Супер крутая работа Олежки!!1

«Тепловая карта предложений жилья»

Оглавление

[Введение 2](#_Toc506167647)

[Функционал проекта 3](#_Toc506167648)

[1. Отображение предложений 3](#_Toc506167649)

[2. Транспортная доступность 4](#_Toc506167650)

[3. Доступность инфраструктуры 5](#_Toc506167651)

[Существующие альтернативы 5](#_Toc506167652)

[Реализация 6](#_Toc506167653)

[1. Структура проекта и инструменты 6](#_Toc506167654)

[1.1 Структура 6](#_Toc506167655)

[1.2 Инструменты 7](#_Toc506167656)

[2. Сервер: 7](#_Toc506167657)

[2.1 Данные 7](#_Toc506167658)

[2.1.1 Геоданные 7](#_Toc506167659)

[2.1.2 Данные о предложениях 9](#_Toc506167660)

[2.2 Квадродерево 9](#_Toc506167661)

[2.3 Дорожный граф: 11](#_Toc506167662)

[2.4 Расчет графа: 12](#_Toc506167663)

[2.5 Доступность инфраструктуры 13](#_Toc506167664)

[2.6 Отрисовка 14](#_Toc506167665)

[2.6.1 Предложения 14](#_Toc506167666)

[2.6.2 Транспортная доступность 14](#_Toc506167667)

[3. Клиент 15](#_Toc506167668)

[Дальнейшее развитие 15](#_Toc506167669)

[Заключение 15](#_Toc506167670)

[Источники 16](#_Toc506167671)

[Приложения 17](#_Toc506167672)

# Введение

В жизни сегодняшнего человека существует множество занятий, когда ему приходится иметь дело с большим объемом данных. Негласным стандартом для такой работы стало использование различный фильтров. Выбор жилья не стал исключением. Но вместо того что бы с помощью фильтров исключать предложения, но можно изменить способ их представления на более удобный и легкий в восприятие, способный отображать намного больший объем информации на экране и благодаря этому пользователь сможет принимать решение на основание намного большего числа факторов, что сделает решение более взвешенным. Кроме информации о самой квартире можно отобразить информацию об её окружение. О транспортной доступности и об инфраструктуре. Выбор методов отображение и их реализация будет описана ниже.

# Функционал проекта

## 1. Отображение предложений

Для предложений я собираюсь использовать вместо классических маркеров заливку зданий определенным цветом. Цвет будет определятся на основание отличия цены, от заданной пользователем.



Рисунок 1.

Для этого отлично подойдет HSB градиент (Рисунок 1), ограниченный снизу на синем цвете. То есть градиент с параметром hue(оттенок) от 0 до 240. Также он должен быть симметричен, то есть где P – цена.

Благодаря интуитивной ассоциации красного цвета как с чем-то выше норма, а синего наоборот с чем-то ниже нормы, пользователю будет легко привыкнуть к значениям цветов. Но при построении градиента на всем диапазоне цен, представленных на карте может возникнуть проблема. Она заключается в том, что градиент может стать слишком широким, из-за чего он перестанет отображать разницу в ценах.

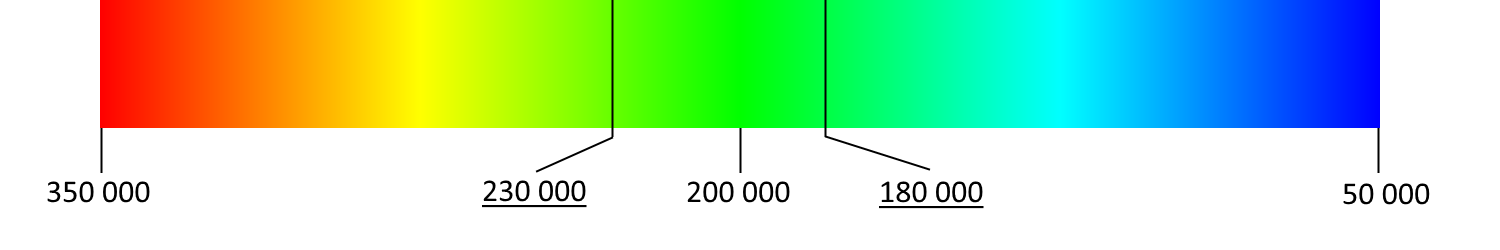


Рисунок 2.

На рисунке 2 показан градиент (все цены в рублях за метр квадратный), отражающий диапазон цен от 50000 до 350000, с указанной ценой в 200000. Также отмечены цены 230000 и 180000. Видно что не смотря на отличие цены на 30000 и 20000 соответственно, их цвет едва различимы. Для решения этой проблемы я решил использовать настраиваемый диапазон цен. Пользователь будет выбирать не только целевую цену, но и диапазон цен.

Используя полученный градиент, я буду использовать для отрисовки домов, причем отрисовывать их я буду на прозрачном фоне, что бы их можно было наложить на карту. Пример представлен на Рисунке 3.

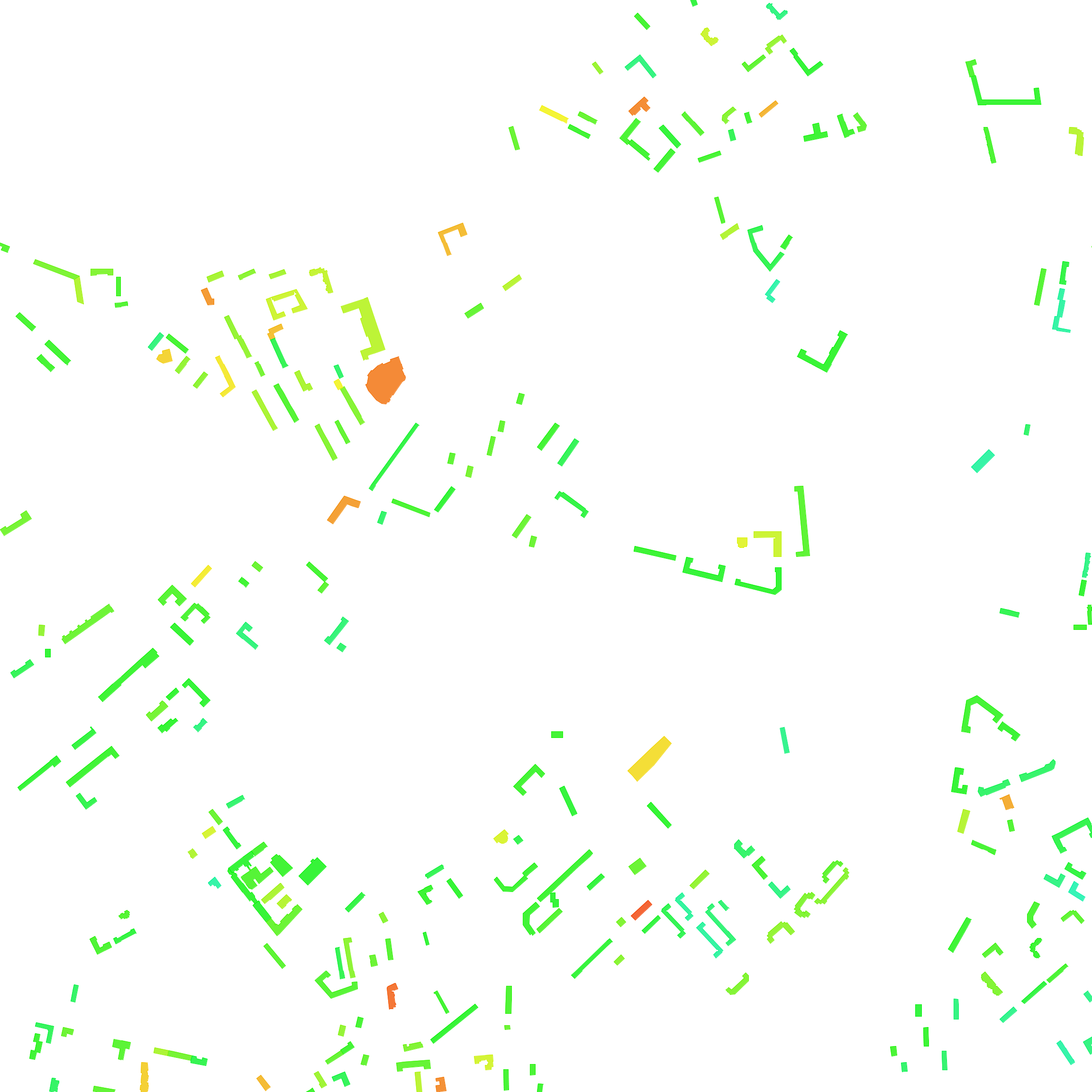


Рисунок 3.

## 2. Транспортная доступность

Для отображения транспортной доступности я буду раскрашивать дороги в цвет зависящий от того, за сколько времени можно до той или иной её части. Для этого имея дорожный граф и начальную точку, я найду значение для каждой вершины, которое будет являться минимально возможным временем, за которое можно добраться из начальной вершины в данную. Дальше на основание этих значений и максимального времени, граф раскрашивается с помощью градиента приведенного на рисунке 4.



Рисунок 4.

Соответственно не доступные части графа будут окрашены в темно-красный цвет(находится за отметкой макс. время). Также кроме автомобильного режима, использующего обычные автомобильные дороги, я добавлю пешеходный режим, учитывающий пешеходные улицы и общественный транспорт. Этот граф также должен быть отрисован на прозрачном фоне, причем и его самого я сделаю полупрозрачным, что бы после наложения на карту можно было видеть названия улиц.

## 3. Доступность инфраструктуры

При выборе того или иного дома я собираюсь показывать список объектов инфраструктуры поблизости. Магазины, кафе, гос. учреждения.

# Существующие альтернативы

Отображение цен с помощью тепловых карт уже было реализовано например здесь <https://квартиры-домики.рф/Карта-цен/Москва/> (статья автора сайта: <https://habrahabr.ru/post/335638/>). Но метод отображения реализованный на сайте выше, не показывает данные о домах по отдельности. К тому же из-за погрешностей тепловой карты, нельзя с точностью сказать у ценах в том или ином доме. К сожалению во время работы над проектом я не знал о нем, и не смог позаимствовать некоторые методы, описанные в статье.

Отображение транспортной доступности имеется у сервиса <https://maps.openrouteservice.org>, но моё отображение отличается тем, что поддерживает общественный транспорт и методом отрисовки. У данного сервиса зоны закрашиваются сплошняком, когда у меня закрашиваются только дороги, из-за чего пользователю легче ориентироваться на карте

Ближайшую инфраструктуру умеют показывать популярные агрегаторы жилья, например <https://cian.ru>. Он отображает объекты инфраструктуры на карте, позволяю человеку самому оценить её доступность. У меня же пользователю будет доступно примерное время, за которое можно добраться до тоги или иного объекта.

# Реализация

## 1. Структура проекта и инструменты

### 1.1 Структура

Структура изображена на схеме в Приложение 1. Клиента отправляют запрос к серверу, на котором запущен сервер nginx на порте 80(стандартный порт для http запросов). Далее в зависимости от запроса nginx либо отдает тайл, хранящийся как файл на диске, либо перенаправляет запрос к серверу написанному на java с использованием netty. После PathRouter в зависимости от uri запроса направляет его к одному из handler’ов.

PriceTileHandler – отдает тайлы, на которых нарисованы дома, закрашенные в определенный цвет согласно запросу и которые в последствие накладываются на карту.

RoadGraphTileHandler – отдает тайлы, на которых нарисованы дороги, закрашенные в определенный цвет согласно запросу и которые в последствие накладываются на карту.

MapPointSearchHandler – производит поиск по точке, координаты который отправлены в запросе и если объект найден, то возвращает его данные в формате json.

StringSearchHandler – производит поиск по данной в запросе строке среди названий улиц, при полном совпадение названий возвращает данные о найденном объекте

StringPredictHandler – производит составления подсказок по префиксу данному в запросе и возвращает их.

### 1.2 Инструменты

Nginx – он был выбран из-за высокой производительности и простоты настройки. Также его используют очень крупные сайты вроде vk.com, yandex.ru(для некоторой части контента) и другие, что говорит о его высокой надежности и степени доверия к нему.

Java – этот язык программирования был выбран мною как хорошо знакомый мне, и зарекомендовавший себя в написание серверов. Благодаря такой направленности у этого языка есть много библиотек сильно упрощающих написание сервера.

Netty – это веб фреймворк выбранный мною для написания сервера на java. Из-за малого количества знаний о веб фреймворках он был выбран на основе его популярности и его использовании крупными компаниями. Также большого сообщества, в котором можно найти рекомендации по его использование.

## 2. Сервер:

Все данные карты, цен и методы работы с ними я решил объединить в классе PropertyMap. Соответственно при обработке запросов к серверу, обработчик обращается только к объекту PropertyMap, а все обращения к внутренним объектам производятся уже самим PropertyMap’ом. Такое разделение позволит потом при необходимости держать несколько карт, в одной jvm, с минимальными изменениями кода.

### 2.1 Данные

#### 2.1.1 Геоданные

Для начала я начал искать источник геоданных, мой выбор пал на openstreetmap.org. Их данные совершенно открыты и регулярно обновляются сообществом. После необходимо было написать парсер для этих данных. Их данные распространяются в разных форматах, но мне пришлось работать с форматом osm, т.к. overpass api[[1]](#footnote-1). Формат osm из себя представляет данные об объектах организованные с помощью xml. Есть три основных элемента Node, Way и Relation.

Node – это обычная точка, она в качестве своих xml атрибутов имеет id, географические координаты и информацию о пользователе что её добавил/изменил. Также в него вложены элементы типа tag, которые содержат в себе информацию о точке в формате ключ - значение. Например автобусная остановка описывается как public\_transport - platform.

Way – это тип описывающий полигоны и линии на карте. Он содержит в себе те же атрибуты что и Node. В него вложены элементы nd и tag, элементы tag также содержат информацию о Way, nd же содержат в себе id Node, из которых он состоит. Соответственно соединив все Node’ы мы получим то, как выглядит Way.

Relation – это элемент группирующий другие элементы. Node, Way и Relation. Обычно эти элементы являются маршрутами общественного транспорта или какими либо сетями.

Для работы над этими данными надо создать классы представляющие геоданные. Они показаны в Приложение 2.

Выбор именно такой схемы классов будет объяснен ниже.

Так как osm файлы обладают внушительным размером(карта Москвы занимает ~1.5GB), то обычные xml парсеры, строящие DOM дерево в памяти не подходят. Для парсинга больших xml файлов используются stream parsers(потоковые парсеры), которые не строят DOM дерево в памяти, а всего лишь сообщают о типе элемента и его атрибутах. С такими парсерами вся забота о правильной иерархии элементов ложится на плечи разработчика. Но поскольку иерархия элементов у osm формата очень проста[[2]](#footnote-2), то это не является проблемой. Java включает в себя StAX парсер.

Загрузка происходит по схеме, представленной в Приложение 3. Далее карта – это фрагмент данных osm полученных с помощь overpass api. В самом начале читаются границы карты и записываются в соответствующие поля, это необходимо для того что при дальнейшей конвертации географических координат в координаты на плоскости, можно было получать не абсолютные их значения, а относительно верхнего левого угла карты(точки с минимальной долготой и максимальной широтой). Сделано это для простоты передачи координат от клиента к серверу.

После в методе load() происходит чтение файла, и создание Node, Way и Relation, которые сохраняются внутри загрузчика, а потом забираются из него с помощью методов getNodes(), getWays(), getRelations(), getSimpleNodes(). Благодарю тому что все реализовано через интерфейс MapLoader можно в дальнейшем написать другие загрузчики, реализующие этот интерфейс и читающие данные другого формата, тем самым получив поддержку новых форматов. Также этот метод создаёт дорожный граф, подробнее об этом в разделе 2.3 Дорожный граф:.

#### 2.1.2 Данные о предложениях

Так как я пытался начать работать над этим проектом ещё в прошлом году(но не смог далеко продвинутся из-за малого объема знаний в то время), то я использовал данные собранные мною в прошлом году. Я парсил сайт сian.ru и вытаскивал оттуда данные о предложениях. Далее сохранил их в файл в формате json, из которого сервер и читает эти данные. Все данные о предложениях используются только для демонстрации возможностей программы.

### 2.2 Квадродерево

Для отрисовки домов, их выбора, поиска близкой к ним инфраструктуры и составления дорожного графа мне необходим быстрый поиск по данным карты. Для этого существует множество методов организации данных на плоскости. Из них я выбрал квадродерево(Quadtree), за его простоту в реализации.

Реализация квадродерева находится в классе QuadTreeNode, а методы для поиска в нем и его корень в классе QuadTree. Оно может в себе содержать как точки, так и полигоны с линиями. У меня оно не сбалансированное.

В квадродереве, у каждого узла может быть по 4 потомка, которые делят своего родителя на 4 равные части. Каждый узел при создание считается конечным. Поиск необходимого узла происходит за время O() (где n количество элементов в дереве, при условие что дерево сбалансированное), но так как O() пренебрежительно мало, то сложность поиска в лучшем случае O(), в худшем O() (где - минимальное кол-во элементов в узле, а - максимальное кол-во элементов).

Добавление точки в квадродереве очень тривиально. Когда с линиями и полигонами уже сложнее, так как необходимо их обрезать, что бы они не выходили за границы узла. Для обрезки линий был сделан написан алгоритм для обрезки линий addRoad() класса QuadTreeNode. Для обрезки же полигонов я реализовал алгоритм Уайлера-Атертона (Weiler–Atherton). Его реализация находится в методе addPoly\_double().

Так как дерево несбалансированное, то узлы делятся после того, как количество элементов в нем превосходит определенный порог. После этого текущей узел делится на 4 новых узла, которые назначаются потомками текущего. И все элементы из текущего перемещаются в его потомков, согласно алгоритму добавления элементов в дерево. При этом текущей элемент перестаёт быть конечным.

Все алгоритмы поиска в квадродереве сводятся к одной последовательности действий. 1. Поиск всех конечных узлов, которые пересекаются или содержатся в фигуре поиска или содержат фигуру поиска. 2. Обход всех найденных узлов и проверка их элементов на вхождение в фигуру поиска или вхождение фигуры поиска в них. 3. Массив элементов что входят в фигуру и будет результатом. Здесь есть пространство для оптимизаций, например, проведя простую проверку на вхождение узла в фигуру поиска, при его полном вхождение мы можем добавить все элементы узла без проверки каждого из них.

Мною реализовано несколько видом поиска в дереве. Поиск по точке, окружности и прямоугольнику. Также они отличаются тем, что они ищут: Way, Node или RoadGraphNode.

### 2.3 Дорожный граф:

Что бы показать транспортную доступность дома, мне необходимо знать минимальное время, за которое можно добраться до фрагментов дорог. Для расчета этого времени необходимо объединить все возможные пути передвижения в единый граф(то есть объединять не только автомобильные дороги, но и пешеходные и маршруты общественного транспорта), и далее имея подобный граф, провести расчет.

Вершинами графа являются объекты класса RoadGraphNode. Они хранят ссылки на другие вершины, с которыми они связаны и время, за которое можно до них добраться. Хранится 2 разных времени: для пешеходного режима и для автомобильного. Причем время хранится не в минутах, а в сотых долях секунд (0,01 сек). Такая малая единица времени была взята из-за того что на карте встречаются очень короткие участки дорог, которые например машиной со скоростью в 60 км/ч преодолеваются за очень малое время. И если взять за минимальную единицу времени минуту, то из-за округления, многие участки будут преодолеваться мгновенно и следовательно, результаты будут некорректны.

Так как экземпляров RoadGraphNode достаточно много(на карте Москвы – 1716727), то очень важно что бы они занимали минимум памяти. Это послужило причиной для отказа от динамических массивов ArrayList в пользу статических массивов java’ы, т.к. согласно библиотеки jol(Java Object Layout) размер ArrayList’a – 24 байта(зависит от конфигурации jvm), а размер обычного Array’я – 16 байт(зависит от конфигурации jvm), но поскольку ArrayList реализован как обертка для Array’я, то в итоге суммарный размер ArrayList равен 40 байт. Соответственно переход от ArrayList к Array сохраняет 24 байта, а если учесть количество RoadGraphNode и количество массивов в них, то сохраняется минимум 4\*24\*1716727=164805792 байт 157 Мбайт. Но т.к. в RoadGraphNode есть ещё и вложенные массивы, то реальный выигрыш будет больше.

Благодаря использованию Array я получил выигрыш по памяти, но из-за того что Array не может автоматически менять свой размер, код сильно усложнился. Для его упрощения мною были написаны Builder’ы для RoadGraphNode и самого дорожного графа – классы RoadGraphNodeBuilder и RoadGraphBuilder. В которых реализована вся логика создания вершин и их соединения. После добавления всех нод в RoadGraphBuilder вызывается его метод getRoadGraph(), который возвращает HashMap<Long, RoadGraphNode>, где ключ это id ноды, а значение сама нода. Все ноды в этом HashMap связаны друг с другом и готовы к использованию.

Используя RoadGraphBuilder, MapLoader добавляет в граф все дороги прочитанные им. Также после загрузки геоданных вызывается метод public\_transport\_init(), который добавляет к дорожному графу маршруты общественного транспорта.

### 2.4 Расчет графа:

Расчет графа – нахождение минимально возможного времени пути из начальной точки до вершины для вершин графа, если это время меньше максимально допустимого.

При написание алгоритма для расчета графа по началу мною был написан рекурсивный алгоритм(PropertyMap.recCalculateDistances()), очень похожий на поиск в глубину, но вместо сравнения вершины с искомой, я записываю в вершину сумму весов уже пройденных ребер, если она меньше уже записанной в вершине. Перед этим во все вершины записывается максимально возможное число(INT\_MAX). Но при его использовании я понял, что время его работы слишком велико. Тогда я решил попробовать реализовать расчет графа на основание поиска в ширину (PropertyMap.widthRecCalculateDistance()). Изменения скорости работы показаны в Приложение 4 и Приложение 5.

При работе с одним пользователем проблем нет, но если будет несколько пользователей запрашивающих тайлы, то начнутся проблемы из-за того что при смене параметров запроса(один пользователь смотрит транспортную доступность одного дома, а другой другого) придется перерассчитывать граф, т.к. даже если он был рассчитан раньше, то результаты предыдущего расчета были перезаписаны текущим. Для решения этой проблемы мною был реализован кэш. Вместо одной числовой переменной, в которую записывается её дальность, будет массив чисел, также хранящий дальности этой ноды в различных расчетах(соответственно дальности всех нод, рассчитанные в рамках одного запроса, хранятся под одним и тем же индексом). Для того что бы определить был ли уже произведен расчет для запрошенных параметров и если был, то по какому индексу в массивах находятся его результаты я создал класс CalculatedGraphCache и CalculatedGraphKey. С помощью второго я проверяю являются ли запросы идентичными, а первый хранит в себе с помощью HashMap’ы с ключом CalculatedGraphKey, индексы по которым находятся результаты расчетов. Также первый класс хранит в себе список свободных для записи индексов и занимается их высвобождением при переполнение кэша.

### 2.5 Доступность инфраструктуры

Имея рассчитанный граф очень просто найти инфраструктуру по близости. Если максимальная её дальность меньше или равна максимальной дальности при расчете графа. Для это необходимо обойти все объекты инфраструктуры, для каждого из них найди ближайшую RoadGraphNode’у. Её удаленность, с поправкой на расстояние между ними и будет удаленностью объекта инфраструктуры. Если же удаленность RoadGraphNode’ы больше максимальной, то данный объект считаем недоступным. После нужно отсортировать объекту по убыванию удаленности и отдать их клиенту.

### 2.6 Отрисовка

#### 2.6.1 Предложения

Предложения отрисовываются с помощью java graphics2d api, на прозрачном фоне. Процесс отрисовки выглядит так. Для здания выбирается цена. Она равна самой близкой к искомой цене из всех предложений. То есть выбирается предложение для которого значение данного выражения минимально ( – целевая цена, – цена текущего предложения). Если же в здание нету предложений, то оно не рисуется. Дальше на основании разницы между целевой ценой и ценой выбранной для дома выбирается цвет. После на основании геоданных рисуется полигон выбранного цвета. Это повторяется для всех зданий попадающих в границы тайла. После готовая картинка отдается клиенту.

#### 2.6.2 Транспортная доступность

Дороги по началу отрисовывались тоже с помощью java graphics2d api, но из-за того что меня не устраивала скорость отрисовки, то я решил перейти на библиотеку cairo, которую использовал через JNI[[3]](#footnote-3). Разница в производительности показаны в Приложение 6. Можно видеть что Cairo выигрывает в среднем в 10 раз по сравнению с Graphics2D.

Отрисовка проводилась следующим методом. У PropertyMap’а запрашиваются все RoadGraphNode’ы, которые входят в границы тайла. Дальше для каждой RoadGraphNode’ы рисуются линии к нодам, с которыми она связана. Для отрисовки линии, сначала с помощью метода getNodeColor() берется цвет для начала и конца линии, выбирается толщина линии, зависящая от типа соединения(RoadType), и после с помощью градиента переходящего из цвета начала к цвету конца она рисуется.

При использование отрисовки через JNI, все происходит также, только вместо самой отрисовки линии, данные о ней записываются в массив чисел в формате (). А дальше этот массив передается в native функцию drawNativeCall(). Дальше уже код на C++ рисует все эти линии с использованием библиотеки cairo, и возвращает массив байт, содержащий готовое png изображение.

## 3. Клиент

Так как основной частью проекта является сервер, то клиент был создан только ради того что бы продемонстрировать работу сервера. Клиент очень прост и состоит из двух частей, карты и боковой панели, на которой отображается информация о предложениях и поля для настройки. Клиент хранит координаты левого верхнего угла, относительно верхнего левого края карты. Далее на основание этого отрисовываются тайлы и на них накладываются изображения сгенерированные сервером.

# Дальнейшее развитие

В дальнейшем я планирую на маленьких уровнях зума, когда отдельные дома становятся едва различимыми, вместо отдельных домов отрисовывать плавную тепловую карту, но подобие <https://квартиры-домики.рф/Карта-цен/Москва/>. Но мне будет необходимо пересмотреть алгоритм генерирования тепловой карты, из-за того что у меня нет возможности для её пререндера

Также я собираюсь заменить клиент на Python на веб клиент. Он будет на порядок удобнее, из-за того что его не понадобится устанавливать, а в будущем такой веб клиент, можно слить с каким-то агрегатором жилья или же превратить его в полноценный агрегатор.

Ещё я хочу попробовать использовать GPU для отрисовки тайлов, ведь производительность графических процессоров при отрисовки на порядок выше обычных.

# Заключение

В результате работы над этим проектом я многое узнал о геоданных и методах их обработки. Также мною были изучены алгоритмы работы с графикой.

Это был мой первый опыт написания крупных проектов на java, а также написания клиент серверных приложений. Я набрался опыта в работе с системой контроля версий Git. И научился писать JNI код для java.

В итоге у меня не получилось приложение способное заменить обычные агрегаторы жилья. Но зато у меня получилось приложение способное дополнить их ещё одним средством для поиска жилья.

# Источники

Проекция меркатора <https://zenodo.org/record/35392>

Все геоданные и тайлы карты <https://www.openstreetmap.org>

Описание формата osm [https://wiki.openstreetmap.org/wiki/OSM\_XML](https://wiki.openstreetmap.org/wiki/OSM_XML\)

Библиотека Cairo <https://www.cairographics.org/>

Библиотека и документация Netty <http://netty.io/>

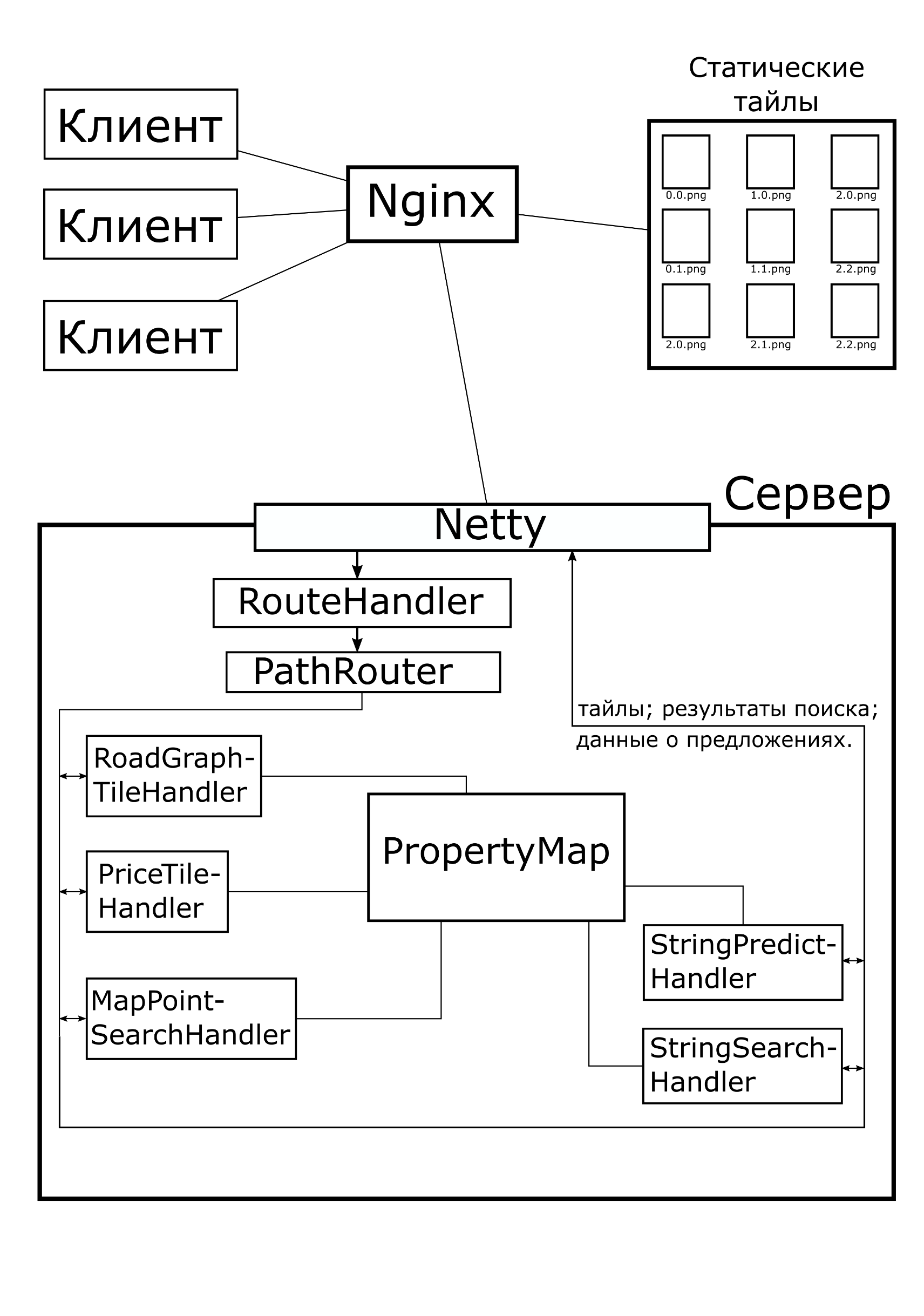
Сервер и документация Nginx <http://nginx.org/>

Библиотека и документация wxPython <https://www.wxpython.org/>

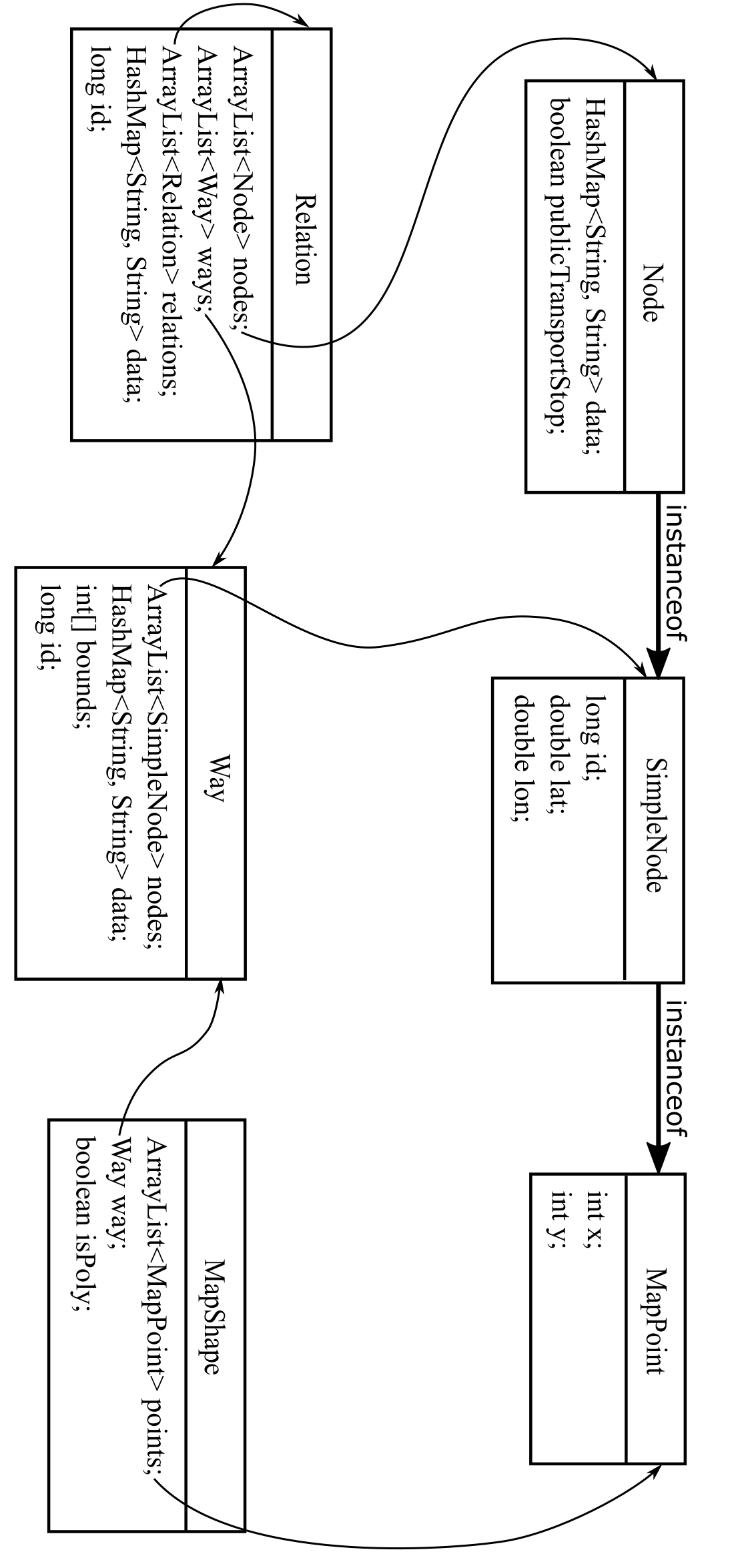
Библиотека minimal json <https://eclipsesource.com/blogs/2013/04/18/minimal-json-parser-for-java/>

Библиотека jol <http://openjdk.java.net/projects/code-tools/jol/>

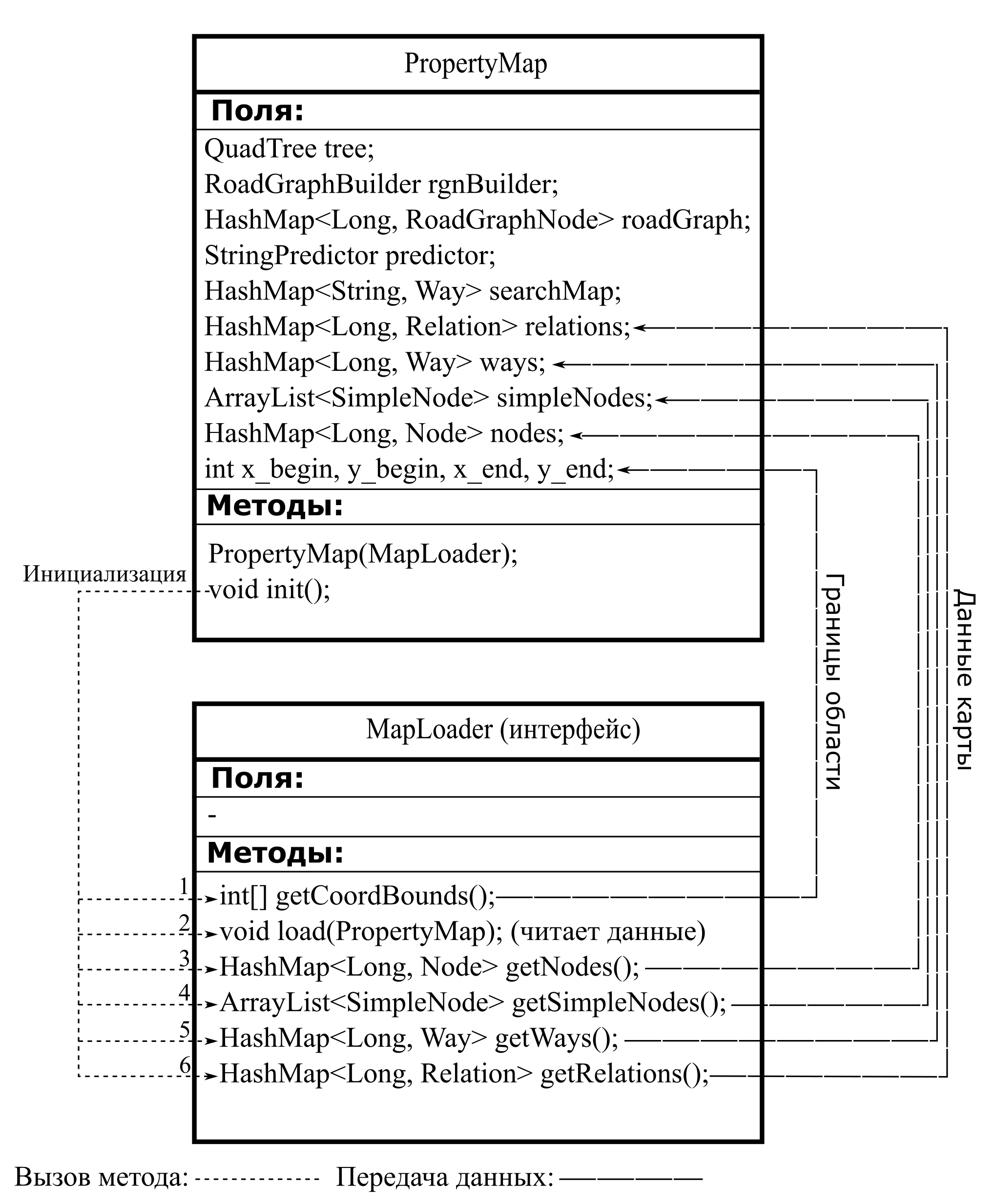
# Приложения

Структура проекта

Приложение 1

Классы, представляющие геоданные.

Приложение 2

Процесс загрузки гео данных из файла.

Приложение 3

Зависимость времени расчета от максимальной дальности.

Приложение 4

График выше, без расчета в глубину.

Приложение 5

Приложение 6

1. Overpass api – позволяет скачивать произвольную, ограниченную прямоугольником область карты. Поддерживает только формат osm. [↑](#footnote-ref-1)
2. Вложенность элементов не превышает 2 уровней, если считать корневой элемент 0 уровнем. [↑](#footnote-ref-2)
3. JNI – Java Native Interface, он позволяет из java кода вызывать методы из динамических библиотек(.dll и .so), имеющие определенную сигнатуру, описанную в java коде. [↑](#footnote-ref-3)