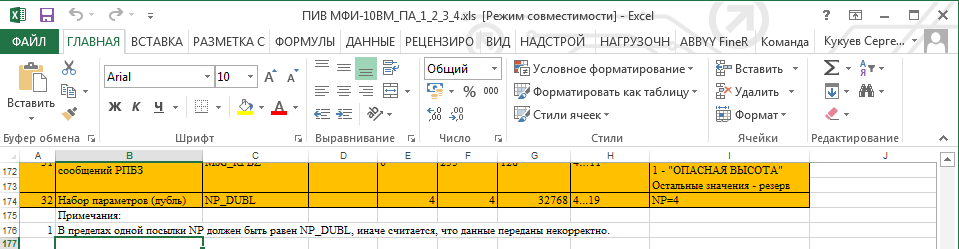


Рис. 3.4 – Окончание таблицы сигналов.

На выходе получаем отчет по анализу на возможные ошибки, показанные на рисунках 3.5 и 3.6. А также сгенерированные записи параметров сигналов для тестового автомата, сохраненные в формате текстового документа, пример которого показан на рисунке 1.3.

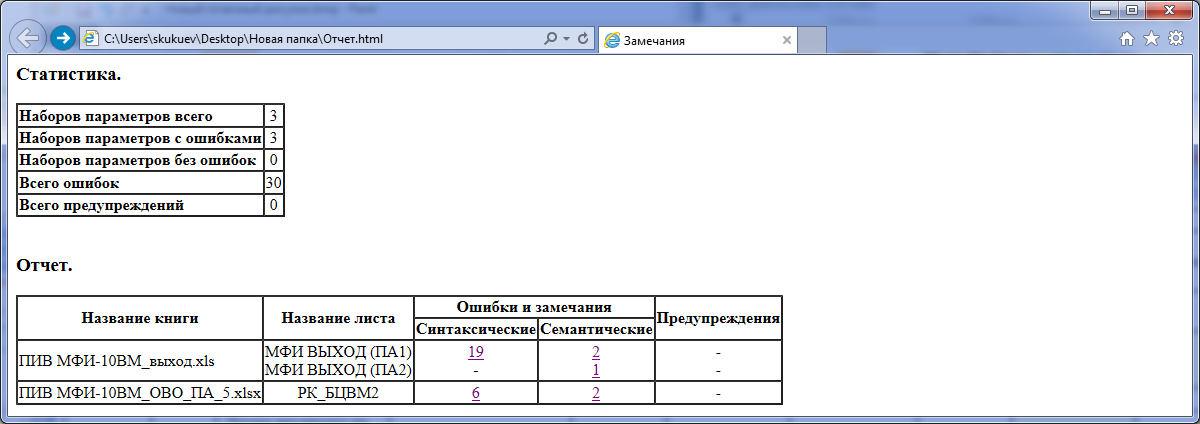


Рис.3.5 – Главная страница отчета.

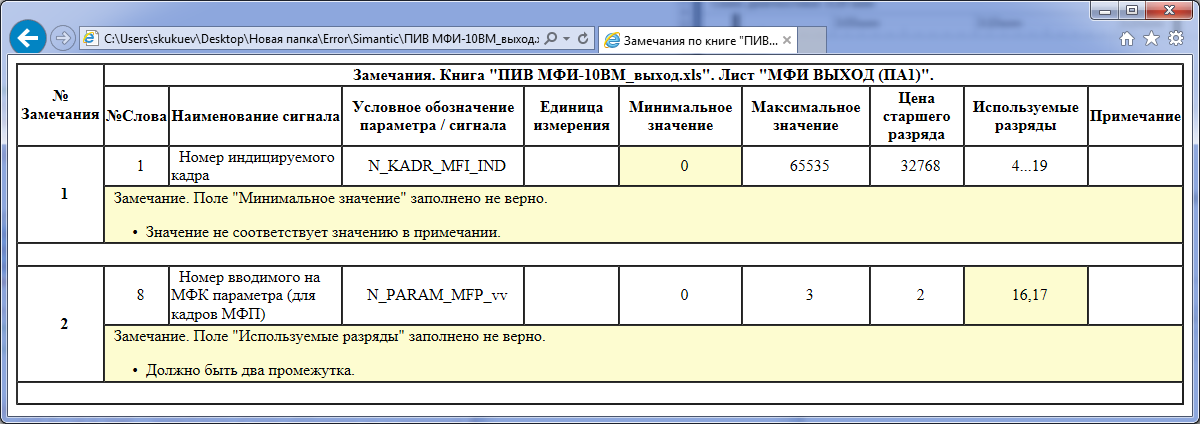


Рис. 3.6 – Страница отчета ошибок и замечаний.

## Структура разработанного ПО

Диаграмма взаимосвязей модулей программного обеспечения выглядит следующим образом:

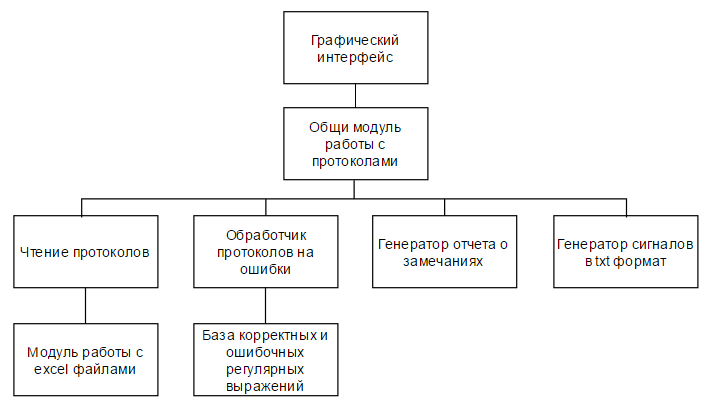


Рис. 3.7 – Схема взаимосвязей модулей программного обеспечения.

Общий модуль работы с протоколами является некой оберткой для основных функций программного обеспечения, отвечает за логику работы программы и является связующим звеном между графическим интерфейсом и функциональной составляющей. Реализованный метод обработки протоколов с помощью набора регулярных выражений находится в модуле обработки протоколов на ошибки.

На рисунке 3.8 представлена диаграмма класса главного модуля приложения, связывающего все функции, сгенерированная средствами Visual Studio. Также ниже представлены классы для генерации отчетов об ошибкаx CReport и протоколов формата txt CRepTxt.

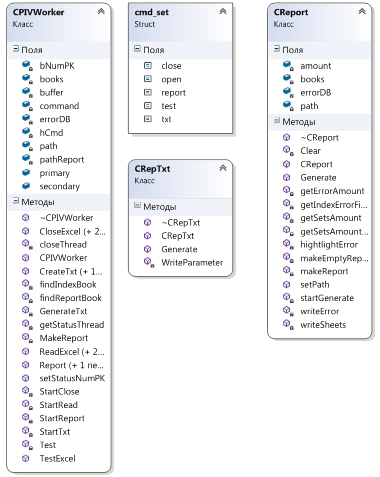


Рис. 3.8 – Диаграмма классов общего модуля, генераторов отчета и формата текстового документа.

Класс «CPIVWorker» реализует доступ к основным функциям и содержит основную информацию о протоколах информационного взаимодействия. Поле *books* содержит в себе набор прочитанных excel файлов. База всех замечаний по книгам содержится в поле *errorDB*. Все функции выполняются через поток *primary*, который принимает команды *cmd\_set.* Основные функции выполняются в следующих методах: ReadExcel, TestExcel, Report, CreateTxt.

Диаграмма класса для работы с excel файлов, описанного в основных моментах программной реализации представлена на рисунке 3.9:

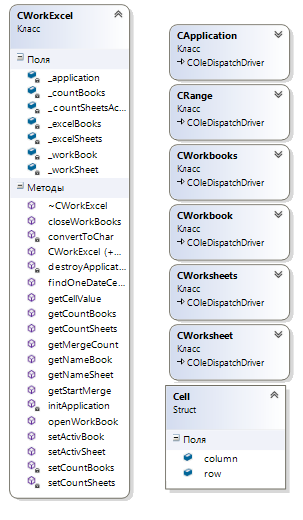


Рис. 3.9 – Диаграмма классов работы с excel файлами.

Диаграмма класса чтения excel файлов представлена на рисунке ниже:

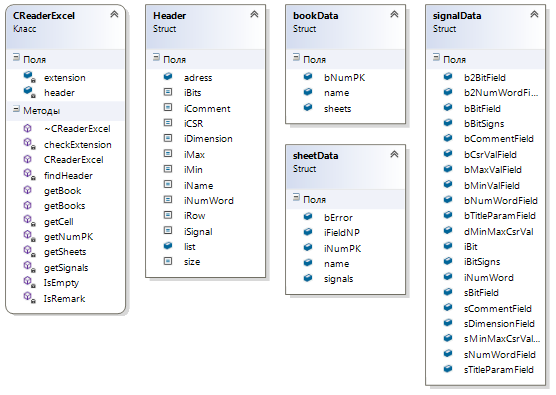


Рис. 3.10 – Диаграмма класса чтения протоколов.

Класс «CReadExcel» реализует чтение всех параметров сигналов из протоколов информационного взаимодействия. Структура *Header* содержит в себе всю информацию о необходимых заголовках таблицы сигналов. Структура *bookData* содержит всю считанную информацию о книге, разделенную на страницы. *sheetData* содержит информацию,находящуюся на текущей странице. Для сохранения параметров сигналов создана структура *signalData*, которая содержится в *sheetData*.

Диаграмма классов основного метода обработки протоколов информационного взаимодействия изображена на рисунке 3.11.

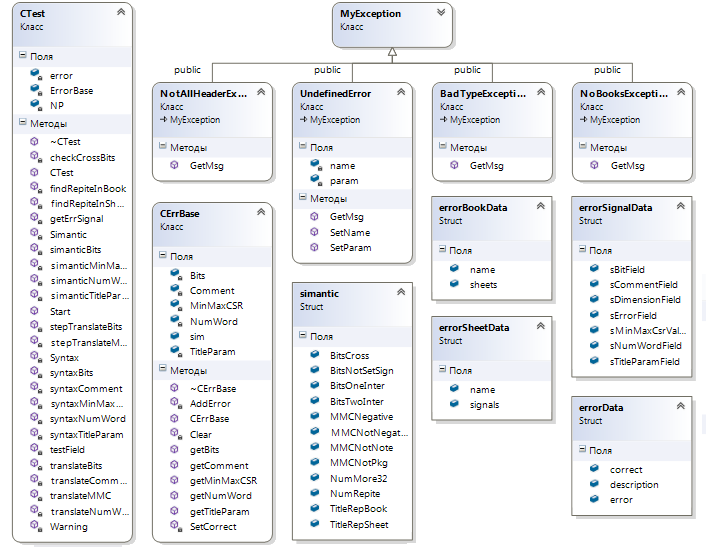


Рис. 3.11 – Диаграмма классов метода обработки excel файлов.

Класс «CTest» предназначен для обработки протоколов информационного взаимодействия. Наборы всех синтаксических и семантических ошибок лежат в классе «CErrBase». Для каждого из параметров содержится поле с корректным регулярным выражением и набором ошибочных регулярных выражений с соответствующими дескрипторами. Для хранения всех записей о замечаниях в протоколах информационного взаимодействия созданы структуры *errorBookData, errorSheetData, errorSignalData*. Данные структуры содержат практически идентичные поля с структурами *bookData, sheetData* и *signalData*, за исключением нескольких полей, предназначенных для описания замечаний.

Кроме проверки на синтаксические и семантические ошибки класс «CTest» содержит методы перевода значений параметров сигнала необходимые для последующего создания txt файлов, необходимого тестовому автомату.

Интерфейс «MyException» представляет абстракцию для использования уникальных исключений при работе с классом «CTest».

## Тестирование

Использование огромного объема входных данных приведет к возникновению ошибок. Таким образом, необходимо модульное тестирование составляющих алгоритма и функциональное тестирование алгоритма в целом для исключения ошибок уже на стадии разработки.

Для проведения модульного тестирования программы использовался компонент Test Explorer среды разработки Visual Studio. При работе с данным компонентом необходимо создать отдельный проект, который будет использоваться для прогонки тестов. Кроме того, тестируемые компоненты ПО должны быть оформлены в виде статических библиотек, которые включаются в тестовый проект при сборке.

Были протестированы следующие части программы:

* ReadExcel;
* WorkExcel;
* Test.

## Интерфейс пользователя

Графический интерфейс приложения показана на рисунке 3.12:

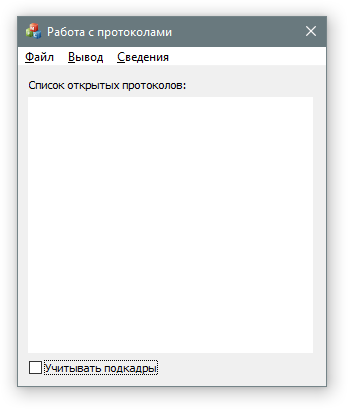


Рис. 3.12 – Графический интерфейс приложения.

При работе с приложением пользователь имеет два всплывающих меню «Файл» и «Вывод», представленных на рисунке 3.13. С помощью всплывающего подменю «Файл» пользователь может открыть один или несколько протоколов информационного взаимодействия, при этом, все открытые на данный момент указаны в «Списке открытых протоколов». Если данный список не будет пустым, станет доступен следующий параметр подменю, который позволяет провести анализ всех открытых файлов. Для активации подменю «Вывод» требуется провести обработку доступных открытых протоколов, тогда генераторы отчета и текстовых документов для тестового автомата, представленные в подменю «Вывод» на рисунке 3.13 станут доступны.

Также для быстрого доступа к каждой функции созданы собственные акселераторы:

* Открыть файл(ы) – CTRL+O;
* Провести анализ открытых файлов – CTRL+A;
* Закрыть выделенный протокол – CTRL+X;
* Создать отчет о замечаниях – CTRL+R;
* Генерировать txt файлы параметров – CTRL+T.

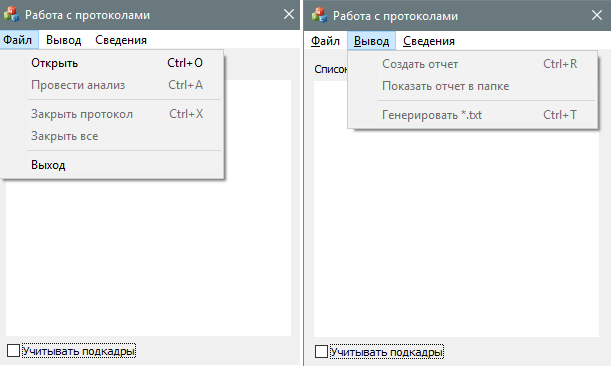


Рис. 3.13 – Всплывающие меню «Файл» и «Вывод».

## Выводы

В соответствии с описанными алгоритмами в конструкторском разделе, на языке С++ в среде разработке Microsoft Visual Studio под управлением OC Windows был реализован метод обработки протоколов информационного взаимодействия с использованием регулярных выражений. Описаны основные моменты программной реализации, структура разработанного программного обеспечения, а также форматы его входных и выходных данных. Продемонстрирован интерфейс приложения. Произведено тестирование и отладка программного продукта.

# Экспериментальный раздел

В данном разделе проводится экспериментальное исследование основных алгоритмов программного обеспечения на основе критериев оценки, которые будут определены в следующем подразделе. В качестве набора входных данных для проведения исследований взяты тестовые артефакты одного из проектов отдела и поделены на следующие группы:

* До 500 сигналов;
* От 500 до 1000 сигналов;
* От 1000 сигналов и выше.

## Критерии оценки

В соответствии с требованиями, предъявленными корпоративной политикой предприятия основным критерием разработанных алгоритмов является быстродействие.

Исследуя архивы документооборота отдела и историю изменения протоколов информационного взаимодействия на сервере, была выведена следующая статистика до разработки данного программного обеспечения:

* Время для поиска и описания одного замечания составляет от 3 до 5 минут;
* Трудоемкость в описании одного набора параметров:
  + Количество замечаний: 0 – 200 штук
  + Время на описание всех замечаний: 0 – 24 часов
* Трудоемкость в описании замечаний всех наборов параметров одного проекта:
  + Количество замечаний: до 3000 штук
  + Время на описание всех замечаний: до 280 часов (2 чел-месяцев)

## Оценка времени работы алгоритмов

Для оценки времени работы алгоритмов было использовано оборудование со следующими основными характеристиками:

* Процессор: Intel Core i7-5775C CPU @ 3.30GHz, 3.30GHz;
* Оперативная память (ОЗУ): 16 Гб.

Общий список протоколов выбранного для исследования проекта с учетом примерного количества сигналов:

* ПИВ МФИ-10ВМ\_КСС\_ПА\_20 (45 сигналов);
* ПИВ МФИ-10ВМ\_ПЛТ\_ПА\_5\_6\_7 (324 сигнала);
* ПИВ МФИ-10ВМ\_РЛС\_ПА\_5\_6\_7 (175 сигналов);
* ПИВ МФИ-10ВМ\_АСП\_5\_6\_7 (588 сигналов);
* ПИВ МФИ-10ВМ\_ИОС\_АСП\_5\_6\_7 (704 сигнала);
* ПИВ МФИ-10ВМ\_ПА\_1\_2\_3\_4 (663 сигнала);
* ПИВ МФИ-10ВМ\_ИСПР\_ПА\_5\_6\_7 (~3000 сигналов);
* ПИВ МФИ-10ВМ\_МФП\_ПА\_5\_6\_7 (~2000 сигналов);
* ПИВ МФИ-10ВМ\_НВГ\_ПА\_5\_6\_7 (1253 сигнала);
* ПИВ МФИ-10ВМ\_ПА\_8...21 (~2000 сигналов).

### Чтение протоколов

Для анализа быстродействия алгоритма чтения протоколов информационного взаимодействия были произведены замеры времени для каждого файла, представленного в списке подраздела 4.1. Поскольку средства для определения времени, представленные ОС Windows имеют такие недостатки, как низкая точность и большая погрешность измерений, было произведено несколько замеров и высчитано их среднее арифметическое значение. Данные измерений представлены в таблице 4.1.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Наименование протокола | Кол-во сигналов | Измерение времени (сек) | | | | | Итог |
| I | II | III | IV | V |
| ПИВ МФИ-10ВМ\_КСС\_ПА\_20 | до 500 | 1,06 | 1,07 | 1,05 | 1,06 | 1,09 | 1,06 |
| ПИВ МФИ-10ВМ\_ПЛТ\_ПА\_5\_6\_7 | 4,01 | 4,08 | 4,09 | 4,57 | 4,69 | 4,29 |
| ПИВ МФИ-10ВМ\_РЛС\_ПА\_5\_6\_7 | 3,7 | 3,38 | 3,29 | 3,21 | 3,19 | 3,35 |
| ПИВ МФИ-10ВМ\_АСП\_5\_6\_7 | от 500 до 1000 | 12,14 | 11,96 | 14,395 | 12,45 | 12,02 | 12,59 |
| ПИВ МФИ-10ВМ\_ИОС\_АСП\_5\_6\_7 | 15,62 | 15,84 | 16,177 | 16,02 | 16,57 | 16,04 |
| ПИВ МФИ-10ВМ\_ПА\_1\_2\_3\_4 | 9,6 | 9,22 | 8,118 | 8,28 | 8,13 | 8,67 |
| ПИВ МФИ-10ВМ\_ИСПР\_ПА\_5\_6\_7 | от 1000 | 391,16 | 354,6 | 454,473 | 403,32 | 385,47 | 397,81 |
| ПИВ МФИ-10ВМ\_МФП\_ПА\_5\_6\_7 | 95,55 | 98,01 | 97,513 | 99,25 | 102,15 | 98,49 |
| ПИВ МФИ-10ВМ\_НВГ\_ПА\_5\_6\_7 | 25,15 | 24,15 | 29,543 | 26,48 | 23,11 | 25,69 |
| ПИВ МФИ-10ВМ\_ПА\_8...21 | 51,95 | 55,87 | 50,21 | 53,48 | 56,77 | 53,66 |

Таблица 4.1. – Измерение времени работы алгоритма чтения.

Из результатов эксперимента видно, что возрастание времени работы алгоритма при увеличении количества сигналов происходит нелинейно. Столь резкое возрастание, представленное на рисунке 4.1, обусловлено большей частотой обращения к excel файлу и постоянным переключением между этим процессом и приложением.

Рис. 4.1 – График изменения времени работы алгоритма чтения.

### Обработка протоколов

Для анализа быстродействия алгоритма обработки протоколов информационного взаимодействия на основе наборов регулярных выражений были произведены расчеты, представленные в таблице 4.2. Измерения проведены аналогично с теми, которые изображены в пункте 4.1.1 при помощи стандартной библиотеки С++ – *ctime*.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Наименование протокола | Кол-во сигналов | Измерение времени (сек) | | | | | Итог |
| I | II | III | IV | V |
| ПИВ МФИ-10ВМ\_КСС\_ПА\_20 | до 500 | 0,11 | 0,15 | 0,13 | 0,12 | 0,13 | 0,12 |
| ПИВ МФИ-10ВМ\_ПЛТ\_ПА\_5\_6\_7 | 0,68 | 0,74 | 0,71 | 0,7 | 0,74 | 0,71 |
| ПИВ МФИ-10ВМ\_РЛС\_ПА\_5\_6\_7 | 0,62 | 0,6 | 0,61 | 0,62 | 0,62 | 0,61 |
| ПИВ МФИ-10ВМ\_АСП\_5\_6\_7 | от 500 до 1000 | 3,65 | 3,59 | 3,55 | 3,57 | 3,59 | 3,59 |
| ПИВ МФИ-10ВМ\_ИОС\_АСП\_5\_6\_7 | 4,51 | 4,21 | 4,35 | 4,18 | 4,35 | 4,32 |
| ПИВ МФИ-10ВМ\_ПА\_1\_2\_3\_4 | 1,4 | 1,4 | 1,38 | 1,41 | 1,39 | 1,39 |
| ПИВ МФИ-10ВМ\_ИСПР\_ПА\_5\_6\_7 | от 1000 | 84,7 | 83,06 | 80,24 | 82,65 | 83,54 | 82,84 |
| ПИВ МФИ-10ВМ\_МФП\_ПА\_5\_6\_7 | 23,12 | 22,26 | 22,15 | 22,12 | 21,24 | 22,18 |
| ПИВ МФИ-10ВМ\_НВГ\_ПА\_5\_6\_7 | 5,53 | 5,41 | 5,43 | 5,4 | 5,41 | 5,44 |
| ПИВ МФИ-10ВМ\_ПА\_8...21 | 12,64 | 12,51 | 12,52 | 12,49 | 12,52 | 12,53 |

Таблица 4.2 – Измерение времени работы алгоритма обработки протоколов.

Из результатов эксперимента видно, что алгоритм обработки протоколов методом регулярных выражений работает значительно быстрее алгоритма чтения. Это обусловлено отсутствием взаимодействия с excel файлом. Кроме того, была замечена существенная разница во времени между обработкой двух протоколов с примерно одинаковым количеством сигналов в категории «<1000». Список файлов, представленных на рисунке 4.2:

1. ПИВ МФИ-10ВМ\_ИСПР\_ПА\_5\_6\_7;
2. ПИВ МФИ-10ВМ\_МФП\_ПА\_5\_6\_7;
3. ПИВ МФИ-10ВМ\_НВГ\_ПА\_5\_6\_7;
4. ПИВ МФИ-10ВМ\_ПА\_8...21.

Как видно из рисунка 4.2, разница во времени обработки протоколов 2 и 4 почти в два раза, при том, что количество сигналов у обоих порядка 2000, обусловлена количеством альтернативных результатов в регулярных выражениях, поскольку при использовании регулярных структур используется несколько кванторов и чередований, что увеличивает риск тупиковых ситуаций.

Рис. 4.2 – Гистограмма времени обработки протоколов категории «<1000».

## Выводы

В данном разделе проведены экспериментальные исследования для оценки быстродействия разработанных алгоритмов. Исходя из статистики, составленной на основе данных отдела, благодаря разработанному алгоритму было сокращено время обработки тестовых артефактов с 280 часов до 20 минут.

В ходе экспериментов было выявлено, что алгоритм чтения тестовых артефактов требует дальнейшей оптимизации.

# ЗАКЛЮЧЕНИЕ

В результате выполнения выпускной квалификационной работы бакалавра было разработано ПО для обработки тестовых артефактов для тестового автомата с использованием регулярных выражений. В частности, были достигнуты следующие результаты:

* Разработан метод обработки параметров протоколов с использованием наборов регулярных выражений;
* Разработано ПО, реализующее данный метод.

Разработанное программное обеспечение полностью соответствует заявленным требованиям и уже внедрена в работу отдела. В качестве дальнейшего развития можно выделить следующие аспекты:

* Добавление обработки тестовых артефактов с линией передачи Arinc;
* Распараллеливание реализованного алгоритма;
* Оптимизация алгоритма чтения тестовых артефактов при помощи использования преобразования в xml-классы.

# СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Кучерявый А.А. Бортовые информационные системы – 2-е изд., Ульяновск: УлГТУ, 2004
2. Серверы предприятия АО «РПКБ»
3. Microsoft, Интерфейс ODBC – MSDN [Электронный ресурс] – Режим доступа: <https://msdn.microsoft.com/ru-ru/library/s9ds2ktb.aspx> (Дата обращения 30.05.2017)
4. Суховилов Б.М. Часть I. Архитектура Microsoft Windows для разработчиков. [Электронный ресурс] – Режим доступа: <http://inf.susu.ac.ru:8000/sukhovilov/MSWinbook1/> (Дата обращения 30.05.2017)
5. Microsoft, Программирование OLE DB – MSDN [Электронный ресурс] – Режим доступа: <https://msdn.microsoft.com/ru-ru/library/502e07a7.aspx> (Дата обращения 31.05.2017)
6. OLE DB или ODBC? – Windows IT Pro [Электронный ресурс] – Режим доступа: <https://www.osp.ru/winitpro/2000/01/174144/> (Дата обращения 30.05.2017)
7. Боб Виллариал, Программирование в Access 2002 – Кудиз-образ, Москва, 2003
8. Лекция 40: Алгоритмы поиска в тексте – НОУ ИНТУИТ [Электронный ресурс] – Режим доступа: <http://www.intuit.ru/studies/courses/648/504/lecture/11468> (Дата обращения 02.06.2017)
9. Префикс-функция. Алгоритм Кнута-Морриса-Пратта. [Электронный ресурс] – Режим доступа: <https://brestprog.neocities.org/lections/prefixfunction.html> (Дата обращения 02.06.2017)
10. Thierry Lecroq, A variation on the Boyer-Moore algorithm – Theoretical Computer Science, 1992
11. Описание работы алгоритма Shift-OR для поиска подстроки в строке [Электронный ресурс] – Режим доступа: <https://habrahabr.ru/post/132128/> (Дата обращения 02.06.2017)

# ПРИЛОЖЕНИЕ А

// Инициализация объекта Excel и указателя

bool CWorkExcel::initApplication()

{

if (!\_application.CreateDispatch(\_T("Excel.Application"))) //запустить сервер

{

AfxMessageBox(\_T("Ошибка при старте Excel!"));

return false;

}

\_excelBooks = \_application.get\_Workbooks(); // указатель на открытые книги

if (!\_excelBooks)

{

AfxMessageBox(\_T("Ошибка при инициализации указателя на Книги!"));

ASSERT(\_excelBooks != NULL);

return false;

}

return true;

}

// Уничтожение объекта Excel и указателя

void CWorkExcel::destroyApplication()

{

\_excelBooks.Close();

\_excelBooks.ReleaseDispatch();

\_application.Quit();

\_application.ReleaseDispatch();

\_application = nullptr;

\_excelBooks = nullptr;

\_countBooks = 0;

}

//Конструктор

CWorkExcel::CWorkExcel()

{

initApplication();

}

//Конструктор с параметром

CWorkExcel::CWorkExcel(CString& pathToTemplate)

{

initApplication();

openWorkBook(pathToTemplate);

}

// Деструктор

CWorkExcel::~CWorkExcel()

{

destroyApplication();

}

//Открытие или добавление рабочей книги

void CWorkExcel::openWorkBook(CString& pathToTemplate)

{

if (!\_excelBooks)

initApplication();

COleVariant covOptional((long)DISP\_E\_PARAMNOTFOUND, VT\_ERROR);

\_excelBooks.Open(pathToTemplate, covOptional, covOptional, covOptional, covOptional, covOptional, covOptional, covOptional,

covOptional, covOptional, covOptional, covOptional, covOptional, covOptional, covOptional);

setCountBooks();

}

//Закрытие всех книг

void CWorkExcel::closeWorkBooks()

{

destroyApplication();

}

//Задание активной книги

void CWorkExcel::setActivBook(long& iBook)

{

\_workBook = \_excelBooks.get\_Item(COleVariant(iBook));

\_excelSheets = \_workBook.get\_Sheets();

setCountSheets();

}

//Задание активного листа в книге

void CWorkExcel::setActivSheet(long& iSheet)

{

\_workSheet = \_excelSheets.get\_Item(COleVariant(iSheet));

}

//Получение названия книги

CString CWorkExcel::getNameBook()

{

return \_workBook.get\_Name();

}

//Получение названия листа

CString CWorkExcel::getNameSheet()

{

return \_workSheet.get\_Name();

}

//Получение количества открытых книг

int CWorkExcel::getCountBooks()

{

return \_countBooks;

}

//Получение количества листов в активной книге

int CWorkExcel::getCountSheets()

{

return \_countSheetsActivBook;

}

//Получение количества объедененных ячеек

int CWorkExcel::getMergeCount(Cell& cell)

{

CString rangeCellX;

int iCol = static\_cast<int>(cell.column);

rangeCellX.Format(\_T("%s%d"), convertToChar(iCol), cell.row);

CRange cellRange = \_workSheet.get\_Range(COleVariant(rangeCellX), COleVariant(rangeCellX));

cellRange = cellRange.get\_MergeArea();

cellRange = cellRange.get\_Rows();

return cellRange.get\_Count();

}

//Получение индекса стартовой ячейки объединения

long CWorkExcel::getStartMerge(Cell& cell)

{

CString rangeCellX;

int iCol = static\_cast<int>(cell.column);

rangeCellX.Format(\_T("%s%d"), convertToChar(iCol), cell.row);

CRange cellRange = \_workSheet.get\_Range(COleVariant(rangeCellX), COleVariant(rangeCellX));

cellRange = cellRange.get\_MergeArea();

cellRange = cellRange.get\_Rows();

return cellRange.get\_Row();

}

// Получение значения из ячеки

CString CWorkExcel::getCellValue(Cell& cell)

{

CRange cellRange = \_workSheet.get\_Cells();

return cellRange.get\_Item(COleVariant(cell.row), COleVariant(cell.column));

}

// Поиск по ячекам в активном листе

bool CWorkExcel::findOneDateCells(CString& findString, Cell& cell)

{

CRange firstFind = \_workSheet.get\_Cells();

CRange Find;

COleVariant covOptional((long)DISP\_E\_PARAMNOTFOUND, VT\_ERROR);

COleVariant covTrue((short)TRUE, VT\_BOOL), covFalse((short)FALSE, VT\_BOOL);

Find = firstFind.Find(COleVariant(findString), covOptional, COleVariant(long(xlValues)), COleVariant(long(xlPart)),

COleVariant(long(xlByRows)), (long)xlNext, covFalse, covFalse, covFalse);

if (Find)

{

cell.column = Find.get\_Column();

cell.row = Find.get\_Row();

return true;

}

else

return false;

}

// Установка значения открытых книг

void CWorkExcel::setCountBooks()

{

\_countBooks = \_excelBooks.get\_Count();

}

//Установка количества листов в активной книги

void CWorkExcel::setCountSheets()

{

\_countSheetsActivBook = \_excelSheets.get\_Count();

}

//Получения буквенного обозначения ячейки из номера колонки

CString CWorkExcel::convertToChar(int & iCol)

{

CString rez = \_T("");

int iA = 0, iB = 0;

if (iCol <= 26)

rez.Format(\_T("%c"), static\_cast<char>(iCol + 64));

else

{

iA = iCol / 26;

iB = iCol % 26;

if (iB == 0)

iB = 26;

rez.Format(\_T("%s%s%s"), rez, convertToChar(iA), static\_cast<char>(iB + 64));

}

return rez;

}

# ПРИЛОЖЕНИЕ Б

// Чтение одной книги

bookData CReaderExcel::getBook(CString pathToExcel)

{

if (!checkExtension(pathToExcel))

throw BadTypeException();

CWorkExcel work;

bookData book;

work.openWorkBook(pathToExcel); // Открытие книги

if (work.getCountBooks() == 0)

throw NoBooksException();

long clBook = (long)1;

work.setActivBook(clBook);

book.name = work.getNameBook();

book.sheets.resize(work.getCountSheets());

book.sheets = getSheets(work);

work.closeWorkBooks(); // Закрытие книги

return book;

}

// Чтение листов

vector <sheetData> CReaderExcel::getSheets(CWorkExcel& work)

{

vector <sheetData> sheets;

sheets.resize(work.getCountSheets());

for (int i = 1; i < work.getCountSheets() + 1; i++)

{

long iSheet = static\_cast<long> (i);

work.setActivSheet(iSheet);

sheets[i - 1].name = work.getNameSheet();

sheets[i - 1].iFieldNP = -1;

sheets[i - 1].iNumPK = -1;

if (!findHeader(work)) // Поиск заголовков на листе

throw NotAllHeaderException();

sheets[i - 1].bError = false;

sheets[i - 1].iNumPK = getNumPK(work);

header.adress[header.iRow]++;

sheets[i - 1].signals = getSignals(work);

}

return sheets;

}

// Поиск номера кадра (в противном случае будет равен -1)

int CReaderExcel::getNumPK(CWorkExcel& work)

{

// Переход к столбцу примечаний

Cell cell;

cell.row = static\_cast<long> (header.adress[header.iRow] - 1);

cell.column = static\_cast<long> (header.adress[header.iComment]);

int iNumPK = -1; // Индекс подкадра

if (work.getCellValue(cell) != "")

{

CString sNumPK = work.getCellValue(cell);

int i = 0;

while (i != -1)

{

i = sNumPK.Find(\_T("№"), i == 0 ? i : i + 1);

i = i == sNumPK.GetLength() ? -1 : i;

iNumPK = i == -1 ? iNumPK : i;

}

if (iNumPK != -1)

sNumPK.Delete(0, iNumPK + 1);

sNumPK = sNumPK.Trim();

iNumPK = \_wtoi(sNumPK);

}

return iNumPK;

}

// Чтение параметров на листе

list <signalData> CReaderExcel::getSignals(CWorkExcel& work)

{

list <signalData> signals;

bool bRemark = false; // Является ли строка примечанием

int cEmpty = 0; // Счетчик пустых строк

do

{

signalData signal;

Cell cell;

cell.row = static\_cast<long> (header.adress[header.iRow]);

// Чтение параметров:

// Наименование параетра и обозначение сигнала

cell.column = static\_cast<long> (header.adress[header.iName]);

signal.sTitleParamField[0] = getCell(work, cell);

cell.column = static\_cast<long> (header.adress[header.iSignal]);

signal.sTitleParamField[1] = getCell(work, cell);

long cMergeName = work.getMergeCount(cell);

signal.bTitleParamField = false;

// Чтение номера слова

cell.column = static\_cast<long> (header.adress[header.iNumWord]);

signal.sNumWordField = getCell(work, cell, cMergeName);

signal.b2NumWordField = false;

signal.bNumWordField = false;

// Чтение размерности, min, max и csr

cell.column = static\_cast<long> (header.adress[header.iDimension]);

signal.sDimensionField = getCell(work, cell, cMergeName);

cell.column = static\_cast<long> (header.adress[header.iMin]);

signal.sMinMaxCsrValField[0] = getCell(work, cell, cMergeName);

signal.bMinValField = false;

cell.column = static\_cast<long> (header.adress[header.iMax]);

signal.sMinMaxCsrValField[1] = getCell(work, cell, cMergeName);

signal.bMaxValField = false;

cell.column = static\_cast<long> (header.adress[header.iCSR]);

signal.sMinMaxCsrValField[2] = getCell(work, cell, cMergeName);

signal.bCsrValField = false;

// Чтение разрядов

cell.column = static\_cast<long> (header.adress[header.iBits]);

signal.sBitField = getCell(work, cell, cMergeName);

signal.bBitField = false;

signal.b2BitField = false;

signal.iBitSigns = 0;

signal.bBitSigns = false;

// Чтение комментариев

cell.column = static\_cast<long> (header.adress[header.iComment]);

signal.sCommentField = getCell(work, cell, cMergeName);

signal.bCommentField = false;

bool bEmpty = IsEmpty(work, cell.row);

bRemark = IsRemark(work, cell.row);

if (bEmpty)

cEmpty++; // инкрементация счетчика пустых строк

else

cEmpty = 0; // сброс счетчика

// Добавление сигнала

if (!bEmpty && !bRemark)

signals.push\_back(signal);

header.adress[header.iRow] += cMergeName;

} while (cEmpty < MAX\_EMPTY\_STRING && !bRemark);

return signals;

}

// Является ли строка примечанием

bool CReaderExcel::IsRemark(CWorkExcel& work, long row)

{

bool result = false;

Cell cell;

cell.row = row;

for (long i = 1; i < header.size; i++)

{

cell.column = static\_cast<long> (header.adress[i]);

result = (work.getCellValue(cell).Find(\_T("Примечания:")) > -1 ||

work.getCellValue(cell).Find(\_T("Примечание:")) > -1) ? true : result;

}

return result;

}

// Чтение ячейки

CString CReaderExcel::getCell(CWorkExcel& work, Cell cell, long cName)

{

CString result;

long size = work.getMergeCount(cell);

// Если слитая ячейка наименования больше, чем текущее, значит читаем все ячейки в количестве слитых ячеек наименования

if (cName > size)

{

long start = work.getStartMerge(cell);

for (long i = start; i < start + cName; i++)

{

cell.row = i;

CString tmp = work.getCellValue(cell);

if (!result.IsEmpty() && !tmp.IsEmpty())

result += \_T(", ");

if (!tmp.IsEmpty())

result += work.getCellValue(cell);

}

}

else

{

cell.row = work.getStartMerge(cell);

result = work.getCellValue(cell);

}

return result;

}

# ПРИЛОЖЕНИЕ В

// Проверка на синтаксические ошибки

void CTest::Syntax(errorBookData& errBook, bookData& book)

{

for (size\_t cSheet = 0; cSheet < errBook.sheets.size(); cSheet++)

{

errBook.sheets[cSheet].name = book.sheets[cSheet].name;

for (list <signalData>::iterator it = book.sheets[cSheet].signals.begin(); it != book.sheets[cSheet].signals.end(); it++)

{

try

{

bool begin = true, result = true;

if (it != book.sheets[cSheet].signals.begin()) begin = false;

if (it->sTitleParamField->Compare(\_T("Резерв")) != 0)

{

result = syntaxNumWord(errBook.sheets[cSheet], it);

(!syntaxTitleParam(errBook.sheets[cSheet], it)) ? result = false : result = result;

(!syntaxMinMaxCSR(errBook.sheets[cSheet], it)) ? result = false : result = result;

(!syntaxBits(errBook.sheets[cSheet], it)) ? result = false : result = result;

(!syntaxComment(errBook.sheets[cSheet], it, begin)) ? result = false : result = result;

}

if (NP != 0)

{

book.sheets[cSheet].iFieldNP = NP; // Установка номера набора (translateComment)

NP = 0;

}

if (!result) // Установка флага, что на листе есть ошибка

book.sheets[cSheet].bError = true;

}

catch (UndefinedError& exc)

{

exc.SetName(errBook.sheets[cSheet].name);

throw exc;

}

}

}

}

// Проверка номера слова

bool CTest::syntaxNumWord(errorSheetData& sheet, list<signalData>::iterator& it)

{

list <CString> error = testField(it->sNumWordField, ErrorBase.getNumWord());

if (error.empty())

{

translateNumWord(it); // Преобразование слов в числа

return true;

}

error.push\_front(errRemarks[0]);

errorSignalData signal = getErrSignal(it, error);

sheet.signals.push\_back(signal);

it->bNumWordField = true;

return false;

}

// Проверка наименований сигнала

bool CTest::syntaxTitleParam(errorSheetData& sheet, list<signalData>::iterator& it)

{

list <CString> error = testField(it->sTitleParamField[1], ErrorBase.getTitleParam());

if (error.empty())

return true;

error.push\_front(errRemarks[1]);

errorSignalData signal = getErrSignal(it, error);

sheet.signals.push\_back(signal);

it->bTitleParamField = true;

return false;

}

// Проверка минимального, максимального и цср

bool CTest::syntaxMinMaxCSR(errorSheetData& sheet, list<signalData>::iterator& it)

{

list <CString> errMin, errMax, errCSR;

bool result = true;

if (!it->sMinMaxCsrValField[0].IsEmpty() || !it->sMinMaxCsrValField[0].IsEmpty() || !it->sMinMaxCsrValField[0].IsEmpty())

{

errMin = testField(it->sMinMaxCsrValField[0], ErrorBase.getMinMaxCSR());

errMax = testField(it->sMinMaxCsrValField[1], ErrorBase.getMinMaxCSR());

errCSR = testField(it->sMinMaxCsrValField[2], ErrorBase.getMinMaxCSR());

}

if (!errMin.empty())

{

errMin.push\_front(errRemarks[2]);

errorSignalData signal = getErrSignal(it, errMin);

sheet.signals.push\_back(signal);

it->bMinValField = true;

result = false;

}

if (!errMax.empty())

{

errMax.push\_front(errRemarks[3]);

errorSignalData signal = getErrSignal(it, errMax);

sheet.signals.push\_back(signal);

it->bMaxValField = true;

result = false;

}

if (!errCSR.empty())

{

errCSR.push\_front(errRemarks[4]);

errorSignalData signal = getErrSignal(it, errCSR);

sheet.signals.push\_back(signal);

it->bCsrValField = true;

result = false;

}

if (result)

translateMMC(it);

return result;

}

// Проверка используемых разрядов

bool CTest::syntaxBits(errorSheetData& sheet, list<signalData>::iterator& it)

{

list <CString> error = testField(it->sBitField, ErrorBase.getBits());

if (error.empty())

{

translateBits(it);

return true;

}

error.push\_front(errRemarks[5]);

errorSignalData signal = getErrSignal(it, error);

sheet.signals.push\_back(signal);

it->bBitField = true;

return false;

}

// Проверка комментариев

bool CTest::syntaxComment(errorSheetData& sheet, list<signalData>::iterator& it, bool begin)

{

list <CString> error;

string field = CT2A(it->sCommentField);

if (begin)

{

bool bNP = regex\_search(field, ErrorBase.getComment().error[0]); // Проверка на набор параметров

if (bNP)

error.push\_back(ErrorBase.getComment().description[0]);

}

else

{

bool zn = regex\_search(field, ErrorBase.getComment().error[1]); // Проверка на знак

if (zn)

{

error.push\_back(ErrorBase.getComment().description[1]);

it->bBitSigns = true;

}

}

if (error.empty())

{

translateComment(it); // Достать значение знакового бита или NP набора, если они есть

return true;

}

error.push\_front(errRemarks[6]);

errorSignalData signal = getErrSignal(it, error);

sheet.signals.push\_back(signal);

return false;

}

// Проверка поля на ошибки

list <CString> CTest::testField(CString field, errorData errStruct)

{

string temp = CT2A(field);

list <CString> error; // Набор найденных ошибок в поле

bool result = false;

result = regex\_match(temp, errStruct.correct); // Проверка на корректную регулярку

if (result) return error;

for (size\_t i = 0; i < errStruct.error.size(); i++) // Перебор базы ошибок

if (regex\_search(temp, errStruct.error[i]))

{

error.push\_back(errStruct.description[i]);

result = true;

}

if (!result) // исключение, если в базе ошибок такая ошибка не предусмотрена

{

UndefinedError exc;

exc.SetParam(field);

throw exc;

}

return error;

}