**МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ**

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ

«ПОВОЛЖСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ»

Факультет информатики и вычислительной техники

Кафедра ИиСП

**ВКР по теме**

**«Разработка нейросетевого игрового AI для военной стратегии на дорожном графе»**

Выполнил:

студент

группы ПСм-21

Колчин И.А.

Проверил:

к.т.н., доцент

Егошин А.В.

г. Йошкар-Ола

2019

Оглавление

[Актуальность 4](#_Toc18671111)

[Почему этим нужно заниматься 4](#_Toc18671112)

[Недостатки существующих решений и предлагаемые решения 4](#_Toc18671113)

[Постановка задачи 5](#_Toc18671114)

[Научная новизна 5](#_Toc18671115)

[Аналитическая часть 6](#_Toc18671116)

[Обзор исследований 6](#_Toc18671117)

[Бот компании OpenAI игры Dota 2 6](#_Toc18671118)

[Бот компании OpenAI играющий в сумо и футбол 6](#_Toc18671119)

[Per-Arne Andersen, Morten Goodwin, Ole-Christoffer Granmo «Deep RTS: A Game Environment for Deep Reinforcement Learning in Real-Time Strategy Games» 6](#_Toc18671120)

[Kun Shao, Yuanheng Zhu, Member, IEEE and Dongbin Zhao, Senior Member «StarCraft Micromanagement with Reinforcement Learning and Curriculum Transfer Learning» 6](#_Toc18671121)

[Hendrik Baier, Peter I. Cowling «Evolutionary MCTS for Multi-Action Adversarial Games» 6](#_Toc18671122)

[AlphaStar: Mastering the Real-Time Strategy Game StarCraft II 6](#_Toc18671123)

[Конструкторская часть 9](#_Toc18671124)

[Используемые методологии разработки и технологии 9](#_Toc18671125)

[Test-driven development 9](#_Toc18671126)

[C4Model 11](#_Toc18671127)

[Нотация ER-диаграмм Crow’s foot 15](#_Toc18671128)

[UML 19](#_Toc18671129)

[Используемые программные продукты и языки программирования 26](#_Toc18671130)

[Язык JavaScript 26](#_Toc18671131)

[Список литературы 28](#_Toc18671132)

[Приложение А. Техническое задание. 29](#_Toc18671133)

[Введение 29](#_Toc18671134)

[Назначение разработки 29](#_Toc18671135)

[Основание для разработки 29](#_Toc18671136)

[Требования к программе или программному изделию 29](#_Toc18671137)

[Требования к программной документации 30](#_Toc18671138)

[Стадии и этапы разработки 30](#_Toc18671139)

[Порядок контроля и приемки 30](#_Toc18671140)

[Приложение Б. Программная спецификация. 31](#_Toc18671141)

[Документация 31](#_Toc18671142)

[Компоненты 31](#_Toc18671143)

[Приложение В. Программная документация. 32](#_Toc18671144)

[Диаграмма контекста 32](#_Toc18671145)

# Актуальность

## Почему этим нужно заниматься

Адекватный игровой искусственный интеллект имеет большое значение в современных военных стратегиях, т.к. зачастую игроку очень сложно найти другого игрока для долгой совместной игры.

Ещё большую сложность представляет разработка ИИ, способного адекватно управлять армией из разнородных юнитов, передвигающихся по графу реальных дорог, учитывая множество условий: рельеф местности, ландшафт, время суток и время года, тактико-технические характеристики техники и вооружений, снабжение по дорогам и т. д. Программирование поведения армии, способной нанести поражение игроку-человеку без численного перевеса на основе жёстко заданных алгоритмов сложная задача, кроме того к такому виду ИИ гораздо проще подобрать стратегию, с помощью которой можно всегда выигрывать, такие стратегии называются доминантными.   
 Перспективным видится использование ИНС, способных к «самообучению». Искусственная нейронная сеть(ИНС, в тексте используется сокращение нейросеть) - математическая модель, а также её программное или аппаратное воплощение, построенная по принципу организации и функционирования биологических нейронных сетей — сетей нервных клеток живого организма. ИНС представляет из себя набор нейронов, которые представляют из себя функции с несколькими входными переменными, которые передают полученный результат следующему нейрону или наружу. ИНС является ИИ, который представляет из себя самообучающуюся программу. Программы такого типа разрабатываются в рамках машинного обучения.

Реализация игрового интеллекта на базе такой технологии в теории позволит обучить программу не только на играх с человеком (что очень медленно), а на множестве ускоренных по времени играх с самим собой. Также при достаточно длительном обучении можно получить ИИ к которому не удастся подобрать доминантную стратегию и поэтому игроку придётся лучше думать, но это также не означает что будет создан непобедимый ИИ.

## Недостатки существующих решений и предлагаемые решения

Рассмотренные далее, решения хорошо справляются с задачей управления на графах похожих на «шахматную доску», но они не могут работать с графом дорог(картой дорог). Графы вида «шахматная доска» похожи на графы дорог, но главное отличие в том, что они не такие плотные и разветвлённые, и в качестве вершин у них клетки вместо мест стыковок дорог.

В рамках диссертационной работы будет разработана нейросеть, которая будет управлять различными видами войск на дорожном графе. Нейросеть будет управлять следующими типами войск: пехота, артиллерия, мотострелковые подразделения, танки, конвои грузовиков для снабжения войск и поезда для переброски на дальние расстояния. У каждого подразделения имеется запас сил, боеприпасов, еды, воды, кроме этого при ведении боя учитываются боевые характеристики(боевой дух, боевой опыт, дальность атаки, скорость передвижения). Все эти характеристики и особенности графа дорог(множество путей, разветвлённость и плотность дорог) будут учитываться нейронной сетью при достижении цели.

## Постановка задачи

Целью данного исследования является исследование применения современных технологий машинного обучения для решения задачи управления мультиагентной системой(виртуальной армией) в многопараметрической системе на графе дорог. Для того, чтобы выполнить эту цель необходимо разработать нейронную сеть, которая будет решать описанную ранее задачу.

## Научная новизна

Разработанная нейронная сеть будет первой нейронной сетью решающая задачу на дорожном графе. Результаты эффективности полученной нейронной сети позволят понять возможности «обучения с подкреплением» в задачах на дорожном графе, что поможет использовать «обучения с подкреплением» в других, возможно более сложных, задачах на дорожных графах.

# Аналитическая часть

## Обзор исследований

### Бот компании OpenAI игры Dota 2

### Бот компании OpenAI играющий в сумо и футбол

### Per-Arne Andersen, Morten Goodwin, Ole-Christoffer Granmo «Deep RTS: A Game Environment for Deep Reinforcement Learning in Real-Time Strategy Games»

### Kun Shao, Yuanheng Zhu, Member, IEEE and Dongbin Zhao, Senior Member «StarCraft Micromanagement with Reinforcement Learning and Curriculum Transfer Learning»

### Hendrik Baier, Peter I. Cowling «Evolutionary MCTS for Multi-Action Adversarial Games»

### AlphaStar: Mastering the Real-Time Strategy Game StarCraft II

Машинное обучение успешно применяется в играх разных жанров. Например, компания OpenAI создала бота для игры Dota 2[1], который победил профессиональных игроков. Представленный бот обучался с нуля, играя сам с собой, постепенно он смог научиться адекватно реагировать на нестандартные ситуации, что и помогло ему обыграть профессиональных игроков.

Кроме этого бота были разработаны боты для забивания голов, защиты футбольных ворот и игры в сумо[2]. Разработанные боты обучались на агентах с различным поведением, чтобы получить более универсальных ботов, также это позволило избежать переобучения. Авторы также протестировали методику трансферного обучения на сумо-боте, которому дали задачу оставаться в центре ринга при наличии ветра, который дул со случайной стороны и различной силой. Суть трансферного обучения заключается в том, что для создания нового слоя нейронов мы берём копию другого слоя, который выполнял похожую задачу, и обучаем его. В результате сумо-бот хорошо справился с задачей, несмотря на то, что с ветром до этого он не сталкивался.

На данный момент наибольших успехов в применении машинного обучения в задачах управления войсками добились: Per-Arne Andersen, Morten Goodwin, Ole-Christoffer Granmo создав игру Deep RTS для исследования технологий машинного обучения[3]; Kun Shao, Yuanheng Zhu и Dongbin Zhao в своей работе использовали постепенное трансферное обучения для обучения нейросетей управлению боевыми единицами в игре StarCraft[4]; Hendrik Baier, Peter I. Cowling использовали эволюционный вариант алгоритма Монте-Карло для более быстрого обучения нейросети в пошаговой стратегии Hero Academy[5]; компания Deepmind создала нейросеть под названием «AlphaStar» для игры Starcraft 2, которая смогла обыграть двух профессиональных игроков со счётом 5:0[6];

Deep RTS[3] - это высокопроизводительная RTS-игра, созданная специально для исследований в области искусственного интеллекта. В данной стратегии игроку нужно добывать ресурсы, строить базу и армию для победы. Она поддерживает ускоренное обучение, что означает, что она может учиться в 50 000 раз быстрее по сравнению с существующими RTS играми. Такого эффекта разработчики добились за счёт того, что Deep RTS использует краткосрочные конфигурации при обучении и имеет возможность настраивать игровой таймер. Действия в краткосрочной конфигурации непосредственно применяются к среде в течение следующих нескольких игровых кадров. Это делает корреляцию между действием и вознаграждением более наблюдаемой. Долгосрочная конфигурация значительно усложняет пространство состояний, потому что чем больше временной промежуток, тем больше возможных вариантов событий может произойти. Игровой таймер включает в себя множитель, который позволяет регулировать количество тиков, равное секунде, что позволяет, по сути, ускорять или замедлять время в игре, а следовательно и влиять на скорость обучения.

Kun Shao, Yuanheng Zhu и Dongbin Zhao в своей работе[4] использовали комбинацию двух подходов трансферного обучения(Transfer Learning) и постепенного обучения(Curriculum Learning) для более быстрого обучения нейронной сети, которая управляла армией в игре Starcraft. Применение трансферного обучения позволило обучить новые слои нейросетей обучается в 10 раз быстрее, чем если бы они создавали их заново(300 тренировок против 3000). Постепенное обучение представляет из себя обучение определённой последовательности постепенно усложняющихся задач, которые помогут в достижении конечной цели. Совмещение двух подходов позволило разработчикам достаточно быстро обучить нейросеть управлять различными видами боевых единиц и кроме того вероятность победы достаточна высока при произвольном количестве боевых единиц(выше 80%)

Hendrik Baier и Peter I. Cowling в своей работе[5] рассмотрели улучшенный вариант алгоритма поиска дерева решений Монте-Карло(Monte-Carlo tree search, MCTS) — это эволюционный MCTS (evolutionary MCTS, EMCTS). MCTS используется для поиска оптимального решения и регулирования параметров нейронной сети. EMCTS отличается от классического MCTS тем, что он использует эволюционные алгоритмы для более быстрого поиска решения, суть которых заключается в том что выбирается действие давшее больший результат в текущем состоянии. MCTS строит полное дерево решений и только после этого оценивает полезность каждого узла, а EMCTS в процессе построения строит дерево, в котором остаются только те узлы, которые принесли наибольшую выгоду. EMCTS согласно результатам исследования хорошо масштабируется и показывает себя лучше чем MCTS на таких сложных играх как стратегии. EMCTS тестировался только на пошаговой стратегии Hero Academy. Авторы отмечают, что данный алгоритм этот алгоритм может хорошо себя показать в более сложных играх, таких как XCOM или Civilization.

Компания Deepmind создала нейросеть под названием «AlphaStar»[6]. AlphaStar использует мультиагентный процесс обучения с подкреплением. Суть этого подхода в том, что сначала создаётся несколько агентов и они обучаются друг на друге какое-то время, затем на их основе создаются новые. Из новых агентов формируется лига внутри которой они сражаются между собой. Для большего охвата различных стратегий, каждому новому агенту ставили свою цель(например, научиться обыгрывать конкретного агента или группу агентов). Скомбинировав различных агентов с наиболее эффективными стратегиями, которые получились при применении такого подхода, разработчикам удалось добиться победы ИИ над игроками.

# Конструкторская часть

## Используемые методологии разработки и технологии

### Test-driven development

**Test-driven development, или TDD** (**разработка через тестирование)** — техника разработки программного обеспечения, которая предполагает сначала написание теста для нового кода, а затем самого кода с дальнейшем тестированием. [28]

**Цикл разработки** через тестирование:

1. Добавление теста.

Сначала нужно написать тест, в который входит описание ,где указывается какой функционал тестируется и каким должно быть поведение, и проверка.

1. Запуск теста.

Новый тест не должен проходиться, так как функционал не написан. Запуск проводиться для проверки работоспособности тестов.

1. Написание кода.

Далее пишется функциональность, которая была ранее описана.

1. Повторный запуск. Проверка функционала.

Запустив тест, программист убедиться, что нужный функционал работает и соответствует требованиям.

1. Рефакторинг.

На этом этапе можно спокойно править код и, запустив тесты, можно будет убедиться в его работоспособности или неработоспособности.

Важное замечание: написанные тесты не гарантируют, что тест действительно тестирует то, для чего он был разработан. Возможна такая ситуация, что тест проверяет не все ситуации, которые могут возникнуть.

**Плюсы технологии**:

* Этот метод позволяет избежать множества ошибок, особенно на ранних этапах. Также TDD препятствует возникновению дорогостоящих ошибок, которые могут возникнуть на поздних этапах разработки.
* Добавляя новую функциональность, риск испортить старый функционал очень мал. Чтобы этот риск был низким код должен быть полностью покрыт тестами, то есть для каждой функциональной единицы должен быть как минимум один тест. Это позволит при тестировании сразу увидеть, где ошибка.
* TDD способствует разработке более модульного, гибкого и расширяемого кода. Это связано с тем, что при этой методологии разработчику необходимо думать о программе как о множестве небольших модулей, которые написаны и протестированы независимо и лишь, потом соединены вместе. Это приводит к меньшим, более специализированным классам, уменьшению связанности и более чистым интерфейсам.
* Тесты могут использоваться в качестве документации. Хороший код расскажет о том, как он работает, лучше любой документации. Документация и комментарии в коде могут устаревать. Это может сбивать с толку разработчиков, изучающих код. А так как документация, в отличие от тестов, не может сказать, что она устарела, такие ситуации, когда документация не соответствует действительности — не редкость.

**Минусы технологии**:

* Существуют задачи, которые невозможно (по крайней мере, на текущий момент) решить только при помощи тестов. В частности, TDD не позволяет механически продемонстрировать адекватность разработанного кода в области безопасности данных и взаимодействия между процессами. Безусловно, безопасность основана на коде, в котором не должно быть дефектов, однако она основана также на участии человека в процедурах защиты данных. Тонкие проблемы, возникающие в области взаимодействия между процессами, невозможно с уверенностью воспроизвести, просто запустив некоторый код.
* Разработку через тестирование сложно применять в тех случаях, когда для тестирования необходимо прохождение функциональных тестов. Например, разработка интерфейса пользователя или СУБД, в которой работа будет зависеть от конфигурации сети.
* Требуется больше времени на разработку и поддержку
* Большое количество используемых тестов может создать ложное ощущение надежности, приводящее к меньшему количеству действий по контролю качества.
* Тесты сами по себе являются источником накладных расходов. Плохо написанные тесты, например, содержат жёстко вшитые строки с сообщениями об ошибках или подвержены ошибкам, дороги при поддержке. Чтобы упростить поддержку тестов, следует повторно использовать сообщения об ошибках из тестируемого кода.
* Уровень покрытия тестами, получаемый в результате разработки через тестирование, не может быть легко получен впоследствии. Исходные тесты становятся всё более ценными с течением времени. Если неудачные архитектура, дизайн или стратегия тестирования приводят к большому количеству непройденных тестов, важно их все исправить в индивидуальном порядке. Простое удаление, отключение или поспешное изменение их может привести к необнаруживаемым пробелам в покрытии тестами.

Данная технология разработки была выбрана по **следующим причинам**:

* Позволяет избежать ошибок на ранних и последующих этапах. Данный микро-сервис будет разрабатываться на javascript, из-за того что он динамически типизированный могут возникнуть различные ошибки, которые без тестов будут заметны только при запуске системы. Например, опечатки или передача аргумента не того типа, на такие ошибки динамически типизированные языки не обращают внимания.
* TDD способствует разработке более модульного, гибкого и расширяемого кода. Код, разделённый на модули, легче тестировать и эти тесты дают уверенность в том, что когда модули будут объединены в единую систему, то всё заработает.
* Тесты могут использоваться в качестве документации. Это значительно ускорит ознакомление новых программистов с системой.

### C4Model

C4Model представляет из себя несколько диаграмм: диаграмма контекста (Context diagram), диаграмма контейнеров (Container diagram), диаграмма компонентов (Component diagram), диаграмма классов (Class diagram). [29]

1. **Диаграмма контекста**.

**Рассматриваемая область**: программные системы и их взаимодействие, люди, взаимодействующие с разрабатываемой системой.

**Основные элементы**: программные системы, пользователи, взаимодействия пользователей и программных систем.

**Вспомогательные элементы:** Люди и программные системы напрямую связанные с разрабатываемой системой.

**Предполагаемая аудитория**: Все, включая людей, которые не являются разработчиками. Например, заказчик или инвестор.

Диаграмма не должна содержать информацию об используемых технологиях, это будет указано на следующей диаграмме.

На схеме должны быть показаны: программные системы в виде прямоугольников, пользователи в виде человечков, связи с описанием воздействия. Внутри каждого элемента сначала указывается название, затем ниже в квадратных скобках тип элемента(программная система, пользователь и т.д), и в конце описание. Ниже будет приведён пример диаграммы контекста для интернет-банкинга (см. рис 1).



Рисунок 2. Пример диаграммы контекста для интернет-банкинга

1. **Диаграмма контейнеров**

Под контейнером понимается то, что можно развернуть отдельно от других компонентов системы. Например, веб-приложение, база данных.

**Рассматриваемая область**: комплекс взаимодействующих контейнеров и программных систем.

**Основные элементы**: контейнеры в разрабатываемой системе.

**Вспомогательные элементы:** Люди и программные системы, взаимодействующие с контейнерами.

**Предполагаемая аудитория**: Разработчики, архитекторы ПО, вспомогательные сотрудники.

На этой диаграмме в каждом контейнере указывается используемая технология в строке с типом элемента следующим образом: [Container:<Используемая технология>]. На связях указывается способ, технология или при помощи чего будет происходить взаимодействие в виде: <Как взаимодействует>(<При помощи чего>). Ниже показан пример диаграммы контейнеров для интернет-банкинга(см. рис 2).



Рисунок 3. Диаграмма контейнеров для интернет-бакинга

1. **Диаграмма компонентов**

Для каждого контейнера нужно составлять эту диаграмму. На ней будет показано, из каких компонентов состоит контейнер и как они взаимодействуют. Также нужно указать взаимодействия с другими контейнерами.

**Рассматриваемая область**: один контейнер.

**Основные элементы**: компоненты контейнера.

**Вспомогательные элементы:** Люди, программные системы и контейнеры, взаимодействующие с данным контейнером и его компонентами.

**Предполагаемая аудитория**: Архитекторы и разработчики программного обеспечения.

Ниже показан пример диаграммы компонентов для контейнера API Application(см. рис 3).

****

Рисунок 4. Диаграмма компонентов для контейнера API Application

1. **Диаграмма классов**

Затем для каждого компонента строится диаграмма класса. Здесь показывается, как компонент будет выглядеть в коде, для этого можно использовать UML или другую подобную технологию. Я использовал диаграммы классов UML, об этом будет сказано ниже.

**Рассматриваемая область**: один компонент.

**Основные элементы**: Элементы кода (например, классы, интерфейсы, объекты, функции, таблицы базы данных и т. д.) В рамках компонента в области.

**Предполагаемая аудитория**: Архитекторы и разработчики программного обеспечения.

Данная технология разработки была выбрана по следующим причинам:

* Использование этих диаграмм позволяет структурировать информацию о разрабатываемом продукте в виде понятном разработчику, его команде и его заказчику (относится только к диаграмме контекста).
* Эти диаграммы позволяют программистам разных категорий понять суть работы системы, не углубляясь в детали.
* Нарисованные диаграммы позволяют увидеть систему целиком и отдельно рассмотреть какую-то её часть, что очень сложно сделать без диаграмм, просто заглядывая в код.

### Нотация ER-диаграмм Crow’s foot

Существуют две нотации дляER-диаграмм: нотация Чена и нотация Crow’s foot[11]. Далее я опишу обе нотации, и затем объясню, почему была выбрана нотация Crow’s foot.

**Crow’s foot (англ. Воронья лапа)** — нотация диаграмм отношений сущностей, где отношения показываются с помощью линий, с указанием названия связи над линией и количественных отношений на концах линии. Это одна из наиболее распространенных нотаций. [30]

В нотации Crow’s foot есть следующие элементы:

1. **Сущность**(Entity). Объект, который хранит в себе нужные вам данные (строки, числа, даты и так далее). Чтобы показать поле первичным ключом, нужно перед названием поставить звёздочку(\*).
2. **Отношения**(Relationship). Показывают связи между сущностями. На схеме показывается простой линией с названием связи над линией.
3. **Мощность**(Cardinality) и **модальность**(Modality). Под мощностью понимается, что, в сущности, может быть несколько экземпляров другой сущности. Под модальностью понимается, что, в сущности, может быть один или не одного экземпляра другой сущности. Чтобы показать мощность или модальность рисуется линия с указанием количества экземпляров на конце. Если количество сущностей равно 1, то рисуется вертикальная линия, если может быть равным 0, то рисуется кружок. Чтобы показать что количество экземпляров может быть больше одного нужно нарисовать воронью лапу, состоящую из 3 пальцев (поэтому нотация и называется «Вороньей лапой»)

Чтобы было проще понять, как элементы должны выглядеть, ниже будет картинка с примерами.



Рисунок 5. Примеры элементов нотации Crow’s foot

На рисунке будут показаны примеры сущностей и отношений. Рисунок сделан в программе “Visio 2016” и есть отличие в обозначении поля с первичным ключом. В Visio первичный ключ обозначается как квадрат с надписью «PK»(primal key). На картинке слева в прямоугольнике указан тип отношения, справа сущности и связь между ними.Чтобы рисунок не занимал слишком много места, только вначале будет показан пример сущностей со всеми полями, далее будет только сокращённая версия без свойств и полей.

**Нотация Чена(англ. Chen notaion)** — нотация в которой отношения показываются с помощью ромба, соединённого линиями с сущностями. [31]

В нотации Чена есть те же элементы, что и в нотации Crow’s foot, но есть отличия:

1. Сущность. На схеме изображается в виде прямоугольника с названием в центре. К нему подсоединяются свойства, которые показываются как овалы с названиями в центре. В нотации Crow’s foot свойства находятся внутри сущности и разделены линией.
2. Отношения. Изображаются ромбом, соединённым линиями с сущностями, Внутри ромба находится название связи.
3. Мощность и модальность. Указываются на линиях, которые подсоединены к ромбу. Вместо обозначений вроде не закрашенного кружка или «Вороньей лапы» линии используются числовое обозначение.

Помимо этого в рассматриваемой нотации дополнительные элементы, которых нет в Crow’s foot:

1. Есть ещё два вида сущности
   1. Слабая сущность (Weak entity) — объект, который не может быть однозначно идентифицирован только по его атрибутам. Существование слабого объекта зависит от другого объекта, называемого сущностью владельца. Идентификатор слабого объекта представляет собой комбинацию идентификатора объекта владельца и частичного ключа слабого объекта. Обозначается как прямоугольная рамка с названием сущности в центре.
   2. Ассоциативный объект (Associative entity) - объект, используемый в отношении «многие ко многим» (представляет собой дополнительную таблицу). Обозначается как прямоугольник с ромбом в центре, внутри которого находится название сущности.
2. Существует несколько видов атрибутов:
   1. Ключевой атрибут выглядит как обычное свойство, только название внутри подчёркивается.
   2. Атрибут частичного ключа (дискриминатор) (Partial key attribute (discriminator)) - атрибут, который в сочетании с ключевым атрибутом объекта владельца предоставляет уникальную идентификацию для слабого объекта. Этот элемент изображается как обычное свойство, в котором название подчёркивается пунктирной линией.
   3. Многозначный атрибут (multivalued attribute) - атрибут, который может иметь много значений (для него есть много разных значений в одном столбце таблицы). Многозначный атрибут изображается овальной рамкой с названием в центре.
   4. Производный атрибут (или вычисленный атрибут)(derived attribute (or computed attribute)) - атрибут, значение которого вычисляется (выводится) из других атрибутов. Полученный атрибут может или не может быть физически сохранен в базе данных. Данный элемент обозначается овалом, граница которого изображена пунктирной линией)
3. Есть второй вид отношений:
   1. Слабое (идентифицирующее) отношение(weak (identifying) relationship) — отношение, в котором дочерняя сущность зависит от родителя, и первичный ключ взят из родительской сущности.
4. Есть необязательное отношение. На рисунке изображается пунктирной линией.
5. Существует ограничения участия. Сущности могут участвовать в отношении полностью или частично.
   1. Полное участие. Означает, что каждая сущность участвует в отношении. Например, каждого студента обучает преподаватель.
   2. Не полное (частичное) участие. Означает, что не каждая сущность может участвовать в отношении. Например, преподаватель обучает студента, но могут быть преподаватели, которые не имеют учеников.

На следующей странице будут показаны примеры элементов нотации Чена.



Рисунок 6. Примеры элементов нотации Чена

Теперь сравним рассмотренные нотации. Нотация Чена имеет больше видов элементов, что позволит подробнее описать базу данных, но для небольших баз данных это лишнее. ER-диаграмма в нотации Чена будет больше по размеру и выглядеть она будет более сложной для восприятия.

Я выбрал нотацию Crow’s foot, потому что итоговая схема будет более компактной и простой, чем в случае использовании нотации Чена. Также не весь функционал нотации Чена мне будет нужен, и ещё при использовании нотации Чена нужно было бы позаботиться о более-менее красивом расположении свойств, что занимает определённое время, в случае с Crow’s foot таких проблем нет.

### UML

**UML** (англ. Unified Modeling Language — унифицированный язык моделирования) — язык графического описания для объектного моделирования в области разработки программного обеспечения, моделирования бизнес-процессов, системного проектирования и отображения организационных структур. [32]

В этот язык входит множество диаграмм(чтобы не было неоднозначности с названиями в скобках будет указано название на английском языке):

* **Структурные диаграммы(Structure Diagrams):**
  + **Диаграмма классов (Class diagram).** Содержит в себе классы, их методы, поля и взаимосвязи.[33]
  + **Диаграмма компонентов (Component diagram).** Содержит в себе компоненты программной системы. Компонента – всё что можно развернуть (поставить, установить) отдельно от других частей системы (например, модуль, библиотека, веб-приложение и так далее).[34]
  + **Диаграмма композитной/составной структуры (Composite structure diagram).** Данная диаграмма показывает внутреннюю структуру и взаимодействие внутренних элементов класса.[35]
  + **Диаграмма развёртывания (Deployment diagram).** На этой диаграмме показываются компоненты, которые нужно установить, чтобы разрабатываемая система заработала.[36]
  + **Диаграмма объектов (Object diagram).** В этой диаграмме показывают экземпляры классов и отношения между ними в какой-то момент времени.[37]
  + **Диаграмма пакетов (Package diagram).** На этой диаграмме показываются пакеты, связи и зависимости между ними. Пакет – это группа моделей (классов, сущностей, случаев использования и т.д).[38]
* **Диаграммы поведения (Behavior Diagrams):**
  + **Диаграмма деятельности (Activity diagram).** Она отображает действия, которые описаны в диаграмме состояний.[39]
  + **Диаграмма состояний (State Machine diagram).** На данной диаграмме изображается конечный автомат.[40]
  + **Диаграмма вариантов использования (Use case diagram).** На этой диаграмме показываются актёры(люди, исполняющие определённую роль) и их действия на описываемую систему, которые изменяют состояние системы.[41]
  + **Диаграммы взаимодействия (Interaction Diagrams):**
    - **Диаграмма коммуникации/кооперации (Communication diagram)**. Содержит взаимодействия между частями композитной структуры или ролями кооперации. На этой диаграмме указываются отношения между объектами и не указывают время для измерения работы, как на диаграмме последовательности.[42]
    - **Диаграмма обзора взаимодействия (Interaction overview diagram)**. Данная диаграмма представляет из себя комбинацию диаграмм деятельности и диаграмм последовательности. Представляет из себя диаграммы последовательности, разбитые на отдельные диаграммы деятельности. С помощью этой диаграммы можно показать поток управления.[43]
    - **Диаграмма последовательности (Sequence diagram)** . На этой диаграмме показывается прецедент и всё, что нужно для его исполнения (объекты, актёров, жизненный цикл объектов). [44]
    - **Диаграмма синхронизации (Timing diagram)** . В этой диаграмме рассматриваются состояния, и их изменение с течением времени.[45]

Большую часть диаграмм я не использовал, так нет в этом необходимости. Далее будет сказано только про те диаграммы, которые были использованы. Я использовал следующие диаграммы:

* Диаграмма активностей или деятельности(activity diagram) для схемы алгоритма обработки гео-данных и диаграммы классов.
* Диаграмма классов. Она была использована при следовании концепции C4Model.

В UML и C4Model есть диаграмма компонентов, но они сильно отличаются. Диаграмма компонентов из UML показывает интерфейсы компонентов и также отличается содержимое контейнера, в нём нет описания. Подробнее диаграмма компонентов UML будет описана ниже.

**Диаграмма компонентов**

**Диаграмма компонентов** – диаграмма, показывающая компоненты разрабатываемой системы и их взаимодействия. [34]

В диаграмме компонентов изображаются:

* Компоненты. Они рисуются как прямоугольники содержащие:
  + Значок компонентного элемента. Ставится, если данный контейнер является элементом другого контейнера. Значок выглядит как прямоугольник с двумя меньшими прямоугольниками, выступающими с левой стороны.
  + Фраза «Component». Эта фраза указывается, если данный контейнер является стереотипным. Здесь под стереотипом имеется ввиду один из механизмов расширения (в UML есть ещё тэги и ограничения), который является специализированной моделью на основе которой можно создать другие компоненты.
  + Фраза «Subsystem». Указывается вместо прошлой фразы, если данный контейнер является подсистемой. Архитектор сам решает что является подсистемой, в UML это жестко не задано.
  + Интерфейсы. Требуемые интерфейсы показываются линией с полукругом на конце и названием интерфейса над линией. Предоставляемые интерфейсы обозначаются почти также как и требуемые, только вместо полукруга рисуется целый не закрашенный круг. Если один контейнер требует интерфейс «А», а другой его предоставляет, то их линии связи соединяются так, чтобы внутри полукруга требуемого интерфейса был круг предоставляемого интерфейса.
* Отношения
  + Зависимость. Показывается также как и в диаграмме классов. Стрелка зависимости должна выходить из гнезда требуемого интерфейса.

Ниже будут показаны примеры элементов диаграммы коспонентов.



Рисунок 7. Примеры элементов UML диаграммы компонентов

**Диаграмма деятельности**

**Диаграмма деятельности** – диаграмма, показывающая процесс и последовательность действий необходимых для его выполнения. [39]

Данный вид диаграмм очень похож на блок-схемы и имеет ту же функцию. В отличие блок-схем рассматриваемый вид более сложный, но у него больше возможностей:

1. С помощью жирных линий можно показать начало (разветвление) и окончание (схождение) ветвления действий. Это используется для изображения параллельных операций и дальнейшей синхронизации потоков.
2. Можно показать состояния объектов.
3. У каждой сущности участника есть своя дорожка, на которой видно, какое действие выполняет конкретная сущность, и на каком этапе.

Данный вид диаграмм был использован из-за возможности использовать дорожки. В алгоритме обработки гео-данных используются несколько функций, которые нужно было показать в удобном виде. Блок схемы не позволили бы мне этого сделать.

**Диаграмма классов**

**Диаграмма классов** – диаграмма, показывающая классы разрабатываемой системы, их взаимодействия, свойства и методы. [33]

В диаграмме классов изображаются:

1. Сущности (классы) в виде прямоугольников. В прямоугольнике содержится:
   * Название. Находится вверху посередине, выделено жирным шрифтом.
   * Поля и свойства. Находятся под заголовком, отделены чертой. Тип указывается после знака двоеточия, который стоит после названия.
   * Методы. Находятся под полями и свойствами, отделены чертой. После названия следуют скобки, внутри которых указываются аргументы через запятую. Тип возвращаемого значения указывается после двоеточия, которое стоит после скобок.

Поля и свойства могут быть:

* Публичными. Это означает, что к ним могут обратиться другие сущности. Обозначается «+» и ставится перед названием.
* Приватными. Это означает, что они будут недоступны другим сущностям, в том числе и тем, которые наследуют этот метод или свойство. Обозначается «-» и ставится там же, где и прошлый признак.
* Защищёнными. Это означает, что они будут недоступны другим сущностям, но при наследовании они могут быть доступны только наследнику. Обозначается «#» и ставится там же, где и другие признаки.
* Статическими. Это означает что данное поле или метод одинаковый для всех экземпляров данного класса. Название такого элемента подчёркивается.

1. Отношения:
   * Ассоциация – связь между сущностями, которая является общим случаем композиции и агрегации. Показывается обычной стрелкой.
   * Наследование – то же самое что и агрегация, но здесь могут включаться не все методы или поля и уровень доступа может измениться, при включении. Показывается линией с не закрашенным треугольником.
   * Реализация – отношение, при котором один класс реализует метод(ы) другого класса. Показывается пунктирной линией с не закрашенным треугольником.
   * Зависимость – отношение, при котором изменение спецификации класса-поставщика может повлиять на работу зависимого класса, но не наоборот.
   * Агрегация – отношение между классами, при котором один класс включает в себя всё содержимое другого. Показывается линией с пустым ромбом на конце.
   * Композиция – отношение между классами, при котором один класс включает в себя экземпляр другого. Показывается линией с закрашенным ромбом на конце.

При обозначении отношения указывается количественное отношение на концах линий: один к одному, один ко многим, многие ко многим. Ниже будут показаны примеры элементов диаграммы классов.



Рисунок 8. Отношения в диаграммах классов



Рисунок 9. Пример сущностей и связей между ними

## Используемые программные продукты и языки программирования

### Язык JavaScript

**JavaScript** — мультипарадигменный язык программирования. Поддерживает объектно-ориентированный, императивный и функциональный стили. Является реализацией языка ECMAScript.[46]

Данный язык используется в основном для доступа и управления объектами различных приложений. Чаще всего применяется в браузерах в качестве языка сценариев для создания интерактивности на веб-странице.

Возможности языка:

* Динамическая типизация
* Сборщик мусора (автоматическое управление памятью)
* Прототипное программирование
* Функции являются объектами первого класса
* Автоматическое приведение типов
* Встраивание в код веб-страницы

В языке отсутствуют такие полезные вещи, как:

* стандартная библиотека: в частности, отсутствует интерфейс программирования приложений по работе с файловой системой, управлению потоками ввода-вывода, базовых типов для бинарных данных;
* стандартные интерфейсы к веб-серверам и базам данных;
* система управления пакетами, которая бы отслеживала зависимости и автоматически устанавливала их.

JavaScript состоит из нескольких частей:

* ядро (ECMAScript)
* объектная модель браузера (Browser Object Model или BOM (англ.))
* объектная модель документа (Document Object Model или DOM)

**ECMAScript** — это встраиваемый, расширяемый, не имеющий средств ввода-вывода язык программирования. ECMAScript является стандартом и основой для других языков: JavaScript, JScript и ActionScript. [47]

ECMAScript предоставляет пять основных примитивных типов:

1. Число
2. Строка
3. Логический (или булевый (англ. boolean))
4. нулевой (англ. Null)
5. неопределённый (англ. Undefined)

Есть составной тип: объектный (англ. Object).

Также есть стандартные для большинства современных языков инструкции (блок, условие, цикл и другие), операторы («.»(доступ к свойству), «++»(инкремент), «\*»( умножение) и другие).

Функции в JavaScript являются объектами и могут быть использованы как обычные переменные. В ECMAScript имеется два типа функций:

1. внутренние функции (например, parseInt)
2. функции, определённые в тексте программы.

В ECMAScript есть перегрузка функций и рекурсия, как и в других языках.

**Объектная модель документа (Document Object Model или DOM)** — интерфейс для программирования приложений, использующие HTML и XML документы. [46] Согласно DOM, документ может быть представлен в виде дерева объектов, обладающих рядом свойств, которые позволяют производить с ним различные манипуляции:

* генерация и добавление узлов,
* получение узлов,
* изменение узлов,
* изменение связей между узлами,
* удаление узлов.

**Объектная модель браузера (Browser Object Model или BOM (англ.))** — часть JavaScript, позволяющая скрипту управлять поведением веб-страницы. Данная часть JavaScript связывает ядро и DOM. [46, 48]

В качестве языка разработки был выбран JavaScript, потому что система трекинга поездок транспортного средства изначально была написана на нём. Разработка на JavaScript позволит переиспользовать код от разных частей системы трекинга, а также упростит администрирование.

# Список литературы

1. OpenAI and Dota 2 [Электронный ресурс] - Режим доступа: <https://openai.com/blog/dota-2/> , свободный
2. Competitive Self-Play [Электронный ресурс] - Режим доступа: <https://openai.com/blog/competitive-self-play/> , свободный
3. Per-Arne Andersen, Morten Goodwin, Ole-Christoffer Granmo «Deep RTS: A Game Environment for Deep Reinforcement Learning in Real-Time Strategy Games» [Электронный ресурс] - Режим доступа: <https://arxiv.org/abs/1808.05032>, свободный
4. Kun Shao, Yuanheng Zhu, Member, IEEE and Dongbin Zhao, Senior Member «StarCraft Micromanagement with Reinforcement Learning and Curriculum Transfer Learning» [Электронный ресурс] - Режим доступа: https://arxiv.org/pdf/1804.00810.pdf, свободный
5. Hendrik Baier, Peter I. Cowling «Evolutionary MCTS for Multi-Action Adversarial Games» [Электронный ресурс] - Режим доступа: https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=1&ved=2ahUKEwjqpb-r-qbgAhWI8ywKHdj5B5YQFjAAegQICRAC&url=https%3A%2F%2Fhendrikbaier.jimdo.com%2Fapp%2Fdownload%2F13604570727%2Fevolutionary%2520mcts%2520for%2520multi-action%2520adversarial%2520games.pdf%3Ft%3D1532703212&usg=AOvVaw3M9xX0AbJl4h5uYAy8GBBh, свободный
6. AlphaStar: Mastering the Real-Time Strategy Game StarCraft II [Электронный ресурс] - Режим доступа: https://deepmind.com/blog/alphastar-mastering-real-time-strategy-game-starcraft-ii/ , свободный

Приложение А. Техническое задание.

Введение

На

Назначение разработки

Данный

Основание для разработки

Так

Требования к программе или программному изделию

1. Архитектура

Требования к программной документации

1. Должны
2. Также

Стадии и этапы разработки

1. С

Порядок контроля и приемки

1. П

Приложение Б. Программная спецификация.

Документация

Ранее в основной части использовались перечисленные ниже диаграммы, поэтому здесь тут даны только ссылки на них.

В документацию входят:

* Диаграмма контекста. (см. рис )

Компоненты

При разработке микро-сервиса будут использоваться следующие программные продукты:

* PostgreSQL – база данных

Документация по программным комплексам:

* PostgreSQL – https://www.postgresql.org/docs/

Приложение В. Программная документация.

Диаграмма контекста

Рисунок .