

Министерство образования и науки Российской Федерации  
Санкт-Петербургский политехнический университет Петра Великого  
Высшая школа программной инженерии



Работа допущена к защите

Директор ВШ ПИ

\_\_\_\_ П.Д. Дробинцев

" \_\_\_\_ " \_\_\_\_\_ 2019г.

## ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА БАКАЛАВРА

Разработка и реализация языка описания сценариев  
тестирования автомобильных систем

По направлению *09.03.01 «Информатика и вычислительная  
техника»*

по образовательной программе

*09.03.01\_06 «Распределенные информационные системы»*

Выполнил  
студент гр. 43504/6  
Руководитель  
д.т.н., проф.

А. А. Спасеева

А. В. Самочадин

Санкт-Петербург  
2019



САНКТ-ПЕТЕРБУРГСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ  
ПЕТРА ВЕЛИКОГО

Институт компьютерных наук и технологий

Утверждаю

Директор ВШ ПИ

\_\_\_\_\_ П.Д. Дробинцев

"\_\_" \_\_\_\_\_ 2019г.

ЗАДАНИЕ

по выполнению выпускной квалификационной работы  
студенту А. А. Спасеевой гр. 43504/6

1. Тема: *Разработка и реализация языка описания сценариев тестирования автомобильных систем*
2. Срок сдачи работы.
3. Исходные данные к проекту (работе).
4. Содержание расчетно-пояснительной записки (перечень подлежащих разработке вопросов).
5. Перечень графического материала с точным указанием обязательных чертежей.
6. Консультанты по проекту (с указанием относящегося к ним разделов проекта, работы).

Дата выдачи задания: \_\_\_\_\_ г.

Руководитель ВКР: \_\_\_\_\_ д.т.н., проф. А. В. Само-  
чадин

Задание принял к исполнению \_\_\_\_\_ г.

Студент \_\_\_\_\_ А. А. Спасеева

# Реферат

На 44 с., , рис. 7

Выпускная квалификационная работа посвящена разработке языка описания сценариев тестирования автомобильных систем. Проводится сравнительный анализ схожих по тематике инструментов.

Так же описываются принципы разработки языков и технических инструментов, выбранных для разработки.

Результат работы – язык описания сценариев тестирования автомобильных систем, являющийся потенциально пригодным для внедрения на производстве.

Ключевые слова: Предметно-ориентированный язык, функциональное тестирование, автомобильные системы.

# Abstract

44 pages , 7 figures

Final year graduation diploma dedicated to the development of the language designed for describing various automobile system testing scenarios. The comparable analysis of systems that share similar instruments is conducted.

Furthermore, this paper aims to describe the development of the programming languages as well as technical instruments used by them.

The final result of this work is a fully functional language used for describing automobile testing scenarios. Also, this language could potentially be integrated into the corporate environment.

Keywords: domain specific language, functional testing, automobile systems.

# Оглавление

<b>Список обозначений</b>	<b>7</b>
<b>Введение</b>	<b>8</b>
<b>1 Обзор литературы и постановка задачи</b>	<b>11</b>
1.1 Цель и задачи . . . . .	11
1.2 Обзор литературы . . . . .	12
1.3 Анализ существующих инструментов для тестирования	17
1.3.1 Выделение критериев сравнения . . . . .	17
1.3.2 Обзор аналогов . . . . .	17
1.3.3 Результаты . . . . .	20
1.4 Уточненные требования к работе . . . . .	21
<b>2 Теоретическая часть</b>	<b>22</b>
2.1 Тестирование автомобильных систем . . . . .	22
2.1.1 Симуляция работы сетей . . . . .	24
2.2 Описание синтаксиса языка . . . . .	25
2.3 Общая структура . . . . .	27
2.3.1 Лексический анализ . . . . .	28
2.3.2 Синтаксический анализ . . . . .	28
2.3.3 Семантический анализ . . . . .	29
<b>3 Реализационная часть</b>	<b>31</b>
3.1 Лексер . . . . .	31
3.2 Парсер . . . . .	35
3.3 Использование грамматики . . . . .	37
3.4 Обработчик ошибок . . . . .	39

<b>4 Экспериментальная часть</b>	<b>42</b>
<b>Заключение</b>	<b>44</b>

# Список обозначений

UML	Унифицированный язык моделирования
ECU	Electronic Control Unit
BDD	Behaviour-driven development
ПО	Программное обеспечение
TTD	Test-driven development
DSL	Domain-specific language
AUTOSAR	Automotive Open System Architecture
БНФ	Форма Бэкуса-Наура
РБНФ	Расширенная форма Бэкуса-Наура
AST	Abstract syntax tree
АС	Автомобильные системы



# Введение

С развитием автомобильной промышленности тестирование автомобильных систем стало неотъемлемой частью жизненного цикла разработки программного и аппаратного обеспечения. Современные автомобили включают множество встроенных систем для повышения уровня безопасности и комфорта водителей и пассажиров путем обеспечения функций, таких как адаптивный круиз-контроль, контроль давления в шине и др.

Современные автомобили имеют несколько блоков управления (electronic control unit, ECU), связанных между собой внутримашинной сетью. Эти блоки взаимодействуют между собой через стандартные шинные архитектуры CAN, FlexRay, LIN и Ethernet. С развитием техники количество ECU в автомобильных системах стремительно растет, что приводит к созданию сложных структур сетей. К основным особенностям автомобильного программного обеспечения можно отнести:

- надежность: в сложной сети ECU в течение всего срока эксплуатации автомобиля автомобильные программные системы должны работать исключительно надежно;
- функциональная безопасность: такие функции, как антиблокировочная тормозная система, требуют безотказной работы, что определяет высокие требования к процессам разработки программного обеспечения и к самим программам;
- работа в режиме реального времени: быстрая реакция (от микросекунд до миллисекунд) на внешние события требует оптимизированных операционных систем и особой программной архитектуры;

- минимальное потребление ресурсов: любое дополнение вычислительных ресурсов или памяти увеличивает стоимость продуктов, что при миллионных тиражах выливается в немалые деньги;
- надежная архитектура: автомобильное программное обеспечение должно выдерживать искажение сигналов и поддерживать электромагнитную совместимость;

Функциональные проверки безопасности автомобильных систем на программном уровне включают:

- Функциональное тестирование, целью которого является предоставление гарантий, что ПО удовлетворяет требованиям высокого и низкого уровней.
- Анализ времени выполнения худшего сценария, для того, чтобы гарантировать, что критические функции ПО выполняются достаточно быстро (такие как срабатывание подушек безопасности).
- Структурный анализ покрытия.

Что касается тестирования автомобильных систем, необходимо хорошо понимать, как устроен автомобиль в целом и его составляющие по отдельности. Однако представления о системе у инженеров и менеджеров зачастую очень сильно расходятся. Взаимодействие участников бизнес процесса, а именно заказчика, бизнес-аналитика, менеджера, разработчика и тестировщика является неотъемлемой частью гибкой методологии разработки ПО (Agile software development).

В данной работе рассматривается разработка языка для функционального тестирования ПО автомобильных систем таким образом, чтобы тестовая документация была понятна всем участникам бизнес цикла. Практика «разработка через поведение» (behaviour-driven development, BDD) была создана для достижения взаимопонимания между всеми участниками процесса. В сфере автомобильных систем BDD подход сильно облегчать процесс разработки.

Данная работа организована следующим образом. В главе 1 представлен обзор проблемы создания языка для описания тестовых сценариев тестирования автомобильных систем и обзор аналогов. Глава 2 содержит описание предлагаемого подхода к непосредственному построению языка. Вопросы его практической реализации рассмотрены

в главе 3. Результаты экспериментального исследования его эффективности приведены в главе 4.

# Глава 1

## Обзор литературы и постановка задачи

### 1.1 Цель и задачи

Целью выпускной квалификационной работы является разработка проблемно-ориентированного языка для функционального тестирования автомобильных систем, который облегчит взаимодействие между тестировщиком, заказчиком, менеджером, аналитиками, разработчиками и предоставит функционал, необходимый для симуляции работы автомобильных сетей. Такой язык позволит достичь большего взаимопонимания между менеджерами, аналитиками и инженерами, что в свою очередь значительно упростит разработку ПО и повысит качество разрабатываемого продукта. Для достижения поставленной цели следует выполнить ряд задач:

- Исследовать предметную область
- Выделить аналоги и сравнить их
- Разработать язык
- Реализовать язык
- Протестировать язык

## 1.2 Обзор литературы

Современный автомобиль обладает огромным количеством компьютеров, которые общаются между собой на разных скоростях и типах коммуникаций, так как призваны выполнять разные задачи. С очень быстрым внедрением электронных встроенных систем в транспортных средствах потребность отследить и управлять различными параметрами транспортного средства стала необходимостью. Таким образом были разработаны диагностические системы, чтобы клиенты (проектировщики, тестировщики и механики) могли обнаружить ошибки в транспортном средстве, с помощью соединения их диагностического оборудования с электронным блоком управления в транспортном средстве.

Бортовые системы современного автомобиля оборудованы электронным блоком управления с функцией самодиагностики, анализирующим основные системы автомобиля и оповещающим водителя об их неисправности. Для компьютерной диагностики электронный блок управления на физическом уровне подключается к стенду через специализированные диагностические интерфейсы, оснащенные датчиками, посредством которых данные о работе систем передаются для дальнейшего анализа на компьютер. На этом этапе удаляется собранная бортовой системой база данных, а также активируется управляющее устройство для сбора информации.

Одной из областей в автомобильном тестировании является диагностика транспортных средств, описанная в серии стандартов UDS ISO 14229. Unified Diagnostic Service (UDS) - автомобильный протокол, который позволяет диагностическим системам общаться с ECU, чтобы диагностировать ошибки и при необходимости повторно программировать ECU. Потребность в стандарте возникла так как производители транспортных средств собирают автомобильные компоненты у разных поставщиков. До возникновения UDS ISO 14229 поставщики должны были заниматься проблемами совместимости между различными диагностическими протоколами, сейчас же все ECU разрабатываются, придерживаясь данного стандарта. Архитектура UDS протокола разработана, основываясь на сетевой модели OSI (Open System Interconnection) и определяет пятый (сетевой уровень) и седьмой (прикладной уровень) модели.

Стек протокола UDS предлагает четыре важных категории сервисов:

- Загрузка данных в ECU для разрешения программных ошибок или добавления вновь разработанных программных модулей;
- Удаленная активация рутин. Диагностика транспортного средства может потребовать тестирования некорректного компонента в определенном диапазоне значений. Более того во время тестирования некоторые шаги могут выполняться в течении определенного времени. Для такого рода активностей используется Remote Routine Service, описанный в ISO 14229.
- Возможность передачи данных позволяет клиентам читать или записывать данные в ECU. Данные могут быть считанны или записаны в ячейки памяти, имеющие 16-битный адрес. Информация может быть статичной, как серийный номер устройства, и динамической, как скорость мотора или статус сенсоров. Допустимые значения данных и их длина могут варьироваться в зависимости от адреса ячейки памяти.
- Диагностика ошибок. Это один из самых важных сервисов UDS протокола. Когда в ECU возникает ошибка, DTC (Diagnostic trouble code), соответствующий возникшей ошибке сохраняется в определенной области памяти, называемой FCM (Fault Code Memory). Каждая ошибка имеет одно или несколько условий возникновения, обычно описываемые в требованиях.

Одно и то же оборудование, разрабатываемое для разных автомобильных компаний с различной внутренней архитектурой программного обеспечения при тестировании ПО может рассматриваться в качестве «черного ящика». Данное понятие предложил У.Р. Эшби в своей книге “ Введение в кибернетику”. Под черным ящиком подразумевается объект исследования, внутреннее устройство которого неизвестно. В электро-вычислительных системах оно помогает изучать поведение систем, абстрагируясь от их внутреннего устройства. Такое тестирование называется поведенческим. В этом случае проверяется функциональное поведение объекта с точки зрения внешнего мира. Под этой стратегией понимается создание тестов для тестового набора, основанных на технических требованиях и их спецификациях. Фактически, при поведенческом тестировании известно какой результат должен быть при определенном наборе данных, которые подаются на вход. Проанализировав требования и спецификации тестиров-



Рис. 1.1. Исследование поведения “черного ящика”

щик может легко определить набор тестовых сценариев для проверки различной функциональности тестируемой системы. Принимая тестируемую систему в качестве черного ящика ее можно исследовать, манипулируя входными данными и данными, полученными на выходе. Чтобы лучше понять, как исследуется оведение “черного ящика”, предположим что имеется некоторая система управления, внутреннее устройство которой неизвестно для пользователя. Эта система управления имеет входы  $X(x_1, x_2, \dots, x_n)$  и выходы  $Y(y_1, y_2, \dots, y_n)$ . Способ исследования поведения данной системы заключается в том, что выходы системы зависят от входов системы и времени  $t_k, k \in \mathbb{Z}$ . При этом величины времени выбираются произвольно (рис. 1.1) .

В марте 2006 года Dan North предложил BDD методологию (behaviour-driven development), основанную на agile методологиях. BDD является своего рода расширением техники разработки программного обеспечения через тестирование (test-driven development, TDD). Идея BDD подхода была в том, что в процессе написания требования для разрабатываемого ПО аналитики должны описать тестовые сценарии таким образом, чтобы их смог понять и разработчик, и тестировщик, и заказчик. При этом тестовые сценарии состоят из набора заранее обговоренных предложений. В своей статье Dan North

описал следующие BDD идеи:

- Название тестового метода должно быть предложением. В этом случае хорошо разработанная документация будет понятна и бизнес пользователям и инженерам.
- Простой шаблон делает тестовые методы более определенными.
- Выразительное название очень помогает, когда тест обрушился.
- Слово «поведение» более полезное, чем «тест».
- Методология BDD предоставляет «общий язык» для анализа.
- Приемочные критерии должны быть выполняемыми.

Почти в то же время Эрик Эванс опубликовал книгу «Проблемно-ориентированное проектирование», в которой описал набор принципов и схем, направленных на создание оптимальных систем объектов. Его идея заключается в том, что для бизнеса удобно смоделировать систему, в которой будет определен единый язык (Domain specific language), основанный на бизнес области, такой, что бизнес словарь смогут использовать как менеджеры так и инженеры.

Совместно Эрик Эванс и Dan North разработали шаблон для выявления приемочных критериев теста. Они разработали структуры тестовых сценариев, в которых каждый шаг определялся ключевым словом

- Given – дано начальное условие
- When – происходит событие
- Then – проверка, что получены некоторые результаты

На основе вышеперечисленных принципов BDD был создан язык Gherkin – человеко-читаемый язык, используемый для описания поведения системы.

С точки зрения автомобильной индустрии, в которых функциональное тестирование ПО подразумевает достаточно нетривиальную задачу данный подход сильно облегчает взаимодействие между всеми бизнес участниками. Однако не существует инструмента облегчающего разработку документации, а как следствия взаимопонимания



между менеджерами и инженерами и, в то же время, учитывающего особенности данной сферы.

Возможность использовать единую всеми участниками модель предметной области позволяет значительно ускорить процесс проектирования ПО. Предметно-ориентированное программирование (Domain-driven design, DDD) основано на трех главных определениях:

- Область (англ. domain, домен) — предметная область, к которой применяется разрабатываемое программное обеспечение.
- Модель (англ. model) — описывает отдельные аспекты области и может быть использована для решения проблемы.
- Язык описания — используется для единого стиля описания домена и модели.

Использование концепции моделирования системы с использованием единого языка, основанного на бизнес области таким образом, что словарь используется в разработке ПО может решить проблему взаимодействия участников бизнес процесса.

В 2003 году была создана Автомобильная Открытая Системная Архитектура (AUTOSAR, Automotive Open System Architecture). Это открытая архитектура, которая стандартизирует архитектуры ПО для автомобильных ECU и жизненный цикл разработки. Сегодня в Autosar входят более 150 компаний, и в рамках этого партнерства разрабатывается архитектура ECU, базовое программное обеспечение, методология и стандартизованные интерфейсы для прикладного программного обеспечения.

Учитывая, что все системы стандартизованы, одну и ту же систему (например, лидары для автомобилей марок Diamler и Audi) можно протестировать используя одни и те же инструменты. Однако, среди современных средств для тестирования программных систем нет специализированных под специфические нужды автомобильной промышленности, и при этом позволяющих разрабатывать спецификации на человеко-читаемом языке для реализации BDD подхода.

С расширением областей применения вычислительной техники возникла необходимость в новом — проблемно-ориентированном языке (domain-specific language, DSL), позволяющем в определенной области использовать специфичные обозначения и термины. Такие языки

обеспечивают пользователям возможность коротко и четко сформулировать задачу и получить результаты в необходимой для них форме.

Хорошо разработанный DSL язык должен быть основан на следующих принципах (Debasish, G. 2011 DSLs in Action, Manning Publications):

- Язык зеркально отображает артефакты предметной области.
- Язык должен использовать общий словарь предметной области. Словарь становится катализатором для лучшей связи между разработчиками и бизнес-пользователями.
- Имплементация языка должна быть абстракцией. В язык не должно быть точных фрагментов имплементации языка.

Использование проблемно-ориентированный языка позволит решить проблему разработки тестовой документации понятной всем участникам бизнес процесса с учетом специфики автомобильной промышленности.

## **1.3 Анализ существующих инструментов для тестирования**

### **1.3.1 Выделение критериев сравнения**

Требования к разрабатываемой системе

- Архитектура тестов, разрабатываемых с помощью языка, должна быть событийно-управляемая (Event based).
- Язык должен быть проблемно-ориентированным
- Язык должен позволять использовать BDD подход.

### **1.3.2 Обзор аналогов**

- CAPL (Communication Access Programming Language)

Компания Vector, разрабатывающая программные инструменты для работы с коммуникационными сетями, основывающихся на шинах CAN, LIN, FlexRay, Ethernet и др., используемыми в автомобильной промышленности создала программный

пакет для разработчиков: CANoe. Этот инструмент поддерживает симуляцию работы сетей, предоставляет диагностические инструменты и т.д. Данный пакет используется большинством OEM-производителей и поставщиков автомобильных компонентов. В среде CANoe есть возможность разрабатывать тестовые сценарии на языке CAPL (Communication Access Programming Language).

CAPL - процедурный язык, на котором выполнение блоков программы управляются событиями. Эти блоки программы упоминаются как событие процедуры.

С помощью CAPL можно описать тесты, полностью покрывающие функциональные требования АС. Однако данный язык является Си подобным, достаточно трудным для понимания менеджерами и аналитиками.

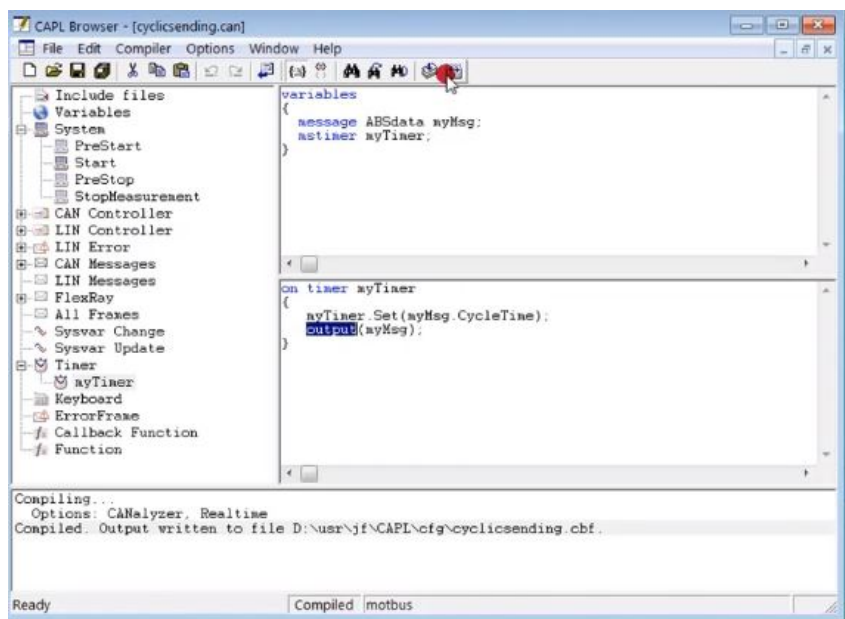


Рис. 1.2. Тестовая спецификация CAPL

- CCDL

Еще одним языком для описания тестовых спецификаций в автомобильной сфере является CCDL. Это язык тестовых спецификаций для тестирования, основанном на требованиях. Предоставляет высокоуровневый язык тестирования. CCDL может использоваться для автоматизированного black box тестирования. Однако, спецификации содержат в себе программный код, непонятный для части бизнес участников (рис. 1.3).

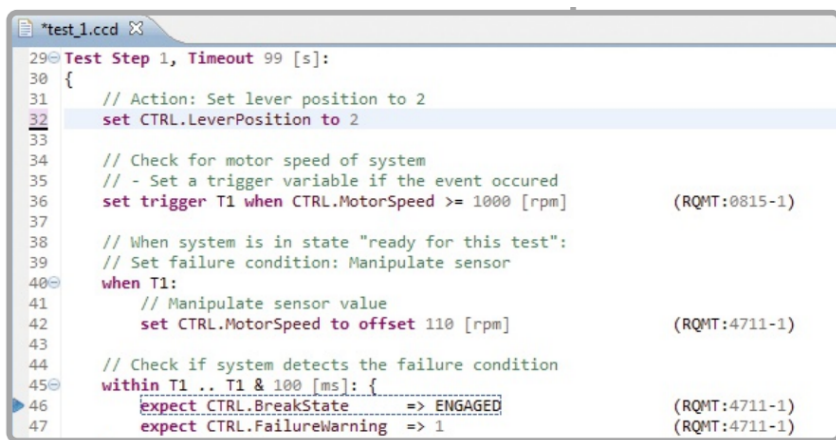
The image shows a screenshot of a code editor window titled "test1.ccdl". The code is written in a syntax-highlighted language for test specifications. It starts with a line "29 Test Step 1, Timeout 99 [s]:" followed by an opening curly brace. Line 31 is a comment: "// Action: Set lever position to 2". Line 32 is a command: "set CTRL.LeverPosition to 2". Line 33 is a comment: "// Check for motor speed of system". Line 34 is a comment: "// - Set a trigger variable if the event occurred". Line 35 is a command: "set trigger T1 when CTRL.MotorSpeed >= 1000 [rpm]" with a reference "(RQMT:0815-1)" on the right. Line 36 is a comment: "// When system is in state 'ready for this test:'". Line 37 is a comment: "// Set failure condition: Manipulate sensor". Line 38 is a command: "when T1:". Line 39 is a comment: "// Manipulate sensor value". Line 40 is a command: "set CTRL.MotorSpeed to offset 110 [rpm]" with a reference "(RQMT:4711-1)" on the right. Line 41 is a comment: "// Check if system detects the failure condition". Line 42 is a command: "within T1 .. T1 & 100 [ms]: {" with a reference "(RQMT:4711-1)" on the right. Line 43 is a command: "expect CTRL.BreakState => ENGAGED" with a reference "(RQMT:4711-1)" on the right. Line 44 is a command: "expect CTRL.FailureWarning => 1" with a reference "(RQMT:4711-1)" on the right. Line 45 is a closing curly brace. Line 46 is a closing curly brace. Line 47 is a closing curly brace.

Рис. 1.3. Тестовый сценарий на языке CCDL

- Cucumber

Cucumber – это фреймворк, реализующий подход BDD. В Cucumber для разработки тестов используется Gherkin-нотация. Она определяет набор ключевых слов и структуру теста. Пример тестового сценария, написанного с помощью Cucumber приведена на рис. 1.4 Программная реализация шагов отделена от теста (рис. 1.5), что очень удобно для бизнеса. Однако для создания полной domain specific модели АС необходимо каждый раз заново реализовывать протокол коммуникации с тестовым окружением.

```

# language: ru
@withdrawal
функция: Снятие денег со счета

@success
Сценарий: Успешное снятие денег со счета
    Дано на счете пользователя имеется 120000 рублей
    Когда пользователь снимает со счета 20000 рублей
    Тогда на счете пользователя имеется 100000 рублей

@fail
Сценарий: Снятие денег со счета - недостаточно денег
    Дано на счете пользователя имеется 100 рублей
    Когда пользователь снимает со счета 120 рублей
    Тогда появляется предупреждение "На счете недостаточно денег"

```

Рис. 1.4. Тестовая спецификация Cucumber

```

}

@Когда("^пользователь снимает со счета (\\d+) рублей$")
public void пользовательСнимаетСоСчетаРублей(int arg1) throws Throwable {
    // Write code here that turns the phrase above into concrete actions
    throw new PendingException();
}

```

Рис. 1.5. Реализация шага Cucumber

### 1.3.3 Результаты

Из вышеперечисленных инструментов для тестирования ПО видно, что ни один из них не удовлетворяет требованиям. Таким образом существует необходимость в создании domain specific языка для функционального тестирования, который объединит в себе принципы BDD и функционал, необходимый для симуляции работы автомобильных сетей CAN, LIN, Ethernet. Наличие такого языка значительно упростит разработку ПО и повысит качество разрабатываемого продукта.

## 1.4 Уточненные требования к работе

Окончательная постановка задачи с явным отсечением лишнего (чужого, нереализуемого и т.д.)

## Глава 2

# Теоретическая часть

Разрабатываемый язык описания сценариев тестирования автомобильных систем должен быть проблемно-ориентированным, при этом прост в использовании и изучении, а так же предоставлять возможность разрабатывать сценарии не только инженерами, но и бизнес-аналитиками.

### 2.1 Тестирование автомобильных систем

Прежде чем начать разрабатывать язык для описания сценариев тестирования автомобильных систем рассмотрим основные стандарты, используемые в транспортной промышленности.

Тестирование автомобильных систем предполагает, что поведение электрических блоков управления транспортного средства должно соответствовать стандарту UDS ISO 14229.

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, LIN и Ethernet).

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 одному или нескольким протоколам. За реализацию данных протоколов отвечает встроенное программно-аппаратное обеспечение данного устройства. Схема подключения тестируемого оборудования к персональному компьютеру представлена на рис. 2.1.

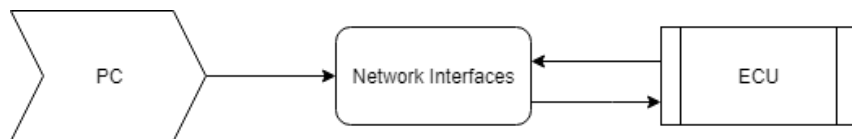


Рис. 2.1. Схема подключения тестируемого оборудования к PC

Алгоритм тестирования ECU можно описать следующим образом:

1. 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 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;

Поведенческое тестирование автомобильных систем может быть выполнено следующими способами:

- Разделение на классы эквивалентности



Часто легко заметить, что входные данные имеют схожие характеристики. В этом случае их можно сгруппировать и подавать на вход только одну единицу данных из каждой группы. Идея заключается в разделении входных данных на классы эквивалентности таким образом, что каждый член класса обладает схожими характеристиками.

Данный способ включает в себя два шага: определение класса эквивалентности и генерацию тестовых сценариев. Под определением класса эквивалентности подразумевается разделение входных данных на два класса: корректные и некорректные данные. Например, если действительный диапазон данных от 0 до 100, то в качестве корректных данных может быть выбрано значение 49, а в качестве некорректного 106.

- Анализ граничных значений

Тестирование граничных значения данных позволяют достаточно хорошо проверить качество программного обеспечения. Следовательно, если тестовые сценарии разработаны для граничных значений входной области, то эффективность тестирования повышается. Например, если действительный диапазон данных от 0 до 100, то в качестве корректных данных выбираются граничные значения 0 и 100 для корректных данных, а в качестве некорректных -1 и 101.

- Тестирование, основанное на требованиях

Данный способ тестирования подразумевает валидацию требований, предоставленных аналитиками и инженерами, которые составляют требования. Например, если в требовании описано, что сеть должна переходить в активный режим при получении любого кадра, то в качестве входных данных формируется любой корректный в данной сети кадр.

### 2.1.1 Симуляция работы сетей

рассказать о шине CAN в промышленной автоматизации, ethernet, fleahug? немного о их пакетах и что нужно для тестирования?

Основываясь на вышесказанном разрабатываемый язык должен предоставлять пользователям возможность получения передачи дан-

ных на сетевом уровне модели ISO по протоколам CAN, FlexRay, LIN и Ethernet.

## 2.2 Описание синтаксиса языка

Существуют три основных метода описания синтаксиса языков программирования: формальные грамматики, формы Бэкуса-Наура и диаграммы Вирта.

Формальной грамматикой называется четверка вида:  $G = (VT, VN, P, S)$ ,

где  $VN$  - конечное множество нетерминальных символов грамматики,  $VT$  - множество терминальных символов грамматики  $P$  - множество правил вывода грамматики,  $S$  - начальный символ грамматики.

Для записи правил вывода с одинаковыми левыми частями вида  $a \rightarrow b_1, a \rightarrow b_2, \dots, a \rightarrow b_n$  используется сокращенная форма записи  $a \rightarrow b_1 | b_2 | \dots | b_n$ .

Во второй половине 20-го века Джон Брэкус и Ноам Хомски независимо друг от друга создали форму записи, которая в последствии стала методом формального описания синтаксиса языков. Форма Бэкуса-Наура (БНФ) - формальная система описания синтаксиса, используемая для описания контекстно-свободных грамматик, в которой одни синтаксические абстракции последовательно определяются через другие абстракции. Для описания синтаксических структур форма БНФ использует абстракции.

Форма БНФ является порождающим устройством для определения языков. С помощью последовательности правил создаются предложения языка. Создание предложений называется выводом. Вывод должен начинаться с начального символа *start symbol*. Сентенциальная форма грамматики - это строка, которая может быть выведена из стартового символа. Предложение (сентенция) грамматики - это сентенциальная форма, состоящая только из терминальных символов. Язык  $L(G)$  грамматики - это множество всех ее предложений.

Метаязык, предложенный Бэкусом и Науром, использует следующие обозначения:

- символ « $::=$ » отделяет левую часть правила от правой (читается: «определяется как»);

- нетерминалы обозначаются произвольной символьной строкой, заключенной в угловые скобки «<» и «>»;
- терминалы - это символы, используемые в описываемом языке;
- правило может определять порождение нескольких альтернативных цепочек, отделяемых друг от друга символом вертикальной черты «|» (читается: «или»).

Из-за некоторых незначительных неудобств в БНФ Никлаус Вирт предложил свой вариант расширенной формы Бэкуса-Наура (РБНФ, расширенная Бэкус — Наурова форма). Эти расширения не увеличивают описательную силу БНФ, а упрощают чтение и использование такой формы.

Для повышения удобства и компактности описаний, в РБНФ вводятся следующие дополнительные конструкции (метасимволы):

- квадратные скобки «[» и «]» означают, что заключенная в них синтаксическая конструкция может отсутствовать;
- фигурные скобки «{» и «}» означают повторение заключенной в них синтаксической конструкции ноль или более раз;
- сочетание фигурных скобок и косой черты «{/» и «/}» используется для обозначения повторения один и более раз;
- круглые скобки «(» и «)» используются для ограничения альтернативных конструкций.

Синтаксическими графами называются ориентированные графы, на которых представляется информация о правилах форм БНФ и РБНФ. Так же их называют синтаксическими диаграммами или синтаксическими схемами. При построении диаграмм учитывают следующие правила:

- каждый графический элемент, соответствующий терминалу или нетерминалу, имеет по одному входу и выходу, которые обычно изображаются на противоположных сторонах;
- каждому правилу соответствует своя графическая диаграмма, на которой терминалы и нетерминалы соединяются посредством дуг;

- альтернативы в правилах задаются ветвлением дуг, а итерации - их слиянием;
- должна быть одна входная дуга (располагается обычно слева или сверху), задающая начало правила и помеченная именем определяемого нетерминала, и одна выходная, задающая его конец (обычно располагается справа и снизу);
- стрелки на дугах диаграмм обычно не ставятся, а направления связей отслеживаются движением от начальной дуги в соответствии с плавными изгибами промежуточных дуг и ветвлений.

## 2.3 Общая структура

Существует два главных вида языков программирования: компилируемый и интерпретируемый. Компилятор выясняет все, что должна выполнить программа, превращает инструкции в машинный код и сохраняет его, чтобы выполнить позже. Интерпретатор проходит всю программу строчку за строчкой и тут же выполняет.

Технически любой язык может быть, как компилируемым так и интерпретируемым. Обычно компилируемый язык выбирают, если в программе важна скорость работы. Интерпретируемый же язык более гибкий.

В случае разработки интерпретатора стадии разработки можно разбить на следующие этапы:

- Лексический анализ – разбор исходного кода на токены. Этот этап выполняется лексером.
- Синтаксический анализ – сбор токенов в абстрактное синтаксическое дерево (AST). Данный этап выполняется синтаксическим анализатором (парсером).
- семантический анализ;
- Исполнение AST

Естественные языки и искусственные, вне зависимости от их происхождения, представляют собой совокупность строк, состоящих из символа некоторого алфавита. Предложения или утверждения – это строки, состоящие из символов языка. Какие именно утверждения существуют в языке определяют синтаксические правила.

### 2.3.1 Лексический анализ

Первым шагом в разработке языков зачастую является лексический анализ – процесс аналитического разбора входной последовательности символов на лексемы, с целью получения на выходе идентифицированных последовательностей, называемых токенами. Токен – это малая единица языка. Токен может быть именем переменной или функции, оператором или числом.

Предполагается, что лексер берет входную строку, содержащую файлы с исходным кодом на разрабатываемом языке и разделяет его на список токенов. Лексер может выполнять такие задачи, как удаление комментариев, определение чисел и т.д.

ЛА необязательный этап, но желательный так как:

1) замена идентификаторов, констант, ограничителей и служебных слов лексемами делает программу более удобной для дальнейшей обработки.

2) Лексический анализатор уменьшает длину программы, устраняя из ее исходного представления комментарии и несущественные пробелы.

3) если будет изменена кодировка в исходном представлении программы, то это отразится только на лексическом анализаторе.

В процедурных языках лексемы обычно делятся на классы: служебные слова и ограничители.

Входные данные ЛА - текст транслируемой программы на входном языке.

Выходные данные ЛА - файл лексем в числовом представлении.

### 2.3.2 Синтаксический анализ

Синтаксический анализ – это процесс сопоставления линейной последовательности лексем (слов, фраз) языка с его формальной грамматикой. Результатом обычно является синтаксическое дерево. Обычно применяется совместно с лексическим анализом. Синтаксический анализатор— это программа или часть программы, выполняющая синтаксический анализ, то есть распознавание входной информации. При этом входные данные преобразуются к виду, пригодному для дальнейшей обработки. Этот вид обычно представляет собой формальную модель входной информации на языке последующего процесса обработки информации.

Во время парсинга входной текст преобразуется в структуру данных, которая отражает синтаксическую структуру входной последовательности и подходит для последующей обработки. Как правило синтаксическая структура представляется в виде дерева зависимостей.

Существуют следующие алгоритмы синтаксического анализа:

- Нисходящий (англ. top-down) — это такой анализ, в котором продукции грамматики раскрываются, начиная со стартового символа, до получения требуемой последовательности лексем.
- Восходящий (англ. bottom-up) — это такой анализ, в котором продукции восстанавливаются из правых частей, начиная с токенов-лексем и кончая стартовым символом.

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).

### 2.3.3 Семантический анализ

В ходе семантического анализа проверяются отдельные правила записи исходных программ, которые не описываются КС-грамматикой. Эти правила носят контекстно-зависимый характер, их называют семантическими соглашениями или контекстными условиями.

Пользовательские инструкции, используемые в тестовых сценариях должны быть заранее объявлены пользователями в специальной библиотеке. Ключевое слово должно отражать смысл инструкции: Send- для отправки пакета, Recieve - для получения пакета и т.д.

В теле метода соответствующей инструкции должны быть реализованы шаги отправки, получения пакетов и т.д. в зависимости от аннотации. Для того, чтобы определить метод, который должен быть выполнен при вызове заданной текстовой инструкции, а так же при-

ведения типов пользовательских параметров можно использовать рефлексию.

Рефлексия (от лат. *reflexio* — обращение назад) — это механизм исследования данных о программе во время её выполнения. Рефлексия позволяет исследовать информацию о полях, методах и конструкторах классов.

Механизм рефлексии позволяет обрабатывать типы, отсутствующие при компиляции, но появившиеся во время выполнения программы. Возможность использования рефлексии реализована в Java с помощью `Reflection API`.

## Глава 3

# Реализационная часть

Задача построения языка довольно трудозатратная. Однако существуют программные инструменты, помогающие сделать его меньше и проще. Для этого был разработан следующий инструментарий для создания компиляторов: генераторы лексических анализаторов (сканеров), генераторы синтаксических анализаторов (парсеров), автоматические генераторы кода.

Для реализации языка было выбрано «еще одно средство распознавания языков» ANTLR4 (ANOther Tool for Language Recognition). Это генератор нисходящих анализаторов (парсеров) для формальных языков. Он преобразует контекстно-свободную грамматику в форме РБНФ в программу на Java, C++, JavaScript, Go. Python и Go.

ANTLR4 удобен для работы с AST, является свободным программным обеспечением, предоставляет сообщение об ошибках и восстановление после них, а так же предоставляет плагины для Eclipse и IntelliJ IDEA, что позволяет удобно создавать и отлаживать грамматики. Интерпретатор реализован на языке java.

### 3.1 Лексер

Предметно-ориентированный язык для описания тестовых сценариев автомобильных систем должен должен позволять описывать следующие виды инструкций, необходимых для описания входных и выходных данных:



- Отправить фрейм, сигнал или набор данных по UDS протоколу на тестируемое ПО
- Получить фрейм, сигнал или набор данных по UDS протоколу от тестируемого ПО
- Сделать паузу между инструкциями, чтобы тестируемое ПО успело сформировать и отправить необходимый пакет
- Выполнять шаги определенное количество раз, т.к. многие системы имеют алгоритмы со счетчики, которые можно выполнить определенное количество раз. Зачастую значения таких счетчиков измеряются сотнями и даже тысячами шагов. При удачной группировке этих шагов количество строк тестового сценария можно сократить в десятки раз, поместив их в цикл.
- Выполнять определенные действия по событию (триггер).
- Установить значение отправляемого сигнала
- Проверить полученное значение сигнала

При этом команды, состоящие из аналитиков, тестировщиков и разработчиков должны сами определять набор инструкций, так как в процессе разработки автомобильного ПО могут выявляться свойства, специфичные для того или иного программного продукта. У пользователей должна быть возможность описать сценарии с использованием специфичных терминов (рычаги у рулевой колонки, расстояние у лидара и т.д). Предполагается, что пользователи сами определяют ограниченный набор шагов, необходимых для описания тестовых сценариев, а тестировщик набор шагов для каждой инструкции.

Допустим, пользователи определили следующий ограниченный набор команд: «Отправить запрос ‘запрос’», «Получить ответ ‘ответ’». При этом формат запроса и ответа является проектно-специфичным, а значит при описании любого шага должна быть возможность описывать кастомизированные параметры. Добиться этого можно, выделив параметр в отдельную лексему.

Предположим, что синтаксис разрабатываемого языка будет иметь ключевые слова, по которым можно определить входные и выходные данные, возможность описывать сценарии на языке, привычном пользователю, будь то русский или английский.

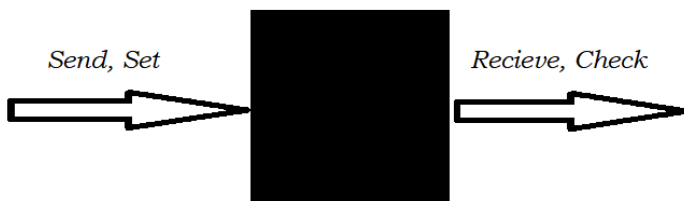


Рис. 3.1. Тестирование автомобильной системы в качестве “черного ящика”

В качестве входов системы рассмотрим пакеты данных, отправляемые тестируемому оборудованию, а так же установку значений параметров (частота передачи данных, значения отправляемых сигналов и д.р.). В этом случае входы системы можно разделить на два класса: отправка данных и настройка данных. В таком случае выходами системы будут получаемые с тестируемого оборудования пакеты данных. Данные могут приниматься единожды (UDS пакеты), или же периодически (синхросигналы). Соответственно, представляется логичным разделить их на две большие группы: получение пакетов и данные, которые необходимо проверить в определенный момент времени. В этом случае система управления имеет входы  $X(Send_{data}, Set_{data})$  и выходы  $Y(Recieve_{data}, Check_{data})$ . На рис. 3.1 в качестве входных и выходных данных “черного ящика” представлены ключевые слова, которыми назовем четыре группы вышеописанных данных.

Так как автомобильные системы тестируются в режиме реального времени, в тестовых сценариях необходимо явно оперировать временем. Ответ на запрос приходит не моментально, так же как и не моментально изменяются значения сигналов. Зачастую возникает необходимость проверить значение полученного сигнала несколько раз в

определенные промежутки времени. В этом случае между отправкой запроса и получением ответа необходимо установить паузу, в течение которой тестовое оборудование не будет выполнять проверку полученных данных на соответствие ожидаемым данным, описанные в тестовом сценарии. Для реализации этой возможности введем ключевое слово `Pause`. После данного ключевого слова должно указываться значение  $p_n$  такое, что  $t_{n+1} = t_n + p_n$ .

Из вышеуказанного можно определить следующий набор ключевых слов с которых должен начинаться любой шаг сценария: `Send`, `Set`, `Check`, `Receive`, `Pause`. Реализация данных лексем представлена в листинге ???. При этом для удобства пользователя ключевые слова языка должны быть регистронезависимы. К сожалению, в ANTLR нет поддержки регистронезависимых токенов, и для токенов приходится использовать следующую запись с использованием фрагментных токенов, которые используются для построения реальных токенов:

```
fragment S: [sS];
```

Листинг 3.1. Ключевые слова

```
Send
    : DOG [Ss][e][n][d]
    ;
Receive
    : DOG [Rr][e][c][i][e][v][e]
    ;
Set
    : DOG [Ss][e][t]
    ;
Check
    : DOG [Cc][h][e][c][k]
    ;
Pause
    : DOG [Pp][a][u][s][e]
    ;
```

`listKeyWords`

После шага должна идти инструкция, определяемая пользователем языка. Например, если необходимо проверить функцию записи определенных данных в энергонезависимой памяти, нужно реализовать следующий набор шагов тестового сценария:

- Включить тестовое оборудование
- С помощью сервиса UDS протокола WriteDataByIdentifier (0x2E) записать корректные данные
- С помощью сервиса UDS протокола ReadDataByIdentifier (0x22) проверить, что данные успешно записаны
- Выключить тестируемое оборудование
- Включить тестируемое оборудование
- С помощью сервиса UDS протокола ReadDataByIdentifier (0x22) прочесть записанные данные
- Выключить тестируемое оборудование

Предположим, что для области памяти с адресом 0xF101 корректными данными являются три байта 0x0001A5. В этом случае, в соответствии с принципами BDD набор тестовых инструкций, определяемых пользователем могут выглядеть следующим образом:

**Set** ignition on, **Send** request [22 F1 01 00 01 A5], **Recieve** response [62 F1 01], **Set** ignition off, **Set** ignition on, **Send** request [2E F1 01], **Pause** 100 ms, **Recieve** response [6E F1 01 00 01 A5], **Set** ignition off.

Такая запись не очень удобна с точки зрения восприимчивости теста. Поэтому разумно ввести правило, что каждое правило должно начинаться с новой строки. Для обозначения новой строки введем лексему **NewLine**. Подробнее правила синтаксического анализа будут рассмотрены в разделе 3.2. Более того, нет обозначения начала тестового сценария. Так же в тестовой спецификации должна быть краткая информация для каждого сценария, чтобы можно было быстро понять зачем нужно данное испытание. Для этого введем ключевое слово **TestCase**, обозначающее начало сценария, после которого должна идти краткая информация. Шаги группируются в тестовые сценарии, которых в тестовой спецификации может быть несколько.

## 3.2 Парсер

Для описания синтаксической структуры языка нужно определить порядок записи:

- Предложений в тексте;
- Фраз в предложении;
- Лексем и фраз в более общих фразах.

Для разрабатываемого языка необходимы такие синтаксические правила как текст инструкции и кастомизированный параметр. При этом текст может содержать пробелы, любые символы английского и русского языков, а так же любые другие символы, кроме символов, обозначающих начало и конец кастомизированного параметра, а так же служебный символ «@», обозначающий начало ключевого слова.

Синтаксические правила одного шага можно описать следующим образом:

- Шаг должен начинаться с ключевого символа, определяемого служебным символом “@”
- Шаг не может начинаться с пользовательского параметра
- Шаг может начинаться с любого символа
- Шаг может иметь сколь угодно много параметров

TODO: описать синтаксические правила шагов и сценария в целом

По описанным правилам ANTLR генерирует файлы для распознавания. `BddParser.java` - это описание класса парсера, то есть синтаксического анализатора, отвечающего грамматике `BddParser`. `BddLexer.java` - это описание класса лексера, или лексического анализатора, отвечающего грамматике

`BddParser.tokens`, `BddLexer.tokens` — это вспомогательные классы, которые содержат информацию о токенах

`BddParserVisitor.java`, `BddParserListener.java`, `BddParserBaseListener` — это интерфейсы и классы, содержащие описания методов, которые позволяют выполнять определенный действия при обходе синтаксического дерева

Вопрос: вставлять ли картинки с построенными ANTLR ast?

### 3.3 Использование грамматики

Для использования разработанного парсера ANTLR4 представляет возможность сгенерировать два паттерна проектирования: Visitor (посетитель) и Listener (слушатель). Каждый из них предполагает анализ определенного подмножества узлов дерева разбора. Узлы дерева разбора, не являющиеся листьями, соответствуют каким-либо синтаксическим правилам грамматики. При анализе узлов дерева разбора нужно обращаться к дочерним узлам, соответствующим фрагментам исходного правила. Причем обращаться можно как к отдельным узлам, так и к группам узлов. Следовательно важным условием создания хорошей грамматики является возможность интуитивно простой доступ к любой части правила. ANTLR4 предоставляет такие сущности как альтернативные и элементарные метки. Альтернативные метки позволяют разбить сложное правило на альтернативные фразы и обрабатывать каждую фразу отдельно. В нашем случае правило инструкции можно разбить на следующие альтернативные метки:

Листинг 3.2. альтернативные метки синтаксического правила  
instruction

```
instruction
: Send annotationText      #send
| Recieve annotationText   #recieve
| Set annotationText       #set
| Check annotationText     #check
| Pause time TEXT?        #pause
```

Элементными метками помечаются отдельные нетерминалы или последовательности терминалов. Они предоставляют доступ к содержимому контекста правила в виде поля с заданным именем. Таким образом, вместо вычисления (извлечения) отдельного элемента содержимого некоторого контекста достаточно просто обратиться к такой элементной метке.

В реализации интерпретатора используется паттерн Listener. При обходе абстрактного синтаксического дерева интерпретатор находит очередную нотированную инструкцию, с помощью рефлексии находит метод, аннотированный соответствующим ключевым словом и добавляет ее в список необходимых для выполнения. При этом инструкция должна быть заранее определена пользователем в библиотеке инструкций. Во время семантического анализа все параметры пользова-

тельских 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.

Листинг 3.3. Пример реализации пользовательской инструкции с аннотацией `Send`

```
@Send(text = "request_[p1]_in_[p2]_addressing_mode")
public void sendRequest(String p1, AddressingMode p2){
    //do something
}
```

Для реализации функционала симуляции автомобильных сетей используется `XL-Driver-Library`. `XL-Driver-Library` – это универсальный программный интерфейс, позволяющий получить доступ к интерфейсам аппаратных средств `Vector`. Он поддерживает следующие шины:

- CAN / CAN FD
- LIN
- FlexRay
- Automotive Ethernet
- MOST
- ARINC

`XL-Driver-Library` предоставляет общие и шинно-специфичные методы, которые облегчают управлять интерфейсами шины от `Vector`. Каналы и порты управляются общими методами. Шинно-специфичные методы используются, чтобы настроить сетевые узлы и послать или получить сообщения. `XL-Driver-Library` позволяет эффективно использовать интерфейсы шины в пользовательских приложениях. Особенно полезно это при реализации специализированных инструментов, которые адаптированы к автоматизированному рабочему месту и его окружению, с целью увеличения производительности. По словам производителей данная библиотека может быть использована при создании инструментов тестирования для тестового обслуживания автомобильных систем.

Для разработки XL Driver Library приложений требуется подключить динамические библиотеки, которые находятся в открытом доступе на сайте компании Vector. необходимые методы реализованы на языке C, поэтому для доступа к методам через java был реализован класс JNIVxlApi.java с использованием механизма JNI.

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

Ниже представлена диаграмма классов интерпретатора.

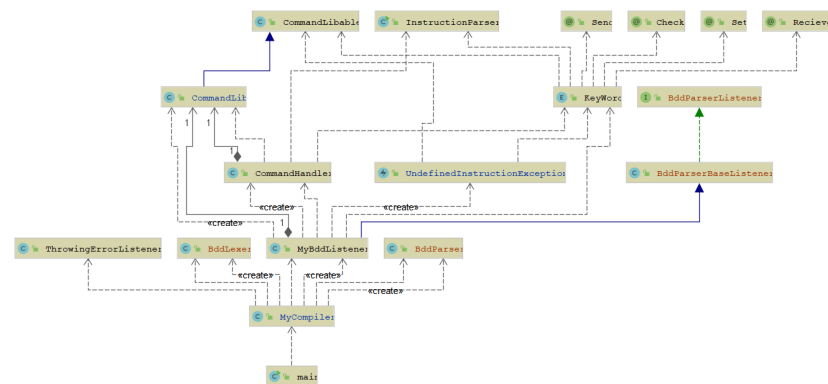


Рис. 3.2. Диаграмма классов

## 3.4 Обработчик ошибок

Важной способностью каждого парсера является обработка ошибок. В случае, если исходный текст не соответствует синтаксическим правилам языка, необходимо корректно реагировать на некорректную цепочку лексем. В этом случае можно завершить синтаксический анализ и вывести сообщение об ошибке, либо попробовать за одну попытку синтаксического анализа найти как можно больше ошибок.

В ANTLR существуют следующие типы ошибок парсинга:



- ошибка распознавания токена (Lexer no viable alt); единственная существующая лексическая ошибка, обозначающая отсутствие правила для формирования токена из существующей лексемы.
- отсутствующий токен (Missing token); в этом случае ANTLR вставляет в поток токенов отсутствующий токен, помечает, что его не хватает, и продолжает парсинг.
- лишний токен (Extraneous token). Генератор помечает, что токен ошибочный и продолжает парсинг дальше.
- несовместимая входная цепочка (Mismatched input). При этом включается «режим паники», цепочка входных токенов игнорируется, а парсер ожидает токена из синхронизирующего множества.
- отсутствующая альтернатива (No viable alternative input). Данная ошибка описывает все остальные возможные ошибки парсинга.

Для обработки ошибок используется класс `ThrowingErrorListener`, наследуемый от `BaseErrorListener`. `BaseErrorListener` 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 в на какой позиции в строке возникла ошибка и ее причину.

Ниже представлен пример тестового сценария с синтаксической ошибкой: на 9 строке после ключевого слова не указана текстовая инструкция. В этом случае `ThrowingErrorListener` должен сообщить о синтаксической ошибке (листинг 3.4)

Листинг 3.4. Пример тестового сценария с синтаксической ошибкой

```
@TestCase [1]
@Send request [00 AB A5] in [functional] addressing mode
@Send request to read did ECU_Serial_Number
@Check signal [RM_LDS_AS_1] is [equal] to [0.0]
@Send request [33 33 33 ]
@Set signal LSD_1 to 1
@Set Signal [LDS_AS_1]
```

```
@Pause [2] ms
@Set
@Pause [0] ms
```

Листинг 3.5. Результат работы интерпретатора для тестового сценария с синтаксической ошибкой

```
Error in the Specification example.cc:
line 9:4 missing TEXT at '\r\n'
```

В случае, если на вход подается тестовая спецификация, в которой указана инструкция, не объявленная в пользовательской библиотеке команд, возникает исключение `UndefinedInstructionException`, сообщающее какая именно инструкция была неопределена.

Листинг 3.6. Пример тестового сценария необъявленной инструкцией

```
@TestCase [1]
@Send request [00 AB A5] in [functional] addressing mode
@Send request [00 AB A5] in functional addressing mode
```

Листинг 3.7. Результат работы интерпретатора для тестового сценария с необъявленной инструкцией

```
Error in the Specification example.cc:
Instruction 'request_[]_in_functional_addressing_mode' is und
```

## Глава 4

# Экспериментальная часть

Для тестирования реализованного языка были разработаны Unit тесты. Модульное тестирование, оно же юнит-тестирование, позволяет проверить корректность отдельных модулей исходного кода программы.

Листинг 4.1. входные данные для теста  
testTwoInstructionsInTheSameLine

```
TestCase 1
Send request [00 AB A5] in [functional] addressing mode
Send request to read did ECU_Serial_Number
Check signal [RM_LDS_AS_1] is [equal] to [0]
Check signal [RM_LDS_AS_1] is [equal] to [0]
Send request [33 33 33 ]
Set signal LSD_1 to 1
Set Signal [LDS_AS_1] Pause [2] ms
```

Листинг 4.2. входные данные для теста testSetEmptyInstruction

```
TestCase 1
Send request [00 AB A5] in [functional] addressing mode
Send request to read did ECU_Serial_Number
Check signal [RM_LDS_AS_1] is [equal] to [0.0]
Send request [33 33 33 ]
Set signal LSD_1 to 1
Set Signal [LDS_AS_1]
```

Входной текст	Ожидаемое поведение	Статус
@Pause [erf] ms	line 1:8 mismatched input 'erf' expecting INTEGER	выполнено
@Pause [] ms	line 1:8 missing INTEGER at ']'	выполнено
@Pause [2] ms	line 2:8 missing INTEGER at ']'	выполнено
@Pause [] ms		выполнено
@Pause [2] ms	line 2:6 missing TEXT at '<EOF>'	выполнено
@Send		
@Pause [2] ms	line 2:6 mismatched input '@Pause' expecting TEXT	выполнено
@Send @Pause [2] ms		

Pause [2] ms

Set

Pause [0] ms

Ниже представлена спецификация для разработанного синтаксического анализатора и результаты выполнения тестов

В результате тестирования выявлено, что интерпретатор работает корректно.

# Заключение