МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение высшего образования

«ПЕНЗЕНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

Факультет
вычислительной техники

Кафедра
«Вычислительная техника»

Направление подготовки: 09.03.01 «Информатика и вычислительная техника»

Профиль подготовки: «Вычислительные машины, комплексы, системы и сети»

БАКАЛАВРСКАЯ РАБОТА

на тему

ГРАФИЧЕСКИЙ РЕДАКТОР ЯЗЫКА ЗАПРОСОВ К RDF-ХРАНИЛИЩАМ ДАННЫХ

Студент _____ (подпись, дата) 19.03.2023 Муромский Дмитрий Александрович

Руководитель _____ (подпись, дата) 19.06.2023 Дубинин В.Н.

Нормоконтролёр _____ (подпись, дата) 19.06.2023 Бычков А.С.

Работа допущена к защите (протокол зас. кафедры от 02.06.2023 г. № 14)

Заведующий кафедрой _____ (подпись) Митрохин М.А.

Работа защищена с отметкой отлично (протокол заседания ГЭК от 20.06.2023 г. № 21)

Секретарь ГЭК _____ (подпись) Гурин Е.И.

Пенза 2023

Федеральное государственное бюджетное образовательное учреждение
высшего образования
«ПЕНЗЕНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

Факультет
вычислительной техники

Кафедра
«Вычислительная техника»

«Утверждаю»
Заведующий кафедрой
_____М.А. Митрохин
«26» апреля 2023

ЗАДАНИЕ
НА БАКАЛАВРСКУЮ РАБОТУ

1. Студент Муромский Дмитрий Александрович гр. 19ВВП1 факультета вычислительной техники

2. Тема работы «Графический редактор языка запросов к RDF-хранилищам данных»

Тема утверждена приказом ПГУ № 458/о от "26" апреля 2023г.

3. Руководитель бакалаврской работы д.т.н. профессор кафедры ВТ ПГУ Дубинин В.Н.

4. Задание на работу

Разработать графический язык для описания и обработки SPARQL-запросов на основе функционально-блочных диаграмм. Язык должен допускать иерархическую структуризацию на основе концепции композитного функционального блока. Атомарные вычислительные функции должны быть реализованы в базисных функциональных блоках на языке JavaScript. Для выполнения запроса к RDF-хранилищам данных используется SPARQL-блок, причём сам SPARQL-запрос формируется на основе графического представления подмножества языка SPARQL.

Разработать и программно реализовать графический редактор для формирования функционально-блочных диаграмм, включая специальный редактор для формирования графического представления SPARQL-запросов.

Язык программирования – C/C++

Среда программирования - QtCreator

5. Перечень подлежащих разработке вопросов

1) Обзор и анализ визуальных языков и графических редакторов
2) Разработка графического языка функционально-блочных диаграмм для описания и обработки SPARQL-запросов
3) Разработка графического подъязыка описания SPARQL-запросов
4) Разработка и программная реализация графического редактора

6. Календарный график выполнения работы

№ п/п	Наименование этапов работы	Объем работы	Срок выполнения	Подпись руководителя
1	Обзор и анализ визуальных языков и графических редакторов	15%	26.04.23-29.04.23	
2	Разработка графического языка функционально-блочных диаграмм для описания и обработки SPARQL-запросов	20%	30.04.23-6.05.23	
3	Разработка графического подъязыка описания SPARQL-запросов	20%	7.05.23-13.05.23	
4	Разработка и программная реализация графического редактора	25%	14.05.23-20.05.23	
5	Тестирование и отладка	10%	21.05.23-23.05.23	
6	Оформление выпускной работы.	10%	24.05.23-30.05.23	

Дата выдачи задания "26" 04 2023 г.

Руководитель бакалаврской работы _____. Дубинин В.Н.

(подпись, дата)

Задание к исполнению принял студент _____. Муромский Д.А.

(подпись, дата)

Работу к защите допустить

Декан факультета _____ 19.06.2023. Фионова Л.Р.

(подпись, дата)

Реферат

Работа содержит 109 лист., 57 рис., 2 табл., 17 источ., 3 прил.

ГРАФИЧЕСКИЙ РЕДАКТОР ЯЗЫКА ЗАПРОСОВ К RDF-ХРАНИЛИЩАМ ДАННЫХ

Цель исследования, в рамках которого выполнялась данная работа – разработка математического, лингвистического и программного обеспечения для поддержки проектирования специализированных информационно-вычислительных систем, предназначенных для оценки объектов энергетической инфраструктуры, на основе функционально-блочных диаграмм и *RDF*-хранилищ данных.

Цель данной работы – разработка графического языка функционально-блочных запросно-вычислительных диаграмм, включающих базисные, композитные и *SPARQL*-блоки, а также программная реализация графического редактора, позволяющего создавать, редактировать и сохранять диаграммы данного вида.

Задачи, решаемые в процессе выполнения проекта:

- разработка конкретного синтаксиса графического языка функционально-блочных запросно-вычислительных диаграмм, включая подъязык графического описания *SPARQL*-запросов;

- разработка и программная реализация графического редактора функционально-блочных запросно-вычислительных диаграмм.

Области применения:

- спецификация (формализованное описание) на языке функционально-блочных запросно-вычислительных диаграмм требований к техническим системам, описанным с использованием *RDF*-онтологий, выраженных в числовом виде;

- проектирование, разработка и реализация специализированных информационно-вычислительных систем на основе функционально-блочных запросно-вычислительных диаграмм.

Информация о реализации и требования к ПО

Язык программирования графического редактора – *C/C++*. Среда программирования – *QtCreator*.

Операционные системы – *Windows 7/10/11* и *Ubuntu 20.04* (и выше).

Содержание

Реферат 2
Содержание 4
Перечень сокращений и обозначений 6
Введение 7
1 Графический редактор языка запросов к rdf-хранилищам данных 9
1.1 Обзор и анализ визуальных языков и графически редакторов 9
1.1.1 Графические языки программирования, моделирования и проектирования 9
1.1.2. Графические языки запросов 14
1.1.3. Графические редакторы 18
1.1.4. Постановка задачи 21
1.2. Язык функционально-блочных диаграмм для описания и обработки SPARQL-запросов 21
1.2.1. Базисные блоки 21
1.2.2. Композитные блоки 25
1.2.3 *SPARQL*-блоки 27
1.2.4 Связи между блоками 29
1.2.5 Ограничения, налагаемые на структуру 30
1.3 Разработка графического редактора для функционально-блочных диаграмм 31
1.3.1 Цели создания программы, её назначение и функциональные возможности 31
1.3.2 Выбор программно-аппаратной платформы 32
1.3.3 Архитектура программы 34
1.3.4 Сохранение данных, блоков и проектов 36
1.3.5 Основные структуры данных 36
1.3.6 Алгоритмы работы графического редактора 41
1.3.7 Руководство пользователя 46
2 Безопасность жизнедеятельности 68

2.1 Введение 68
2.2 Требования к помещениям для работы с ПЭВМ 70
2.3 Требования к освещению на рабочих местах, оборудованных ПЭВМ 71
Заключение 74
Список использованных источников 76
Приложение А - Листинг программы 78
Приложение Б - *UML*-диаграммы 98
Приложение В - Презентация 103

Перечень сокращений и обозначений

В настоящей работе применяются следующие сокращения и обозначения

ПО - программное обеспечение

ПЭВМ - персональная электронная вычислительная машина

FBD (Function Block Diagram) – это графический язык программирования высокого уровня

FB (Function Block) - функциональный блок

FBQL (Function Block Query Language)- название языка который мы использовали для нашего редактора

FBQL Editor - название нашего редактора

GraphViz - пакет утилит для автоматической визуализации графов, заданных в виде описания на языке *DOT*

GUI (Graphical User Interface)- графический интерфейс пользователя

IDE (Integrated Development Environment)- интегрированная среда разработки

JavaScript - язык программирования

JSON - текстовый формат обмена данными

KDE - окружение среды для *Linux*

Linux - операционная система

RDF (Resource Description Framework) - модель данных

SPARQL - язык запросов к данным, представленным по модели *RDF*

UML (Unified Modeling Language) - язык моделирования

WYSIWYG - свойство прикладных программ или веб-интерфейсов

Введение

Современное информационное общество охватывает огромные объёмы данных, которые постоянно растут и становятся все более сложными для обработки и анализа. Одним из основных источников данных являются *RDF*-хранилища данных, которые представляют собой распределенные системы для хранения и обработки знаний в виде *RDF*-графов.

RDF-графы являются стандартным форматом представления семантических данных, используемым в семантическом вебе и других областях, где требуется описывать информацию с использованием триплетов субъект-предикат-объект. Однако, для эффективной работы с *RDF*-хранилищами и выполнения запросов к данным требуется мощный инструмент, который бы обеспечивал гибкость и удобство взаимодействия с графическим представлением *RDF*-данных.

В данном дипломном проекте мы сфокусировались на разработке графического редактора языка запросов к *RDF*-хранилищам данных. Наша цель состояла в создании интуитивно понятного и удобного инструмента, который позволит пользователям легко формулировать и выполнять запросы к *RDF*-графам без необходимости вручную писать запросы на специализированном языке, таком как *SPARQL*.

В процессе разработки мы уделяли особое внимание функциональности и эргономике редактора. Мы стремились создать инструмент, который позволит пользователям интуитивно создавать и модифицировать запросы, визуально представлять структуру данных и результаты запросов, а также предоставлять расширенные возможности по обработке результатов.

Наш графический редактор языка запросов к *RDF*-хранилищам данных является результатом инновационного подхода к визуализации и обработке семантических данных. Мы уверены, что его использование значительно упростит работу с *RDF*-графами и повысит эффективность анализа данных в различных областях, включая семантический веб, биоинформатику, социальные сети и многие другие.

Данный дипломный проект представляет собой детальное описание разработки графического редактора языка запросов к *RDF*-хранилищам данных, включая анализ требований, проектирование архитектуры, разработку и реализацию функциональности.

1 Графический редактор языка запросов к rdf-хранилищам данных

1.1 Обзор и анализ визуальных языков и графически редакторов

1.1.1 Графические языки программирования, моделирования и проектирования

Существует множество графических языков программирования, моделирования и проектирования, каждый из которых предназначен для определённых задач и областей применения. Ниже приведены некоторые из наиболее распространённых и известных графических языков:

- *UML* является стандартным языком моделирования, используемым для разработки и документирования программных систем. *UML* предоставляет набор нотаций и диаграмм, таких как диаграммы классов, диаграммы последовательности и диаграммы состояний, которые помогают визуализировать и описывать структуру и поведение системы;

- *BPMN* предоставляет нотацию для моделирования бизнес-процессов. Он позволяет описывать шаги, потоки данных, события и роли в рамках бизнес-процесса. *BPMN*-диаграммы используются для анализа, улучшения и автоматизации бизнес-процессов;

- *ER*-диаграммы используются для моделирования структуры баз данных. Они помогают визуализировать сущности, атрибуты и связи между ними. *ER*-диаграммы широко применяются при проектировании баз данных;

- *SysML* является расширением *UML* и предназначен для моделирования сложных системных архитектур. Он используется для анализа, проектирования и документирования систем, таких как авионика, автомобильные системы и промышленные процессы;

- *SDL* - язык моделирования, который широко используется в телекоммуникационной индустрии для описания и проектирования протоколов и систем связи;

- *MATLAB* и *Simulink. MATLAB* - это язык программирования и среда разработки, специализированная для численных вычислений и алгоритмического моделирования. *Simulink* - это графическая среда моделирования и симуляции, расширение *MATLAB*, которая используется для моделирования и проектирования динамических систем, таких как управление и сигнальная обработка.

Это лишь небольшой набор из множества графических языков программирования, моделирования и проектирования, доступных в различных областях разработки программного обеспечения, инженерии и бизнес-процессов. Выбор конкретного языка зависит от задачи, которую необходимо решить, и от предпочтений разработчика или инженера.

UML [1] - это стандартный графический язык моделирования, который широко применяется для анализа, проектирования и документирования программных систем и бизнес-процессов. *UML* предоставляет набор нотаций и диаграмм, которые позволяют разработчикам и аналитикам визуализировать и описывать различные аспекты системы. Некоторые из основных диаграмм *UML* включают:

- диаграммы классов используются для моделирования структуры системы. Они позволяют описывать классы, атрибуты, методы и отношения между классами;

- диаграммы объектов представляют конкретные экземпляры классов и связи между ними. Они полезны для демонстрации взаимодействия объектов в рамках определённой операции или сценария;

- диаграммы последовательности описывают взаимодействие объектов в рамках определённого сценария или функции. Они демонстрируют, как объекты обмениваются сообщениями и взаимодействуют во временном порядке;

- диаграммы состояний моделируют поведение объекта или системы в различных состояниях и переходах между ними. Они полезны для моделирования систем, которые имеют определённые состояния и изменяются в зависимости от внешних событий;

- диаграммы деятельности описывают поток управления и поведение системы в виде серии действий. Они используются для моделирования бизнес-процессов, алгоритмов или процессов принятия решений;

- диаграммы компонентов позволяют описать структуру и взаимодействие компонентов системы, а диаграммы развёртывания моделируют физическое размещение компонентов и их взаимодействие на аппаратном уровне.

UML является стандартом, поддерживаемым многими инструментами и платформами разработки, и он широко используется в индустрии разработки программного обеспечения для обеспечения согласованности и понимания системы между различными участниками проекта.

Диаграммы сущность-связь [2] - это графический язык моделирования, который используется для описания структуры и взаимосвязей данных в базе данных. Они позволяют представить сущности (объекты, концепции) и связи между ними, а также атрибуты, которые описывают каждую сущность. В *ER*-диаграммах есть несколько основных элементов:

- сущности представляют различные объекты или концепции, которые хранятся в базе данных. Например, веб-приложение для электронной коммерции может иметь сущности, такие как "пользователь", "продукт", "заказ" и т. д. Каждая сущность обычно имеет уникальный идентификатор (поле ключа);

- связи определяют отношения между сущностями. Например, у пользователя может быть связь с заказом, что означает, что каждый пользователь может иметь несколько заказов. Связи могут быть однонаправленными или двунаправленными и иметь определённые ограничения, такие как кардинальность (сколько объектов может быть связано) и обязательность связи;

- атрибуты представляют характеристики или данные, связанные с каждой сущностью. Например, атрибутами пользователя могут быть имя, адрес электронной почты и пароль. Атрибуты могут быть простыми (например, строка, число) или составными (например, структура данных);

- ключи используются для однозначной идентификации сущностей в базе данных. Основным ключом (*Primary Key*) является уникальный идентификатор для каждой сущности, который обеспечивает уникальность записей. Дополнительные ключи (*Secondary Keys*) могут быть использованы для быстрого доступа к данным или для установления связей между сущностями.

ER-диаграммы помогают разработчикам и аналитикам лучше понять структуру данных и взаимосвязи между ними в базе данных. Они служат основой для проектирования схемы базы данных и документации для разработки систем, включая создание таблиц, определение связей и атрибутов.

IEC 61131-3 и IEC 61499 [3] - это два международных стандарта, связанных с программным обеспечением для промышленной автоматизации. Они определяют функционально-блочные диаграммы, которые используются для моделирования и программирования систем автоматизации и управления процессами.

IEC 61131-3 - это стандарт, который определяет языки программирования для контроллеров промышленной автоматизации. Он включает несколько языков, одним из которых является функционально-блочная диаграмма, известная как *FBD*.

FBD в *IEC 61131-3* используют блоки и провода для представления логической структуры и потока данных в системе. Блоки представляют функциональные элементы или операции, такие как логические операторы, арифметические вычисления или считывание/запись данных. Провода обозначают потоки данных между блоками. *FBD* позволяет разработчикам моделировать и программировать логику управления и обработки данных в системе автоматизации.

IEC 61499 - это стандарт, который расширяет и дополняет *IEC 61131-3*, предоставляя модель программирования для распределенных систем автоматизации и управления процессами. В *IEC 61499* используется функционально-блочная диаграмма, известная как *FB*-диаграмма.

FB-диаграммы в *IEC 61499* также используют блоки и провода для моделирования функциональности и потока данных в распределенной системе. Однако в отличие от *IEC 61131-3*, *FB*-диаграммы в *IEC 61499* поддерживают концепцию "функционального блока", который представляет собой автономную единицу, имеющую входы, выходы и внутреннее состояние. Функциональные блоки могут взаимодействовать между собой через события и данные, обеспечивая более гибкую и модульную модель программирования для распределенных систем.

Итак, как *IEC 61131-3*, так и *IEC 61499* предоставляют функционально-блочные диаграммы для моделирования и программирования систем автоматизации.

IEC 61131-3 ориентирован на программирование контроллеров, а *IEC 61499* предоставляет модель программирования для распределенных систем. Оба стандарта используют блоки и провода для моделирования логики и потока данных в системе.

Диаграммы потоков данных [4] - это графический язык моделирования, который используется для визуализации потоков данных и их обработки в информационных системах. Они представляют собой сеть узлов, называемых блоками, которые обрабатывают данные, и стрелки, которые представляют потоки данных между блоками. В диаграммах потоков данных используются следующие элементы:

- блоки представляют операции, функции или процессы, которые обрабатывают данные. Они могут быть представлены в виде прямоугольников или окружностей с названием операции внутри;

- потоки данных представляют передачу информации или данных между блоками. Они обычно изображаются в виде стрелок, указывающих направление потока данных;

- хранилища данных представляют места, где данные могут быть сохранены или извлечены. Они обычно изображаются в виде прямоугольников с названием хранилища данных;

- внешние сущности представляют источники или назначения данных, которые находятся вне моделируемой системы. Они могут быть изображены в виде прямоугольников с названием внешней сущности;

Диаграммы потоков данных помогают визуализировать потоки данных и процессы обработки данных в информационной системе. Они широко используются при анализе и проектировании систем, таких как бизнес-процессы, программные системы или системы управления данными. Диаграммы потоков данных помогают идентифицировать потоки данных, проследить их путь и выявить потенциальные узкие места или проблемы в системе. Они также облегчают коммуникацию между разработчиками и заинтересованными сторонами, так как предоставляют наглядное представление процессов и взаимодействий в системе.

1.1.2. Графические языки запросов

Графические языки запросов (*Graphical Query Languages*) - это языки программирования и нотации, которые используют графические элементы для задания запросов к базам данных или другим источникам данных. Они предоставляют визуальный способ формулирования запросов, упрощая процесс создания и взаимодействия с базами данных. Ниже представлены некоторые из наиболее распространённых графических языков запросов.

GQL (Graph Query Language) [5] - это язык запросов, разработанный для работы с графовыми базами данных. Он представляет собой декларативный язык запросов, который позволяет эффективно извлекать данные из графовой структуры.

GQL был разработан в контексте графовых баз данных, таких как *GraphQL* и *Neo4j*, и обладает схожим синтаксисом и подходом к формулировке запросов. Однако *GQL* не следует путать с *GraphQL*, так как они предназначены для разных целей. В отличие от *GraphQL*, который является языком запросов для *API*, *GQL* является языком запросов, специализированным для работы с графовыми базами данных. Основные черты *GQL*:

- графовая структура. *GQL* основан на представлении данных в виде графов. Он позволяет формулировать запросы, которые оперируют узлами и рёбрами графа, а также задавать сложные пути и связи между данными;

- декларативный подход. *GQL* использует декларативный подход к формулировке запросов, что означает, что пользователь описывает желаемую структуру данных, а не указывает точные шаги выполнения запроса. Это позволяет оптимизировать процесс выполнения запросов и упростить работу с данными;

- гибкость и эффективность. *GQL* предоставляет гибкость в выборе и фильтрации данных. Он позволяет выбирать только нужные поля и связи, а также применять условия и фильтры для получения конкретных результатов. Это повышает эффективность запросов и уменьшает объем передаваемых данных;

- расширяемость. *GQL* обладает возможностью расширения и настройки в соответствии с потребностями конкретного графового хранилища данных. Это позволяет адаптировать язык под специфические требования и особенности базы данных.

GQL имеет широкое применение в различных областях, где требуется работа с графовыми данными, включая социальные сети, рекомендательные системы, генеалогические исследования, анализ связей и другие области, где данные естественным образом представляются в виде графов. Он упрощает извлечение и манипуляцию данными, предоставляя удобный и эффективный инструмент для работы с графовыми структурами.

PGQL (Property Graph Query Language) [6] - это язык запросов, специально разработанный для работы с графовыми базами данных, основанными на модели собственности (*property graph model*). Он предоставляет мощные средства для формулировки и выполнения сложных запросов к графовым структурам, состоящим из узлов, рёбер и свойств.

PGQL является декларативным языком запросов, что означает, что пользователь описывает желаемую структуру данных и условия для получения

результатов, а не указывает точные шаги выполнения запроса. Это позволяет оптимизировать процесс выполнения запросов и упростить работу с данными. Основные особенности *PGQL*:

- графовая структура. *PGQL* оперирует узлами, рёбрами и свойствами графовой структуры. Он позволяет формулировать запросы, которые учитывают связи между узлами и свойства элементов графа;

- выразительность. *PGQL* обладает высокой выразительностью, что позволяет формулировать сложные запросы с использованием различных операторов и функций. Он поддерживает фильтрацию, сортировку, агрегацию и другие операции над данными графа;

- семантическое расширение. *PGQL* предоставляет возможность определения пользовательских функций и алгоритмов, что позволяет расширить его функциональность в соответствии с конкретными потребностями и особенностями приложений;

- поддержка транзакций. *PGQL* обладает возможностью выполнения запросов в рамках транзакций, что обеспечивает целостность данных и согласованность изменений в графе.

PGQL является мощным инструментом для работы с графовыми базами данных и широко применяется в различных областях, включая социальные сети, анализ связей, рекомендательные системы, биоинформатику и другие области, где данные естественным образом представляются в виде графовых структур. Он упрощает формулировку и выполнение запросов к графам, обеспечивая эффективное извлечение и анализ данных.

G-CORE (Graph Constraint Language - Core) [7] - это язык запросов, разработанный для работы с графовыми базами данных, основанными на модели собственности (*property graph model*). Он предоставляет возможность формулировки сложных запросов и ограничений на графовые структуры, а также проверку их выполнимости.

G-CORE является декларативным языком запросов, что означает, что пользователь описывает желаемую структуру данных и условия для получения результатов, а не указывает точные шаги выполнения запроса. Это позволяет

упростить процесс формулировки запросов и повысить удобство работы с графовыми данными.Основные особенности *G-CORE*:

- графовая структура. *G-CORE* оперирует узлами, рёбрами и свойствами графовой структуры. Он позволяет формулировать запросы, которые учитывают связи между узлами и свойства элементов графа;

- ограничения на граф. *G-CORE* позволяет формулировать ограничения на графовые структуры, определяя условия, которым должен удовлетворять граф для того, чтобы соответствовать запросу. Это позволяет проверять структурные и семантические свойства графа;

- расширяемость. *G-CORE* обладает возможностью расширения и настройки в соответствии с потребностями конкретного графового хранилища данных. Это позволяет адаптировать язык под специфические требования и особенности базы данных;

- эффективность запросов. *G-CORE* оптимизирует выполнение запросов, позволяя использовать индексы и другие методы оптимизации для ускорения поиска и обработки данных в графе.

G-CORE является мощным инструментом для работы с графовыми базами данных и может применяться в различных областях, включая анализ связей, социальные сети, биоинформатику и другие области, где данные представляются в виде графовых структур. Он обеспечивает гибкость и эффективность запросов, позволяя анализировать и извлекать информацию из графовых данных.

В нашем проекте принято решение разработать собственный графический язык запросов вместо использования существующих *GQL*, *PGQL* и *G-CORE* по следующим причинам:

- специфические требования проекта. Существующие графические языки запросов могут не полностью соответствовать требованиям и специфике проекта. Разработка собственного языка позволяет точно определить и реализовать необходимые функциональности, операторы и возможности для удовлетворения требований проекта;

- гибкость и расширяемость. Создание собственного языка запросов предоставляет возможность гибко настроить его под специфические потребности проекта. Разработчик может включить только необходимые функции и операторы, а также расширить его функциональность по мере развития проекта, без ограничений, которые могут быть присущи существующим языкам;

- уникальные возможности: Разработка собственного графического языка запросов может предоставить уникальные возможности и инструменты, которых нет в существующих языках. Это может быть особенно полезно, если проект требует специфических аналитических, визуализационных или манипуляционных возможностей с данными;

- обучение и простота использования: Создание собственного графического языка запросов позволяет разработать простой и интуитивно понятный интерфейс, что облегчает использование языка для пользователей, которые не имеют опыта работы с существующими языками запросов.

Выводы из данного проекта указывают на то, что разработка собственного графического языка запросов была выбрана для удовлетворения специфических требований проекта, обеспечения гибкости, расширяемости, предоставления уникальных возможностей и простоты использования. Это позволяет достичь наилучшей адаптации языка к потребностям проекта и удовлетворить требования конечных пользователей.

1.1.3. Графические редакторы

Для создания и редактирования функциональных блоков в различных областях, таких как программирование, автоматизация или системы управления, могут использоваться следующие графические редакторы:

- *LabVIEW* [8] является мощным графическим программным обеспечением, разработанным для инженеров и учёных. Он использует графическую нотацию, называемую *G (Graphical)* языком, для создания функциональных блоков, известных как виртуальные приборы или виртуальные инструменты *(Virtual Instruments). LabVIEW* позволяет создавать сложные системы управления, измерений и обработки данных, используя

блоки и провода для связи между ними. На рисунке 1 представлен внешний вид программы.

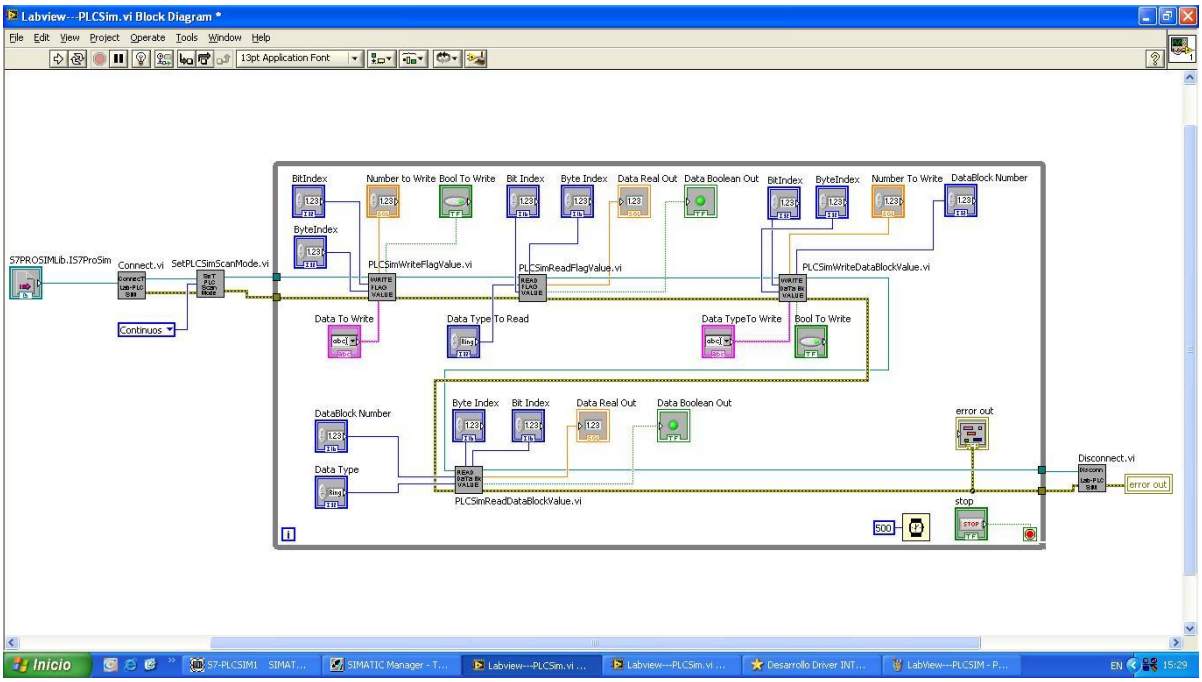


Рисунок 1 - Внешний вид графического редактора *LabVIEW*

Simulink [9] - это графическая среда моделирования и симуляции, разработанная компанией *MathWorks*. Он предоставляет блочную нотацию для создания и моделирования динамических систем. В *Simulink* функциональные блоки называются блоками и представляют различные элементы системы, такие как математические операции, фильтры, логические операторы и т. д. Блоки могут быть связаны проводами для передачи данных и управления. Внешний виг представлен на рисунке 2.

GRAFCET (GRAphe Fonctionnel de Commande Etape/Transition) - это графический язык моделирования для автоматизации и управления системами. Он используется для создания функционально-блочных диаграмм, представляющих последовательность шагов и переходов в системе управления.

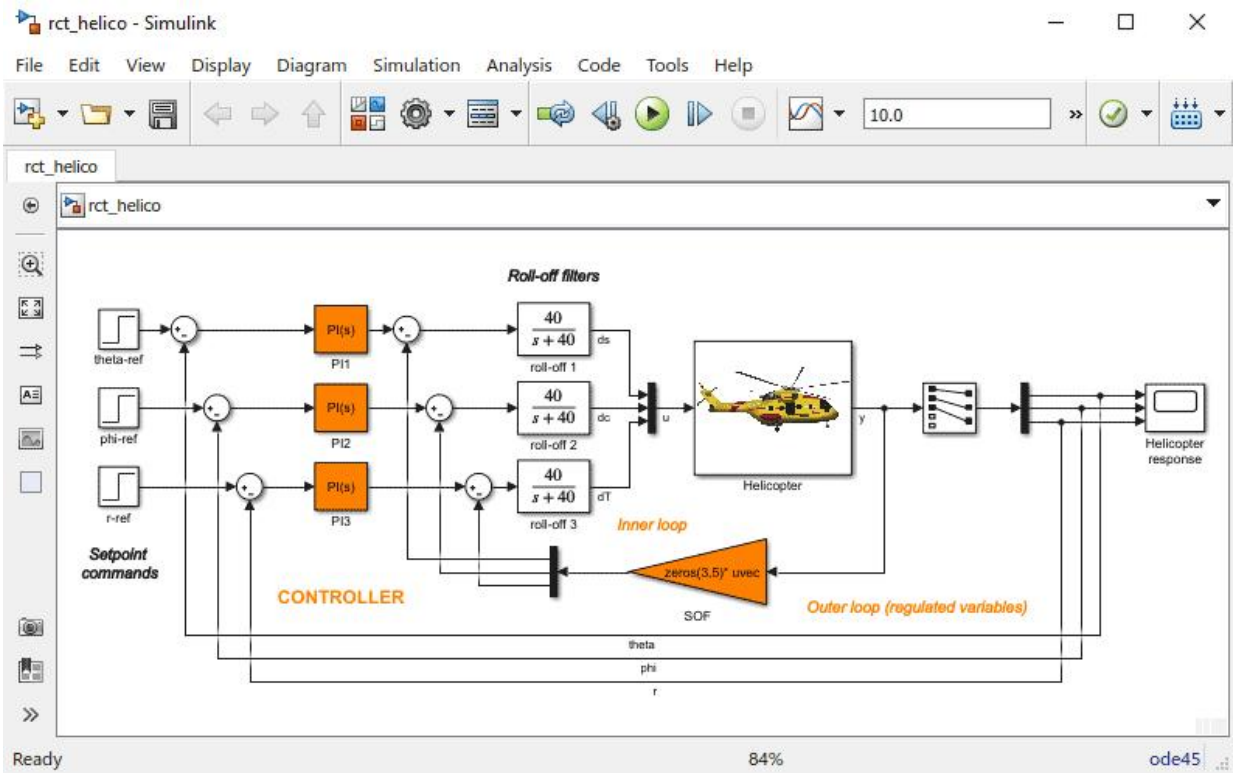


Рисунок 2 - Внешний вид графического редактора *Simulink*

GRAFCET [10] обычно используется в промышленных системах и процессах, где важно управление последовательностью операций и переходами между ними. Внешний вид представлен на рисунке 3.

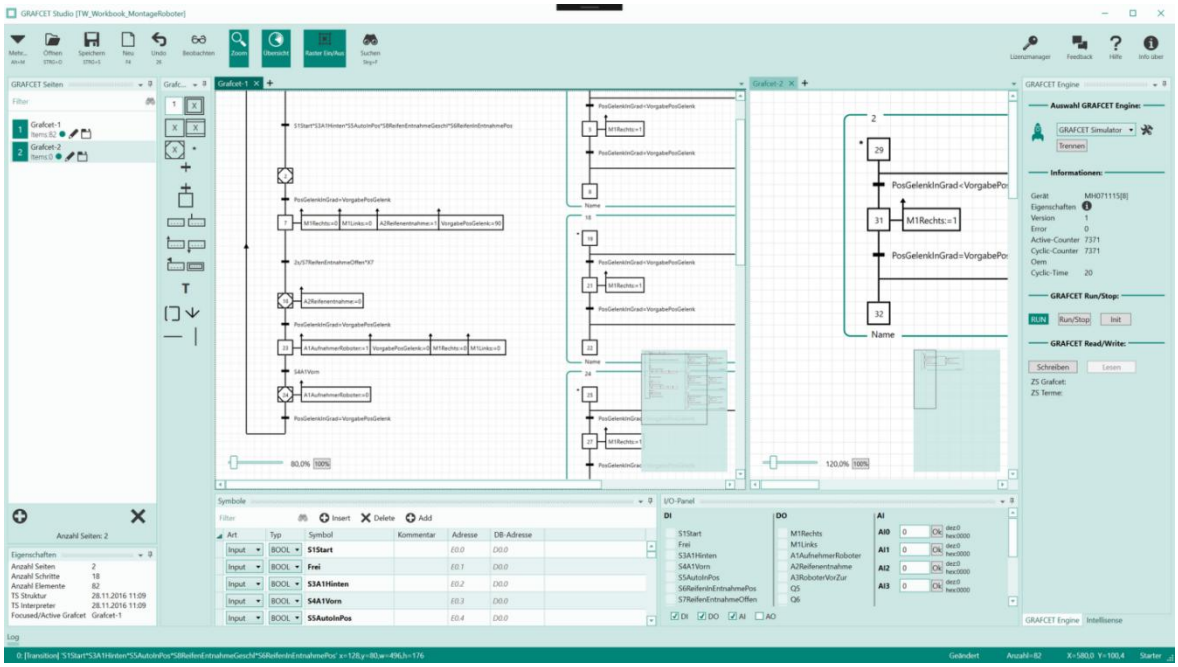


Рисунок 3 - Внешний вид графического редактора *GRAFCET*

Это лишь несколько примеров графических редакторов, которые используются для создания и редактирования функциональных блоков в различных областях. В каждой области могут использоваться специфические инструменты и редакторы, а выбор конкретного редактора зависит от требований и предпочтений пользователя.

1.1.4. Постановка задачи

Разработать графический язык для описания и обработки *SPARQL*-запросов на основе функционально-блочных диаграмм [11]. Язык должен допускать иерархическую структуризацию на основе концепции композитного функционального блока. Атомарные вычислительные функции должны быть реализованы в базисных функциональных блоках на языке *JavaScript*. Для выполнения запроса к *RDF*-хранилищам данных используется *SPARQL*-блок, причём сам *SPARQL*-запрос формируется на основе графического представления подмножества языка *SPARQL* [12][13].

Разработать и программно реализовать графический редактор для формирования функционально-блочных диаграмм, включая специальный редактор для формирования графического представления SPARQL-запросов [14].

Язык программирования – *C/C++*

Среда программирования - *QtCreator*

1.2. Язык функционально-блочных диаграмм для описания и обработки SPARQL-запросов

Язык допускает иерархическую структуризацию на основе концепции композитного функционального блока. Атомарные вычислительные функции реализованы в базисных функциональных блоках на языке *JavaScript*. Для выполнения запроса к *RDF*-хранилищам данных используется *SPARQL*-блок, при чем сам *SPARQL*-запрос формируется на основе графического представления подмножества языка *SPARQL*.

1.2.1. Базисные блоки

Описание базисных блоков

Базисные блоки используются для выполнения самых простых, атомарных операций, например умножение или сложение. Данные блоки представляют собой квадрат жёлтого цвета, сверху в левом углу которого указано название, а в правом иконка. На рисунке 4 приведён пример внешнего вида данного блока.

Рисунок 4 - Внешний вид базисных блоков

На рисунке 5 показан внешний вид блоков с полем для ввода пользовательских данных. Данное поле является не обязательным и его присутствие или отсутствие можно настроить в редакторе для базисных блоков. Также стоит обратить внимание, что над полем ввода располагается текстовая надпись, которая даёт пользователю информацию о запрашиваемых данных.

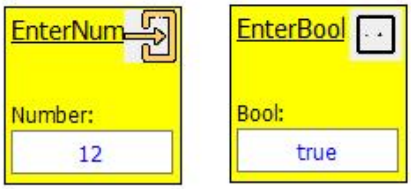


Рисунок 5 - Внешний вид базисного блока с полем для ввода пользовательских данных

Особенности базисных блоков:

- базисные блоки могут иметь сколько угодно много входов и выходов. При этом все входы являются равнозначными и предсказать в каком порядке они будут обрабатываться не представляется возможным. Базисный блок на все выходы выдаёт один и тот же результат работы;

- в отличие от *SPARQL* и композитного блока, в базисном блоке пользователь может сам задать иконку;

- для работы блока используется *JavaScript* код в котором описывается логика обработки входных данных и получение результата.

Принцип использования в системе

Данный блок при создании проекта применяется для атомарных действий, например умножить, сложить или логических функций по типу «И», «ИЛИ».

На рисунке 6 продемонстрировано применение базисных блоков, в данном случае программа должна получить сумму двух чисел. Все блоки взяты из стандартных блоков в нашей программе, так что их поведение мы можем предсказать. Базисный блок «EnterNumber» получает данные от пользователя, в нашем случае числа для сложения. Чтобы передать полученное значение, производится соединение блоков с помощью стрелки, направление которой показывает от какого блока к какому будут передавать данные, в нашем случае от блока «EnterNumber» к блоку «Sum». Блок «Sum» принимает на вход числа «3» и «6» и на выходе получится 9. Блок «OutputBlock» выдаёт информацию пользователю на экран. На рисунке 7 как раз продемонстрирован результат работы блоков в нашей программе.

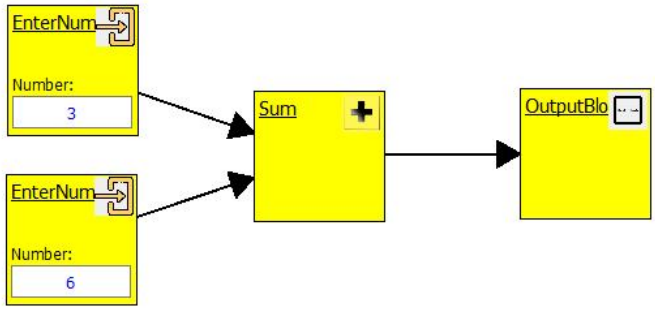


Рисунок 6 - Пример использования базисного блока в проекте

Как формируется базисный блок

На рисунке 8 можно рассмотреть какие параметры используются для настройки базисного блока. Поле «*Name Block*» для настройки имени блока, «*Text label*» для надписи над полем ввода, «*Text lineEdit*» для значения по умолчанию в базисном блоке, «*Script*» для задания пользовательского скрипта.

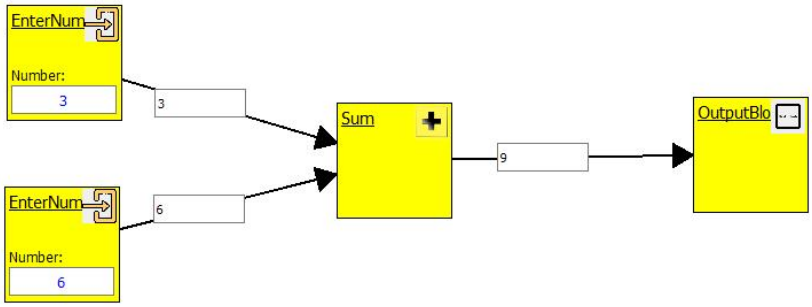


Рисунок 7 - Результат работы схемы, значения на стрелках показывают данные, которые через них прошли

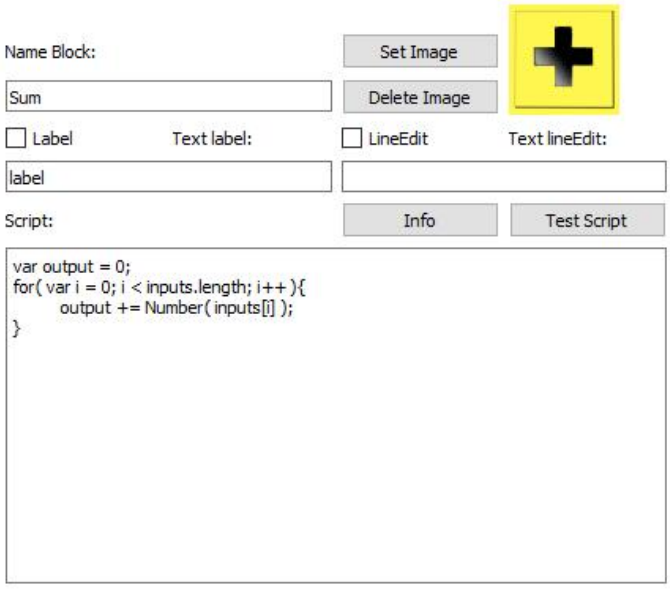


Рисунок 8 - Окно редактора базисного блока

Флажки «*Label*» и «*LineEdit*» нужны для отображения на блоке текста и поля ввода соответственно.

Логика работы блока описывается на языке *JavaScript* самим пользователем при создании. Блок работает по принципу обработки поступающего массива данных и выдачи результата. Входные данные в программе подаются в виде массива «*inputs*», выходные же данные представлены в виде переменной «*output*» в которую помещается результат. Помимо «*inputs*» и «*output*» присутствует массив «*causes*» в который можно записать объяснение результата, путём добавления порядкового номера входного значения из массива «*inputs*». Пользователь может создать дополнительные внутренние переменные для реализации обработки.

Поскольку мы используем *JavaScript* (язык с динамической типизацией) для кодирования логики, переменные не имеют ограничений по типу. Пример скрипта для блока сложения можно посмотреть на рисунке 8.

1.2.2. Композитные блоки

Описание композитных блоков

Композитные блоки нужны для создания более сложных комбинаций блоков. Композитные блоки это своего рода функции из стандартных языков программирования, только у нас в языке функционально-блочных диаграмм. При создании композитных блоков могут быть использованы: базисные, композитные блоки, а также блоки *SPARQL*-запросов.

На рисунке 9 показан внешний вид блока, он представляет собой прямоугольник зелёного цвета, поделённого на 2 части: заголовок сверху и тело снизу. Заголовок состоит из названия в левом углу и стандартная для композитного блока иконка в правом. В теле располагаются входы слева и выходы справа. Задаются они с помощью названия находящегося на самом блоке и соответствующий ему выступ для соединения стрелок.

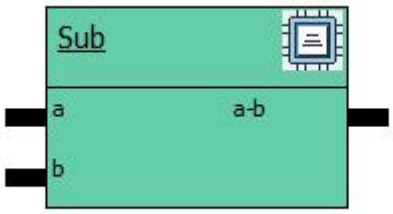


Рисунок 9 - Внешний вид композитного блока

Особенности композитных блоков:

- при исполнении программы композитный блок заменяется на находящиеся внутри себя блоки.;

- композитный блок может содержать в себе другие композитные блоки;

- в связи с особенностями организации сохранения и исполнения блоков, нельзя соединять выходы и входы с одинаковыми названиями.

Принцип использования в системе

Композитный блок в проекте призван облегчить работу со схемой, ведь в него можно записать какие-либо шаблонные схемы и использовать также, как используются функции в стандартных языках программирования.

На рисунке 10 продемонстрирован пример использования композитного блока для вычитания одного числа из другого. В схеме используется стандартных композитный блок вычитания «*Sub*». У данного блока имеется 2 входа «*a*» и «*b*» и один выход «*a-b*». К входу «*a*» мы подключаем значение из которого хотим вычесть, а к «*b*» вычитаемое. В результате работы блока будет выдан результат, в схеме на рисунке 11 это показано.

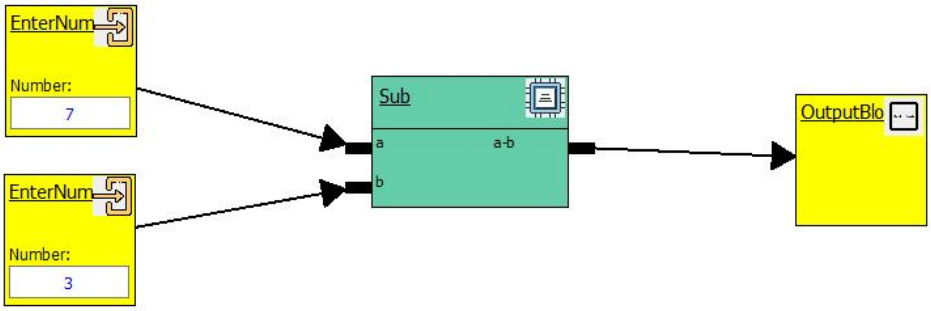


Рисунок 10 - Пример использования композитного блока в схеме

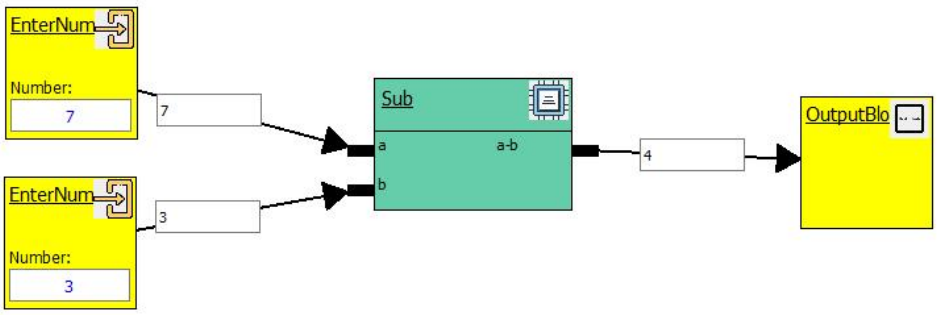


Рисунок 11 - Результат работы схемы

Как формируется композитный блок

Рисунок 12 прекрасно показывает как формируется композитный блок, на его примере и будет разбирать. В редакторе композитных блоков есть вспомогательные блоки *IO*, которые присутствуют в 2 вариациях: «*Input*» и «*Output*», для входа и выхода соответственно. Чтобы создать вход или выход, достаточно добавить данный блок на схему и написать внутри название

входа/выхода. На рисунке 12 видно, что создано 2 блока входа «*a*» и «*b*» и 1 блок выхода «*a-b*», они как раз используются для формирования входов/выходов на композитных блоков на рисунке 9. Так как «*b*» это вычитаемое, то помножим его на «-1» и сложим с «*a*», таким образом реализовано вычитание в нашей программе. Результат же отправится на выход «*a-b*».

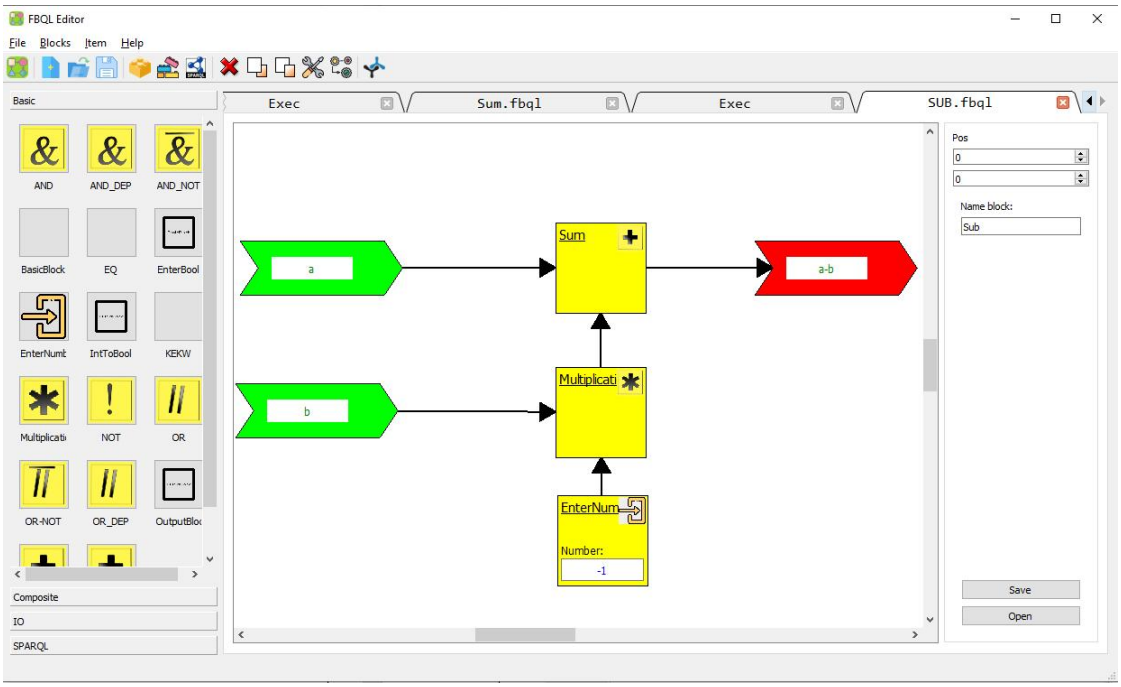


Рисунок 12 - Блок вычитания «*Sub*» открытый в редакторе

1.2.3 *SPARQL*-блоки

Описание *SPARQL*-блоков

SPARQL-блоки нужны для создания запросов к *RDF*-хранилищам данных. *SPARQL*-блоки обращается к онтологии которая загружена на *Jena Fuseki Server* и обрабатывает результат запроса, отображая количество найденных записей, соответствующих запросу.

На рисунке 13 можно увидеть внешний вид блока, он представляет собой квадрат зелёного цвета, сверху в левом углу которого указано название, а в правом иконка.

Рисунок 13 - Внешний вид SPARQL-блока

Особенности *SPARQL*-блоков:

- блок не может иметь входов, так как он и есть источник данных.

Принцип использования в системе

SPARQL-блок соединяется по тому же принципу, что и базисный блок, только не может иметь входов. В рисунках 14 и 15 можно увидеть, что в результате работы *SPARQL*-запросов от *SPARQL*-блоков выдаётся количество найденных записей, далее они суммируются и выдаётся результат.

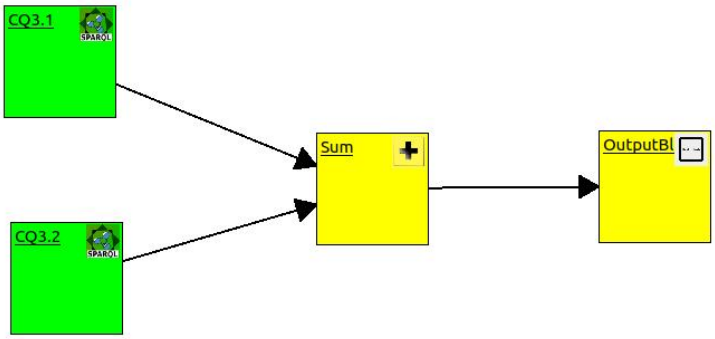


Рисунок 14 - Пример использования *SPARQL*-блока

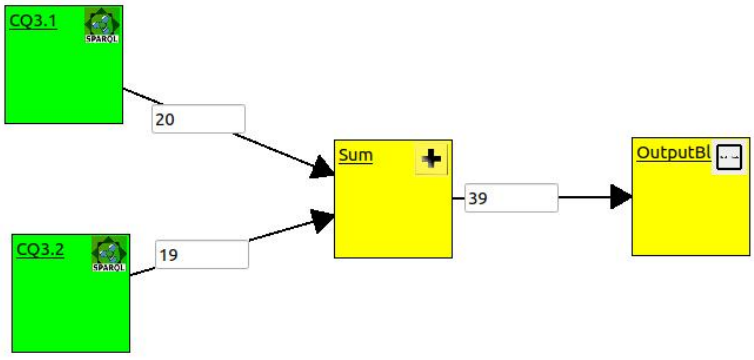


Рисунок 15 - Результат работы схемы

Как формируется *SPARQL*-блок

SPARQL-блок создаётся при помощи вспомогательных *Atom*-блоков. *Atom*-блоки включают в себя 3 типа: «*AREA*», «*VAR*», «*VALUE*». Блок «*AREA*» используется для задания областей, которые потом соединяются с помощью связей «*MINUS*», «*UNION*» и тд. Блок «*VAR*» нужен для обозначения переменных в *SPARQL*-запросе, «*VALUE*» же для константных значений. Формирование запроса происходит по принципу триплетов, стрелкой соединяются 2 блока типа «*VAR*» или «*VALUE*». При конвертации субъектом является блок из которого идёт стрелка, объектом тот блок в который она идёт, а отношением будет надпись на стрелке. Также *SPARQL*-блоки можно задать с помощью текста *SPARQL*-запроса, но тогда диаграмма *SPARQL*-блока сохраняться не будет. На рисунках 16 и 17 можно увидеть наполнение *SPARQL*-блоков.

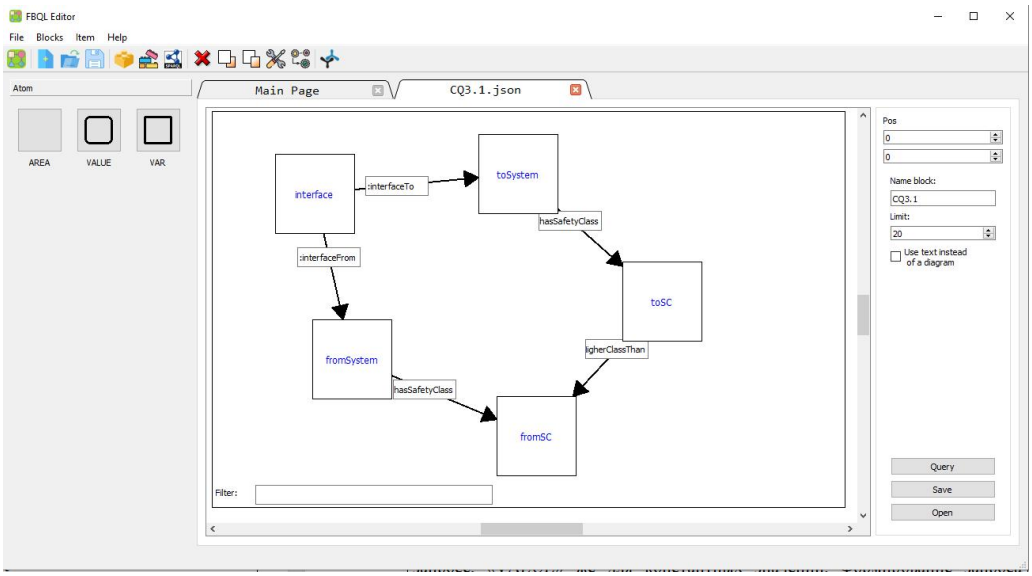


Рисунок 16 - *SPARQL*-блок «*CQ3.1*» открыт в редакторе *SPARQL*-блоков

1.2.4 Связи между блоками

При создании проекта могут быть использованы базисные, композитные и *SPARQL*-блоки. Для связи между блоками используются стрелки. Направление стрелок показывает от какого блока к какому будут передаваться данные.

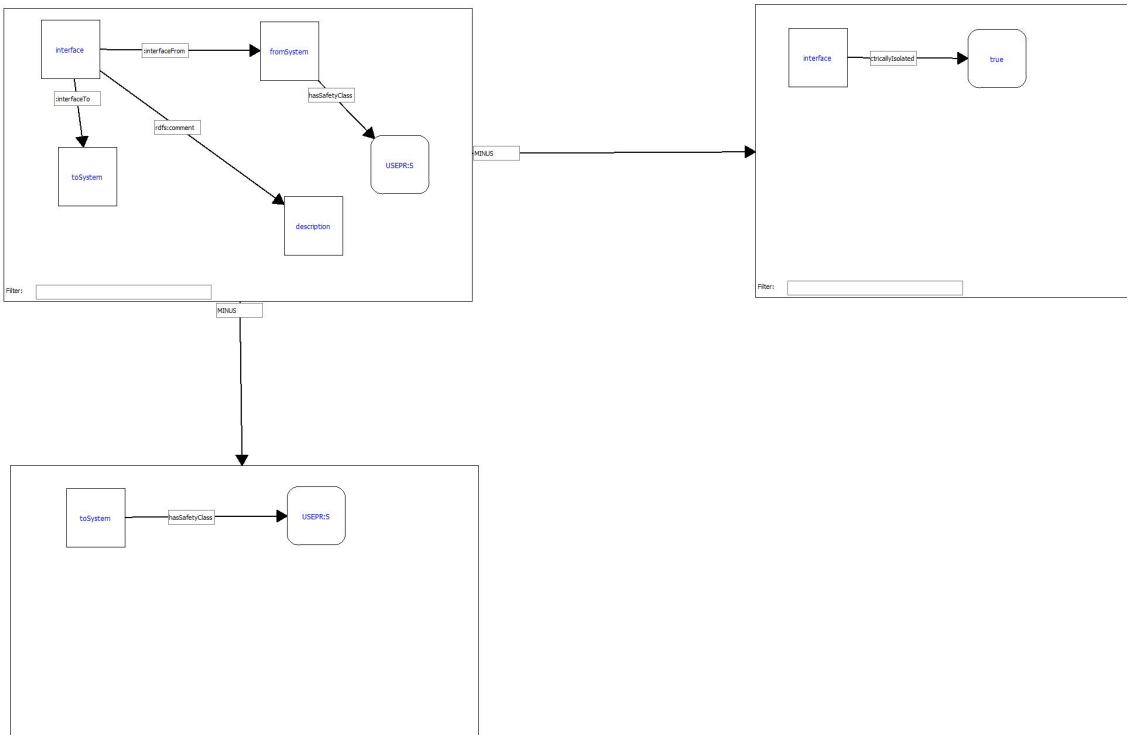


Рисунок 17 - SPARQL-блок с использованием связей «*MINUS*»

На вход базовым блокам может быть сколь угодно много стрелок, для базового блока все эти входы равнозначны и нельзя предсказать в какой последовательность будут переданы данные. От базового блока может идти сколько угодно много стрелок к другим блокам, по каждой стрелке будет передан один и тот же результат после того как блок отработает.

В композитном блоке есть несколько входов и выходов. Для каждого отдельного входа действуют те же самые правила, как и для входа базисного блока, но такой блок не может иметь выходов. Для каждого отдельного выхода действуют те же самые правила, как и для выхода базисного блока, но такой блок не может иметь входов.

SPARQL-блок это блок без входов, так как он сам генерирует данные путём обращения к *RDF*-хранилищу. Для выхода данного блока действуют те же правила, как и для выхода базисного блока.

1.2.5 Ограничения, налагаемые на структуру

В логике работы блоков не предусмотрены зацикливания, так что в диаграммах нельзя использовать зацикливания.

При создании *SPARQL*-блоков, нельзя соединять блоки находящиеся в разных областях.

Нельзя делать бесконечные циклы в базисных блоках, так как это приведёт к паданию программы.

Для *SPARQL*-блоков наложены ограничения на обращение к серверу в виде ожидания ответа 3 сек, если же будет больше то, обращение считается не прошедшим, сделано дабы не допустить зависания программы.

В программе пока не предусмотренно блоков условий по причине особенностей работы блоков, но можно получить ответ на вопрос равны ли значения или нет, к примеру в виде «*true*» или «*false*».

1.3 Разработка графического редактора для функционально-блочных диаграмм

1.3.1 Цели создания программы, её назначение и функциональные возможности

Программа предназначена для создания, редактирования и выполнения и обработки результатов обращения к хранилищам данных *RDF*. Так как мы используем для обращения к RDF собственный графический язык *FBQL*, программа для его создания, редактирования и выполнения будет называться *FBQL Editor*. Данную программу можно использовать для оценки свойств систем, представленных в виде онтологии на основе *RDF*. *FBD* может включать в себя три типа (функциональных) блоков: базовые, композитные и *SPARQL* блоки. Можно использовать как библиотечные, так и пользовательские блоки.

Наша программа предоставляет:

- редактор для базовых блоков с возможностью их создания и редактирования;

- редактор для композитных блоков с возможностью создания в них сложных схем;

- редактор для *SPARQL*-блоков, который предоставляет огромный функционал по созданию *SPARQL*-запросов как в виде диаграмм, так и в текстовом представлении;

- редактор для создания функциональной схемы, которая может исполняться и выдавать результат своей работы;

- возможность настройки работы с *Jena Fuseki Server*[16] как на локальном компьютере, так и в сети;

- блок исполнения схемы с выводом промежуточных и итоговых результатов.

1.3.2 Выбор программно-аппаратной платформы

Для своего диплома мы выбрали язык программирования *C++*, потому что это один из самых востребованных языков программирования в мире. *C++* позволяет создавать сложные и эффективные приложения, которые могут работать на разных платформах и устройствах. Кроме того, *C++* имеет множество библиотек и фреймворков, которые упрощают разработку и повышают производительность приложений. Также *C++* является языком, который используется во многих областях, таких как разработка игр, веб-разработка, научное программирование и многие другие. В целом, *C++* - это отличный выбор для тех, кто хочет создавать сложные и производительные приложения.

В качестве фреймворка для нашего приложения был выбран *Qt[15]*. *Qt* — это фреймворк для разработки кроссплатформенного программного обеспечения на языке программирования *C++*. Со времени своего появления в 1996 библиотека легла в основу многих программных проектов. Кроме того, *Qt* является фундаментом популярной рабочей среды *KDE*, входящей в состав многих дистрибутивов *Linux*. *Qt* позволяет запускать написанное с его помощью программное обеспечение в большинстве современных операционных систем путём простой компиляции программы для каждой системы без изменения исходного кода. Включает в себя все основные классы, которые могут потребоваться при разработке прикладного программного обеспечения, начиная от элементов графического интерфейса и заканчивая

классами для работы с сетью, базами данных и *XML*. Является полностью объектно-ориентированным, расширяемым и поддерживающим технику компонентного программирования.

Отличительная особенность — использование метаобъектного компилятора — предварительной системы обработки исходного кода. Расширение возможностей обеспечивается системой плагинов, которые возможно размещать непосредственно в панели визуального редактора. Также существует возможность расширения привычной функциональности виджетов, связанной с размещением их на экране, отображением, перерисовкой при изменении размеров окна.

Комплектуется визуальной средой разработки графического интерфейса *Qt Designer*, позволяющей создавать диалоги и формы в режиме *WYSIWYG*. В поставке *Qt* есть *Qt Linguist* — графическая утилита, позволяющая упростить локализацию и перевод программы на многие языки; и *Qt Assistant* — справочная система *Qt*, упрощающая работу с документацией по библиотеке, а также позволяющая создавать кроссплатформенную справку для разрабатываемого на основе *Qt* программного обеспечения. Начиная с версии 4.5.0 в комплект включена среда разработки *Qt Creator*, которая включает редактор кода, справку, графические средства *Qt Designer* и возможность отладки приложений. *Qt Creator* может использовать *GCC* или *Microsoft VC++* в качестве компилятора и *GDB* в качестве отладчика. Для *Windows*-версий библиотека комплектуется компилятором, заголовочными и объектными файлами *MinGW*.

В качестве *IDE* для разработки приложения был выбран *Qt Creator*. *Qt Creator* — свободная *IDE* для разработки на *C*, *C++*, *JavaScript* и *QML*. Разработана *Trolltech (Digia)* для работы с фреймворком *Qt*. Включает в себя графический интерфейс отладчика и визуальные средства разработки интерфейса как с использованием *QtWidgets*, так и *QML*. Поддерживаемые компиляторы: *GCC, Clang, MinGW, MSVC, Linux ICC, GCCE, RVCT, WINSCW*. Основная задача *Qt Creator* — упростить разработку приложения с помощью фреймворка *Qt* на разных платформах. Поэтому среди возможностей,

присущих любой среде разработки, есть и специфичные, такие как отладка приложений на *QML* и отображение в отладчике данных из контейнеров Qt, встроенный дизайнер интерфейсов: как на *QML*, так и на *QtWidgets*.

1.3.3 Архитектура программы

В нашей программе присутствует много различных модулей которые используются для различных задач. На рисунке 18 представлены следующие модули программы:

- основной модуль - модуль отвечающий за создание главного окна и установке в него стилей;

- модуль главного окна - модуль отрисовки главного окна и взаимодействия с пользователем. Данный модуль подгружает настройки, вызывает загрузки блоков из сохранения, создаёт внешний вид приложения (*GUI*);

- модуль библиотеки блоков занимается поиском блоков и проектов для загрузки. Также предоставляет главному окну доступ к настройкам блоков;

- модуль графического редактора отвечает за создание окон с графическими редакторами для создания и редактирования блоков и проекта. Использует Библиотеку *GraphViz* [17] для создания изображения проекта при запуске;

- модуль визуализации графических элементов - отвечает за отображение блоков на графической сцене и их поведение при взаимодействии с пользователем;

- мастер Настроек - позволяет получать доступ к глобальным настройкам программы. Данный модуль использует файл для хранения настроек, так что они не теряются при закрытии программы;

- модуль исполнения проекта - модуль отвечающий за создание схемы выполнения блоков и их последующий запуск. Модуль исполнения блоков, отвечает за непосредственное исполнение скриптов блоков и передачу данных от одного блока к другому;

- модуль работы с сервером Jena Fuseki отвечает за API взаимодействия с сервером для проведения SPARQL-запросов.

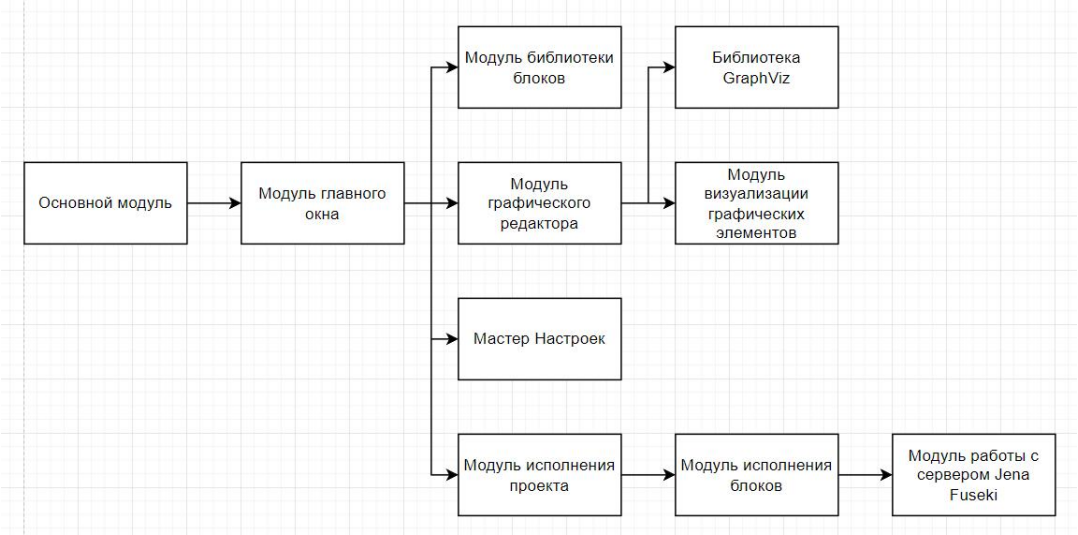


Рисунок 18 - Схема взаимодействия модулей программы

На рисунке 19 можно увидеть диаграмму вариантов использования для нашей программы. На ней представлено взаимодействие основных блоков нашей программы в упрощённом виде, более понятном для пользователя.

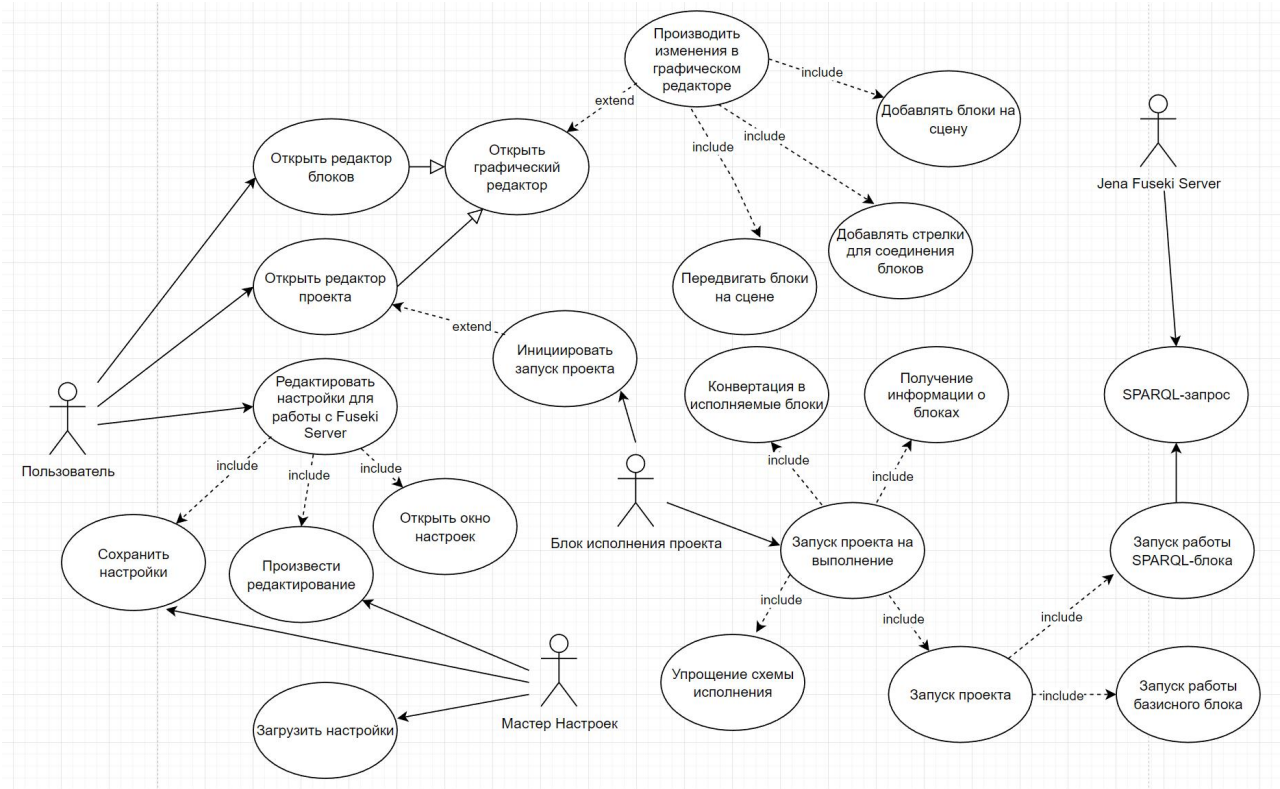


Рисунок 19 - Диаграмма вариантов использования для нашей программы

1.3.4 Сохранение данных, блоков и проектов

В программе присутствует возможность сохранять и открывать различные данные, чтобы их не потерять в случаи закрытия программы.

Сохранение блоков и проектов

При сохранении блоков и проектов используется формат *JSON*. *JSON* — текстовый формат обмена данными, основанный на *JavaScript*. Как и многие другие текстовые форматы, *JSON* легко читается людьми. Формат *JSON* был разработан Дугласом Крокфордом.

Несмотря на происхождение от *JavaScript* (точнее, от подмножества языка стандарта *ECMA-262* 1999 года), формат считается независимым от языка и может использоваться практически с любым языком программирования. Для многих языков существует готовый код для создания и обработки данных в формате *JSON*.

За счёт своей лаконичности по сравнению с *XML* формат *JSON* может быть более подходящим для сериализации сложных структур. Применяется в веб-приложениях как для обмена данными между браузером и сервером (*AJAX*), так и между серверами (программные *HTTP*-сопряжения).

Структуры хранящие настройки для блоков и проекта сохраняются в файл при помощи записи важных полей в формат *JSON*. Формат сохранения файлов блоков «*fbql*», для проектов используется формат «*fbd*». Пример файлов сохранения представлены на рисунках 20 и 21.

Сохранение настроек программы

В программе присутствует специальный модуль отвечающий за сохранение и загрузку настроек программы. Сохранение происходит при помощи сериализации. Формат сохранения файла настроек «*fbs*».

1.3.5 Основные структуры данных

В программе используется огромное количество классов и структур. Мы же попробуем рассмотреть основные классы и их значение в программе.

```
{
    "Body": {
        "Flag_Text": true,
        "Flag_Edit": false,
        "Text_Edit": "edit",
        "Text_Label": "label",
        "Polygon": [
        ],
        "Pos": {
            "X": 38,
            "Y": -112
        },
        "Script":"...",
        "Flag_Custom_Image": true,
        "Image": "..."
    },
    "Header": {
        "Name": "BasicBlock",
        "Type": "Basic"
    }
}
```


Рисунок 20 - Пример файла сохранения базисного блока


```
{
    "Body": {
        "Areas": [
            {
                "Blocks": [
                    {
                        "Header": {
                            "Name": "AtomBlock",
                            "Type": "Atom",
                            "Type_Block": "VALUE"
                        }
                    },
                    {
                        "Header": {
                            "Name": "AtomBlock",
                            "Type": "Atom",
                            "Type_Block": "VAR"
                        }
                    }
                ],
                "Lines": [
                    {
                        "End": 0,
                        "Start": 1,
                        "Text": ""
                    }
                ]
            }
        ],
        "Limit": 20,
        "Lines": [
        ],
        "Start_Area": 0
    },
    "Header": {
        "Name": "SparqlBlock",
        "Type": "Sparql"
    }
}
```


Рисунок 21 - Пример файла сохранения SPARQL-блока

Основные классы для формирования главного окна

Самый главный класс в программе это *MainWindow*, этот класс отвечает за наполнение главного окна и взаимодействие с пользователем. Класс MainWindow использует несколько других, как показано на рисунке 22:

- *SMenuBar* - класс для формирования меню;

- *SToolBar* - класс для формирования панели инструментов;

- *SToolBox* - класс боковой панели блоков;

- *STabWidget* - класс отвечающий за виджет отображения окон настроек и редакторов наследованных от *SWidget*.

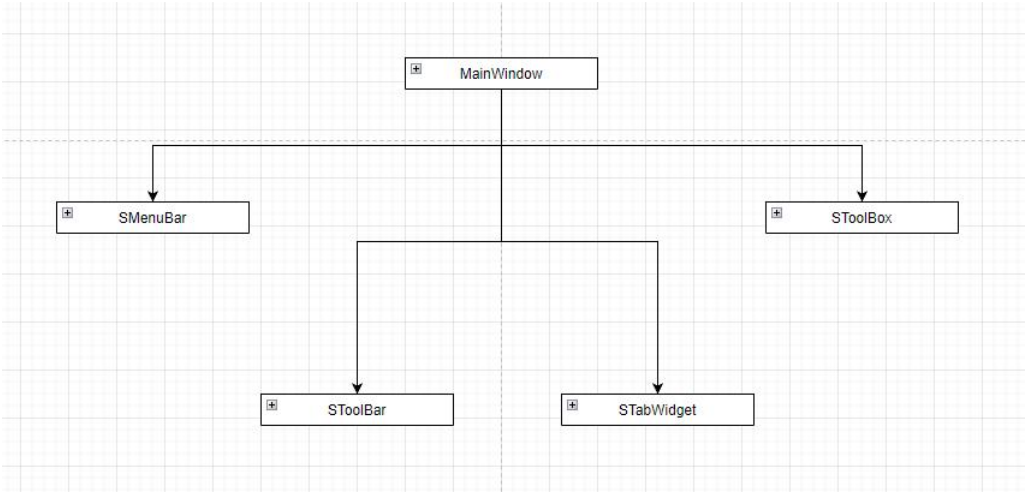


Рисунок 22 - Диаграмма классов для главного окна

Классы отвечающие за графические редакторы

Для создания окон которые могут отображаться в *STabWidget* существует класс *SWidget*. От класса *SWidget* наследуются следующие окнаб как показано на рисунке 23:

- *BasicBlockWindow* - класс отвечающий за окно редактора базисных блоков;

- *FusekiServerSettings* - класс отвечающий за окно настроек работы с Jena Fuseki Server;

- *MainPage* - класс отвечающий за окно отображения приветственной html-страницы;

- *DiagramExecutor* - класс отвечающий за окно исполнения проектов;

- *SGraphicsView* - абстрактный класс для редакторов диаграмм.

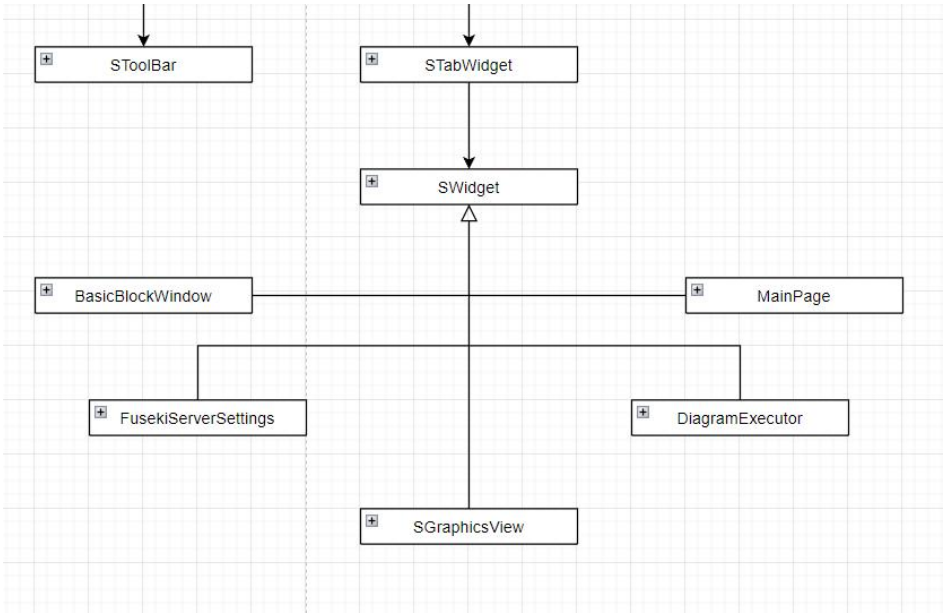


Рисунок 23 - Диаграмма классов для SWidget

SGraphicsView это основной класс для графических редакторов. Класс использует класс *DiagramScene* для рисования. *DiagramScene* - отвечает за графические элементы на сцене. Графические редакторы используют наследников от *DiagramItem* для отображения блоков и наследников *DiagramItemSettings* для сохранения настроек блоков. От *SGraphicsView* наследуются следующие редакторы (рисунок 24):

- *CompositeBlockWindow* - класс графического редактора композитных блоков;

- *SparqlBlockWindow* - класс графического редактора *SPARQL*-блоков;

- *ProjectWindow* - класс графического редактора проектов.

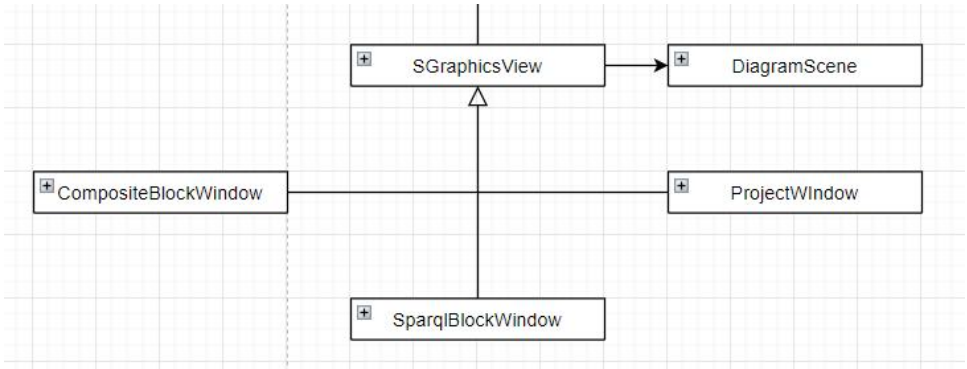


Рисунок 24 - Диаграмма классов для *SGraphicsView*

Классы отвечающие за отображение блоков

За внешний вид и поведение блоков на сцене в нашей программе отвечает класс «*DiagramItem*», который наследуется от стандартного класса библиотеки *Qt* «*QGraphicsPolygonItem*». От класса «*DiagramItem*» наследуются следующие классы (рисунок 25):

- *DiagramItemBasic* - класс отвечающий за поведение базисного блока на сцене;

- *DiagramItemComposite* - класс отвечающий за поведение композитного блока на сцене. Для отрисовки входов и выходов используется класс *DiagramItemIO;*

- *DiagramItemSparql* - класс отвечающий за поведение *SPARQL*-блока;

- *DiagramItemAtom* - класс отвечающий за поведение *Atom*-блоков, использующихся для создания *SPARQL*-запроса.

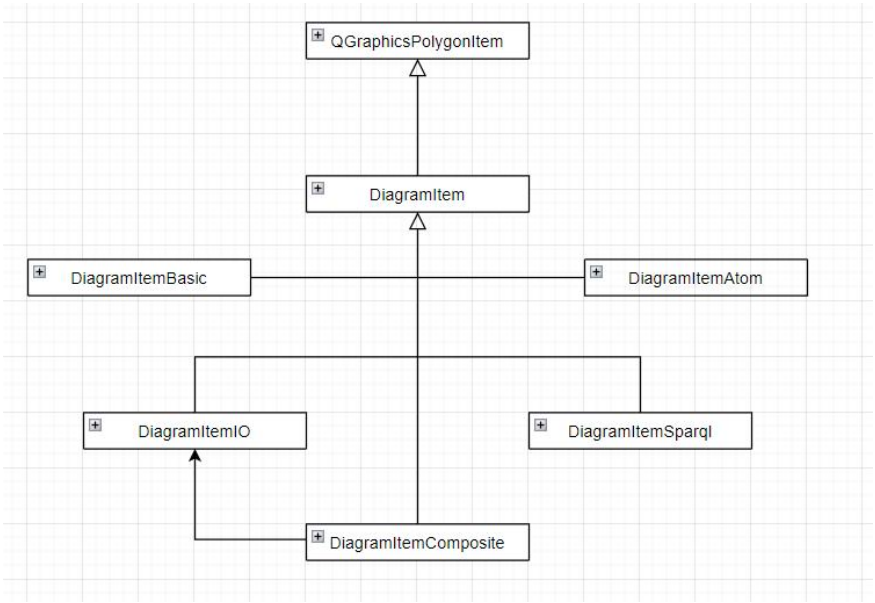


Рисунок 25 - Диаграмма классов для отображения блоков

Классы отвечающие за хранение настроек блоков и проекта

Хранение настроек для блоков очень важная задача, поэтому получилась очень сложная схема, где множество настроек переплетено между собой. Абстрактным классом для всех выступает «*DiagramItemSettings*», в котором описаны функции сохранения настроек в файлы и множество других полезных возможностей.

От «*DiagramItemSettings*» наследуются следующие классы (рисунок 26):

- *BasicBlockSettings* - класс отвечает за сохранение параметров для исполнения и отрисовки базисного блока;

- *AtomBlockSettings* - класс отвечающий за параметры для *Atom*-блока;

- *SparqlBlockSettings* - класс хранящий параметры для *SPARQL*-блока, для хранения диаграммы запроса использует класс «*AtomBlockSettings*»;

- *IOBlockSettings* - класс для хранения параметров *IO*-блоков;

- *CompositeBlockSettings* - класс параметров композитного блока, так как внутри могут находиться базисные, *SPARQL*, композитные и *IO*-блоки, то он хранит их настройки, для возможности последующей отрисовки схемы пользователю;

- *ProjectWindowSettings* - класс отвечающий за параметры проекта, включает в себя все другие классы по той же причине, что и «*CompositeBlockSettings*».

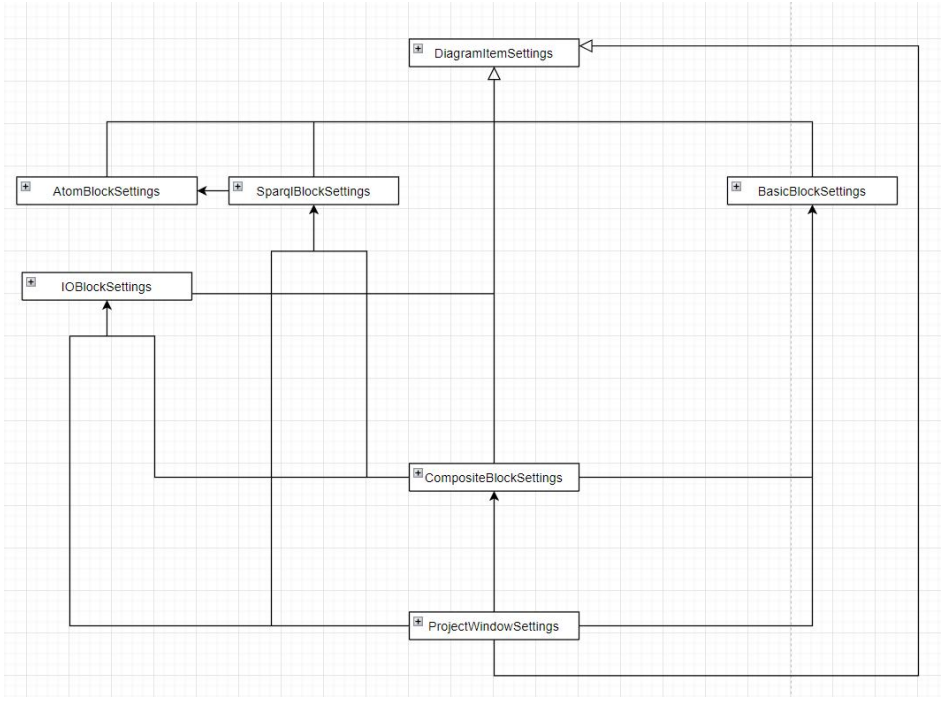


Рисунок 26 - Диаграмма классов настроек для блоков и проекта

1.3.6 Алгоритмы работы графического редактора

Формирование главного окна

На рисунке 27 можно увидеть алгоритм работы программы при её открытии. При старте программы начинается создание главного окна, за которое отвечает класс «*MainWindow*». Сначала происходит инициализация и

загрузка библиотеки блоков, которая отвечает за набор блоков отображаемых в боковой панели блоков. Далее начинается создание пользовательского меню, в которое входит и инициализация событий, таких как: нажатия на кнопки, выбор пунктов меню и использование горячих клавиш. После того как все события были инициализированы и пользовательское меню было создано, начинается создание панели инструментов, используя события из меню. Далее создаётся и устанавливается на форму главный виджет для отображения вкладок, а также создаётся боковая панель блоков. И наконец после вывода приветственной *html*-страницы, происходит отображение программы пользователю.

Рисунок 27 - Блок-схема алгоритма формирования главного окна

На рисунке 28 показано как реагирует программа на действия пользователя. Возможные действия пользователя описаны в пункте 1.3.3.

Можно заметить, что «*Basic*», «*Composite*», «*SPARQL*», «*New project*» и «*MainPage*». Пример реализации добавления новых окон в главный виджет вкладок:

```
void MainWindow::slotOnOpenMainPage()
{
    auto window = new MainPage( this );
    tab_widget->addWidget( window, tr( "Main Page" ) );
}

void MainWindow::slotCreateNewProject()
{
    auto window = new ProjectWindow( item_menu, this );
    tab_widget->addWidget( window, tr( "New Project" ) );
}
```

Реализация открытия окна *DiagramExecutor* для выполнения проекта:

```
void MainWindow::slotOnClickedBlockExecutor()
{
    auto list_blocks = tab_widget->getBlocksForRun();
    if ( !list_blocks.empty() )
    {
        auto window = new DiagramExecutor( this );
        window->setDiagramItem( list_blocks );
        tab_widget->addWidget( window, "Exec" );
    }
}
```

В случае открытия нового проекта или редактора для композитных и *SPARQL*-блоков открывается графический редактор. Далее мы рассмотрим как работает графические редактор для создания и редактирования композитных и *SPARQL*-блоков, а также проекта.

На рисунке 29 продемонстрирован алгоритм создания графического редактора. Сначала мы создаём «*DiagramScene*», которая отвечает за отображение сцены и отрисовку на ней объектов «*DiagramItem*». Далее эта сцена передаётся на виджет *SGraphicsView*, который позволяет пользователю просматривать графические сцены. После создания графического редактора он добавляется в главный виджет, который создаёт нужную вкладку с выбранным графическим редактором.

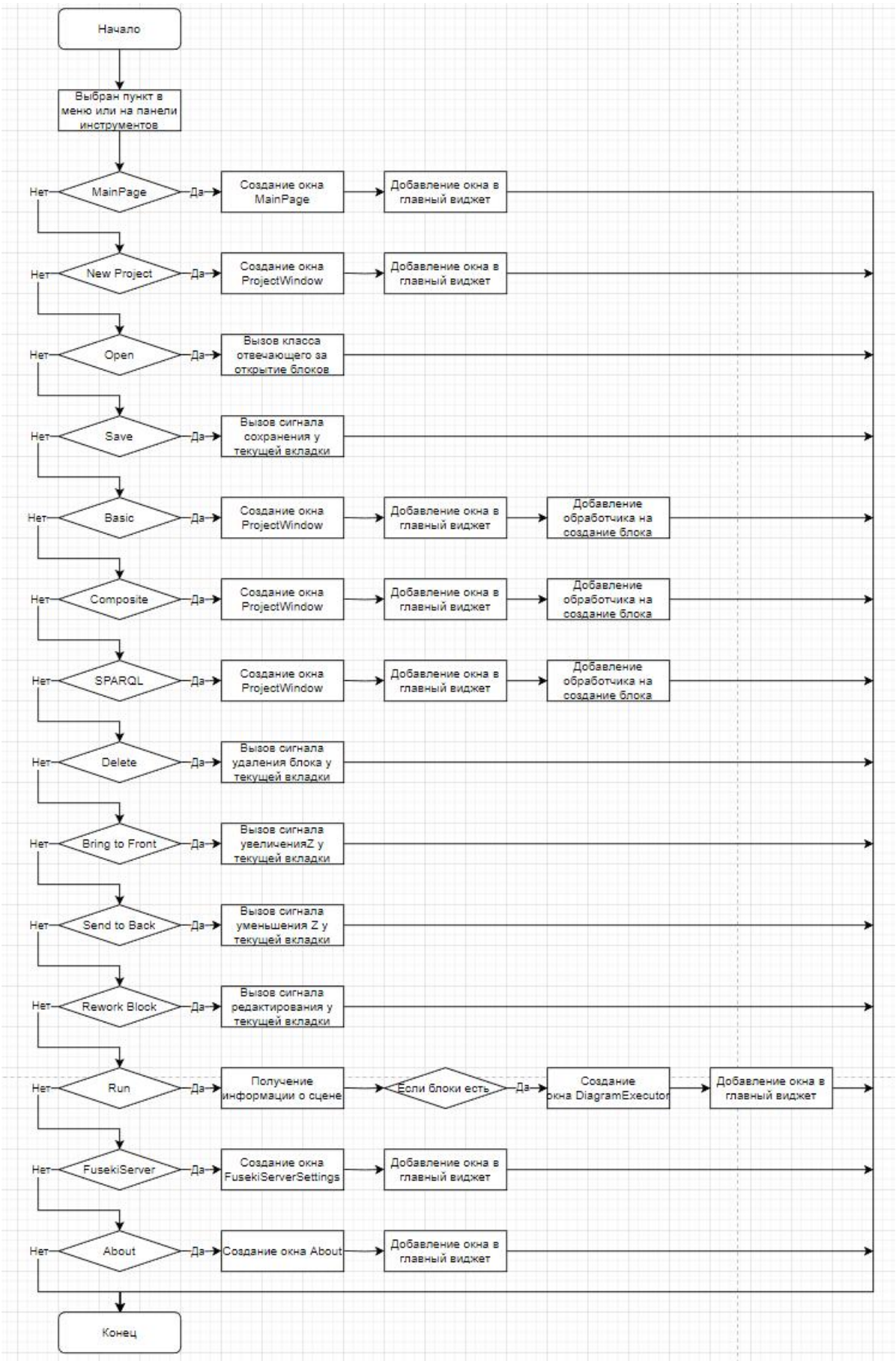


Рисунок 28 - Блок-схема реакции программы на действия пользователя

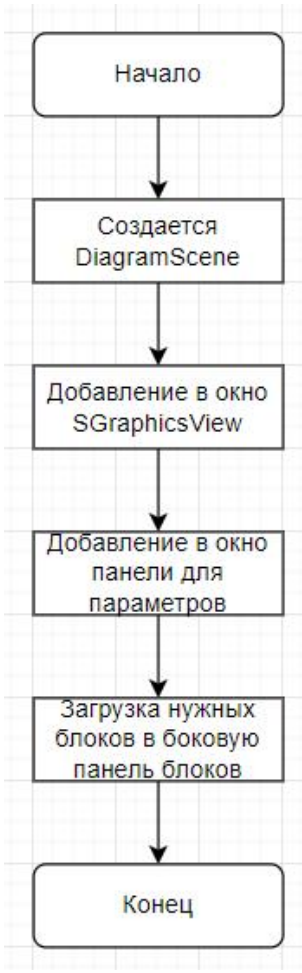


Рисунок 29 - Блок-схема алгоритма создания окна графического редактора

На рисунке 30 блок-схема алгоритма обработки событий в графическом редакторе на различные события, вызванные пользователем. Если с *Save* и *Open* всё понятно, так как было описано ранее, то с другими пунктами мы сейчас разберёмся.

При нажатии на блок в боковой панели блоков, происходит срабатывание сигнала на добавление элемента на сцену, также сцена переходит в режим ожидания нажатия на сцену и при нажатии добавляет элемент в то место, где находится курсор.

При нажатии правой клавишей мыши на блок происходит его выделение для дальнейших действий, например передвижение, удаление, редактирование блока.

При нажатии левой кнопкой мыши начинает рисоваться линия, если отпустить нажатую клавишу на том же блоке, то просто откроется меню

действий, которые можно совершить с блоком. Если отпустить кнопку мыши на другом блоке, то произойдёт соединение.

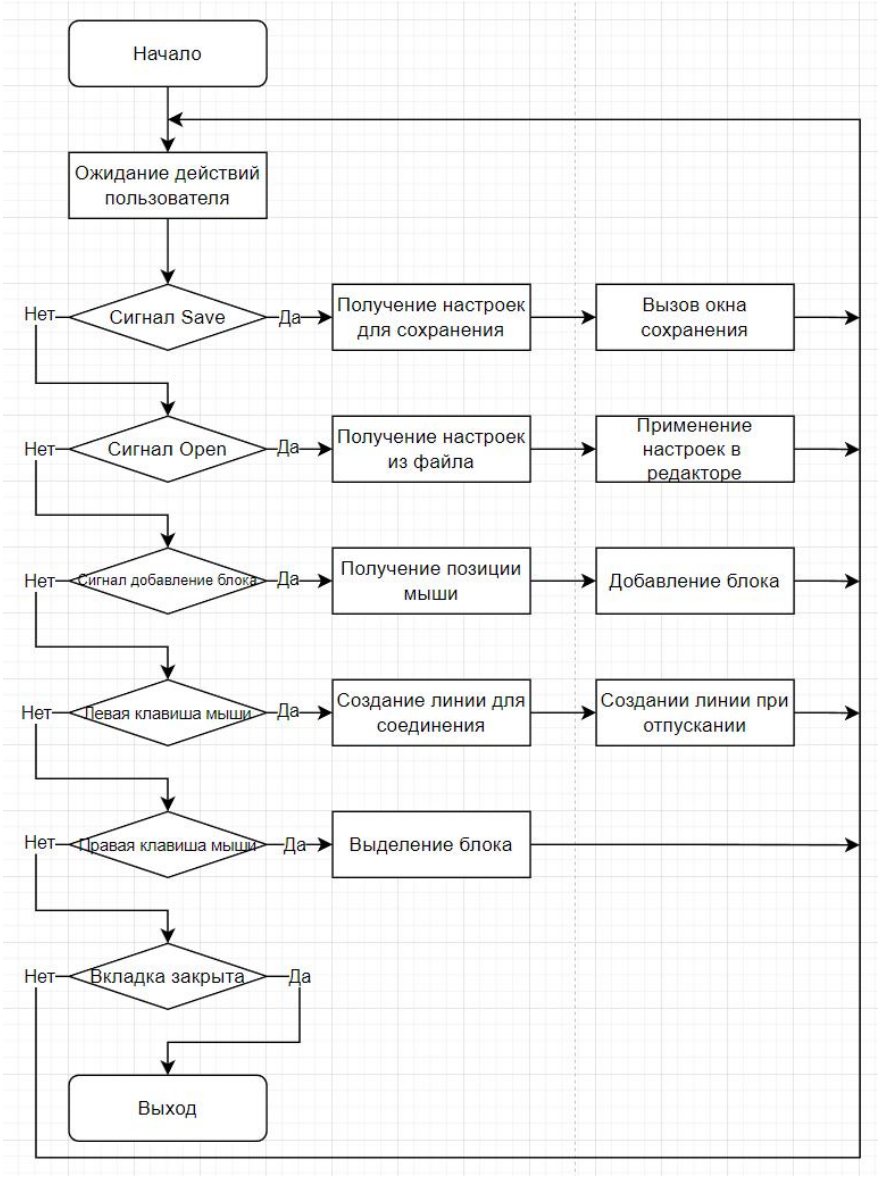


Рисунок 30 - Блок-схема реакции программы на действия пользователя в графическом редакторе

1.3.7 Руководство пользователя

При разработке программного продукта необходимо составить руководство пользователя для того, что у пользователя не возникало никаких проблем при работе с программой. Модель интерпретатора и концепция объяснения решений были внедрены в графический редактор *FBQL Editor*.

Главное окно программы и его возможности

При запуске программы мы попадаем на главное окно программы (рисунок 31). Оно делится на несколько областей: навигационная панель, боковая панель блоков и основной виджет с вкладками.

Навигационная панель пользователя располагается в верхней части программы и состоит из меню и панели инструментов.

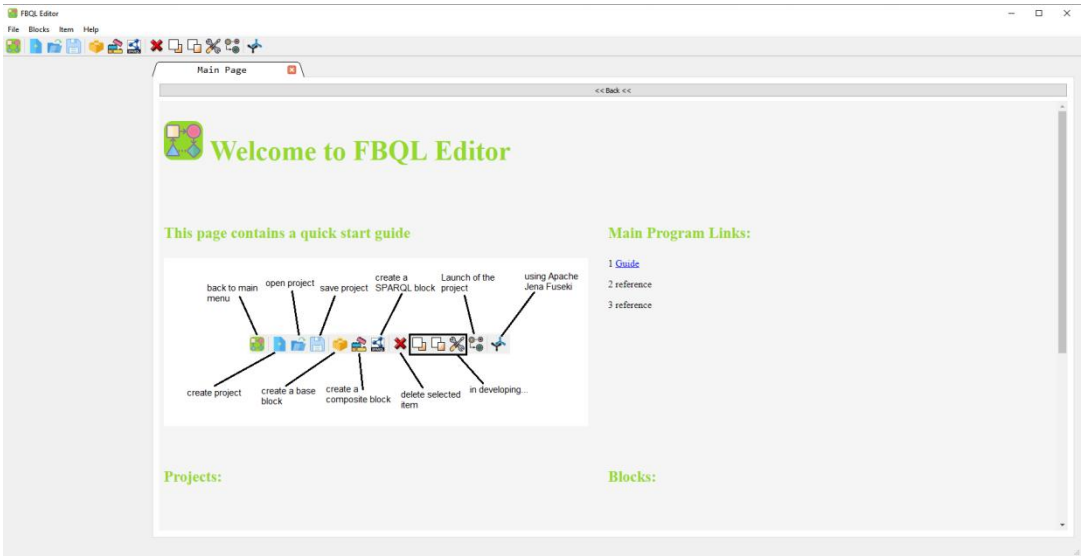


Рисунок 31 - Главное окно программы

На рисунках 32, 33, 34, 35 представлено меню пользователя с различным функциями программы.

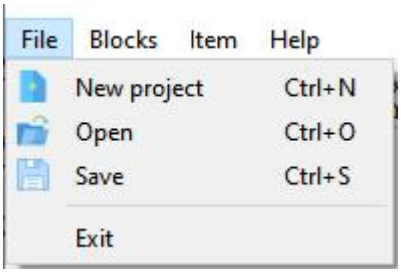


Рисунок 32 - Меню, вкладка «File»

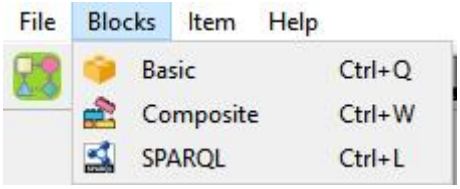


Рисунок 33 - Меню, вкладка «Blocks»

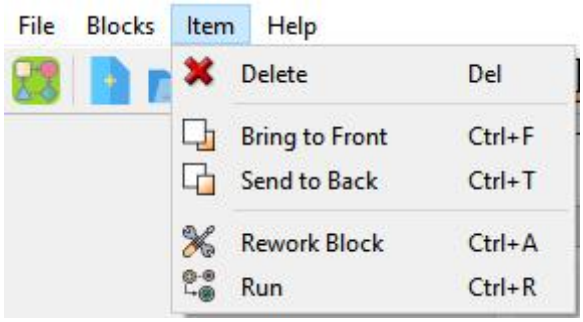


Рисунок 34 - Меню, вкладка «Item»

Рисунок 35 - Меню, вкладка «Help»

Панель инструментов на рисунке 36 дублирует функционал меню, но в более человеко-ориентированном виде.

Рисунок 36 - Панель инструментов

Используя меню и панель инструментов, можно получить доступ к следующим функциям программы:

- отрыть *Main Page* окно, с основной информацией о программе.

1) вид на панели инструментов: ;

2) доступ к данной функции через меню: нет возможности использовать через меню;

3) горячие клавиши: *Ctrl + M*;

- создать новый проект:

1) вид на панели инструментов: ;

2) доступ к данной функции через меню: *File -> New project;*

3) горячие клавиши: *Ctrl + N;*

- открыть блоки или проект:

1) вид на панели инструментов: ;

2) доступ к данной функции через меню: *File -> Open;*

3) горячие клавиши: *Ctrl + O;*

- сохранить блоки или проект:

1) вид на панели инструментов: ;

2) доступ к данной функции через меню: *File -> Save;*

3) горячие клавиши: *Ctrl + S;*

- открыть редактор для создания базисных блоков:

1) вид на панели инструментов: ;

2) доступ к данной функции через меню: *Blocks -> Basic;*

3) горячие клавиши: *Ctrl + Q;*

- открыть редактор для создания композитных блоков:

1) вид на панели инструментов: ;

2) доступ к данной функции через меню: *Blocks -> Composite;*

3) горячие клавиши: *Ctrl + W;*

- открыть редактор для создания *SPARQL*-блоков:

1) вид на панели инструментов: ;

2) доступ к данной функции через меню: *Blocks -> SPARQL;*

3) горячие клавиши: *Ctrl + L;*

- удалить выбеленные блоки:

1) вид на панели инструментов: ;

2) доступ к данной функции через меню: *Item -> Delete;*

3) горячие клавиши: *Del;*

- поместить выбранный блок поверх других:

1) вид на панели инструментов: ;

2) доступ к данной функции через меню: *Item -> Bring to Front;*

3) горячие клавиши: *Ctrl + F;*

- поместить выбранный блок под другие:

1) вид на панели инструментов: ;

2) доступ к данной функции через меню: *Item -> Send to Back;*

3) горячие клавиши: *Ctrl + T;*

- открыть выбранный блок на редактирование:

1) вид на панели инструментов: ;

2) доступ к данной функции через меню: *Item -> Rework Block;*

3) горячие клавиши:

- запустить открытый проект:

1) вид на панели инструментов: ;

2) доступ к данной функции через меню: *Item -> Run;*

3) горячие клавиши: *Ctrl + R;*

- открыть редактор для настроек работы с *Jena Fuseki Server*:

1) вид на панели инструментов: ;

2) доступ к данной функции через меню: нет;

3) горячие клавиши: *Ctrl + F;*

- о программе:

1) вид на панели инструментов: нет возможность использовать через панель инструментов;

2) доступ к данной функции через меню: *Help -> About ;*

3) горячие клавиши: *F1;*

В левой части программы расположилась боковая панель блоков, наполнение данной панели меняется в зависимости от редактора который сейчас выбран.

На рисунке 37 можно увидеть, что боковая панель состоит из 3 разделов «*Basic*», «*Composite*» и «*SPARQL*». Слева на рисунке 37 отображён внешний вид боковой панели при нажатии клавиши «*Basic*», по центре вид панели при нажатии клавиши «Composite», а справа при нажатии «*SPARQL*». Как можно заметить в зависимости от выбранного раздела отображаются различные типы блоков. Кроме уже объявленных типов, могут быть специфические разделы для конкретных редакторов.

И наконец самая важная часть нашей программы занимает большую часть программы. В данной области находится виджет позволяющий

пользоваться функциями программы (рисунок 38 и 39), например редакторами и окнами настроек программы. Данный виджет имеет возможность отображать окна в виде вкладок, делая программу более лёгкой и понятной в использовании.

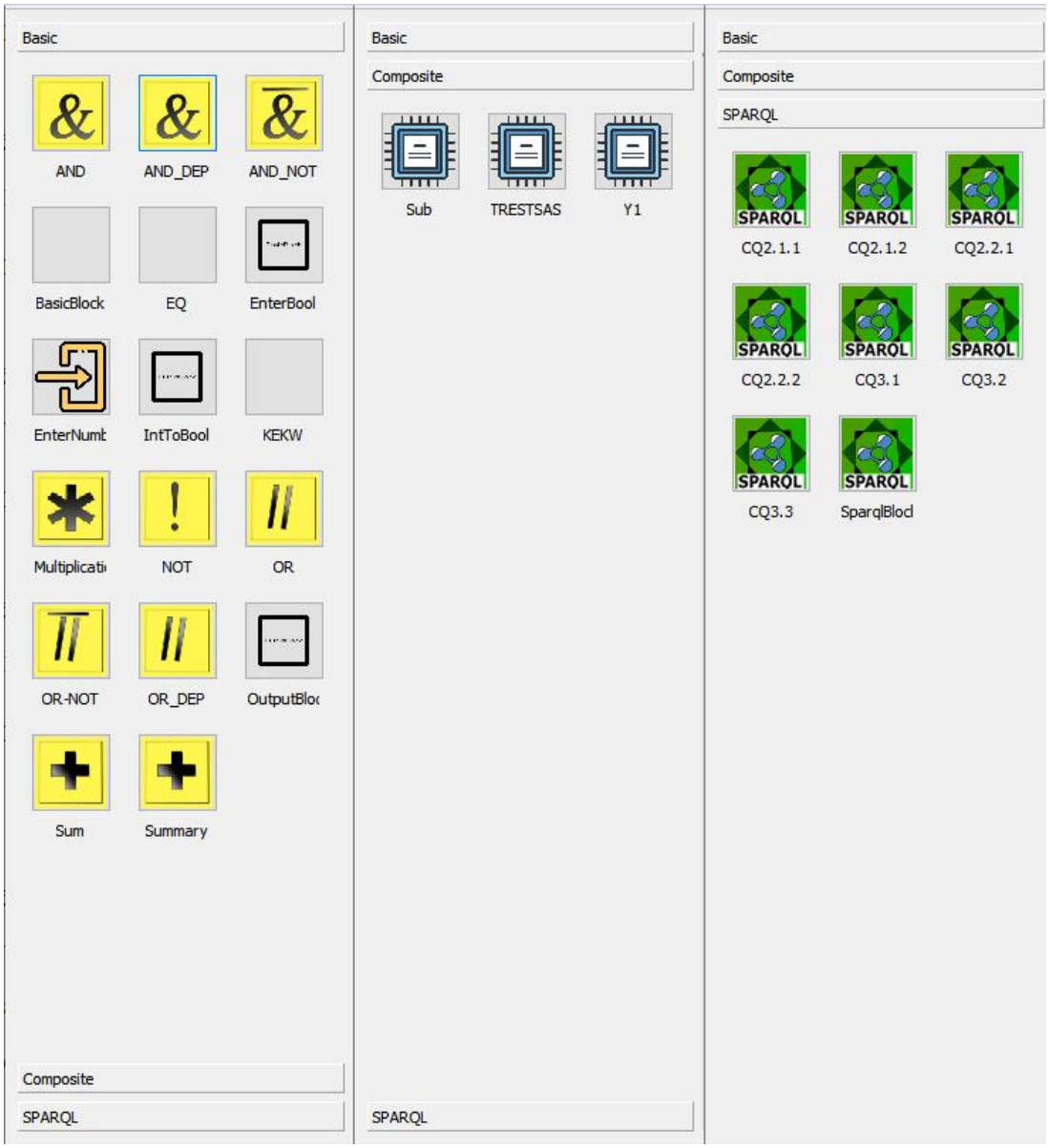


Рисунок 37 - Боковая панель блоков в 3 различных состояниях

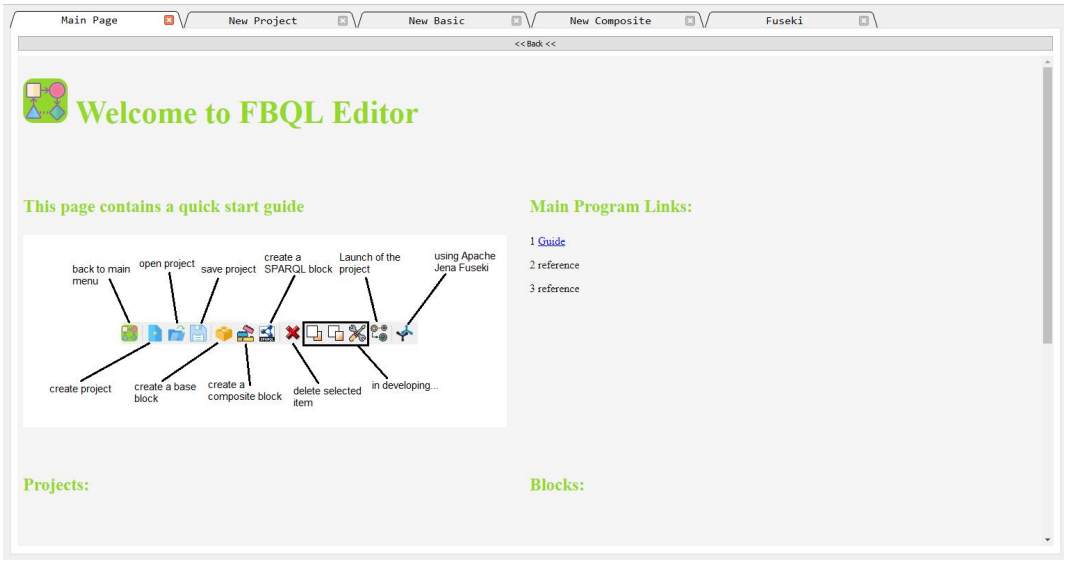


Рисунок 38 - Виджет с вкладками

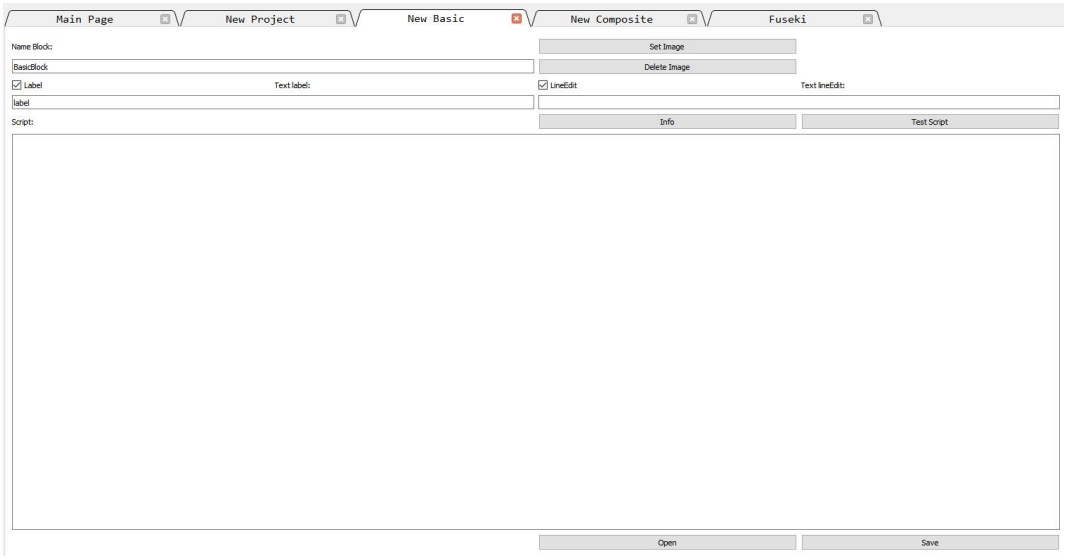


Рисунок 39 - Виджет при переключении вкладки

Окна открытия и сохранения блоков и проекта

Для открытия и сохранения блоков и проектов в нашей программе используются стандартные окна *Windows*, показанные на рисунках 40 и 41. Данные окна не отличаются в зависимости от открытого редактора, но могут меняться параметры для поиска. Форматы хранения данных:

- *.fbql* - формат для блоков;
- *.fbd* - формат хранения проектов;
- *.json* - старый формат хранения данных.

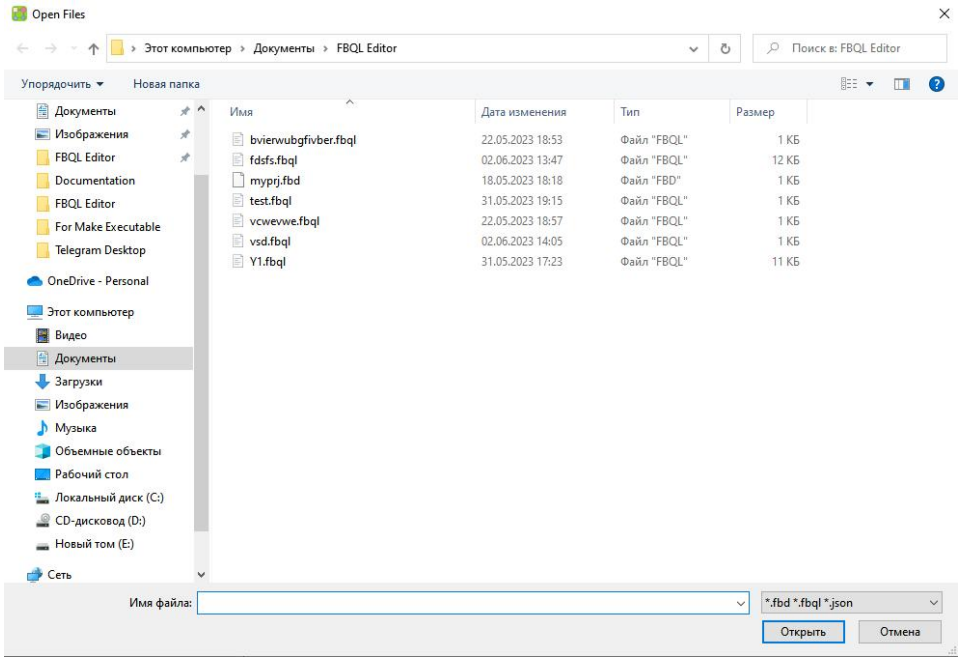


Рисунок 40 - Окно открытия файлов

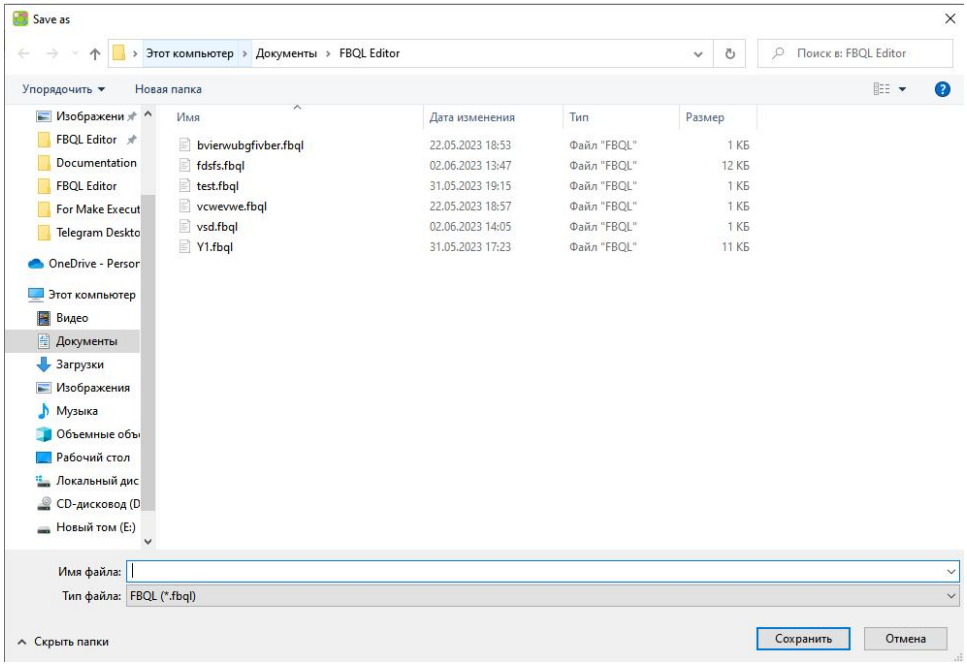


Рисунок 41 - Окно сохранения файлов

Редактор базисных блоков

Редактор базисных блоков изображённый на рисунке 42 нужен для создания и редактирования базисных блоков. Сверху в редакторе находятся поля с настройками базисного блока. Как они влияют на базисный блок описано в разделе 1.2.1.4. Снизу в текстовом поле “*Script*” вы сможете написать логику работы создаваемого блока.

Для примера, рассмотрим реализацию блока “логического ИЛИ”. Код на *JavaScript*, используемый для реализации данного блока.

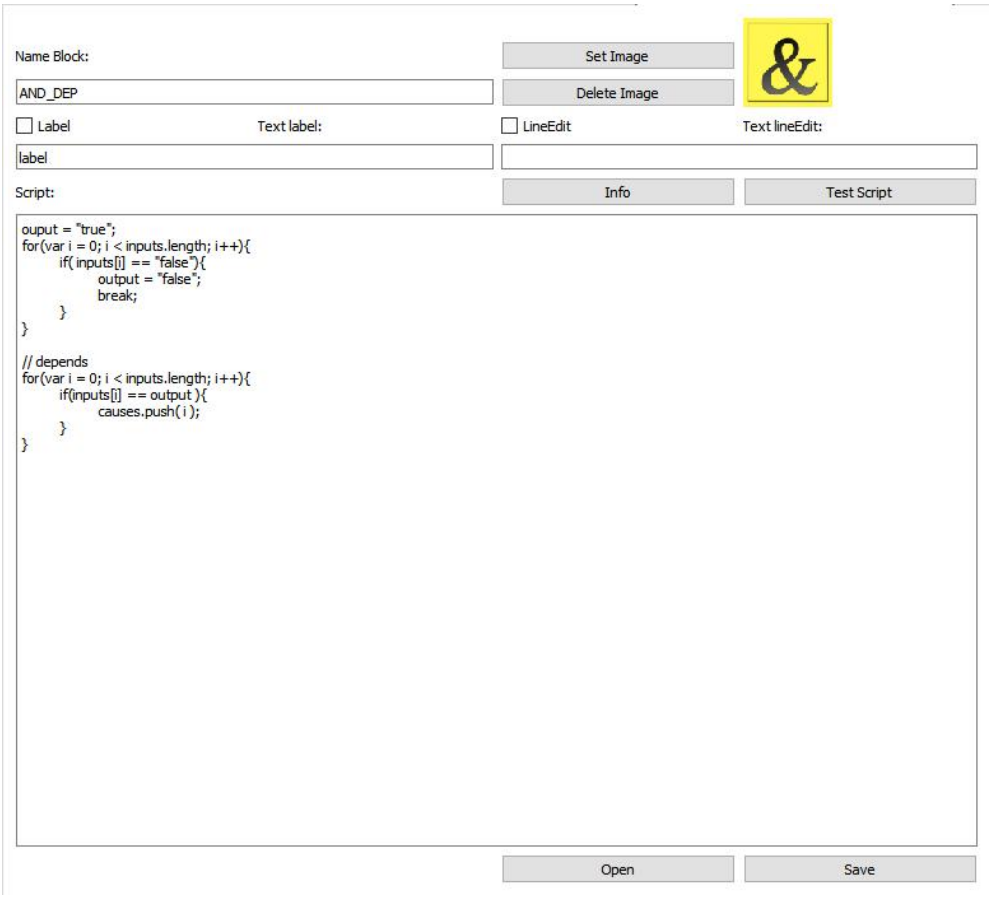


Рисунок 42 - Внешний вид редактора базисных блоков

Входы данного блока являются массивом Булевых значений – “*inputs*”. Результат будет записан в переменную “*output*”. Логика работы заключается в прогоне всех значений входного массива, если хотя бы одно значение является истинным, то результат работы блока будет истинным, иначе ложным (рисунок 43).

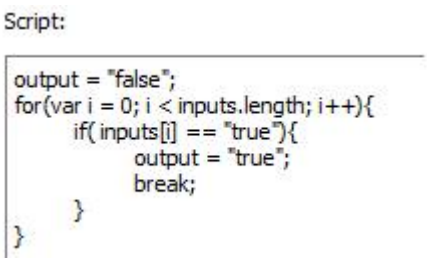


Рисунок 43 – Реализация скрипта для блока логического ИЛИ.

Для того чтобы выставить зависимости блока с другими блоками, а также для работы метода объяснения решений, используется секция кода, показанная на рисунке 44.

```
// depends
for(var i = 0; i < inputs.length; i++){
        if(inputs[i] == output ){
                causes.push( i );
        }
}
```

Рисунок 44 – Реализация блока кода для создания зависимостей.

Чтобы вход блока был записан в массив зависимостей – “*causes*”, значение входа должно быть равным выходному значению, поскольку именно эти входы влияют на работу блока. Массивы *causes* и *inputs*, а также переменная *output* – являются зарезервированными переменными, то есть они используются при работе интерпретатора, а также метода объяснения решений.

Для загрузки иконки базисного блока нужно воспользоваться кнопкой «*Set Image*». После нажатия откроется окно (рисунок 45) в котором можно будет выбрать изображения для конки на блоке.

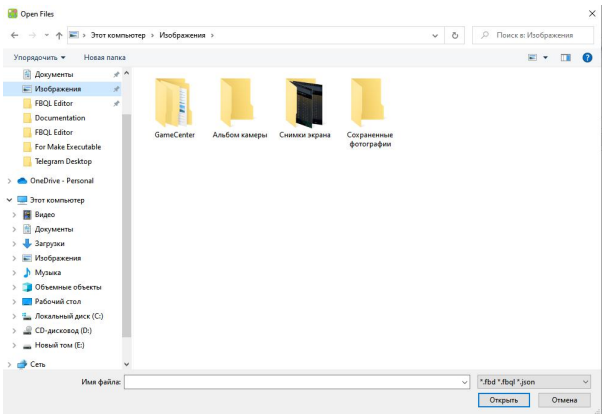


Рисунок 45 - Стандартное окно *Windows* для открытия файлов

Кнопка «*Info*» предназначена, для отображения информации о том, как писать скрипты для базисного блока.

Кнопка «*Test Script*» проверяет скрипт на наличие ошибок.

Кнопки «*Open*» и «*Save*» для открытия блока и сохранения соответственно.

Редактор композитных блоков

Редактор композитных блоков представлен на рисунке 46. Слева в боковой панели блоков кроме стандартных «*Basic*», «*Composite*» и «*SPARQL*» добавляется «*IO*». В котором можно использовать блоки «*Input*» и «*Output*» для формирования входов / выходов.

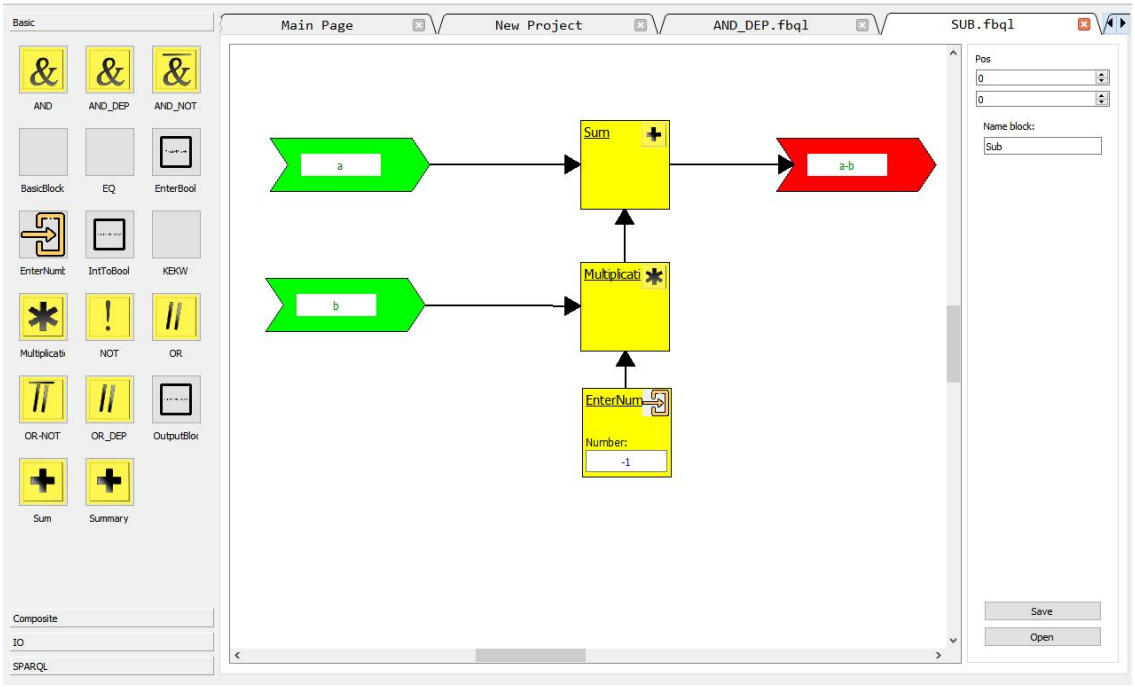


Рисунок 46 - Внешний вид редактора композитных блоков

Для добавления блока на сцену нужно нажать на нужный вам блок на боковой панели, а потом в место на сцене куда вам надо добавить блок. Для соединения блоков между собой, нужно сначала выбрать блок от которого хотите провести стрелку и зажав правую кнопку мыши перетащить курсор на блок к которому хотите провести стрелку. Отпустив кнопку мыши, вы увидите как появится стрелка соединяющая эти блоки.

Справа в редакторе находится функционал по настройке уникальных свойств блока. Поля «*Pos*» показывают позицию того блока, который вы выбрали. Также с помощью изменения значений можно менять позицию блока. «*Name block*» нужно для задания имени блока. Кнопки «*Open*» и «*Save*» для открытия блока и сохранения соответственно.

Редактор *SPARQL*-блоков

Слева в боковой панели редактора блоков находятся только 3 блока для создания *SPARQL*-запроса. Их непосредственное участие в формировании *SPARQL*-блока описано в пункте 1.2.2.4.

Для добавления блока на сцену нужно нажать на нужный вам блок на боковой панели, а потом в место на сцене куда вам надо добавить блок. Блоки «*VALUE*» и «*VAR*» должны обязательно находится внутри блока «*AREA*», иначе они не будут учитываться для формирования запроса. Для соединения блоков между собой, нужно сначала выбрать блок от которого хотите провести стрелку и зажав правую кнопку мыши перетащить курсор на блок к которому хотите провести стрелку. Отпустив кнопку мыши, вы увидите как появится стрелка соединяющая эти блоки. Два раза кликнув по стрелке появится поле для ввода текста, туда как раз и стоит внести отношение субъекта к объекту.

Справа в редакторе находится функционал по настройке уникальных свойств блока. Поля «*Pos*» показывают позицию того блока, который вы выбрали. Также с помощью изменения значений можно менять позицию блока. «*Name block*» нужно для задания имени блока. «*Limit*» используется для формирования «*Limit*» в *SPARQL*-запросе. Галочка «*Use text instead of a diagram*» позволяет включить режим работы редактора в виде текста, а не диаграмм. При сохранении блока с включённой галочкой текстовый запрос становится главным, а диаграмма вообще не будет сохраняться. Кнопки «*Open*» и «*Save*» для открытия блока и сохранения соответственно. Кнопка «*Query*» позволяет после редактирования диаграммы увидеть как программа воспринимает ваш запрос. Если же нажать на кнопку повторно, то происходит возврат к диаграмме.На рисунку 47 продемонстрирован внешний вид редактора с открытым блоков SPARQL. На рисунке 48 этот же блок с рисунка 47, но в текстовом виде.

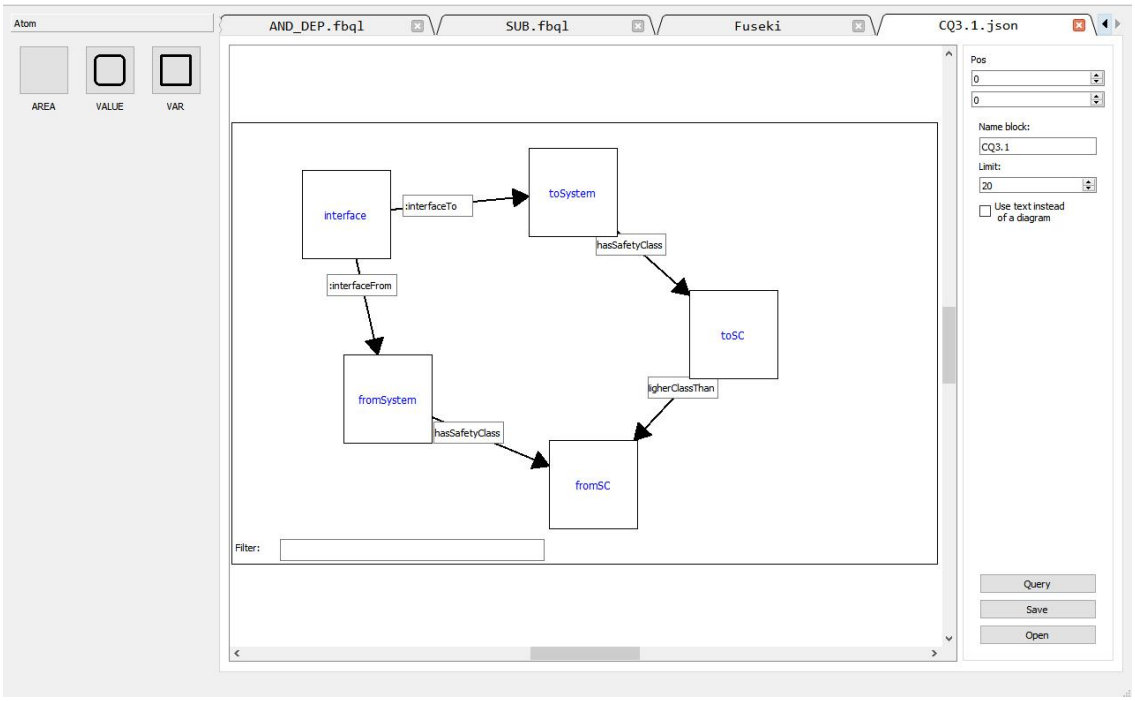


Рисунок 47 - Внешний вид редактора *SPARQL*-блоков

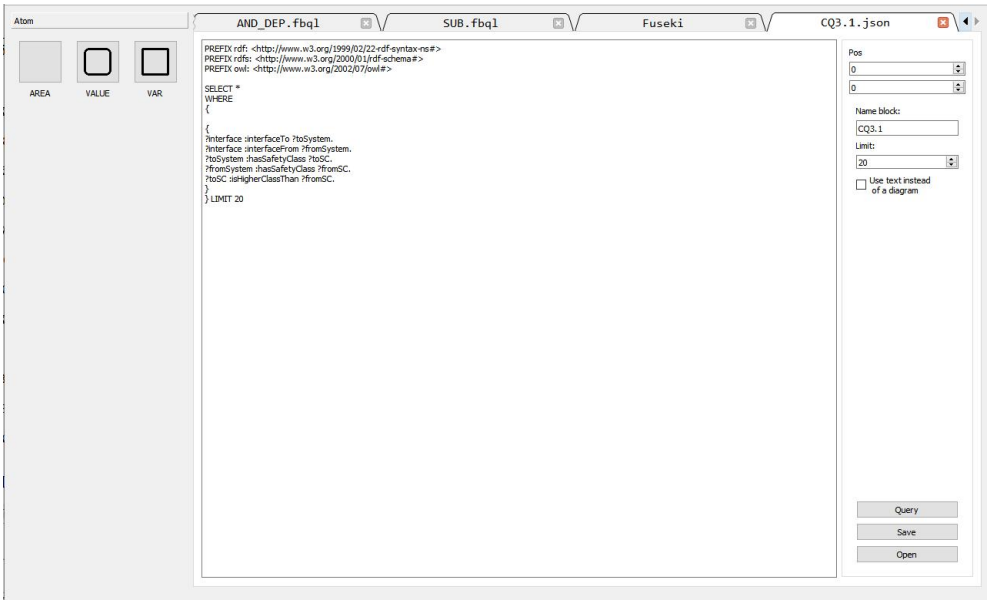


Рисунок 48 - Внешний вид редактора после нажатия кнопки «*Query*»

Редактор проектов

На рисунке 49 представлен внешний вид редактора проектов. Слева в боковой панели присутствуют только стандартные блоки. Для добавления блока на сцену нужно нажать на нужный вам блок на боковой панели, а потом в место на сцене куда вам надо добавить блок. Для соединения блоков между собой, нужно сначала выбрать блок от которого хотите провести стрелку и

зажав правую кнопку мыши перетащить курсор на блок к которому хотите провести стрелку. Отпустив кнопку мыши, вы увидите как появится стрелка соединяющая эти блоки.

Справа в редакторе находится функционал по настройке уникальных свойств блока. Поля «*Pos*» показывают позицию того блока, который вы выбрали. Также с помощью изменения значений можно менять позицию блока.

После того как пользователь завершит редактирование проекта, он может запустить его на выполнение и увидеть результат работы.

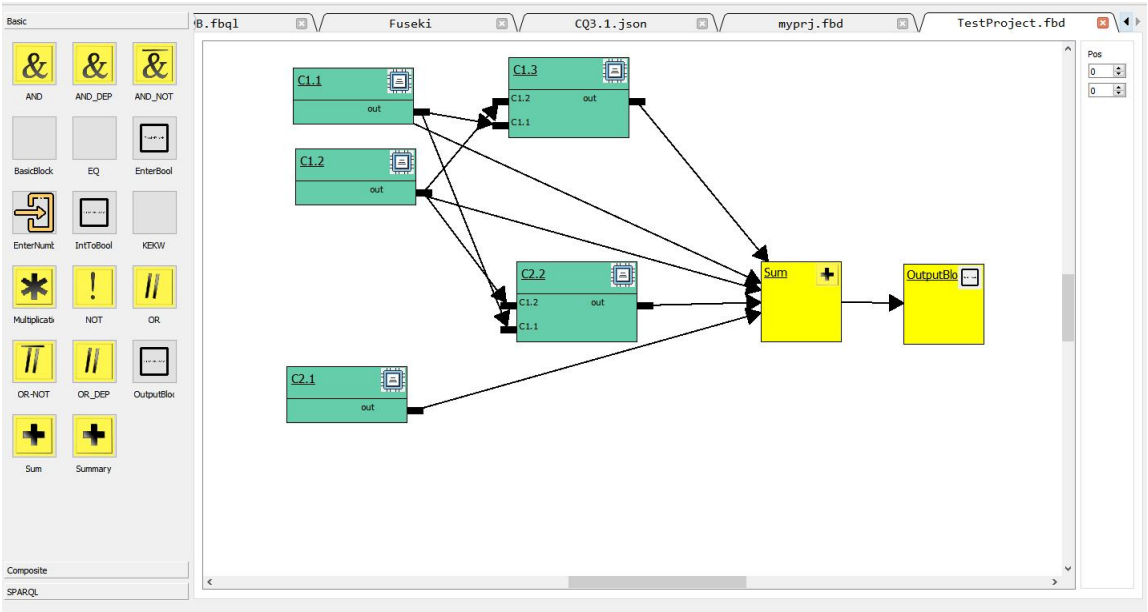


Рисунок 49 - Внешний вид редактора проектов

В нашей программе также представлена возможность объяснения решения (рисунок 50), которая позволяет понять, почему получился именно такой результат. Воспользоваться данной функцией можно после того, как ваш проект был запущен и завершил работу. Для отображения объяснения нужно навести курс на интересующий вас блок и появится подсветка, показывающая какие блоки привели к такому результату.

На рисунке 50 можно увидеть как при наведении на блок «*OutputBlock*» подсвечивается сначала блок «*Sum*», а далее те значения которые повлияли на результат блока. Так как «*false*» равноценно нулю, то при сложении он не влияет на результат, поэтому блоки которые на выход дают «*false*» не были подсвечены объяснением решения.

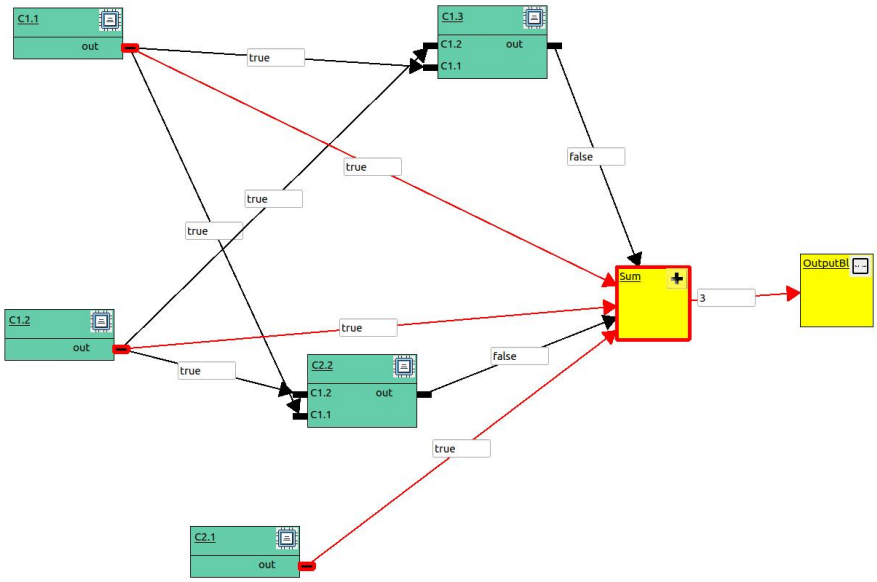


Рисунок 50 - Пример объяснения решения

Окно настройки Jena Fuseki Server

Данное окно создано для настройки программы в «общении» с *Jena Fuseki Server* (рисунок 51). Первый параметр для настройки это *IP*-адресс, куда мы можем занести его в формате *IPv4*. Далее идёт порт, для указания порта на котором работает *Jena Fuseki Server*. «*Dataset Name*» нужно для указания названия набора данных с которым мы будем работать. Следом можно увидеть поле «*Prefixes*», где можно отредактировать список префиксов для формирования *SPARQL*-запроса.

Кнопка «*Add Row*» нужна для добавления ещё одной записи в список префиксов. Кнопка «*Remove Row*» для удаления записи из списка.

Кнопка «*Test Settings*» нужна для проверки доступа к *Jena Fuseki Server* по введённым данным.

Кнопка «*Save Settings*» для сохранения настроек. Они сохраняются в файле настроек, поэтому не пропадут даже при перезагрузке программы.

Кнопка «*Close*» закрывает окно редактирования настроек без сохранения.

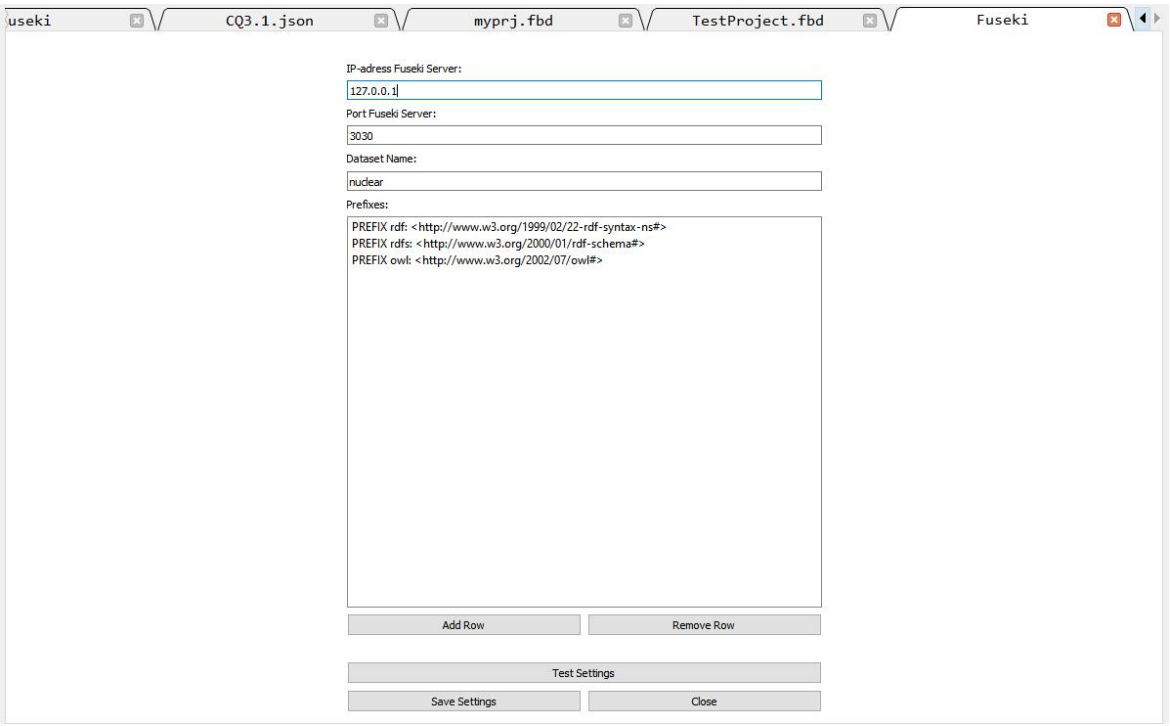


Рисунок 51 - Окно настройки *Jena Fuseki Server*

Окно запуска программы

Окно запуска программы делится на 2 части (рисунок 52). В верхней части окна отображается графическая интерпретация проекта который будет исполняться, на схеме присутствуют только базисные и *SPARQL*-блоки, потому как на данном этапе композитные блоки уже были раскрыты.

После нажатия кнопки «*Execute*» произойдёт выполнение программы и результат выведется в нижней части окна. Также там могут выводиться ошибки и системные сообщения *JavaScript*.

Использование *GraphViz* для визуализации работы программы.

В программе есть встроенная библиотека – *GraphViz*. С её помощью можно наглядно отследить работу блоков и их связи. После запуска проекта, программа построит схему взаимосвязи блоков. В состав блока входят: его тип, название, количество входов/выходов, значения на входе/выходе и пользовательские данные.

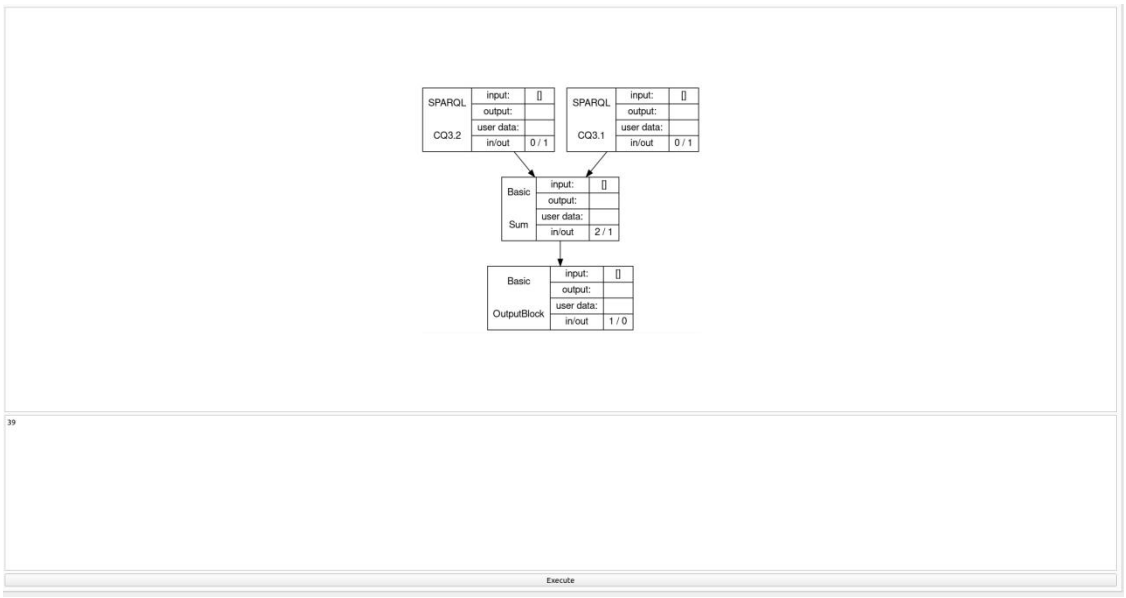


Рисунок 52 - Внешний вид окна запуска программы

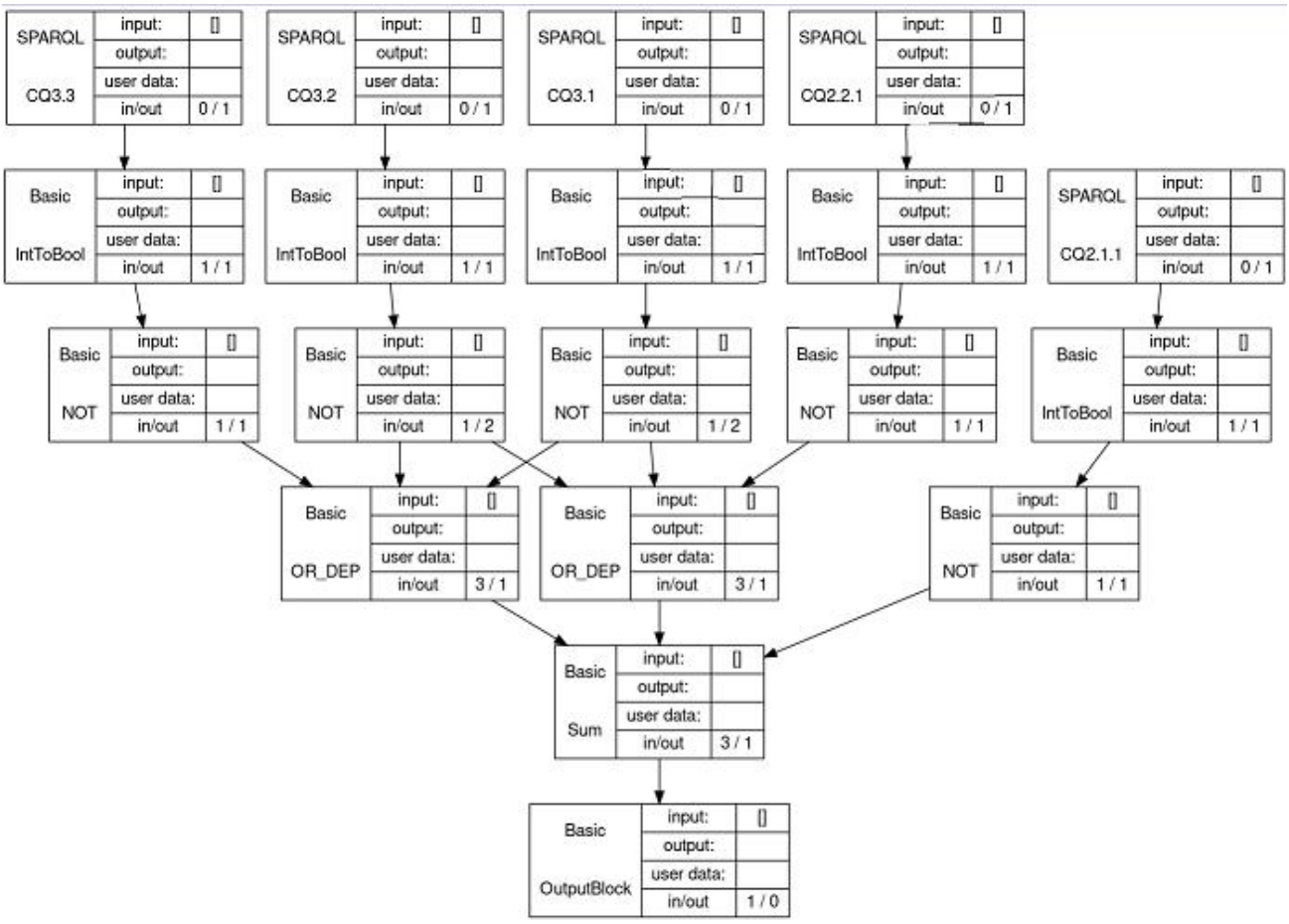


Рисунок 53 – Визуальный пример использования библиотеки GraphViz

1.3.8. Отладка и тестирование

Тестирования главного окна

При тестировании главного окна мы проверяли работоспособность всех функции связанных с GUI нашей программы. Результаты тестирования представлены в таблице 1.

Таблица 1 - Тестирование главного окна

Название теста	Ожидаемый результат	Полученный результат	Оценка
Открытие MainPage	При нажатии кнопки «MainPage» на панели инструментов, откроется окно приветствия пользователя.	Окно открылось	Верно
Открытие редактора проекта	При нажатии кнопки «New project» на панели инструментов, откроется окна редактора проекта.	Окно открылось	Верно
Открытие редактора базисных блоков	При нажатии кнопки «Basic» на панели инструментов, откроется окно редактора базисных блоков.	Окно открылось	Верно
Открытие редактора композитных блоков	При нажатии кнопки «Composite» на панели инструментов, откроется окно редактора композитных блоков.	Окно открылось	Верно
Открытие редактора SPARQL-блоков	При нажатии кнопки «SPARQL» на панели инструментов, откроется окно редактора SPARQL-блоков.	Окно открылось	Верно
Открытие окна запуска проекта	Создаём проект для тестирования. При нажатии кнопки «Run» на панели инструментов, откроется окно запуска проекта.	Окно открылось	Верно
Открытие редактирования настроек Jena Fuseki Server	При нажатии кнопки «FusekiServer» на панели инструментов, откроется окно редактирования настроек Jena Fuseki Server.	Окно открылось	Верно

Название теста	Ожидаемый результат	Полученный результат	Оценка
Открытие блоков из файлов	При нажатии кнопки «Open» на панели инструментов, откроется окно выбора файлов для загрузки. Выбрав несколько файлов нажимаем открыть и видим, что они открылись.	Файлы успешно открылись	Верно
Сохранение блока в файл	Создаём блок. При нажатии кнопки «Save» на панели инструментов, откроется окно сохранения файла. Выбрав настройки для сохранения файла нажимаем кнопку сохранить. Проверяем сохранённый файл и видим, что всё сохранилось как нужно.	Файлы успешно сохранён	Верно
Удаление блока	Выбираем нужный нам блок на сцене. Нажимает кнопку «Delete» и видим как блок удалился со сцены.	Блок успешно удалился	Верно

Тестирование графического редактора

При тестировании графического окна мы проверяли работоспособность всех функции связанных с взаимодействие с графическими элементами. Результаты тестирования представлены в таблице 1.

Таблица 2 - Тестирование графического редактора

Название теста	Ожидаемый результат	Полученный результат	Оценка
Добавление блока	Выбираем нужный нам блок на боковой панели блоков. Видим, что он выделился. Далее нажимаем на сцену и наблюдаем, что выбранный нами блок появился на месте расположения курсора.	Блок был успешно добавлен на сцену	Верно

Название теста	Ожидаемый результат	Полученный результат	Оценка
Удаление блока	Выбираем нужный нам блок на сцене. Нажимает кнопку «Delete» и видим как блок удалился со сцены.	Блок успешно удалился	Верно
Соединение 2 блоков с помощью стрелки	Нажимаем правой кнопкой мыши на блок и не отжимая, ведём к блоку с которым хотим объединить и отпускаем. Появляется стрелка.	Стрелка появилась	Верно
Открытие меню блока	Нажимаем правой кнопкой мыши на блок и сразу отпускаем. Появляется меню блока.	Открылось меню блока	Верно
Передвижение блока	Выбираем блок и начинаем двигать, блок передвигается.	Блок передвинулся	Верно

Тестовый пример

Задача запустить программу и открыть выкладку создания нового проекта. Далее с помощью блоков создать программу суммирования 2 чисел и запустить проект. Проверить результаты.

На рисунке 54 и 55 видно, как мы запустили программу и открыли окно создания проекта, при этом всё выполнилось в пределах наших ожиданий.

Рисунок 54 - Окно программы после запуска

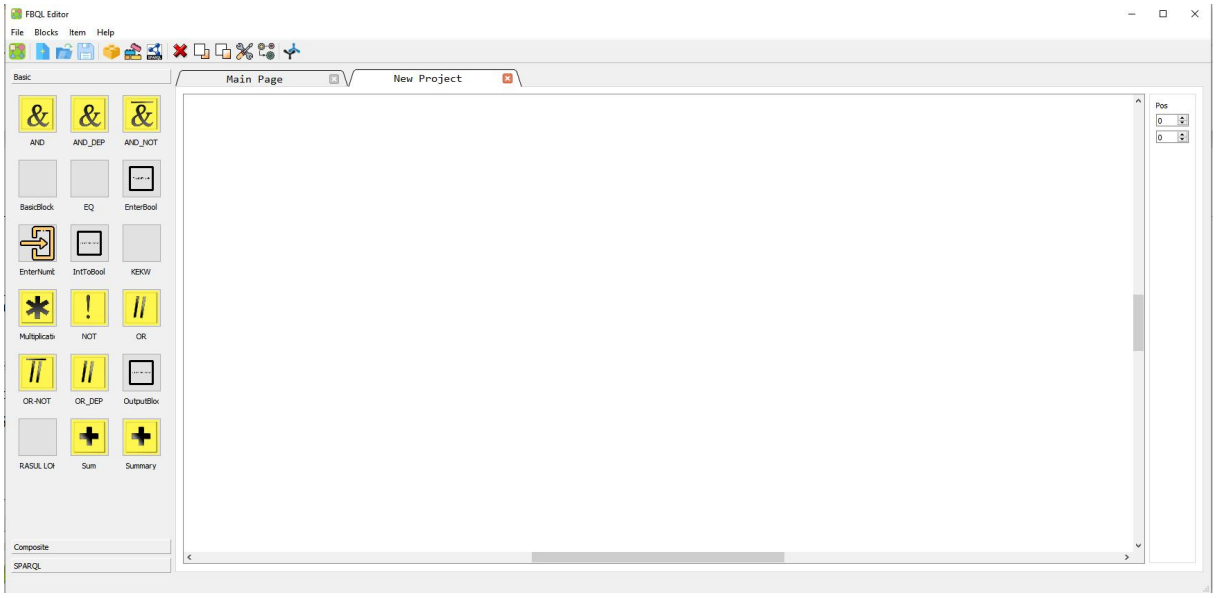


Рисунок 55 - Окно программы после нажатия кнопки создания проекта

На рисунке 56 продемонстрирован результат добавления на сцену блоков и соединения их стрелками. По данному рисунку понятно, что добавление и передвижение блоков работает без ошибок, как и соединение стрелками.

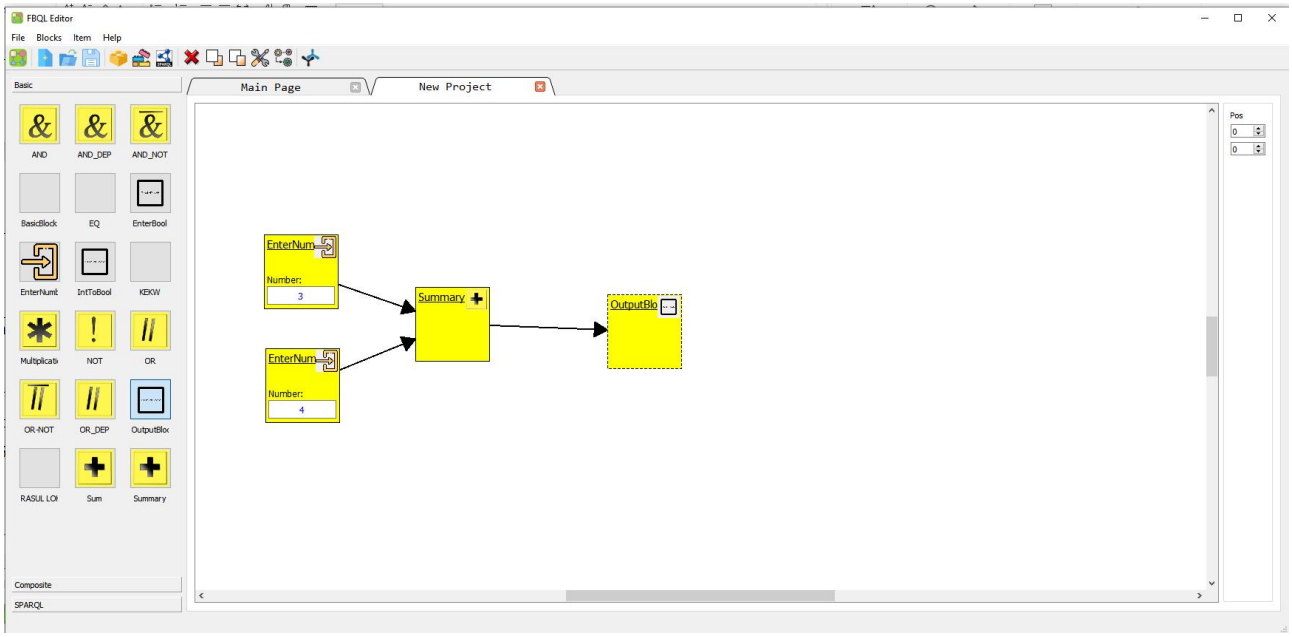


Рисунок 56 - Окно программы после добавления блоков

На рисунке 57 можем увидеть результат выполнннения нашего проекта. Мы складывали 2 числа «3» и «4» и получили результата «7», что является верным.

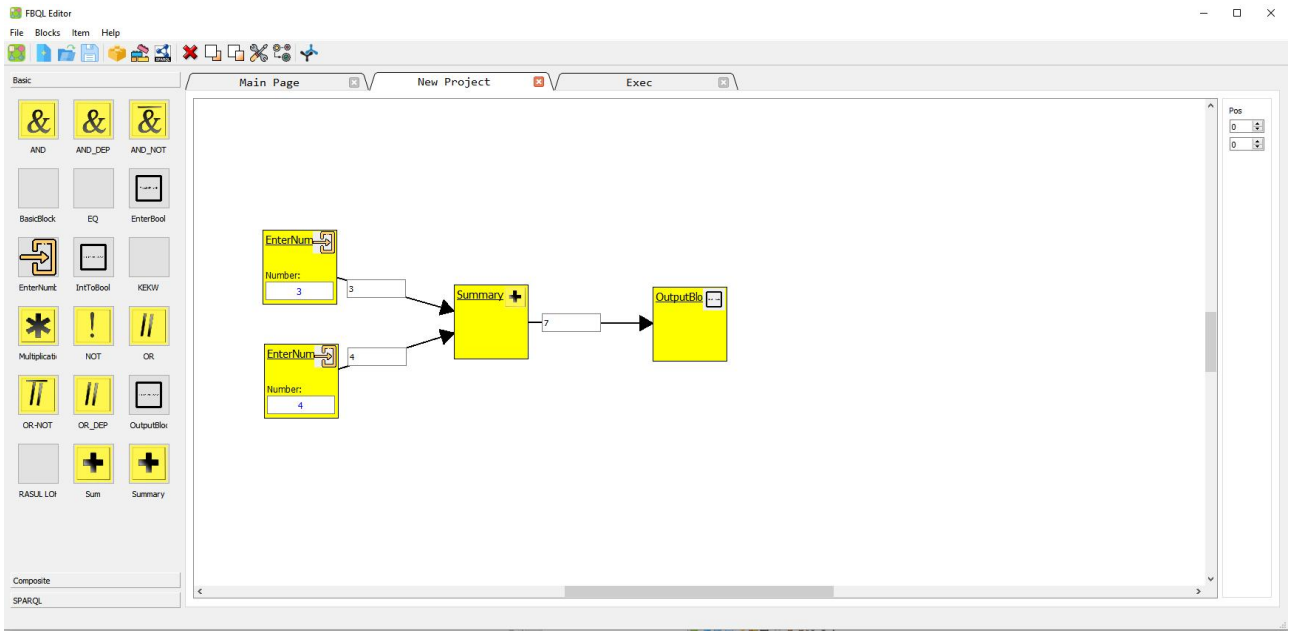


Рисунок 57 - Окно программы после того как проект отработал

В ходе тестирования были проверены работоспособность главного окна и графических редакторов, недочёты не были выявлены, все возможности программы отработали правильно.

2 Безопасность жизнедеятельности

2.1 Введение

Освещение рабочего места – важнейший фактор создания нормальных условий труда. Практически возникает необходимость освещения как естественным, так и искусственным светом. Первый случай характерен для светлого времени суток и при работе в помещениях, в которых имеются проёмы в стенах и крыше здания, во втором случае применяются соответствующие осветительные установки искусственного света.

Естественное освещение по своему спектральному составу является наиболее приемлемым. Искусственное же, наоборот, отличается относительной сложностью восприятия его зрительным органом человека. Это связано с тем, что суточные переходные режимы естественной освещённости имеют малую частоту при достаточно высокой (днём) или очень низкой (ночью) интенсивности светового потока, а искусственные – довольно большую частоту при недостаточной в целом освещённости. Поэтому при искусственном освещении начинают возникать неустойчивые зрительные процессы, которые из-за большой частоты сменяемости световых условий накладываются друг на друга, не давая глазу времени адаптироваться к новым условиям. От усиленной деятельности приспособительных механизмов глаза быстро утомляются, что вызывает физическую усталость организма.

Несмотря на это, искусственное освещение необходимо, как важнейший фактор для приближения ночных условий труда к дневным. Основное отличие ночных условий труда от дневных состоит в том, что при ночных условиях труда отсутствует достаточная освещённость поля зрения работающих равномерно распределенным световым потоком. Стимулирующее действие света на организм при недостаточной освещенности снижается, поэтому ночные условия труда более тяжёлые с физиологической точки зрения. Однако основа естественного и искусственного света общая – энергетическая, поэтому их разделение вызвано разницей в спектре и интенсивности. Последнее

предположение можно принять только в первом приближении, поскольку спектр естественного света полностью не выяснен и не воспроизведён. Более реально создание световой среды, обеспечивающей психофизиологический комфорт и заданное эмоционально-эстетическое воздействие с учётом свето-цветовой доминанты в поле зрения.

По функциональному назначению искусственное освещение подразделяют на рабочее, аварийное и специальное, которое может быть охранным, дежурным, эвакуационным, бактерицидным и др. Нормы освещенности для каждого из видов установлены в СНиП 23-05-19 «Естественное и искусственное освещение» от 01.01.2019.

Рабочее освещение предназначено для обеспечения нормального выполнения производственного процесса, прохода людей, движения транспорта и является обязательным для всех производственных помещений.

Аварийное освещение устраивают для продолжения работы в тех случаях, когда внезапное отключение рабочего освещения (при авариях) и связанное с этим нарушение нормального обслуживания оборудования могут вызвать взрыв, пожар, отравление людей, нарушение технологического процесса и т.д. Минимальная освещённость рабочих поверхностей при аварийном освещении должна составлять 5% нормируемой освещенности рабочего освещения, но не менее 2 лк.

Эвакуационное освещение предназначено для обеспечения эвакуации людей из производственного помещения при авариях и отключении рабочего освещения; организуется в местах, опасных для прохода людей: на лестничных клетках, вдоль основных проходов производственных помещений, в которых работают более 50 чел. Минимальная освещённость на полу основных проходов и на ступеньках при эвакуационном освещении должна быть не менее 0,5 лк, на открытых территориях – не менее 0,2 лк.

Охранное освещение устраивают вдоль границ территорий, охраняемых специальным персоналом. Наименьшая освещённость в ночное время 0,5 лк.

Сигнальное освещение применяют для фиксации границ опасных зон; оно указывает на наличие опасности, либо на безопасный путь эвакуации.

2.2 Требования к помещениям для работы с ПЭВМ

Требования к помещениям для работы с ПЭВМ определены в СанПиН 2.2.2/2.4.1340-03 «Гигиенические требования к персональным электронно-вычислительным машинам и организации работы» и содержат следующие положения:

- помещения для эксплуатации ПЭВМ должны иметь естественное и искусственное освещение. Эксплуатация ПЭВМ в помещениях без естественного освещения допускается только при соответствующем обосновании и наличии положительного санитарно-эпидемиологического заключения, выданного в установленном порядке;

- естественное и искусственное освещение должно соответствовать требованиям действующей нормативной документации. Окна в помещениях, где эксплуатируется вычислительная техника, преимущественно должны быть ориентированы на север и северо-восток. Оконные проёмы должны быть оборудованы регулируемыми устройствами типа: жалюзи, занавесей, внешних козырьков и др;

- не допускается размещение мест пользователей ПЭВМ во всех образовательных и культурно-развлекательных учреждениях для детей и подростков в цокольных и подвальных помещениях;

- площадь на одно рабочее место пользователей ПЭВМ с ВДТ на базе электронно-лучевой трубки (ЭЛТ) должна составлять не менее 6 м2 , в помещениях культурно-развлекательных учреждений и с ВДТ на базе плоских дискретных экранов (жидкокристаллические, плазменные) - 4,5 м2 . При использовании ПВЭМ с ВДТ на базе ЭЛТ (без вспомогательных устройств - принтер, сканер и др.), отвечающих требованиям международных стандартов безопасности компьютеров, с продолжительностью работы менее 4-х часов в день допускается минимальная площадь 4,5 м2 на одно рабочее место пользователя (взрослого и учащегося высшего профессионального образования);

- для внутренней отделки интерьера помещений, где расположены ПЭВМ, должны использоваться диффузно отражающие материалы с коэффициентом отражения для потолка - 0,7 - 0,8; для стен - 0,5 - 0,6; для пола - 0,3 - 0,5.

- полимерные материалы используются для внутренней отделки интерьера помещений с ПЭВМ при наличии санитарно-эпидемиологического заключения;

- помещения, где размещаются рабочие места с ПЭВМ, должны быть оборудованы защитным заземлением (занулением) в соответствии с техническими требованиями по эксплуатации;

- не следует размещать рабочие места с ПЭВМ вблизи силовых кабелей и вводов, высоковольтных трансформаторов, технологического оборудования, создающего помехи в работе ПЭВМ.

2.3 Требования к освещению на рабочих местах, оборудованных ПЭВМ

Требования к освещению на рабочих местах, оборудованных ПЭВМ, определены в СанПиН 2.2.2/2.4.1340-03 «Гигиенические требования к персональным электронно-вычислительным машинам и организации работы» и содержат следующие положения:

- рабочие столы следует размещать таким образом, чтобы видеодисплейные терминалы были ориентированы боковой стороной к световым проёмам, чтобы естественный свет падал преимущественно слева;

- искусственное освещение в помещениях для эксплуатации ПЭВМ должно осуществляться системой общего равномерного освещения. В производственных и административно-общественных помещениях, в случаях преимущественной работы с документами, следует применять системы комбинированного освещения (к общему освещению дополнительно устанавливаются светильники местного освещения, предназначенные для освещения зоны расположения документов);

- освещённость на поверхности стола в зоне размещения рабочего документа должна быть 300 - 500 лк. Освещение не должно создавать бликов на поверхности экрана. Освещённость поверхности экрана не должна быть более 300 лк;

- следует ограничивать прямую блескость от источников освещения, при этом яркость светящихся поверхностей (окна, светильники и др.), находящихся в поле зрения, должна быть не более 200 кд/м2;

- следует ограничивать отражённую блескость на рабочих поверхностях (экран, стол, клавиатура и др.) за счёт правильного выбора типов светильников и расположения рабочих мест по отношению к источникам естественного и искусственного освещения, при этом яркость бликов на экране ПЭВМ не должна превышать 40 кд/м2 и яркость потолка не должна превышать 200 кд/м2;

- показатель ослепленности для источников общего искусственного освещения в производственных помещениях должен быть не более 20. Показатель дискомфорта в административно-общественных помещениях - не более 40, в дошкольных и учебных помещениях - не более 15;

- яркость светильников общего освещения в зоне углов излучения от 50 до 90 градусов с вертикалью в продольной и поперечной плоскостях должна составлять не более 200 кд/м2 , защитный угол светильников должен быть не менее 40 градусов;

- светильники местного освещения должны иметь не просвечивающий отражатель с защитным углом не менее 40 градусов;

- следует ограничивать неравномерность распределения яркости в поле зрения пользователя ПЭВМ, при этом соотношение яркости между рабочими поверхностями не должно превышать 3:1 - 5:1, а между рабочими поверхностями и поверхностями стен и оборудования - 10:1;

- в качестве источников света при искусственном освещении следует применять преимущественно люминесцентные лампы типа ЛБ и компактные люминесцентные лампы (КЛЛ). При устройстве отражённого освещения в производственных и административно-общественных помещениях

допускается применение метало-галогенных ламп. В светильниках местного освещения допускается применение ламп накаливания, в том числе галогенных;

- для освещения помещений с ПЭВМ следует применять светильники с зеркальными параболическими решётками, укомплектованными электронными пускорегулирующими аппаратами (ЭПРА). Допускается использование многоламповых светильников с электромагнитными пускорегулирующими аппаратами (ЭПРА), состоящими из равного числа опережающих и отстающих ветвей. Применение светильников без рассеивателей и экранирующих решёток не допускается. При отсутствии светильников с ЭПРА лампы многоламповых светильников или рядом расположенные светильники общего освещения следует включать на разные фазы трехфазной сети;

- общее освещение при использовании люминесцентных светильников следует выполнять в виде сплошных или прерывистых линий светильников, расположенных сбоку от рабочих мест, параллельно линии зрения пользователя при рядном расположении видеодисплейные терминалов. При перемитральном расположении компьютеров линии светильников должны располагаться локализовано над рабочим столом ближе к его переднему краю, обращённому к оператору;

- коэффициент запаса для осветительных установок общего освещения должен приниматься равным 1,4;

- коэффициент пульсации не должен превышать 5%;

- для обеспечения нормируемых значений освещенности в помещениях для использования ПЭВМ следует проводить чистку стёкол оконных рам и светильников не реже двух раз в год и проводить своевременную замену перегоревших ламп.

Заключение

В ходе выполнения выпускной квалификационной работы была разработана программа предназначенная для создания, редактирования и выполнения и обработки результатов обращения к хранилищам данных *RDF*. Данная программа используется для оценки свойств систем, представленных в виде онтологии на основе *RDF*. *FBD* включает в себя три типа (функциональных) блоков: базовые, композитные и *SPARQL* блоки. Имеется возможность использовать как библиотечные, так и пользовательские блоки.

В рамках выпускной работы были разработаны:

- графический язык для описания и обработки *SPARQL*-запросов на основе функционально-блочных диаграмм;

- редактор для базовых блоков с возможностью их создания и редактирования;

- редактор для композитных блоков с возможностью создания в них сложных схем;

- редактор для *SPARQL*-блоков, который предоставляет огромный функционал по созданию *SPARQL*-запросов как в виде диаграмм, так и в текстовом представлении;

- редактор для создания функциональной схемы, которая может исполняться и выдавать результат своей работы;

- возможность настройки работы с *Jena Fuseki Server* как на локальном компьютере, так и в сети;

- блок исполнения схемы с выводом промежуточных и итоговых результатов.

При создании программы мы столкнулись со следующими трудностями и проблемами:

- не получилось реализовать все возможности *SPARQL*, например выстраивание чёткой последовательности блоков «*MINUS*» и «*UNION*»;

- не удалось получать количество записей внутри самого запроса с помощью «*count*». Происходило, то что не удалось пока понять, если запрос

производился без данного оператора, то всё отрабатывало как ожидалось, но как только его использовали запрос проходит с триплетами расположенными в обратной последовательности, от чего ломался *Fuseki Server*. Причину неправильной работы Fuseki выяснить не удалось, как и причину почему триплеты переворачивались;

- из-за активного использования указателей, если небольшие утечки памяти, но где именно выяснить не удалось.

Список использованных источников

1. Документация UML. [Электронный ресурс]: Режим доступа: https://www.uml.org

2. Документация ER-диаграммы. [Электронный ресурс]: Режим доступа: https://www.lucidchart.com/pages/ru/erd-диаграмма

3. Документация для функционально-блочных диаграмм международных стандартов IEC 61131-3 и IEC 61499. [Электронный ресурс]: Режим доступа: https://ru.wikipedia.org/wiki/IEC_61499

4. Документация для диаграмм потоков данных dataflow. [Электронный ресурс]: Режим доступа: https://habr.com/ru/articles/668684/

5. Документация GQL. [Электронный ресурс]: Режим доступа: https://grooviter.github.io/gql/docs/html5/index.html

6. Документация PGQL. [Электронный ресурс]: Режим доступа: https://pgql-lang.org/spec/1.4/

7. Документация G-CORE. [Электронный ресурс]: Режим доступа: https://github.com/G-Core/product-documentation

8. Документация LabVIEW. [Электронный ресурс]: Режим доступа: https://engineering.purdue.edu/~aae520/Labview_manuals/LVUser.pdf

9. Документация Simulink. [Электронный ресурс]: Режим доступа: https://www.mathworks.com/help/simulink/

10. Документация GRAFCET. [Электронный ресурс]: Режим доступа: https://www.mhj-wiki.de/en/grafcet-studio/documenting-a-project/

11. Ovsiannikova P., Pakonen A., Muromsky D., Kobzev M., Dubinin V., Vyatkin V. Formal verification of non-functional requirements of overall instrumentation and control architectures // IEEE Open Journal of the Industrial Electronics Society. 2023 (submitted).

12. Документация SPARQL. [Электронный ресурс]: Режим доступа: https://www.w3.org/TR/rdf-sparql-query/

13. Ontology for assessing defence-in-depth properties of nuclear overall I&C architectures. [Электронный ресурс]: Режим доступа: https://zenodo.org/record/5010644#.Yg96EuhBwuU

14. Муромский Д.А. Кобзев М.И. Условные операторы для SPARQL запросов к RDF-хранилищам данных // Сб. ст. X Ежегодн. всеросс. межвуз. науч.-практ. конф. «Информационные технологии в науке и образовании. Проблемы и перспективы (ВМНПК – 2023)», Пенза, 2023. – с. 310-313.

15. Документация Qt. [Электронный ресурс]: Режим доступа: https://doc.qt.io

16. Документация Apache Jena Fuseki Server. [Электронный ресурс]: Режим доступа: https://jena.apache.org/documentation/fuseki2/

17. Документация Graphviz. [Электронный ресурс]: Режим доступа: https://graphviz.org/documentation/

Приложение А

(Обязательное)

Графический редактор языка запросов к rdf-хранилищам данных

Листинг программы

Файл MainWindow.cpp

```
#include "mainwindow.h"
#include "ui_mainwindow.h"

#include "projectwindow.h"
#include <QDebug>

#include <atomblocksettings.h>
#include <basicblockwindow.h>
#include <compositeblockwindow.h>
#include <diagramexecutor.h>
#include <fusekiserversetting.h>
#include <ioblocksettings.h>
#include <mainpage.h>
#include <sparqlblockwindow.h>

#define APPLICATION_NAME "FBQL Editor"

MainWindow::MainWindow( QWidget* parent )
    : QMainWindow( parent )
    , ui( new Ui::MainWindow )
{
    ui->setupUi( this );

    QCoreApplication::setApplicationName( APPLICATION_NAME );
    setWindowTitle( QCoreApplication::applicationName() );

QDir().mkdir( QStandardPaths::writableLocation( QStandardPaths::AppDataLocation ) );
    QDir().mkdir( QStandardPaths::writableLocation( QStandardPaths::DocumentsLocation )
+ "/" + APPLICATION_NAME );

    library = new BlocksLibrary();
    library->loadBlocksFromFiles( FOLDER_FOR_DEFAULT_BLOCKS );
    library->loadBlocksFromFiles( FOLDER_FOR_USERS_BLOCKS );
    library->addBlocks( AtomBlockSettings::GetBasicAtomBlocks() );
    library->addBlocks( IOBlockSettings::GetBasicIOBlocks() );

    createMainForm();
    slotOnOpenMainPage();

    open_save_processor = new SOpenSaveProcessor( item_menu, this );
    connect(  open_save_processor,  SIGNAL(  widgetCreate(  SWidget*,  QString  )  ),
tab_widget, SLOT( addWidget( SWidget*, QString ) ) );
}

MainWindow::~MainWindow()
{
    delete ui;
}

void MainWindow::createMainForm()
{
    menu_bar = new SMenuBar( this );
    setMenuBar( menu_bar );

    QWidget* widget = new QWidget( this );
    main_layout = new QGridLayout( widget );
    widget->setLayout( main_layout );

    tab_widget = new STabWidget( this );
    connect( tab_widget, SIGNAL( newCurrentMode( int ) ),
        this, SLOT( slotCurrentTabMode( int ) ) );
```

```
    tool_box = new SToolBox( this );
    connect( tool_box, SIGNAL( itemPressed( DiagramItemSettings* ) ),
        tab_widget, SLOT( itemForAdd( DiagramItemSettings* ) ) );

    tool_bar = new SToolBar( *menu_bar, this );
    addToolBar( Qt::TopToolBarArea, tool_bar );

    main_layout->addWidget( tool_box, 0, 0 );
    main_layout->addWidget( tab_widget, 0, 1 );

    setCentralWidget( widget );
}

void MainWindow::slotOnOpenMainPage()
{
    auto window = new MainPage( this );
    tab_widget->addWidget( window, tr( "Main Page" ) );
}

void MainWindow::slotCreateNewProject()
{
    auto window = new ProjectWindow( item_menu, this );
    tab_widget->addWidget( window, tr( "New Project" ) );
}

void MainWindow::slotCreateBasicBlock()
{
    auto window = new BasicBlockWindow( BasicBlockWindow::CreateMode, tab_widget );
    tab_widget->addWidget( window, tr( "New Basic" ) );
    connect(   window,   SIGNAL(   blockCreated(   DiagramItemSettings*   )   ),   this,
SLOT( slotOnCreateBlock( DiagramItemSettings* ) ) );
}

void MainWindow::slotCreateCompositeBlock()
{
    auto window = new CompositeBlockWindow( item_menu, this );
    tab_widget->addWidget( window, tr( "New Composite" ) );
    connect(   window,   SIGNAL(   blockCreated(   DiagramItemSettings*   )   ),   this,
SLOT( slotOnCreateBlock( DiagramItemSettings* ) ) );
}

void MainWindow::slotCreateSparqlBlock()
{
    auto window = new SparqlBlockWindow( item_menu, this );
    tab_widget->addWidget( window, tr( "New Sparql" ) );
    connect(   window,   SIGNAL(   blockCreated(   DiagramItemSettings*   )   ),   this,
SLOT( slotOnCreateBlock( DiagramItemSettings* ) ) );
}

void MainWindow::slotOnCreateBlock( DiagramItemSettings* settings )
{
    library->addBlock( settings );
    tab_widget->resetBlocks();
}

void MainWindow::slotCurrentTabMode( int mode )
{
    tool_box->setDiagramItems( library->getBlocks( mode ) );
}

void MainWindow::slotOnClickedBlockExecutor()
{
```

```
    auto list_blocks = tab_widget->getBlocksForRun();
    if ( !list_blocks.empty() )
    {
        auto window = new DiagramExecutor( this );
        window->setDiagramItem( list_blocks );
        tab_widget->addWidget( window, "Exec" );
    }
}

void MainWindow::setItemMenu( QMenu* menu )
{
    item_menu = menu;
}

void MainWindow::slotOnDeleteItemOnScene()
{
    tab_widget->deleteItemOnScene();
}

void MainWindow::slotOnOpen()
{
    open_save_processor->openFBAndCreateWidget();
}

void MainWindow::slotOnSave()
{
    tab_widget->callSaveCurrentWidget();
}

void MainWindow::slotOnSettingFusekiServer()
{
    auto window = new FusekiServerSetting( this );
    tab_widget->addWidget( window, tr( "Fuseki" ) );
}
```

Файл SGraphicsView.cpp

```
#include "sgraphicsview.h"

#include <QFile>
#include <QFileDialog>
#include <QLabel>
#include <QMessageBox>

SGraphicsView::SGraphicsView( QMenu* context_menu, QWidget* parent )
    : SWidget( parent )
    , context_menu( context_menu )
{
    setAttribute( Qt::WA_DeleteOnClose );
    createDiagramView();
}

void SGraphicsView::createDiagramView()
{
    grid_layout = new QGridLayout( this );
    setLayout( grid_layout );

    graphics_view = new QGraphicsView( this );
    group_box = new QGroupBox( this );
    group_box->setMaximumWidth( 220 );

    stacked_widget = new QStackedWidget( this );
    stacked_widget->insertWidget( stacked_widget->count(), graphics_view );
```

```
    grid_layout->addWidget( stacked_widget, 0, 0 );
    grid_layout->addWidget( group_box, 0, 1 );

    setDiagramScene( createDiagramScene() );
}

QStackedWidget* SGraphicsView::getStackedWidget()
{
    return stacked_widget;
}

void SGraphicsView::setDiagramScene( DiagramScene* scene )
{
    diagram_scene = scene;
    scene->addRect( 0, 0, width_scene, heigth_scene );
    graphics_view->setScene( diagram_scene );
}

DiagramScene* SGraphicsView::createDiagramScene()
{
    auto scene = new DiagramScene( graphics_view );
    scene->setContextMenu( context_menu );
    return scene;
}

void SGraphicsView::createSidePanel()
{
    QLabel* label = new QLabel( "Pos", group_box );

    x_spin_box = new QSpinBox( group_box );
    x_spin_box->setMaximum( width_scene );
    x_spin_box->setMinimum( 0 );
    connect( x_spin_box, SIGNAL( valueChanged( int ) ),
        this, SLOT( updatePosItems() ) );

    y_spin_box = new QSpinBox( group_box );
    y_spin_box->setMaximum( width_scene );
    y_spin_box->setMinimum( 0 );
    connect( y_spin_box, SIGNAL( valueChanged( int ) ),
        this, SLOT( updatePosItems() ) );

    QVBoxLayout* layout = new QVBoxLayout();
    layout->addWidget( label );
    layout->addWidget( x_spin_box );
    layout->addWidget( y_spin_box );

    group_box_layout = new QVBoxLayout( group_box );
    group_box_layout->addLayout( layout, 0 );
    group_box_layout->addWidget( addCustomWidget(), 1 );
    group_box_layout->addSpacerItem( new QSpacerItem( 0, 3000, QSizePolicy::Ignored,
QSizePolicy::Ignored ) );
    group_box_layout->addWidget( addCustomBotWidget() );

    group_box->setLayout( group_box_layout );
}

void SGraphicsView::updatePosData()
{
}

void SGraphicsView::updatePosItems()
{
```

```
}

DiagramScene* SGraphicsView::getScene()
{
    return diagram_scene;
}

void SGraphicsView::setItemForScene( DiagramItemSettings* settings )
{
    getScene()->setMode( DiagramScene::InsertItem );
    getScene()->setDiagramItemForInserted( settings );
}

void SGraphicsView::setSceneMode( int mode )
{
    getScene()->setMode( static_cast<DiagramScene::SceneMode>( mode ) );
}

void SGraphicsView::removeSelectedGraphicsItems()
{
    getScene()->removeSelectedItems();
}
```

Файл DiagramScene.cpp

```
#include "diagramscene.h"

DiagramScene::DiagramScene( QObject* parent )
    : QGraphicsScene( parent )
{
    scene_mode = MoveItem;
    line_for_arrow = 0;
}

void DiagramScene::setContextMenu( QMenu* context_menu )
{
    my_context_menu = context_menu;
}

QMenu* DiagramScene::getContextMenu()
{
    return my_context_menu;
}

void DiagramScene::setMode( SceneMode mode )
{
    scene_mode = mode;
}

void DiagramScene::setDiagramItemForInserted( DiagramItemSettings* settings )
{
    item_for_insert = settings;
}

DiagramArrow* DiagramScene::createArrow( DiagramItem* startItem, DiagramItem* endItem )
{
    DiagramArrow* arrow;
    // arrow->setColor(myLineColor);
    if ( nullptr == startItem->parentItem()
        || nullptr == endItem->parentItem()
        || startItem->parentItem() != endItem->parentItem() )
    {
        arrow = new DiagramArrow( startItem, endItem );
```

```
            addItem( arrow );
        }
        else
        {
            arrow = new DiagramArrow( startItem, endItem, startItem->parentItem() );
        }
        startItem->addArrow( arrow );
        endItem->addArrow( arrow );
        arrow->setZValue( qMin( startItem->zValue(), endItem->zValue() ) - 1 );
        arrow->updatePosition();
        return arrow;
    }

    void DiagramScene::mousePressEvent( QGraphicsSceneMouseEvent* mouseEvent )
    {
        if ( mouseEvent->button() == Qt::LeftButton )
        {
            DiagramItem* item;
            auto items_for_add = items( mouseEvent->scenePos() );
            bool flag_add = false;
            switch ( scene_mode )
            {
            case InsertItem:
                for ( auto& parent_item : items_for_add )
                {
                    if ( DiagramItem::CheckItemOnDiagramItem( parent_item->type() )
                        && static_cast<DiagramItem*>( parent_item )->getSupportAddItem() )
                    {
                        item    =    DiagramItem::FactoryDiagramItem(    my_context_menu,
    item_for_insert, parent_item );
                        flag_add = true;
                        item->setPos( mouseEvent->scenePos() - parent_item->pos() );
                        break;
                    }
                }
                if ( !flag_add )
                {
                    item     =     DiagramItem::FactoryDiagramItem(     my_context_menu,
    item_for_insert );
                    addItem( item );
                    if ( DiagramItem::CompositeItemType == item->type() )
                    {
                        auto arrows = item->getArrows();
                        for ( auto& arrow : arrows )
                        {
                            addItem( arrow );
                            arrow->updatePosition();
                        }
                    };
                    item->setPos( mouseEvent->scenePos() );
                }
                setMode( MoveItem );
                // emit itemSelected( item );
                break;
            case InsertArrow:
                line_for_arrow = new QGraphicsLineItem( QLineF( mouseEvent->scenePos(),
                    mouseEvent->scenePos() ) );
                line_for_arrow->setPen( QPen( Qt::black, 2 ) );
                line_for_arrow->setZValue( 1000 );
                addItem( line_for_arrow );
                break;
            case MoveItem:
                if ( resize_item != nullptr )
```

```
            {
                setMode( Resize );
            }
            break;
        default:;
        }
    }
    else if ( mouseEvent->button() == Qt::RightButton )
    {
        line_for_arrow = new QGraphicsLineItem( QLineF( mouseEvent->scenePos(),
            mouseEvent->scenePos() ) );
        line_for_arrow->setPen( QPen( Qt::black, 2 ) );
        line_for_arrow->setZValue( 1000 );
        addItem( line_for_arrow );
        setMode( InsertArrow );
    }
    /*
        foreach ( QGraphicsItem* temp, this->selectedItems() )
        {
            if ( DiagramItem::CheckItemOnDiagramItem( temp->type() ) )
            {
                // emit itemSelected( static_cast<DiagramItem*>( temp ) );
                break;
            }
        }
    */
    QGraphicsScene::mousePressEvent( mouseEvent );
}

void DiagramScene::mouseMoveEvent( QGraphicsSceneMouseEvent* mouseEvent )
{

    if ( InsertArrow == scene_mode && 0 != line_for_arrow )
    {
        QLineF new_line( line_for_arrow->line().p1(), mouseEvent->scenePos() );
        line_for_arrow->setLine( new_line );
    }
    else if ( MoveItem == scene_mode )
    {
        QApplication::restoreOverrideCursor();
        resize_item = nullptr;
        auto selected_items = items();
        for ( auto item : selected_items )
        {
            if ( DiagramItem::CheckItemOnDiagramItem( item->type() ) )
            {

                auto diagram_item = static_cast<DiagramItem*>( item );
                if ( diagram_item->getAllowResize() )
                {
                    auto right_diff = mouseEvent->scenePos().x() - ( diagram_item-
>pos().x() + diagram_item->getEndPos().x() * diagram_item->scale() );
                    auto bottom_diff = mouseEvent->scenePos().y() - ( diagram_item-
>pos().y() + diagram_item->getEndPos().y() * diagram_item->scale() );
                    auto offset = 5;
                    if ( qAbs( right_diff ) <= offset )
                    {
                        if ( qAbs( bottom_diff ) <= offset )
                        {
                            QApplication::setOverrideCursor( Qt::SizeFDiagCursor );
                            resize_item = diagram_item;
                        }
                    }
```

```
                }
            }
        }
        auto blocks = items();
        for ( auto block : blocks )
        {
            if ( DiagramItem::CheckItemOnDiagramItem( block->type() ) )
            {
                static_cast<DiagramItem*>( block )->setPen( QPen( Qt::black, 1 ) );
                static_cast<DiagramItem*>( block )->setColorArrows( Qt::black );
            }
        }

        blocks = items( mouseEvent->scenePos() );
        if ( !blocks.empty() )
        {
            for ( auto block : blocks )
            {
                if ( DiagramItem::CheckItemOnDiagramItem( block->type() ) )
                {
                    auto    dep_items    =    static_cast<DiagramItem*>(    block    )-
>getDependecies();
                    for ( auto pair : dep_items )
                    {
                        pair.first->setPen( QPen( Qt::red, 5 ) );
                        pair.first->setColorArrowToItem( Qt::red, pair.second );
                    }
                }
            }
        }

        QGraphicsScene::mouseMoveEvent( mouseEvent );
    }
    else if ( scene_mode == Resize )
    {
        double diff_x = ( mouseEvent->scenePos().x() - ( resize_item->getStartPos().x()
*   resize_item->scale()   +   resize_item->pos().x()   )   )   /   (   resize_item-
>boundingRect().width() );
        resize_item->setScale( diff_x );
    }
}

QGraphicsItem* DiagramScene::getParentItem( QGraphicsItem* item )
{
    auto temp = item;
    while ( true )
    {
        auto parent = temp->parentItem();
        if ( nullptr != parent
            && DiagramItem::CheckItemOnDiagramItem( parent->type() )
            && !static_cast<DiagramItem*>( parent )->getAllowLineToChild()
            && !static_cast<DiagramItem*>( parent )->getSupportAddItem() )
        {
            temp = parent;
        }
        else
        {
            break;
        }
    }
    return temp;
}
```

```
void DiagramScene::mouseReleaseEvent( QGraphicsSceneMouseEvent* mouseEvent )
{
    if ( 0 != line_for_arrow && InsertArrow == scene_mode )
    {
        QList<QGraphicsItem*> startItems = items( line_for_arrow->line().p1() );
        if ( startItems.count() && startItems.first() == line_for_arrow )
            startItems.removeFirst();
        QList<QGraphicsItem*> endItems = items( line_for_arrow->line().p2() );
        if ( endItems.count() && endItems.first() == line_for_arrow )
            endItems.removeFirst();

        removeItem( line_for_arrow );
        delete line_for_arrow;

        if ( !startItems.empty() && !endItems.empty() )
        {
            auto start = startItems.first();
            auto end = endItems.first();

            start = getParentItem( start );
            end = getParentItem( end );

            if ( DiagramItem::CheckItemOnDiagramItem( start->type() )
                && DiagramItem::CheckItemOnDiagramItem( end->type() ) )
            {
                if ( start == end && nullptr != start )
                {
                    static_cast<DiagramItem*>( start )->contextMenuShow( mouseEvent );
                }
                else
                {
                    createArrow( static_cast<DiagramItem*>( start ),
                        static_cast<DiagramItem*>( end ) );
                }
            }
        }
    }
    line_for_arrow = 0;
    setMode( MoveItem );
    resize_item = nullptr;
    QGraphicsScene::mouseReleaseEvent( mouseEvent );
}

void DiagramScene::mouseDoubleClickEvent( QGraphicsSceneMouseEvent* mouseEvent )
{
    QGraphicsScene::mouseDoubleClickEvent( mouseEvent );
}

void DiagramScene::removeSelectedItems()
{
    auto selected_items_list = selectedItems();
    for ( auto item : selected_items_list )
    {
        removeItemSmart( item );
    }
}

void DiagramScene::removeItemSmart( QGraphicsItem* item )
{
    removeItem( item );
    delete item;
}
```

Файл DiagramItem.cpp

```
#include "diagramitem.h"

#include <QGraphicsScene>
#include <QGraphicsSceneContextMenuEvent>
#include <QMenu>
#include <QPainter>
#include <QtMath>

#include <diagramarrow.h>

DiagramItem::DiagramItem( QMenu* contextMenu, QGraphicsItem* parent )
    : QGraphicsPolygonItem( parent )
{
    my_context_menu = contextMenu;

    my_polygon << QPointF( -50, -50 ) << QPointF( 50, -50 )
               << QPointF( 50, 50 ) << QPointF( -50, 50 )
               << QPointF( -50, -50 );

    setItemPolygon( my_polygon );
    setFlag( QGraphicsItem::ItemIsMovable, true );
    setFlag( QGraphicsItem::ItemIsSelectable, true );
    setFlag( QGraphicsItem::ItemSendsGeometryChanges, true );
    setSupportAddItem( false );
    setAllowLineToChild( false );
}

DiagramItem::~DiagramItem()
{
    removeArrows();
}

void DiagramItem::setOutputText( const QString& output_text )
{
    auto arrows = getStartArrows();
    for ( auto arrow : arrows )
    {
        arrow->setText( output_text );
        // arrow->setVisibleText( false );
    }
}

DiagramItemSettings* DiagramItem::getSettings()
{
    return {};
}

void DiagramItem::setSupportAddItem( bool flag )
{
    support_add_item = flag;
}

bool DiagramItem::getSupportAddItem()
{
    return support_add_item;
}

void DiagramItem::setAllowLineToChild( bool flag )
{
    allow_line_to_child = flag;
}
```

```
bool DiagramItem::getAllowLineToChild()
{
    return allow_line_to_child;
}

void DiagramItem::setContextMenu( QMenu* contextMenu )
{
    my_context_menu = contextMenu;
}

QMenu* DiagramItem::getContextMenu()
{
    return my_context_menu;
}

void DiagramItem::setItemPolygon( QPolygonF& polygon )
{
    my_polygon = polygon;
    setPolygon( my_polygon );
}

QPointF DiagramItem::getStartPos()
{
    if ( my_polygon.size() == 0 )
        return QPointF();
    QPointF min = my_polygon[0];
    foreach ( QPointF temp, my_polygon )
    {
        if ( min.x() > temp.x() )
        {
            min.setX( temp.x() );
        }
        if ( min.y() > temp.y() )
        {
            min.setY( temp.y() );
        }
    }
    return min;
}

QPointF DiagramItem::getEndPos()
{
    if ( my_polygon.size() == 0 )
        return QPointF();
    QPointF max = my_polygon[0];
    foreach ( QPointF temp, my_polygon )
    {
        if ( max.x() < temp.x() )
        {
            max.setX( temp.x() );
        }
        if ( max.y() < temp.y() )
        {
            max.setY( temp.y() );
        }
    }
    return max;
}

void DiagramItem::removeArrow( DiagramArrow* arrow )
{
    auto index = arrows.indexOf( arrow );
```

```
    if ( -1 != index )
    {
        arrows.removeAt( index );
    }
}

void DiagramItem::removeArrows()
{
    foreach ( DiagramArrow* arrow, arrows )
    {
        delete arrow;
    }
}

void DiagramItem::addArrow( DiagramArrow* arrow )
{
    arrows.append( arrow );
}

void DiagramItem::setAllowResize( bool flag )
{
    allow_resize = flag;
}

bool DiagramItem::getAllowResize()
{
    return allow_resize;
}

void DiagramItem::mousePressEvent( QGraphicsSceneMouseEvent* mouseEvent )
{
    if ( mouseEvent->button() == Qt::LeftButton )
    {
        if ( mouseEvent->pos().x() >= getEndPos().x() - 5 )
        {
            QApplication::setOverrideCursor( Qt::SizeFDiagCursor );
        }
        else
        {
            QApplication::restoreOverrideCursor();
        }
    }
    QGraphicsPolygonItem::mousePressEvent( mouseEvent );
}

void DiagramItem::contextMenuShow( QGraphicsSceneMouseEvent* mouseEvent )
{
    scene()->clearSelection();
    setSelected( true );
    if ( nullptr != my_context_menu )
        my_context_menu->exec( mouseEvent->screenPos() );
}

QVariant DiagramItem::itemChange( GraphicsItemChange change, const QVariant& value )
{
    if ( change == QGraphicsItem::ItemPositionChange )
    {
        foreach ( DiagramArrow* arrow, arrows )
        {
            arrow->updatePosition();
        }
    }
```

```
    return value;
}

QVector<DiagramArrow*> DiagramItem::getArrows()
{
    return arrows;
}

QVector<QPair<DiagramItem*, DiagramItem*>> DiagramItem::getDependecies()
{
    /*if ( dependencies.empty() )
    {
        QVector<QPair<DiagramItem*, DiagramItem*>> data;
        auto arrows = getEndArrows();
        for ( auto arrow : arrows )
        {
            data.push_back( { arrow->startItem(), this } );
        }
        return data;
    }*/
    return dependencies;
}

void DiagramItem::setColorArrowToItem( const QColor& color, DiagramItem* item )
{
    auto arrows = getStartArrows();
    for ( auto arrow : arrows )
    {
        if ( arrow->endItem() == item )
        {
            arrow->setColor( color );
            arrow->updatePosition();
        }
    }
}

void DiagramItem::setColorArrows( const QColor& color )
{
    for ( auto arrow : arrows )
    {
        arrow->setColor( color );
        arrow->updatePosition();
    }
}

void DiagramItem::addDependecies( DiagramItem* start, DiagramItem* end )
{
    if ( nullptr != start && nullptr != end )
    {
        dependencies.push_back( { start, end } );
    }
}

void DiagramItem::clearDependecies()
{
    dependencies.clear();
}

void DiagramItem::setDependecies( QVector<QPair<DiagramItem*, DiagramItem*>> data )
{
    if ( data.empty() )
    {
        auto arrows = getEndArrows();
```

```
        for ( auto arrow : arrows )
        {
            data.push_back( { arrow->startItem(), this } );
        }
    }
    dependencies = data;
}

bool DiagramItem::CheckItemOnDiagramItem( const qint64 code )
{
    if ( DiagramItem::BasicItemType == code
        || DiagramItem::CompositeItemType == code
        || DiagramItem::SparqlItemType == code
        || DiagramItem::AtomItemType == code
        || DiagramItem::IOItemType == code )
    {
        return true;
    }
    return false;
}

#include <diagramitematom.h>
#include <diagramitembasic.h>
#include <diagramitemcomposite.h>
#include <diagramitemio.h>
#include <diagramitemsparql.h>

DiagramItem* DiagramItem::FactoryDiagramItem( QMenu* context_menu,
    DiagramItemSettings* settings, QGraphicsItem* parent )
{
    switch ( settings->typeSettings() )
    {
    case DiagramItemSettings::BasicItemSettingsType:
        return new DiagramItemBasic( context_menu, parent,
            new BasicBlockSettings( *static_cast<BasicBlockSettings*>( settings ) ) );
        break;
    case DiagramItemSettings::CompositeItemSettingsType:
        return new DiagramItemComposite( context_menu, parent,
            new
CompositeBlockSettings( *static_cast<CompositeBlockSettings*>( settings ) ) );
        break;
    case DiagramItemSettings::SparqlItemSettinsType:
        return new DiagramItemSparql( context_menu, parent,
            new
SparqlBlockSettings( *static_cast<SparqlBlockSettings*>( settings ) ) );
        break;
    case DiagramItemSettings::AtomItemSettingsType:
        return new DiagramItemAtom( context_menu, parent,
            new AtomBlockSettings( *static_cast<AtomBlockSettings*>( settings ) ) );
        break;
    case DiagramItemSettings::IOItemSettingsType:
        return new DiagramItemIO( context_menu, parent,
            new IOBlockSettings( *static_cast<IOBlockSettings*>( settings ) ) );
        break;

    default:
        break;
    }
    return nullptr;
}

QVector<DiagramArrow*> DiagramItem::getStartArrows()
{
```

```
    QVector<DiagramArrow*> result;
    auto arrows = getArrows();
    for ( auto arrow : arrows )
    {
        if ( this == arrow->startItem() )
            result.push_back( arrow );
    }
    return result;
}

QVector<DiagramArrow*> DiagramItem::getEndArrows()
{
    QVector<DiagramArrow*> result;
    auto arrows = getArrows();
    for ( auto arrow : arrows )
    {
        if ( this == arrow->endItem() )
            result.push_back( arrow );
    }
    return result;
}
```

Файл SparqlBlockSettings.cpp

```
#include "sparqlblocksettings.h"

#include <QJsonArray>
#include <QJsonObject>
#include <QMap>
#include <QUrl>

SparqlBlockSettings::SparqlBlockSettings()
{
    block_name = "SparqlBlock";
    pixmap = image();
    limit = 20;
    use_query = false;
}

SparqlBlockSettings::~SparqlBlockSettings()
{
    clear();
}

bool SparqlBlockSettings::CheckSettings( const QJsonObject& object )
{
    if ( object["Header"]["Type"].toString() == "Sparql" )
    {
        return true;
    }
    return false;
}

void SparqlBlockSettings::setSettingFromJson( const QJsonObject& object )
{
    if ( CheckSettings( object ) )
    {
        areas.clear();
        lines.clear();

        QJsonValue header = object["Header"];
        QJsonValue body = object["Body"];
```

```
        block_name = header["Name"].toString();
        // type_image = value["type_img"].toString();
        // pixmap = pixmapFrom( body["Image"] );
        limit = body["Limit"].toInt();
        start_area = body["Start_Area"].toInt();
        pos = pointFromJsonObject( body["Pos"] );
        query = body["Query"].toString();
        use_query = body["Use_Query"].toBool();

        QJsonArray areas_array = body["Areas"].toArray();
        for ( const QJsonValue& area : areas_array )
        {
            auto blocks_array = area["Blocks"].toArray();
            auto lines_array = area["Lines"].toArray();
            auto settings = area["Settings"];

            AreaSaver area_saver;
            for ( const QJsonValue& block : blocks_array )
            {
                AtomBlockSettings* setting = new AtomBlockSettings();
                setting->setSettingFromJson( block.toObject() );
                area_saver.blocks.push_back( setting );
            }

            for ( const QJsonValue& line : lines_array )
            {
                area_saver.lines.push_back(          {          line["Start"].toInt(),
line["End"].toInt(), line["Text"].toString() } );
            }

            area_saver.settings = new AtomBlockSettings();
            area_saver.settings->setSettingFromJson( settings.toObject() );
            area_saver.filter = area["Filter"].toString();

            areas.push_back( area_saver );
        }

        QJsonArray lines_array = body["Lines"].toArray();
        for ( const QJsonValue& line : lines_array )
        {
            lines.push_back(     {     line["Start"].toInt(),     line["End"].toInt(),
line["Text"].toString() } );
        }
    }
}

QJsonArray  SparqlBlockSettings::getJsonArrayFromLineSaver(  const  QVector<LineSaver>&
lines )
{
    QJsonArray lines_array;
    for ( const auto& line : lines )
    {
        QJsonObject line_obj;
        line_obj.insert( "Text", line.text );
        line_obj.insert( "Start", line.start_block );
        line_obj.insert( "End", line.end_block );
        lines_array.push_back( line_obj );
    }
    return lines_array;
}

QJsonObject SparqlBlockSettings::getJsonFromSetting()
{
```

```
    QJsonObject json_object;

    QJsonObject header;
    QJsonObject body;

    header.insert( "Type", "Sparql" );
    header.insert( "Name", block_name );

    body.insert( "Limit", limit );
    body.insert( "Start_Area", start_area );
    body.insert( "Pos", jsonFromPointF( pos ) );
    body.insert( "Query", query );
    body.insert( "Use_Query", use_query );

    body.insert( "Lines", getJsonArrayFromLineSaver( lines ) );

    QJsonArray areas_array;
    for ( const auto& area : areas )
    {
        QJsonObject area_obj;
        area_obj.insert( "Filter", area.filter );
        area_obj.insert( "Settings", area.settings->getJsonFromSetting() );
        area_obj.insert( "Lines", getJsonArrayFromLineSaver( area.lines ) );
        QJsonArray blocks_array;
        for ( const auto& block : area.blocks )
        {
            blocks_array.push_back( block->getJsonFromSetting() );
        }
        area_obj.insert( "Blocks", blocks_array );
        areas_array.push_back( area_obj );
    }
    body.insert( "Areas", areas_array );

    json_object.insert( "Header", header );
    json_object.insert( "Body", body );

    return json_object;
}

QString SparqlBlockSettings::getQuery()
{
    if ( use_query )
        return query;

    QString request;
    MainSettingsMaster master( "SPARQL" );
    auto prefixes = master.getSettings<QStringList>( "prefixes" );
    for ( auto& prefix : prefixes )
    {
        request += prefix + "\n";
    }
    request += "\nSELECT *";

    QString body = "";
    QMap<int, QString> stack = { { start_area, "" } };
    while ( !stack.empty() )
    {
        int index = stack.keys().at( 0 );
        QString name = stack[index];
        AreaSaver area = areas.at( index );
        stack.remove( index );

        QString area_str = name + "\n{\n";
```

```
        for ( const auto& line : area.lines )
        {
            QString str = ( ( DEFAULT_VAR == area.blocks.at( line.start_block )-
>type_block ) ? "?" : "" )
                + area.blocks.at( line.start_block )->text;
            str += " " + line.text;
            str += " "
                + QString( ( DEFAULT_VAR == area.blocks.at( line.end_block )-
>type_block ) ? "?" : "" )
                + area.blocks.at( line.end_block )->text;
            str += ".\n";
            area_str += str;
        }
        if ( !area.filter.isEmpty() )
        {
            area_str += "FILTER ( " + area.filter + ")\n";
        }
        area_str += "}\n";
        body += area_str;

        for ( const auto& line : lines )
        {
            if ( line.start_block == index )
            {
                stack.insert( line.end_block, line.text );
            }
        }
    }

    request += "\nWHERE\n"
               "{\n";
    request += "  " + body;
    request += "} LIMIT " + QString::number( limit );
    return request;
}

QString SparqlBlockSettings::getScript()
{
    MainSettingsMaster master( "Fuseki" );
    auto dataset_name = master.getSettings<QString>( "dataset_name" );
    auto fuseki_ip = master.getSettings<QString>( "ip" );
    auto fuseki_port = master.getSettings<QString>( "port" );

    if ( dataset_name.isEmpty() )
    {
        qDebug() << "Dataset is Empty";
    }

    QString script = "var xmlHttp = new XMLHttpRequest(network);\n"
                     "xmlHttp.setUrl(\"http://"
        + fuseki_ip + ":"
        + fuseki_port + "/"
        + dataset_name + "/query\");\n"
                         "xmlHttp.open(\"POST\", \"/\");\n"
                         "xmlHttp.setRequestHeader(\"Connection\", \"keep-alive\");\n"
                         "xmlHttp.setRequestHeader(\"Accept\",    \"application/sparql-
results+json\");\n"
                         "var answer = xmlHttp.send(";

    script += "\"query=" + QUrl::toPercentEncoding( query.isEmpty() ? getQuery() :
query ) + "\");\n";
    script += "json_obj = JSON.parse( answer );\n"
              "array = json_obj.results.bindings;\n"
```

```
                "output = array.length;";

    return script;
}

QPixmap SparqlBlockSettings::image() const
{
    return QPixmap( ":/Sources/images/sparqlicon.jpg" );
}

SparqlBlockSettings* SparqlBlockSettings::CreateTemplateSparqlSettings()
{
    auto settings = new SparqlBlockSettings();
    AreaSaver area_saver;
    area_saver.settings = AtomBlockSettings::GetAreaAtomBlock();
    area_saver.settings->pos = QPointF( 2500, 2500 );
    settings->areas.push_back( area_saver );
    return settings;
}

void SparqlBlockSettings::clear()
{
    for ( auto& area : areas )
    {
        /*if ( area.settings != nullptr )
            delete area.settings;
        for ( auto& block : area.blocks )
        {
            if ( block != nullptr )
                delete block;
        }*/
        area.blocks.clear();
    }
    areas.clear();
}
```

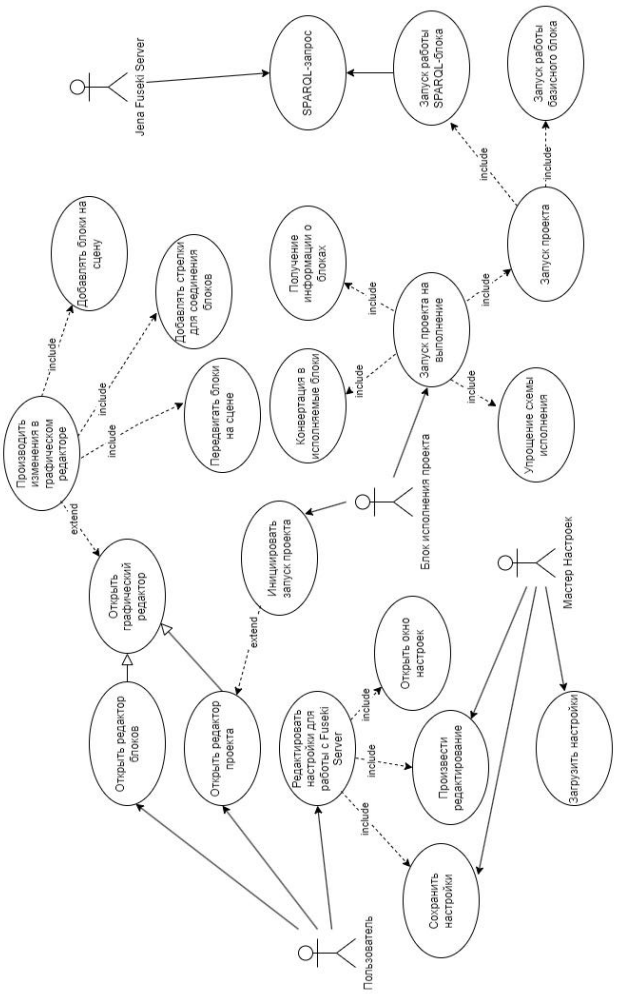
Приложение Б

(Обязательное)

Графический редактор языка запросов к rdf-хранилищам данных

UML-диаграммы

Приложение Б.1 Диаграмма прецедентов



ПГУ1.09.03.01.15.002 Д13

Изм.	Лист	№ докум.	Подпись	Дата
Разраб.		Мирошников Н.А.		
Провер.		Мирошников Н.А.		
Т. контр.				
Реценз.		Антонов А.С.		
Н. контр.		Мирошников Н.А.		
Утверд.				

Графический редактор языка запросов к RDF-совместимым данным

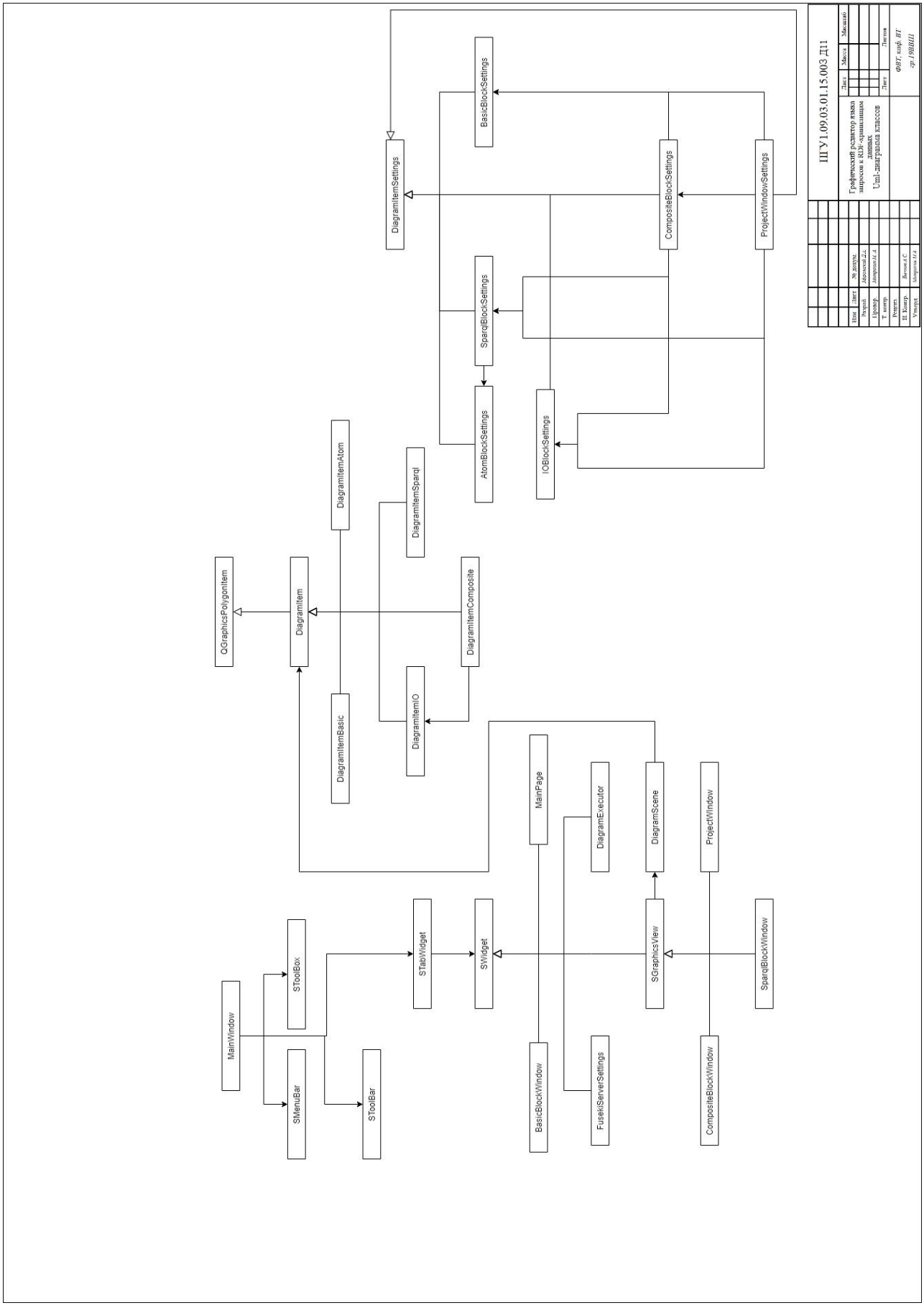
UML-диаграмма прецедентов

Лит.	Масса	Масштаб

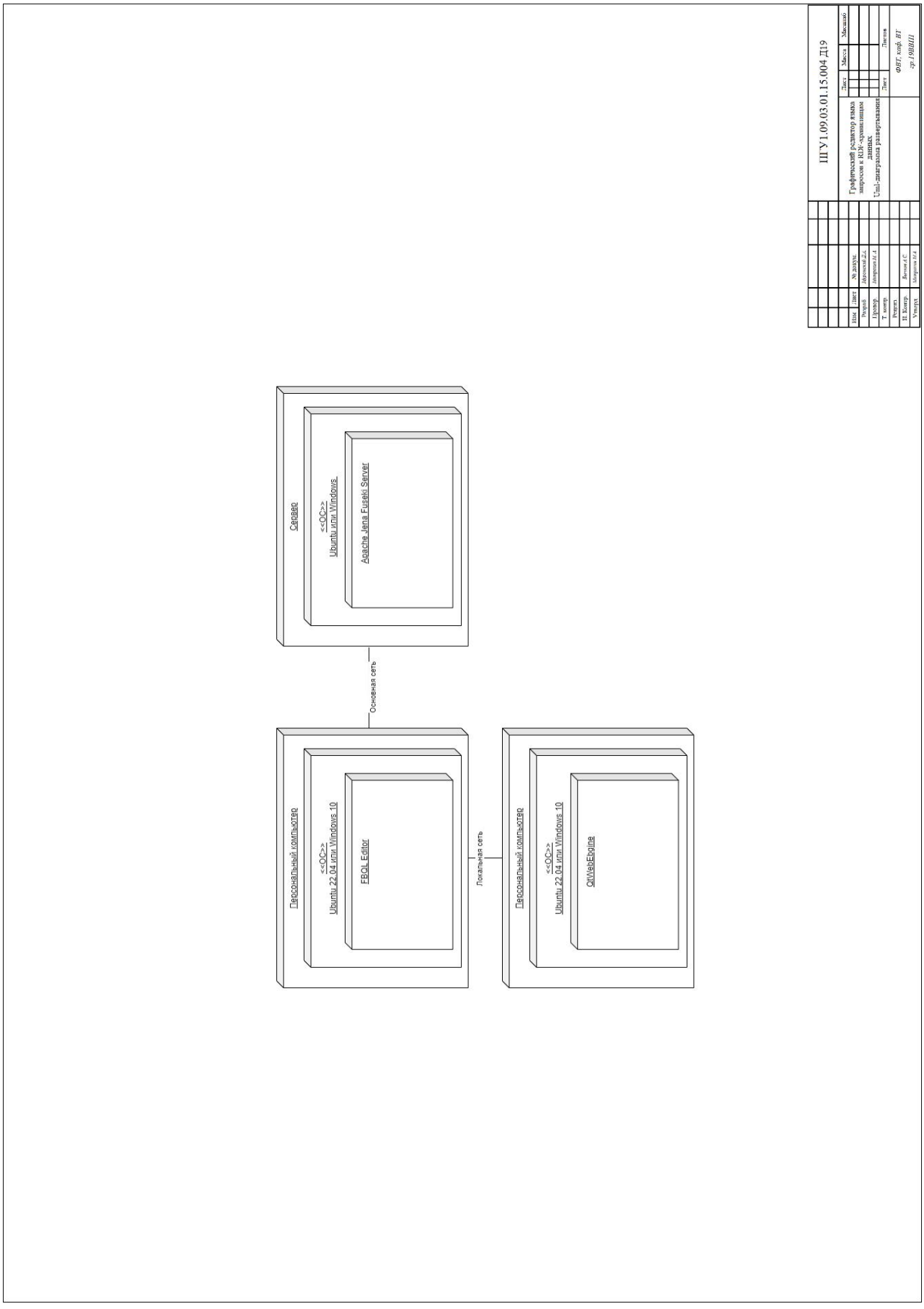
Лист	Листов
	1

ФВТ, гр. ВТ гр. 19ВВП1

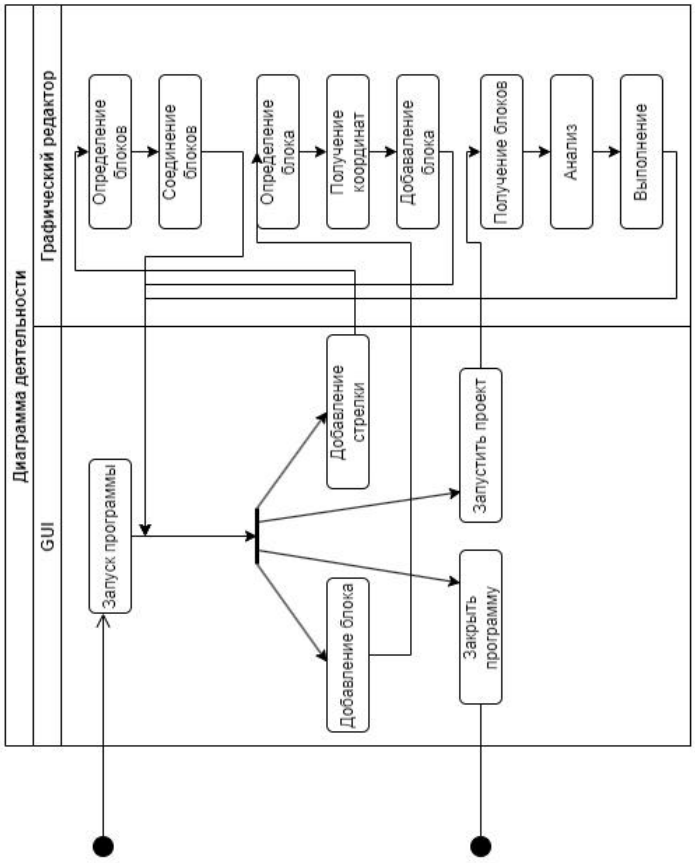
Приложение Б.2 Диаграмма классов



Приложение Б.3 Диаграмма развертывания



Приложение Б.4 Диаграмма деятельности



Приложение В

(Обязательное)

Графический редактор языка запросов к rdf-хранилищам данных

Презентация

ГРАФИЧЕСКИЙ РЕДАКТОР ЯЗЫКА ЗАПРОСОВ К RDF-ХРАНИЛИЩАМ ДАННЫХ

Автор работы:

студент гр. 19ВВП1 Муромский Дмитрий Александрович

Научный руководитель:

д.т.н., профессор кафедры ВТ Дубинин Виктор Николаевич

Пенза 2023

Цель работы

Цель исследования, в рамках которого выполнялась данная работа – разработка математического, лингвистического и программного обеспечения для поддержки проектирования специализированных информационно-вычислительных систем, предназначенных для оценки объектов энергетической инфраструктуры, на основе функционально-блочных диаграмм и RDF-хранилищ данных.

Цель данной работы – разработка графического языка функционально-блочных запросно-вычислительных диаграмм, включающих базисные, композитные и SPARQL-блоки, а также программная реализация графического редактора, позволяющего создавать, редактировать и сохранять диаграммы данного вида.

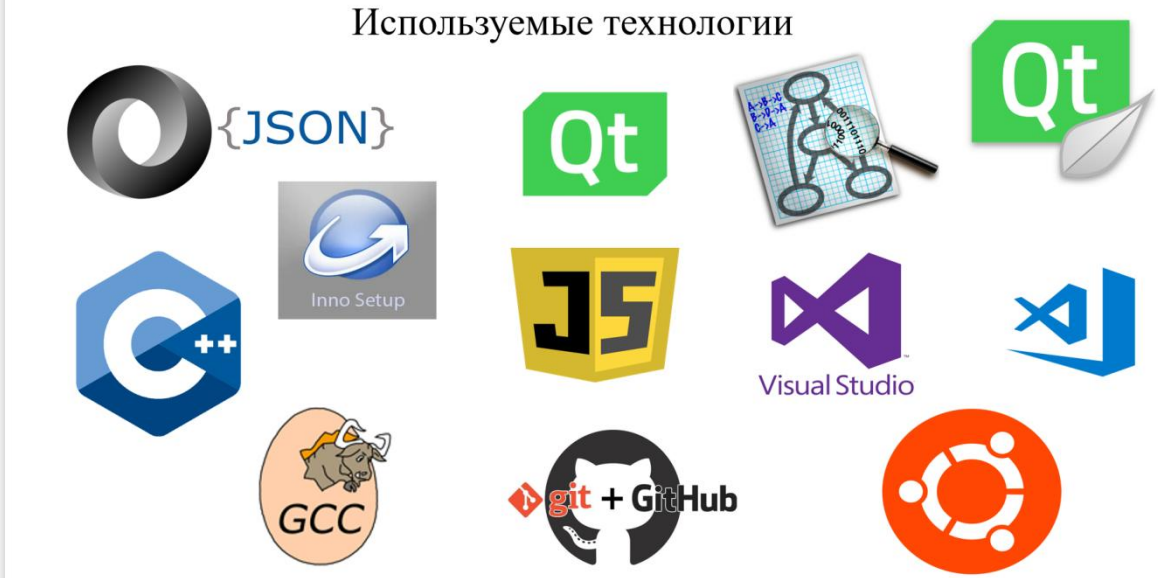
Задачи, решаемые в процессе выполнения проекта

1. Разработка конкретного синтаксиса графического языка функционально-блочных запросно-вычислительных диаграмм, включая подъязык графического описания SPARQL-запросов;

2. Разработка и программная реализация графического редактора функционально-блочных запросно-вычислительных диаграмм;

Области применения:

- Спецификация (формализованное описание) на языке функционально-блочных запросно-вычислительных диаграмм требований к техническим системам, описанным с использованием RDF-онтологий, выраженных в числовом виде;
- Проектирование, разработка и реализация специализированных информационно-вычислительных систем на основе функционально-блочных запросно-вычислительных диаграмм.



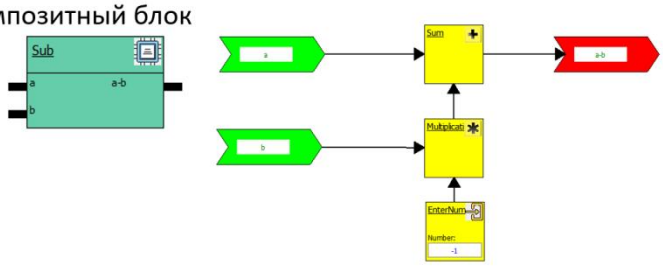
Графический язык функционально-блочных запросно-вычислительных диаграмм

Базисный блок

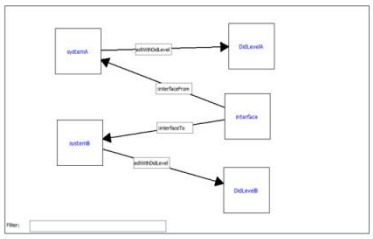
JavaScript

```
output = "false";
for(var i = 0; i < inputs.length; i++){
    if( inputs[i] == "true"){
        output = "true";
        break;
    }
}
```

Композитный блок

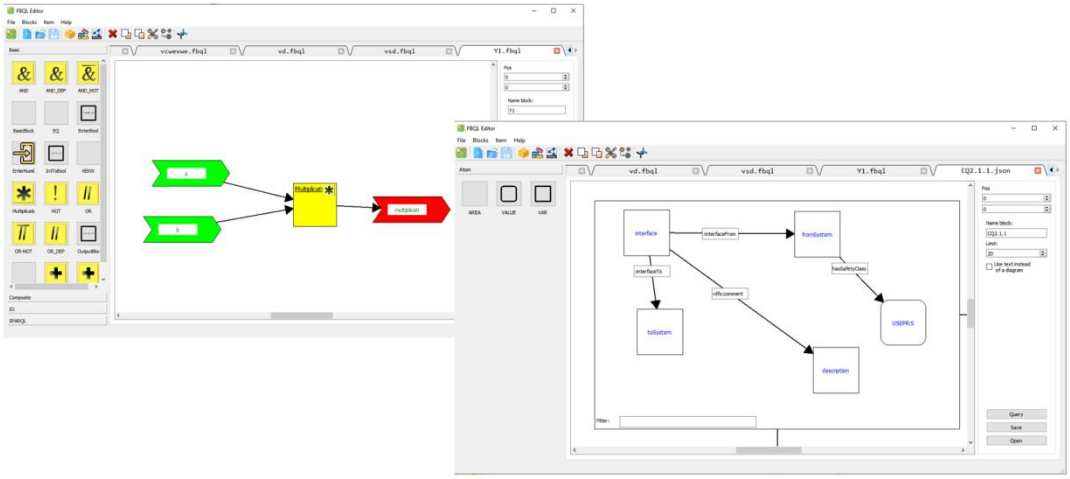


SPARQL-блок




```
SELECT ?systemA ?DidLevelA ?systemB
       ?DidLevelB ?interface
WHERE {
  ?systemA :associatedWithDidLevel ?DidLevelA.
  ?systemB :associatedWithDidLevel ?DidLevelB.
  ?interface :interfaceFrom ?systemA.
  ?interface :interfaceTo ?systemB.
  FILTER (?DidLevelA != ?DidLevelB)
}
```

Графический редактор (FBQL Editor)



Архитектура программы

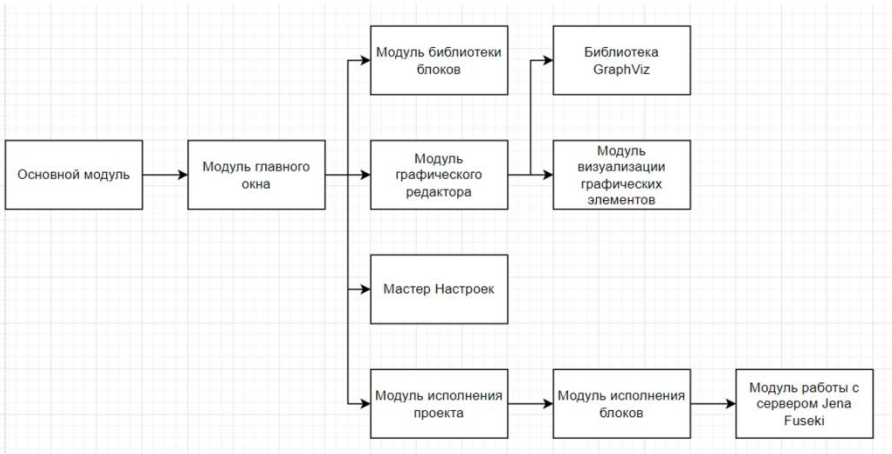


Диаграмма вариантов использования

Диаграммы классов

- ApiJS
- AtomBlockSettings
- BasicBlockSettings
- BasicBlockWindow
- BlocksExec
- BlocksLibrary
- CompositeBlockSettings
- CompositeBlockWindow
- DiagramArrow
- DiagramExecutor
- DiagramItem
- DiagramItemAtom
- DiagramItemBasic
- DiagramItemComposite
- DiagramItemIO
- DiagramItemSettings
- DiagramItemSparql
- DiagramScene
- Downloader
- FusekiServerSetting
- IOBlockSettings
- MainPage
- MainSettingsMaster
- MainWindow
- MyProcess
- ProjectWindow
- ProjectWindowSettings
- SGraphicsView
- SMenuBar
- SNetwork
- SOpenSaveProcessor
- SparqlBlockSettings
- SparqlBlockWindow
- STabWidget
- SToolBar
- SToolBox
- SWidget
- main (int, char *[])

Список разработанных классов

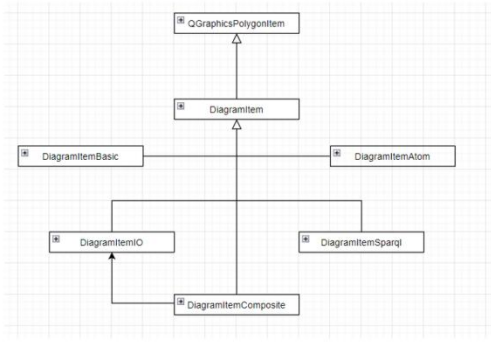


Диаграмма классов для отображения блоков

Результаты работы

- Разработан графический язык для описания и обработки SPARQL-запросов на основе функционально-блочных диаграмм;
- Редактор для базовых блоков с возможностью их создания и редактирования;
- Редактор для композитных блоков с возможностью создания в них сложных схем;
- Редактор для SPARQL-блоков, который предоставляет огромный функционал по созданию SPARQL-запросов как в виде диаграмм, так и в текстовом представлении;
- Редактор для создания функциональной схемы, которая может исполняться и выдавать результат своей работы;

Публикации по теме работы

Муромский Д.А. Кобзев М.И. Условные операторы для SPARQL запросов к RDF-хранилищам данных // Сб. ст. X Ежегодн. всеросс. межвуз. науч.-практ. конф. «Информационные технологии в науке и образовании. Проблемы и перспективы (ВМНПК – 2023)», Пенза, 2023. – с. 310-313.

Ovsiannikova P., Pakonen A., Muromsky D., Kobzev M., Dubinin V., Vyatkin V. Formal verification of non-functional requirements of overall instrumentation and control architectures // IEEE Open Journal of the Industrial Electronics Society. 2023 (submitted).

Спасибо за внимание!