”Федеральное государственное бюджетное образовательное учреждение
высшего образования «Санкт-Петербургский государственный
университет

Кафедра системного программирования

Юрьев Семен Юрьевич

Архитектура клиента в Web Modeling Project

Курсовая работа

Научный руководитель:
к.т.н., ст. преп. Брыксин Т. А.

Санкт-Петербург
2016

Оглавление

Введение

Существует такое понятие, как визуальное программирование – способ создания программы для ЭВМ путём манипулирования графическими объектами вместо написания её текста. В качестве примера такого типа программирования можно привести проектирование интерфейса с помощью форм в некоторых IDE (к примеру, Qt Creator) и составление UML-диаграмм. В связи с развитием сети Internet и популярностью мобильных сенсорных устройств (КПК, смартфоны, планшеты), появилась идея предоставить веб-инструмент для визуального программирования. Действительно, доступность инструмента отовсюду и его независимость от платформы могут сделать его весьма удобным для разработки.

Эта идея положила начало проекту QReal-web[1]. Он задумывался как редактор диаграмм – один из самых наглядных и популярных подвидов визуального программирования. Основными целями этого проекта были нахождение необходимых инструментов для реализации такого проекта и написания прототипа редактора даграмм. По аналогии с разрабатываемым на кафедре системного программирования СПбГУ проекта TRIK-studio[2], в качестве языка диаграмм был выбран язык диаграмм роботов.

QReal-web справился со своей задачей, однако он имеет потенциал быть средой не только для моделирования поведения роботов, но и для других процессов, например, BPMN (Business Process Model and Notation). К тому же, сам редактор можно улучшить множеством способов. На базе старого проекта был основан новый – WMP (Web Modeling Project)[4]. Его основные принципы:

- Открытый код
- Взаимозаменяемость модулей
- Простота добавления новой функциональности

Web Modeling Project задумывался так, чтобы иметь возможность постоянно расширяться. Система QReal-web уже имела внушительные размеры, и плохая архитектура нового проекта при расширении может сделать его очень сложным для понимания и последующего расширения. QReal-web отлично выполнял свои задачи, но был рассчитан только на работу с диаграммами роботов. Проект представлял из себя "монолит", добавление функциональности к которому представляло сложную задачу. Появилась необходимость разработать новую архитектуру, содержащую все сделанные в QReal-web наработки, которая, к тому же, будет легко дополняться новой функциональностью. Если это будет сделано, то WMP при его огромном потенциале

[1]https://github.com/qreal/qreal-web/wiki
[2]http://blog.trikset.com/p/trik-studio.html

для роста будет хорошим студенческим проектом и впоследствии сможет потягаться с аналогичными крупными проектами.

Первые шаги в этой области были сделаны в курсовой работе Артемия Безгузикова – разделение приложения на взаимодействующие друг с другом сервисы[3]. Данная работа является следующим шагом на пути к решению описанной выше задачи.

Постановка задачи

Таким образом, основные три этапа решения поставленной задачи таковы:

1. Документирование текущей архитектуры клиента
2. Разработка новой архитектуры клиента
3. Реализация разработанной архитектуры клиента

1. Обзор текущего решения

1.1. Обзор текущей сервисной архитектуры

Проект QReal-web имеет монолитную архитектуру, где в качестве каркаса выступает Spring MVC[2] (рис.1), реализующий шаблон проектирования Model-View-Controller.

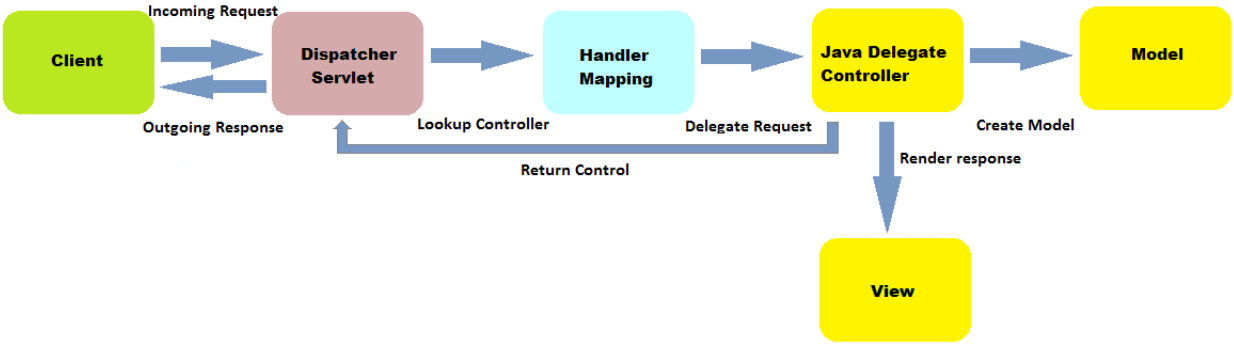

Рис. 1: Структура Spring MVC

Параллельно с разработкой последней версии QReal-web проводилась работа по переходу к сервисной архитектуре[1] в рамках курсовой работы Артемия Безгузикова[3]. Был создан прототип и исследованы различные подходы к построению сервисной архитектуры. На этом этапе уже была произведена декомпозиция монолитной серверной части. Серверный код был разбит на сервисы, а взаимодействие между ними, в свою очередь, осуществлялось с помощью инструментария Apache Thrift[3]. Он предполагает описание интерфейсов на языке Thrift для клиент-серверного взаимодействия и последующую их компиляцию на необходимые языки программирования. Таким образом, общение сервисов друг с другом происходит посредством RPC-вызовов, сгенерированных с помощью Thrift.

Клиентский код также в свою очередь разбивается на плагины – независимые совокупности JavaScript и TypeScript кода, описывающие определенный функционал. Были реализованы плагины для аутентификации, редактирования диаграмм и панели инструментов.

[3]https://thrift.apache.org/docs

Таким образом, был проведен следующий переход (рис.2):

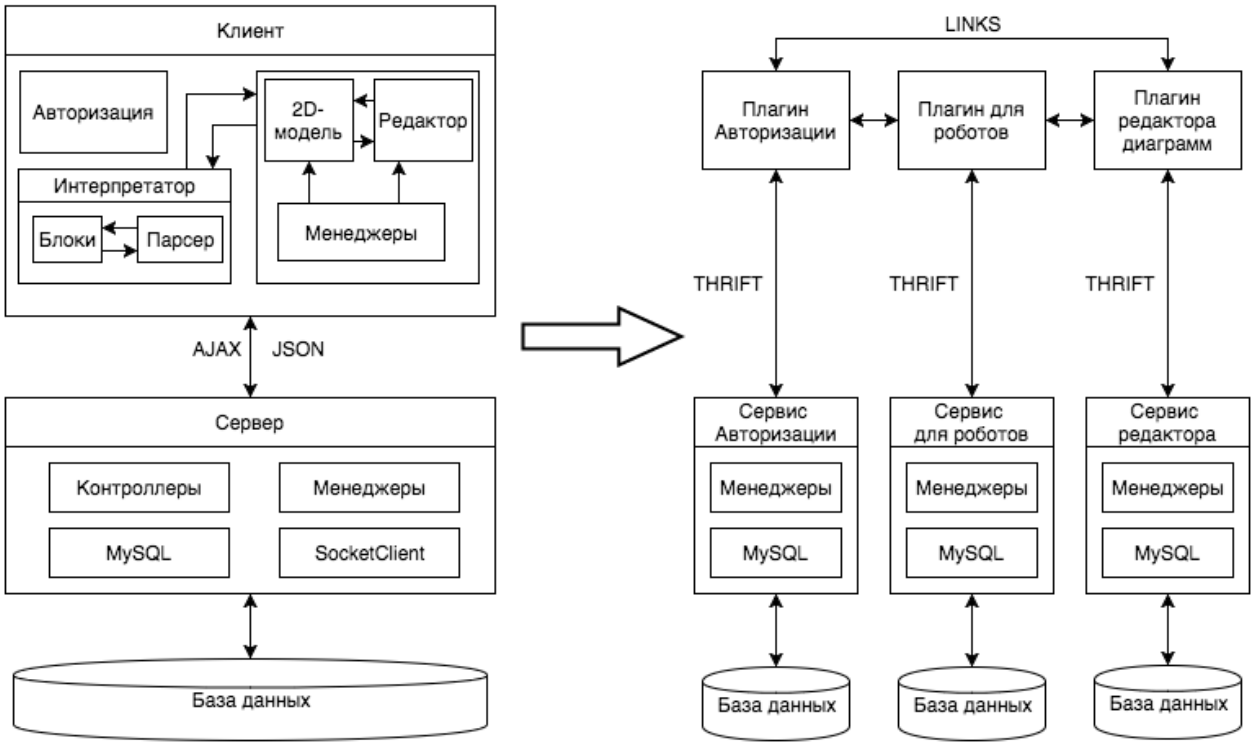

Рис. 2: Переход от монолитной архитектуры к сервисной

Созданный прототип позволял выбрать тот или иной плагин на web-странице и запускать его отдельно от остальных. Сервисы работали самостоятельно и независимо ни от клиетских плагинов, ни от друг друга.

Эта работа показала, что придуманные методы и подходы позволяют справиться с задачей декомпозиции и упрощения приложения. Аналогично тому, как разбита на сервисы серверная часть приложения в описанной архитектуре, необходимо разбить и клиентскую часть.

1.2. Новый проект сервисной архитектуры

В качестве новой сервисной архитектуры для Web Modeling Project был взят проект, предложенный в курсовой работе Артемия Безгузикова[3], и немного доработан (рис.3). Самое важное отличие состоит в наличии Mapping service. Он конструирует веб-страницу, передаваемую пользователю, в зависимости от того, какие ему необходимы приложения. Например, если нужны редактор диаграмм и 2D-модель, то он с помощью заданных ему шаблонов совместит два приложения на одной странице (рис.4). Таким образом, пользователь может сам настроить для себя веб-приложение, в котором ему будет удобно работать.

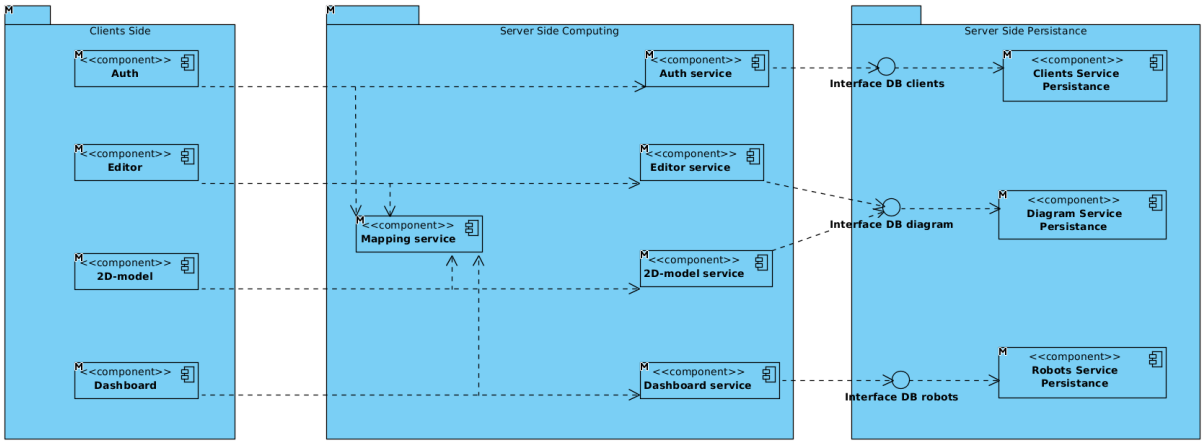

Рис. 3: Новая сервисная архитектура

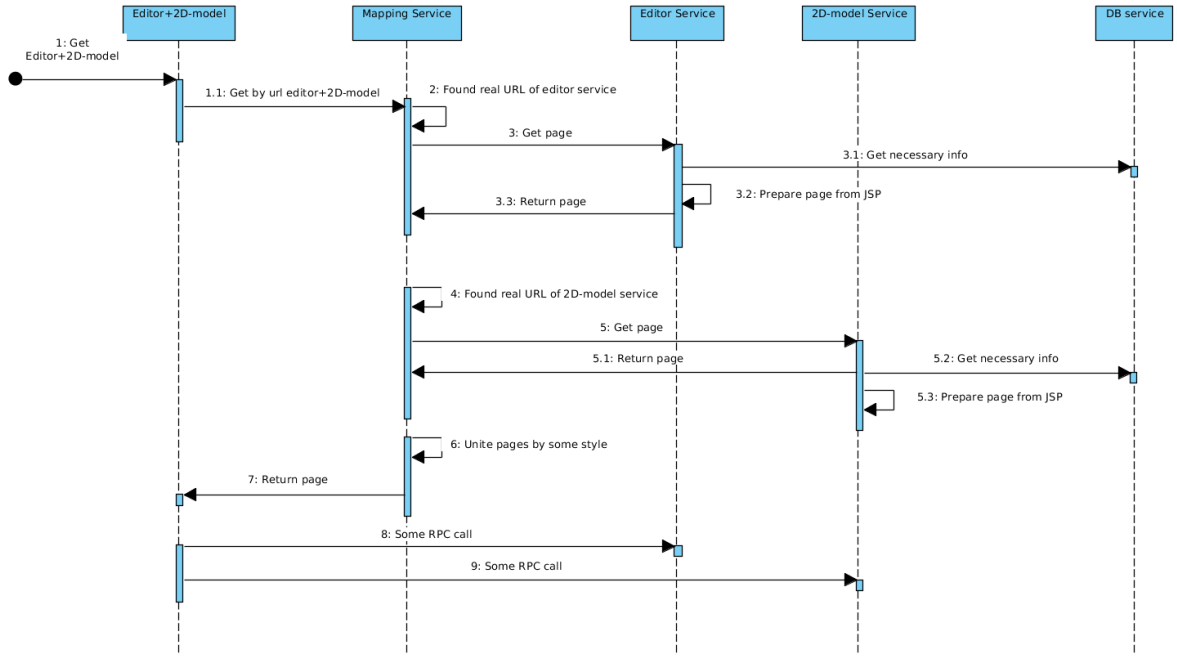

Рис. 4: Процесс взаимодействия сервисов в новой архитектуре

1.3. Документирование текущей архитектуры клиента

Для того, чтобы понять взаимосвязи в текущем клиенте, необходимо было составить диаграмму классов клиенсткого приложения. Клиент приложения QReal-web состоял из двух основных частей – модуля SharedResources и редактора диаграмм роботов. SharedResources (рис.5) объединял в себе максимально абстрактные код и и ресурсы, которые могут понадобиться в различных частях клиента. Его составные части:

- DiagramCore – ядро редактора диаграмм
- TwoDModelCore – ядро 2D-модели

- Utils – вспомогательные модули
- Resources – библиотеки Javascript, стили CSS, шрифты, изображения и веб-страницы

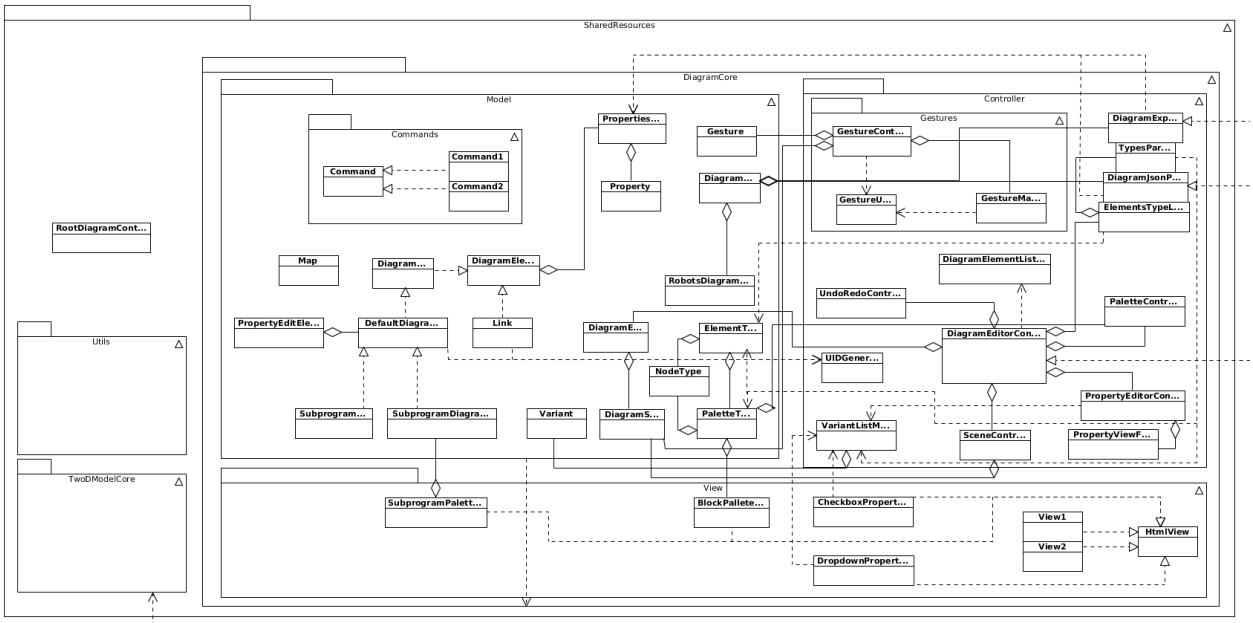

Рис. 5: Модуль SharedResources

У этого решения сразу видны несколько проблем:

- Независимо от того, какая часть SharedResources была нужна приложению, "подтягивался" модуль полностью
  - Например, сервисам проектной панели и авторизации, нуждавшихся только в Resources, был совсем не нужен код, относящийся к диаграммам и 2D-модели
- Объединен код для двух потенциально независимых приложений – редактора и 2D-модели

Следующие проблемы касались DiagramCore:

- Сильная взаимосвязь компонент
  - Чтобы добавить свою функциональность, необходимо разбираться во всей системе в целом и, возможно, значительно ее поменять
- Перегруженность ядра
  - Часть этой функциональности является излишней и может быть убрана из ядра без ущерба основному назначению – рисованию диаграмм

К самому редактору (рис.6) претензий (кроме объединения редактора и 2D-модели) нет. Он устроен достаточно просто, связи между его компонентами прозрачны.

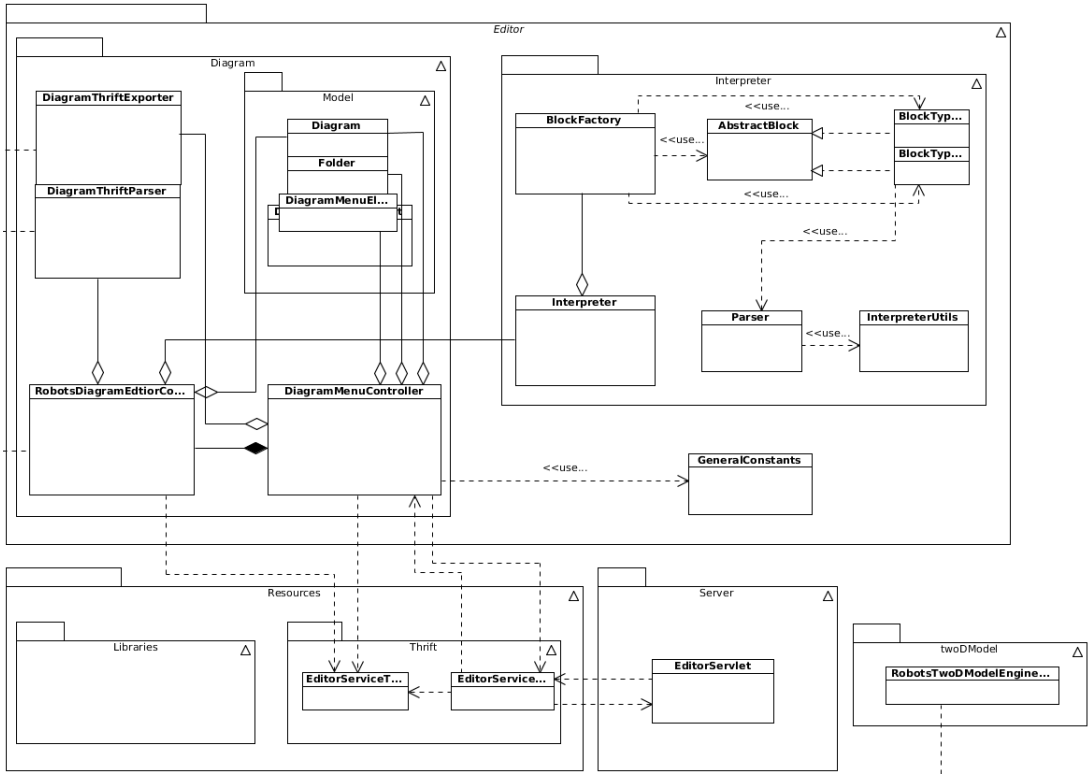

Рис. 6: Редактор и его взаимодействие с сервером

1.4. Требования к новой архитектуре клиента

Обзор текущей архитектуры помог выявить конкретные проблемы, которые мешают дальнейшему продвижению проекта. Новая архитектура таких проблем иметь не должна. Поэтому был составлен список требований к архитектуре, с помощью которого впоследствии будет оценена данная работа:

- Редактор и 2D-модель – отдельные независимые приложения
- Модуль SharedResources переформирован
  - Ядро редактора и ядро 2D-модели находятся в соответствующих приложениях, при необходимости подключаться в клиентские приложения будут только ресурсы
- Ядро редактора лишено функциональности, которая не относится к непосредственно составлению диаграмм
- Наличие механизма, позволяющего просто добавлять функциональность и отключать ее при необходимости

2. Разработка новой архитектуры клиента

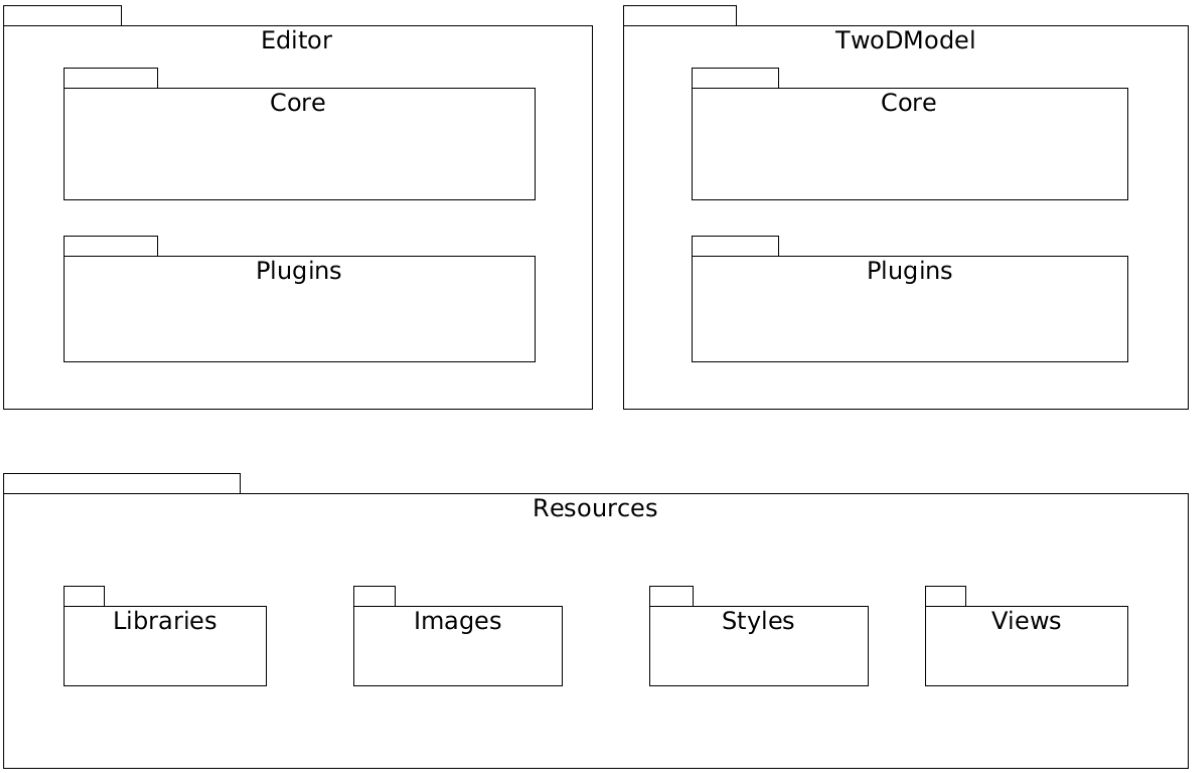

Рис. 7: Проект новой архитектуры клиента

На рис.7 представлен проект архитектуры, удовлетворяющий описанным выше требованиям. В нем два приложения Editor (редактор) и TwoDModel (2D-модель) независимы друг от друга. Каждое из двух приложений состоит из основных двух частей – Core (ядро) и Plugins (система плагинов).

Ядро содержит минимальную функциональность, необходимую для запуска приложения. Оно должно иметь как можно более простую структуру для того, чтобы в нем можно было проще разобраться и им было проще пользоваться.

Система плагинов была разработана, чтобы достичь одновременно двух целей – разгрузить ядро от излишней функциональности, и заодно дать возможность начинающим разработчикам написать что-то несложное для этого проекта. Плагин – это небольшой модуль, по желанию подключаемый к сборке проекта. Основная идея состоит в том, что независимо от того, подключили мы плагин или нет, проект будет работать исправно. Подключение плагина дает приложению новую функциональность. Контроль за работой плагинов осуществляетс класс PluginController. Любое обращение к функциональности плагинов идет через него. PluginController контролирует создание и использование классов плагинов. Если код плагина не был включен в сборку, PluginController не даст приложению упасть при создании или использовании несуще-

ствующего класса. Соответственно, новые разработчики имеют возможность написать свой плагин, и с минимальными проблемами вставить его в программу – если что-то пойдет не так, плагин всегда можно отключить. На рис.8 наглядно показан описанная выше схема: DiagramScene может хранить экземпляр класса GesturesController, но создавать и использовать их напрямую не может.

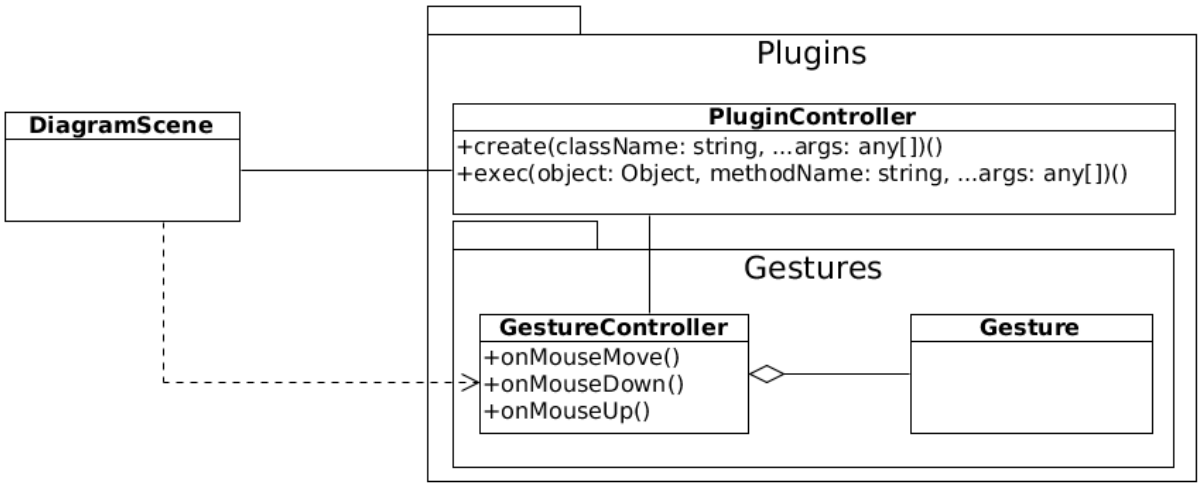

Рис. 8: Система плагинов

Рис.9 содержит более подробный план архитектуры клиента.

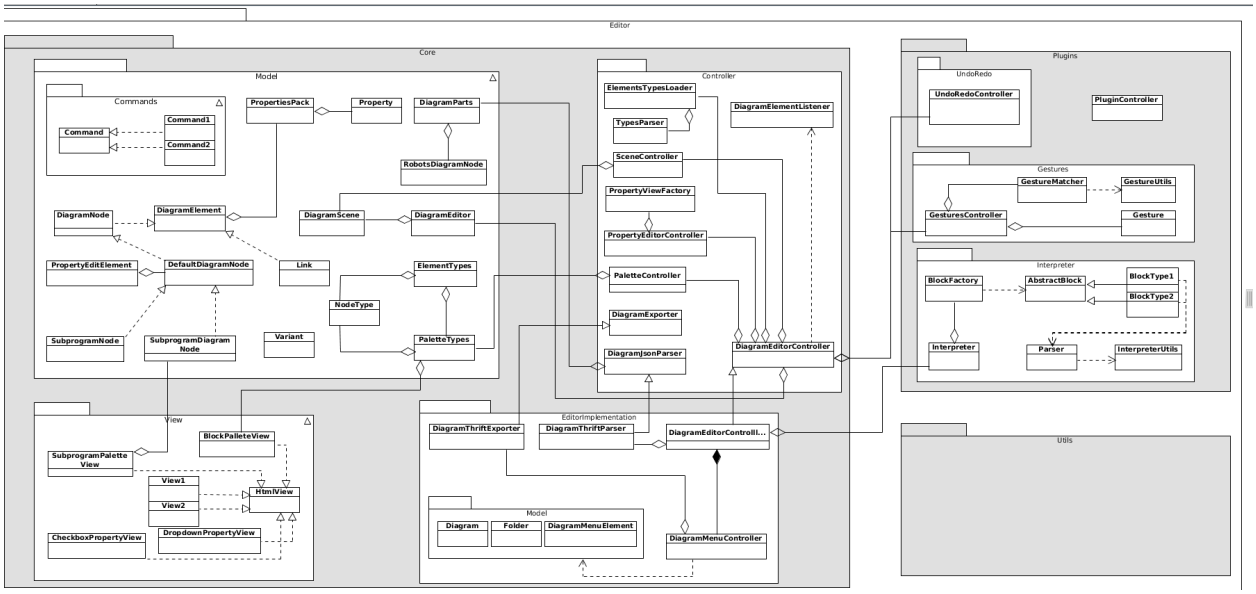

Рис. 9: Диаграмма классов новой архитектуры клиента

3. Реализация

3.1. Подготовительная работа (реализация новой сервисной архитектуры)

К сожалению, код, написанный Артемием Безгузиковым в его курсовой работе, был основан на старой версии QReal-web. В новых версиях добавился большой объем функциональности, которую не хотелось терять. Было принято решение о пошаговом переходе на новую архитектуру, включавшем в себя:

1. Разделение кода на независимые подразделы
2. Переход на межъязыковое взаимодействие с помощью Thrift
3. Выделение сервисов (редактора и 2D-модели, авторизации, проектной панели)

Благодаря имеющимся курсовой работе и наработкам кода, трудности в реализации были скорее техническими. Изменения были необходимы не только в сервере и клиенте, но и в самом процессе их взаимодействия. Также требовалось значительное понимание архитектуры проекта и его внутренних взаимосвязей.

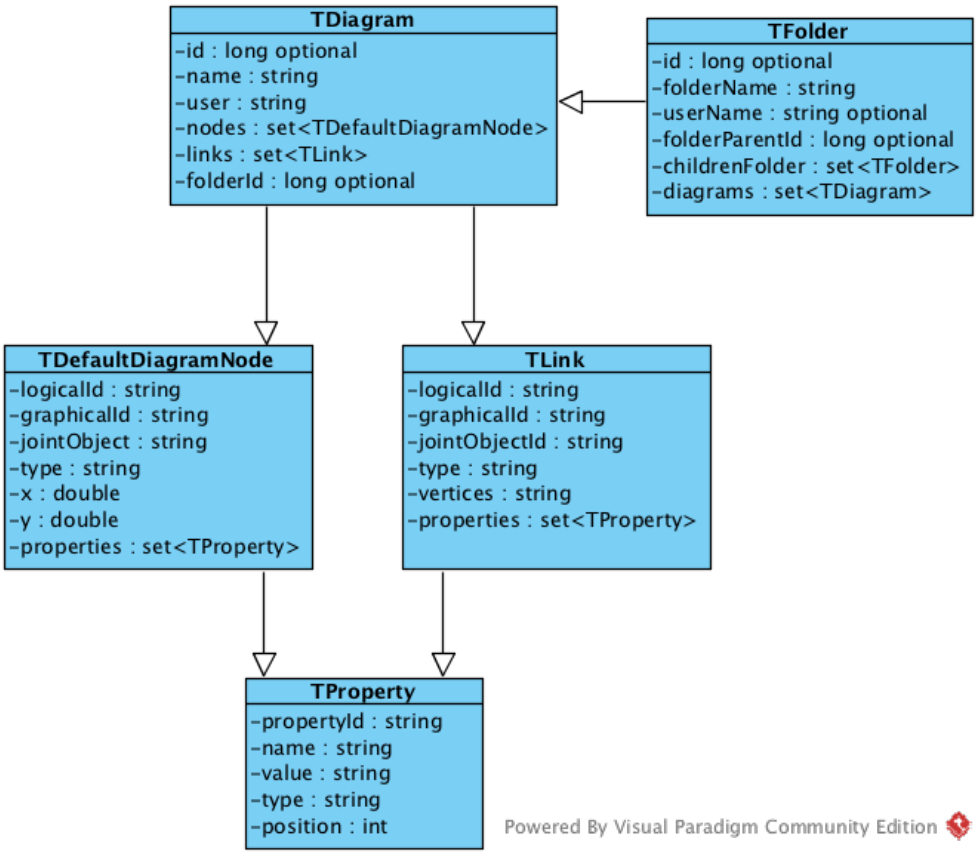

Рис. 10: Диаграмма структур на языке Thrift

В рамках своей курсовой на подготовительном этапе я занимался выделением сервиса для работы с диаграммами. Основной задачей был переход на межъязыковое взаимодействие с помощью Thrift редактора и 2D-модели. Для достижения этой цели была разработана система Thrift-структур (рис.10)

Также был реализован сервис на языке Thrift, позволяющий манипулировать диаграммами. В серверной части приложения был реализован обработчик сервисных вызовов. Этот обработчик, в свою очередь, оборачивался в сервлет и запускался при инициализации приложения. На клиентской стороне приложения были реализованы классы DiagramThriftExporter и DiagramThriftConverter, позволяющие конвертировать внутренние структуры клиента во Thrift-структуры и обратно.

3.2. Реализация новой архитектуры клиента

Реализация разработанной архитектуры проходила в несколько этапов:

1. Разделение редактора и 2D-модели
2. Расформирование SharedResources
3. Реализация системы плагинов
4. Перенос части функционала ядра редактора в систему плагинов
5. Разделение редактора диаграмм

3.2.1. Разделение редактора и 2D-модели

Этот этап состоял целиком из технической работы. Основной трудностью в нем была необходимость понимания того, как работает система сборки Maven[4] и компиляция из TypeScript в JavaScript с помощью Grunt[5].

3.2.2. Расформирование SharedResources

Необходимо было переместить и подключить к сборке кодовую базу для редактора и 2D-модели. Также важным частью этого этапа было распределение ресурсов: какие-то ресурсы были нужны только для редактора, какие-то только 2D-модели. В SharedResources остались некоторые библиотеки, изображения и веб-страницы, необходимые всем или почти всем клиентским сервисам.

[4]https://maven.apache.org/pom.html
[5]http://gruntjs.com/

3.2.3. Реализация системы плагинов

В текущем виде у этой системы такая логика:

- Если необходимо создать экземпляр класса:
  - Проверить, что класс состоит в текущей сборке. Если состоит, то вернуть новый экземпляр
- Если необходимо вызвать метод класса:
  - Узнать, к какому классу принадлежит данный объект
  - Проверить, существует ли запрашиваемый метод у данного класса. Если существует, то выполнить этот метод

Гарантия того, что новый экземпляр класса или его методы не будут запускаться вне этой системы (и тем самым не прервут работу приложения), очень проста. Grunt компилирует TypeScript в JavaScript помодульно. Использование классов не из своего модуля осуществляется с помощью интерфейсных файлов. Интерфейсы для каждого плагина пусты и не содержат ни одного метода. Тем самым, и использовать их извне никто не может.

3.2.4. Перенос части функционала ядра редактора в систему плагинов

Не относящимися к основой функциональности были признаны два модуля: undo-redo и gestures. Первый позволял откатывать сделанные изменения в диаграмме назад (undo) и возращать откаченное вперед (redo). Второй модуль давал возможность рисовать жесты на поле для диаграмм и тем самым быстро (без поиска в палитре элементов) помещать на сцену команды для диаграмм. Что еще важнее, он позволял в одно движение создавать связь между командами. И тот, и другой модуль были, безусловно, полезны, но не относились к основной функциональности системы.

3.2.5. Разделение редактора диаграмм

Редактор состоял из двух основных частей – самого редактора и интерпретатора. Редактор является посредником между сервером и пользователем. Его функциональность фундаментальна, и, безусловно, должна находиться в ядре приложения. Интерпретатор – модуль, который запускает команды диаграмм в определенной в них последовательности. В текущем своем виде он был завязан на диаграммы роботов, и занял свое месте в числе плагинов. Этот этап также помог избавиться от огромных интерфейсных файлов – до этого редактору и интерпретатору для работы было необходимо знать о почти всей функциональности ядра.

4. Оценка результатов

После обзора текущего решения были выдвинуты критерии качественного решения для данной задачи. Чтобы оценить проделанную работу, пройдемся по каждому из критериев:

- Редактор и 2D-модель – отдельные независимые приложения

Выполнено.

- Модуль SharedResources переформирован
  - Ядро редактора и ядро 2D-модели находятся в соответствующих приложениях, при необходимости подключаться в клиентские приложения будут только ресурсы

Выполнено.

- Ядро редактора лишено функциональности, которая не относится к непосредственно составлению диаграмм

Выполнено. Возможно, некоторая функицональность еще может быть выделена из ядра в плагины, но этого точно нельзя сделать без значительной его переработки.

- Наличие механизма, позволяющего просто добавлять функциональность и отключать ее при необходимости

Выполнено частично: был реализован простой механизм, позволяющий контролировать использование дополнительной функциональности. К сожалению, данная система не очень изящна и эффективна, и безусловно нуждается в доработке или замене. Одним из вариантов замены является паттерн Dependency Injection[6].

[6]https://en.wikipedia.org/wiki/Dependency_injection

Заключение

В рамках данной работы было выполнено:

- Задокументирована текущая архитектура клиента
- Выявлены проблемы текущей архитектуры
- Разработана новая архитектура клиента, лишенная описанных проблем
- Реализована разработанная архитектура клиента

Основная цель данной работы также выполнена – проект стал проще для понимания и расширения силами студентов.

Список литературы

[1] Jenkov Jakob. SOA - Service Oriented Architecture. — URL: http://tutorials.jenkov.com/soa/index.html (online; accessed: 01.08.2016).

[2] Wheeler Willie White Joshua. Spring in Practice. — Manning Publications Co, 2013.

[3] Безгузиков Артемий. Микросервисная архитектура QReal-Web. — 2016. — URL: http://se.math.spbu.ru/SE/YearlyProjects/spring-2016/344/344-Bezguzikov-report.pdf/view (дата обращения: 01.08.2016).

[4] СПбГУ Кафедра системного программирования. Web Modeling Project. — 2016. — URL: https://github.com/qreal/wmp (дата обращения: 01.08.2016).