# Титульный лист

Тема моей работы:

Разработка и реализация языка описания сценариев тестирования автомобильных систем

# Введение

С развитием автомобильной промышленности тестирование автомобильных систем стало неотъемлемой частью жизненного цикла разработки программного и аппаратного обеспечения транспортных средств.

Для тестирования автомобильных систем, необходимо хорошо понимать, как устроен автомобиль в целом и его составляющие по отдельности.

# Тестирование «черного ящика»

ПО, разрабатываемое для автомобильных систем, при тестировании может рассматриваться в качестве «черного ящика». **ССЫЛКА** Под черным ящиком подразумевается объект исследования, внутреннее устройство которого неизвестно. В кибернетике оно помогает изучать поведение системы, абстрагируясь от ее внутреннего устройства. Такое тестирование называется поведенческим.

При поведенческом тестировании известно какой результат должен быть при определенном наборе данных, которые подаются на вход. Проанализировав требования и спецификации тестировщик может определяет набор тестовых сценариев для проверки различной функциональности тестируемой системы и разрабатывает тесты на каком-либо языке программирования.

* В марте 2006 года Дэн Норт предложил методологию «разработки через поведение» (behaviour-driven development, BDD), основанную на гибкой методологии разработки.

Идея: в процессе написания требования для разрабатываемого ПО аналитики должны описать тестовые сценарии таким образом, чтобы их смог понять и разработчик, и тестировщик, и заказчик.

Зачастую заказчики и аналитики не имеют достаточных знаний в языках программирования общего назначения, либо использование этих языков не удобно из-за большого количества не относящихся к решаемой задаче деталей (управление памятью, низкоуровневые управляющие конструкции и так далее).

* Почти в то же время Эрик Эванс опубликовал книгу «Предметно-ориентированное проектирование», в которой описал

идею, что для бизнеса удобно смоделировать систему, в которой будет определен единый язык (Domain specific language, DSL), основанный на бизнес области, такой, что бизнес словарь смогут использовать как менеджеры, так и инженеры.

Такой язык обеспечит пользователям возможность коротко и четко сформулировать задачу и позволит уменьшить количество тестовой документации.

**В данный момент производители АС пытаются найти решение, предоставляющее возможность описывать тестовые сценарии в форме, оговариваемой внутри команды, состоящей заказчика, аналитика, разработчика и тестировщика.**

# Формулировка задачи

В ходе данной работы необходимо разработать проблемно-ориентированный язык (DSL) для функционального тестирования автомобильных систем, который позволит использовать методологию «разработки через поведение» (BDD), предоставит функционал, необходимый для симуляции работы автомобильных сетей, а также позволит пользователям описывать инструкции сценариев в форме, оговариваемой внутри команды.

# Аналоги

Рассмотрим существующие инструменты, которые можно использовать при тестировании АС

**CAPL**

С помощью языка CAPL (Communication Access Programming Language) можно описать тесты, полностью покрывающие функциональные требования АС. Однако данный язык является Си подобным, достаточно трудным для понимания менеджерами и аналитиками, и не предоставляют пользователям возможность в свободной форме описывать тестовые шаги

**ССDL**

Еще одним языком для описания тестовых спецификаций в автомобильной сфере является CCDL. Он может использоваться для автоматизированного тестирования «черного ящика». Однако спецификации содержат в себе программный код, и не предоставляют пользователям возможность в свободной форме описывать тестовые шаги.

**Cucumber**

Cucumber – это фреймворк, реализующий подход BDD. Программная реализация шагов отделена от теста, что очень удобно для бизнеса. Однако для создания полной проблемно-ориентированной модели АС необходимо каждый раз заново реализовывать протокол коммуникации с тестовым оборудованием. Кроме того, в тестировании АС зачастую одни и те же шаги необходимо выполнять десятки и сотни раз. Cucumber не позволяет описать в сценарии повторяющиеся шаги, а значит тестовая документация будет очень длинной.

**Вывод**

Из вышеперечисленных инструментов для тестирования ПО видно, что ни один из них не позволяет описывать тестовые сценарии в свободной форме, при этом предоставляя возможность проверки поведения автомобильных систем.

Таким образом существует необходимость в создании предметно-ориентированного языка для функционального тестирования, который объединит в себе принципы BDD и функционал, необходимый для симуляции работы автомобильных сетей. Наличие такого языка значительно упростит разработку ПО и повысит качество разрабатываемого продукта.

# постановка задачи

Требования к работе:

* Язык должен позволять пользователям описывать инструкции для тестовых сценариев в форме, оговариваемой внутри бизнес-команды.
* Синтаксис языка должен позволять описывать несколько тестовых сценариев в документе.
* Тестовые сценарии должны выполняться в той очередности, в которой описаны в документе.
* Необходимо предоставить пользователю возможность работы с коммуникационными сетями автомобильных систем.
* Результаты выполнения тестов должны сохраняться в файл.
* Необходимо реализовать обработчик ошибок.

Качество решения, представленного в данной работе должно быть проверено следующим образом:

* необходимо провести опрос среди потенциальных пользователей языка для выявления степени удобства использования синтаксиса.
* выполнить тестирование реализации языка.

Аппаратная платформа

Под тестируемым оборудованием автомобильной системы подразумевается электрический блок управления (ECU), представляющий собой набор плат. Для того, чтобы управлять данным блоком, необходимо наладить канал взаимодействия с ним. Для этого, ввиду отсутствия у платы интерфейсных выходов, используется внешний интерфейсный блок. В качестве интерфейсного блока выступает аппаратное обеспечение, разработанное компанией Vector, называемое VectorBox. Данный блок представляет собой набор интерфейсов для взаимодействия по протоколам CAN, FlexRay. За реализацию данных протоколов отвечает встроенное программно-аппаратное обеспечение данного устройства. Схема подключения тестируемого оборудования к персональному компьютеру представлена на рис

алгоритм тестирования ПО АС можно описать следующим образом:

1. на PC из тестовой инструкции извлекаются данные, необходимые для отправки на ECU;
2. данные преобразуются в кадры управления VectorBox'ом;
3. интерфейсный блок управления (VectorBox) генерирует управляющие сигналы и отправляет их на ECU;
4. после выполнения полученной инструкции ECU генерирует ответ, который отправляется на интерфейсный блок управления VectorBox;
5. после получения ответа от ECU VectorBox генерирует кадр и передает через интерфейс пользовательскому компьютеру (PC);
6. полученный кадр ответа транслирующая система сравнивает с ожидаемым значением, описанным в тестовой спецификации;

Разработка языка

Принимая АС в качестве черного ящика их можно исследовать, манипулируя входными данными и обрабатывая данные, полученные на выходе. В этом случае под данными понимаются кадры автомобильной сети, передающиеся на сетевом уровне модели OSI через стандартные протоколы (CAN, FlexRay).

В этом случае **входы** системы можно разделить на два класса: отправка данных, и настройка данных.

**Выходами** системы будут получаемые с тестируемого оборудования данные, которые могут приниматься единожды (UDS пакеты), или же периодически (синхросигналы).

# ANTLR4

В ходе разработки языка был определен следующий набор ключевых слов с которых должен начинаться любой шаг сценария: {Send}, {Set}, {Check}, {Receive}, {Pause}, {Repeat}. При этом ключевые слова должны начинаться с символа *коммерческое at(@)*, обозначающего начало служебного слово.

Для реализации языка был выбран генератор нисходящих анализаторов для формальных языков ANTLR4. Он преобразует контекстно-свободную грамматику в Расширенной форме Бэкуса-Наура в программу на Java.

***Нисходящий (англ. top-down) – это такой анализ, в котором продукции грамматики раскрываются, начиная со стартового символа, до получения требуемой последовательности лексем.***

***Нисходящий анализ (метод рекурсивного спуска) является наиболее эффективным методом синтаксического анализа. В его основе лежит левосторонний разбор строки языка. Исходной сентенциальной формой является начальный символ грамматики, а целевой – заданная строка языка. На каждом шаге разбора правило грамматики применяется к самому левому нетерминалу сентенции. Данный процесс соответствует построению дерева разбора цепочки сверху вниз (от корня к листьям).***

На слайде представлены терминальные символы (**лексемы**) для ключевых слов Repeat и Send, а так же правило вывода тестовой спецификации.

Любой сценарий должен начинаться с ключевого слова TestCase, после которого указывается краткая информация о нем.

Для описания шагов сценария необходимо указать ключевое слово, и текстовую инструкцию. Инструкции могут содержать параметры, заключенные в квадратные скобки.

# AST

Данный исходный код тестовой спецификации разработанный мной Лексический анализатор разобрал на токены, которые Синтаксический анализотор собрал в Абстрактное синтаксическое дерево, представленное на слайде.

Для наглядности красным выделены ключевые слова.

# Реализация языка

При разработке тестовых спецификаций пользователи Должны обговорить формулировку текстовых инструкций.

Затем на языке java сами определить методы, описывающие набор шагов, который необходимо выполнить для каждой инструкции, аннотируя каждый метод ключевым словом и указывая текст самой инструкции.

Названия реализованных аннотации соответствуют ключевым словам входных и выходных данных: Send, Set, Check и Receive.

Для использования, разработанного парсера ANTLR4 представляет возможность сгенерировать паттерн Listener (слушатель). Который предполагает анализ определенного подмножества узлов дерева разбора. Узлы дерева разбора, не являющиеся листьями, соответствуют каким-либо синтаксическим правилам грамматики.

Класс Listener’a содержит описания методов, которые позволяют выполнять действия, необходимые при обходе синтаксического дерева.

При обходе абстрактного синтаксического дерева интерпретатор находит очередную инструкцию. Затем помощью рефлексии определяет метод с соответствующей аннотацией и добавляет ее в список необходимых для выполнения, все параметры пользовательских параметров приводятся к типам, объявленным в параметрах метода. Методы инструкций могут принимать параметры следующих типов: boolean, byte, short, integer, long, float, double, String, Enum.

При завершении обхода AST инструкции поочередно выполняются в том порядке, в котором были заданы в тестовой спецификации.

**XL Driver Library**

Для реализации функционала симуляции автомобильных сетей используется универсальный программный интерфейс XL-Driver-Library, позволяющий получить доступ к интерфейсам аппаратных средств Vector.

Для разработки XL Driver Library приложений требуется подключить динамические библиотеки, которые находятся в открытом доступе на сайте компании Vector. Необходимые методы реализованы на языке С, поэтому для доступа к методам через java был реализован класс JNIVxlApi.java c использованием механизма Java Native Interface .

Java Native Interface (JNI) — стандартный механизм для запуска кода, под управлением виртуальной машины Java (JVM), который написан на языках С/С++ или Ассемблера, и скомпонован в виде динамических библиотек. Это даёт возможность вызова функции С/С++ из программы на Java, и наоборот.

Пользовательская библиотека наследует класс, который содержит методы, предоставляющие возможность использовать XL Driver Library.

Результаты выполнения тестовых сценариев записываются в текстовый файл results.txt.

# Обработчик ошибок

Для обработки ошибок используется класс ThrowingErrorListener.java, наследуемый от BaseErrorListener.java. BaseErrorListener.java предоставляет пустую имплементацию интерфейса ANTLRErrorListener. Реализация каждого метода по умолчанию ничего не делает, но может быть переписана в наследнике по мере необходимости. Для обработки синтаксических ошибок используется метод syntaxError, уведомляющий в какой строке и на какой позиции в строке возникла ошибка и ее причину.

# Тестирование интерпретатора

Для тестирования интерпретатора были разработаны Unit тесты. Модульное тестирование, оно же юнит-тестирование, позволяет проверить корректность отдельных модулей исходного кода программы. Для этого было реализовано junit-тестов с использованием пакета org.junit.

. Результаты приведены на слайде

# Описание полученных результатов

В качестве примера применения разработанного языка представлен пример тестового сценария для проверки функционала записи данных в энергонезависимую память. В качестве тестируемого оборудования использовался лидар.

Результаты теста, записанные в файл result.txt представлены на слайде. Из данного файла видно, что ожидаемый ответ от тестируемого оборудования о том, что данные успешно записаны (62 F1 01) был получен. Однако после перезапуска тестируемого оборудования ожидаемый ответ (62 F1 01 00 01 A5) на запрос о прочтении данных не был получен. Вместо этого тестируемое оборудование отправило ответ (7F 22 31), означающий, что данный сервис не поддерживается в текущей сессии.

# Результат опроса потенциальных пользователей

# Заключение

В результате проведенной работы был разработан и реализован язык описания сценариев тестирование программного обеспечения автомобильных систем.

Язык позволяет пользователям описывать инструкции для тестовых сценариев в свободной форме.

В одном документе можно описать несколько тестовых сценариев, выполняющихся поочередно в той последовательности, в которой были указаны в тестовой спецификации. Данные, полученные в результате выполнения теста сохраняются в отдельный файл.

Для удобства использования разработанного языка в дальнейшем стоит изменить формат вывода результатов и представлять их в качестве таблицы, или же в каком-либо другом формате. Так же стоит реализовать графический интерфейс для удобства пользователей, в котором в режиме реального времени могли бы выводиться результаты тестового прогона.