**САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ**

Математико-Механический факультет

Кафедра Системного Программирования

# Реализация кроссплатформенной архитектуры CASE-пакета

Никандров Георгий

Научный руководитель ..... /А. Н. Терехов/
/подпись/

Рецензент,
аспирант ..... /Д. В. Луцив/
/подпись/

"Допустить к защите"
Заведующий кафедрой,
профессор,
д. ф.-м. н-к ..... /А.Н. Терехов/
/подпись/

---

Санкт-Петербург
2007

# Содержание

**1 Введение** **3**

**2 Постановка задачи** **5**

**3 Обзор аналогов и предшественников** **6**
3.1 Обзор существующих технологий . . . . . 6
3.1.1 BOUML . . . . . 6
3.1.2 ArgoUML . . . . . 8
3.1.3 Umbrello . . . . . 9
3.1.4 Eclipse GMF . . . . . 10
3.1.5 Итоги . . . . . 10
3.2 История проекта . . . . . 11
3.2.1 Переписывание графической библиотеки . . . . . 11
3.2.2 Добавление репозитория . . . . . 11
3.2.3 Отбрасывание прослойки совместимости . . . . . 11
3.2.4 СУБД как репозиторий . . . . . 12

**4 Предлагаемое решение** **13**
4.1 Архитектура построения приложения . . . . . 13
4.1.1 Model-View-Controller . . . . . 13
4.1.2 Model-View . . . . . 14
4.1.3 Model-View-Delegate . . . . . 14
4.1.4 Применение к CASE-пакету . . . . . 14
4.2 Выбор инструментария . . . . . 15
4.3 Выбор технологий . . . . . 18

**5 Практическая реализация** **19**
5.1 Пользовательский интерфейс . . . . . 19
5.1.1 Интерфейсы современных средств разработки . . . 19
5.1.2 Интерфейс новой реализации . . . . . 22
5.2 Представление: графическая библиотека, редактор . . . . 23
5.3 Модель: репозиторий . . . . . 27
5.3.1 Выбор способа хранения данных . . . . . 28
5.3.2 Представление данных внутри модели . . . . . 29
5.3.3 Представление данных вне модели . . . . . 32
5.3.4 Обработка данных. . . . . 32

**6 Выводы** **35**

**A Схема базы данных репозитория** **36**

**B Пример XML-описания редактора** **46**

# 1 Введение

На протяжении всей истории разработки ПО его сложность неуклонно возрастала. Не углубляясь в подробности, можно указать несколько путей решения этой проблемы:

- Использование языков программирования всё более и более высокого уровня и с всё возрастающим уровнем абстракции
- Развитие и улучшение процесса разработки программного обеспечения
- Создание и использование инструментальных средств для повышения скорости и качества разработки программного обеспечения.

При проектировании сложных программных систем распространено использование визуального моделирования, т.е. описание системы с разных точек зрения при помощи диаграмм, которые представляют разрабатываемую систему на более высоком уровне абстракции, чем программный код. Существует множество средств (CASE-пакетов), автоматизирующих эту деятельность. В настоящий момент основным стандартом в области визуализации ПО является язык UML, позволяющий представить различные аспекты системы в виде диаграмм.

Однако, та же самая тенденция прослеживается не только для кода ПО, но также и для его моделей. Постепенное увеличение масштабов моделируемого ПО привело к необходимости описывать его всё более абстрактными методами. Для разработки нового ПО зачастую уже не хватает старых CASE-средств, базированных на старых версиях UML. В новой редакции UML 2.0 [1] были добавлены средства для моделирования современных систем, как то:

- Значительно улучшена способность моделирования широкомасштабных программных систем: Некоторые современные приложения представляют собой интеграцию существующих автономных приложений в более комплексные системы. Эта тенденция вероятно будет продолжаться, что приведет к появлению еще более сложных систем. Для поддержки этих тенденций в язык были добавлены новые гибкие иерархические возможности для поддержки моделирования программного обеспечения на различных уровнях сложности.
- Улучшена поддержка зависимой от домена (domain-specific) специализации: Практика использования UML продемонстрировала ценность так называемых механизмов «расширения». Они были объединены и улучшены для разрешения более простой и более

точной детализации базового языка. Такие специализации позволяют еще более упростить процесс разработки и сконцентрироваться на логике приложения [2].

Становится ясно, что уже недостаточно просто реализовывать набор общих редакторов, необходимо создавать средство, которое было бы легко расширяемо. При этом недостаточно просто обеспечить возможность расширения на языке программирования — необходимо как-то описывать желаемые редакторы и по описанию получать их в рабочем варианте.

Однако за последнее время требования, предъявляемые к CASE-пакету возросли. Одних возможностей моделирования недостаточно.

К примеру, для создания встроенных систем для государственных заказов важна открытость архитектуры, причем не только самой системы, но и платформы. Таким образом, все редакторы для платформы Microsoft Windows становятся недоступны. Но недостаточно написать CASE-пакет под открытую ОС, потому что большинство пользователей используют закрытые. Необходимо реализовать пакет с учетом кроссплатформенности.

Также важна возможность сохранять данные не только в файлах, но и в репозитории, причем репозиторий должен быть не только локальный, но и удаленный. Очень важен совместный доступ к проекту, потому что зачастую над разработкой сложной системы трудятся множество пользователей, и необходимо обеспечивать им разграничение прав и одновременный доступ.

Современный пользовательский интерфейс также является важной частью. При помощи различных визуальных элементов пользователь может изучать и изменять диаграммы гораздо быстрее.

Задача объединения всех этих требований в одном работающем пакете и рассмотрена в настоящей работе.

# 2 Постановка задачи

В текущей реализации CASE-пакета REAL для создания нового типа редактора приходится писать код на языке C++, что служит потенциальным источником ошибок и неоправданной тратой времени программистов. Хочется реализовать методологию, которая бы избавила от необходимости писать программный код при добавлении нового типа редакторов. Для этого необходимо спроектировать способ описания модели и воплотить его на практике.

Таким образом, задача сводится к:

- Разработке простой схемы описания редакторов, из которой можно легко получить нужную информацию посредством несложного преобразования
- Разработке архитектуры взаимодействия компонент в CASE-пакете с учетом кроссплатформенности.
- Разработке и реализации графического представления и редактора диаграмм
  - Разработке графово-графической библиотеки
  - Реализации (при помощи разработанной графово-графической библиотеки) компоненты для представления и редактирования диаграммы в графическом виде.
- Разработке и реализации хранилища данных (репозитория)
  - Разработка и проверка схемы базы данных для сервера.
  - Разработка SQL-запросов, которые позволят получить нужные данные.
  - Разработка внутреннего представления структур данных репозитория.
  - Реализация клиентской части
- Разработке нового пользовательского интерфейса
  - Изучение интерфейса современных CASE-средств.
- Апробации выбранного подхода на примере набора редакторов UML 2.1

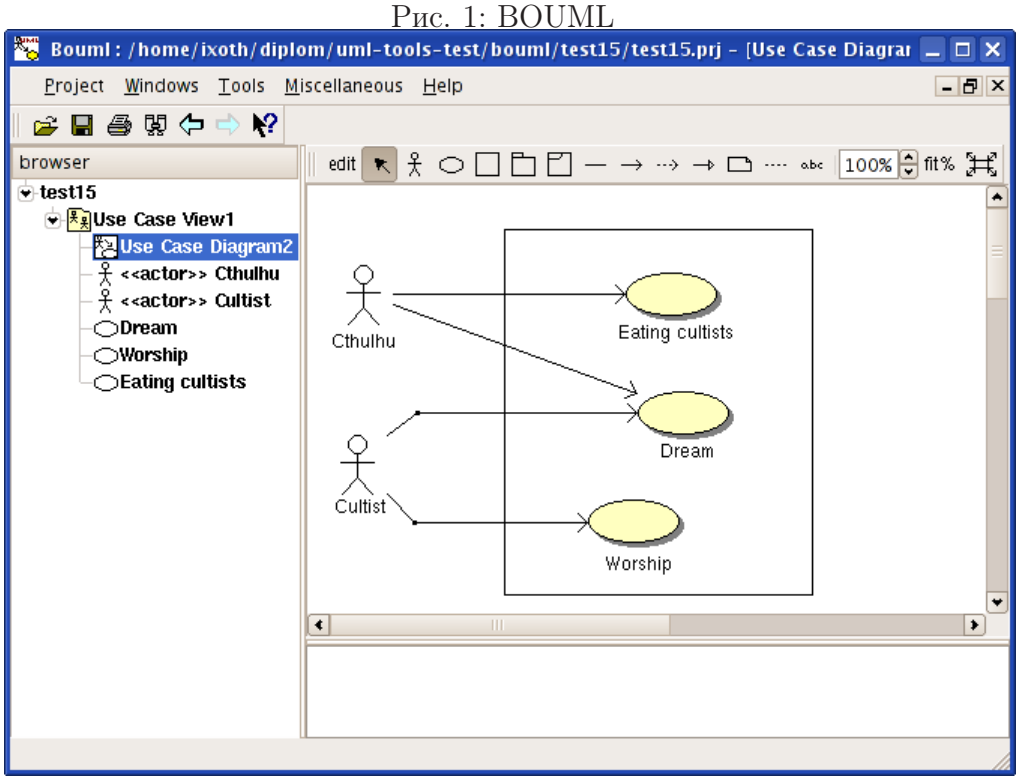
# 3 Обзор аналогов и предшественников

## 3.1 Обзор существующих технологий

Рассмотрение существующих технологий весьма затруднено по причине того, что созданная нами технология сочетает как UML-редактор, так и возможности создания редакторов DSL, причем с возможностью наследования от уже объявленных сущностей. Поэтому здесь будет более подробно рассмотрены открытые технологии моделирования UML, причем с архитектурной точки зрения, а подробные обзоры утилит для ускоренного создания DSL-редакторов можно найти в дипломных работах Симоновой Александры и Брыксина Тимофея. В таблице 1 показано более формальное сравнение возможностей моделирующих утилит, а ниже мы рассмотрим каждую из них подробнее.

### 3.1.1 BOUML

Рис. 1: BOUML




BOUML — наверное один из самых интересных открытых UML-редакторов. Он базируется на технологии Qt, быстрый, расширяемый, умеет работать с форматом XMI, и в целом достаточно удобен в использовании.

Из минусов BOUML можно выделить следущие:

Таблица 1: Сравнение открытых редакторов и фреймворков

| Средство | Real/MV | BOUML | ArgoUML | Umbrello | Eclipse GMF |
|---|---|---|---|---|---|
| Тип | UML редактор/фреймворк | UML редактор | UML редактор | UML редактор | Фреймворк |
| Метамодель UML | UML 2.1 | UML 2.0 | UML 1.4 | UML 2.1 | нет[a] |
| Расширяемость | да | нет | нет | нет | да |
| Многопользовательность | да | ограниченная | нет | нет | нет |
| Тулкит | Qt 4 | Qt 3 | Java | KDE | Java |
| Совместимость | Репозиторий, мост в XMI в разработке | XMI 2.1 | XMI 1.2 | XMI 1.x | нет |
| Независимость от IDE | да | да | да | да | нет |
| Кроссплатформенность | да | да | Java[b] | нет | Java |

[a] Среда Eclipse GMF требует задание собственной метамодели с нуля, наследование от сущностей и понятий UML невозможно.

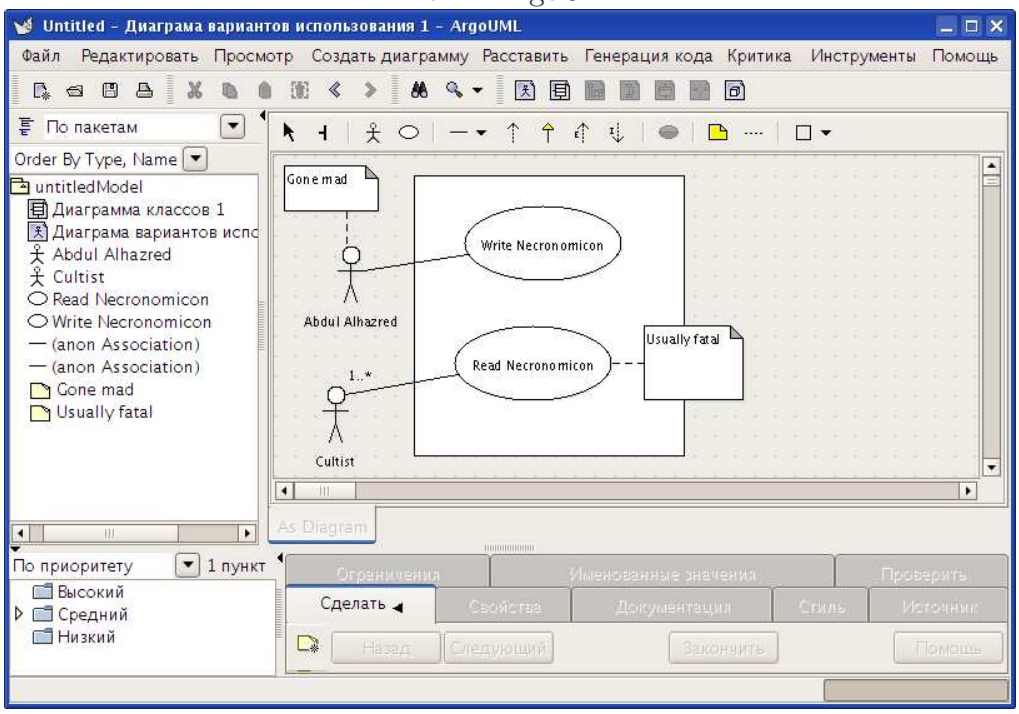
[b] Кроссплатформенность технологии Java сильно преувеличена. Несмотря на заявления фирмы Sun Microsystems, Java работает далеко не везде. Поддержка новых систем осуществляется с трудом, а использование Java под некоторыми устаревшими Unix-системами (например, под МСВС) может быть весьма затруднено: «...развёртывание кода при использовании JVM может фактически снизить его переносимость, несмотря на обратные заверения рекламы»[3]

- Невозможность расширения метамодели. Несмотря на мощную систему модулей, ядро BOUML — это реализация UML 2 на языке C++, поэтому расширение метамодели может проводить только квалифицированный программист.

- Запутанный исходный код. Проведенный анализ показывает, что исходный код состоит из 1200 классов (не считая автоматически сгенерированных), большинство из которых отвечает за обработку различных элементов UML, причем за один элемент отвечает минимум 3 класса. В текущей реализации Real/MV содержится чуть больше 100 классов, из них около 80 - автоматически сгенерированные по XML-описаниям классы, причем за каждую сущность UML отвечает только один класс.

### 3.1.2 ArgoUML

Рис. 2: ArgoUML




ArgoUML — утилита для рисования UML, написанная на Java. В целом, она используется достаточно часто, однако она проигрывает многим другим. В ней нет реализации UML 2, только UML 1.4, и работает она достаточно медленно. Однако её пользовательский интерфейс вполне удобен, и для простого моделирования на базе UML 1.4, не связанного с диаграммами поведения, она вполне подходит.

Из её минусов можно выделить:

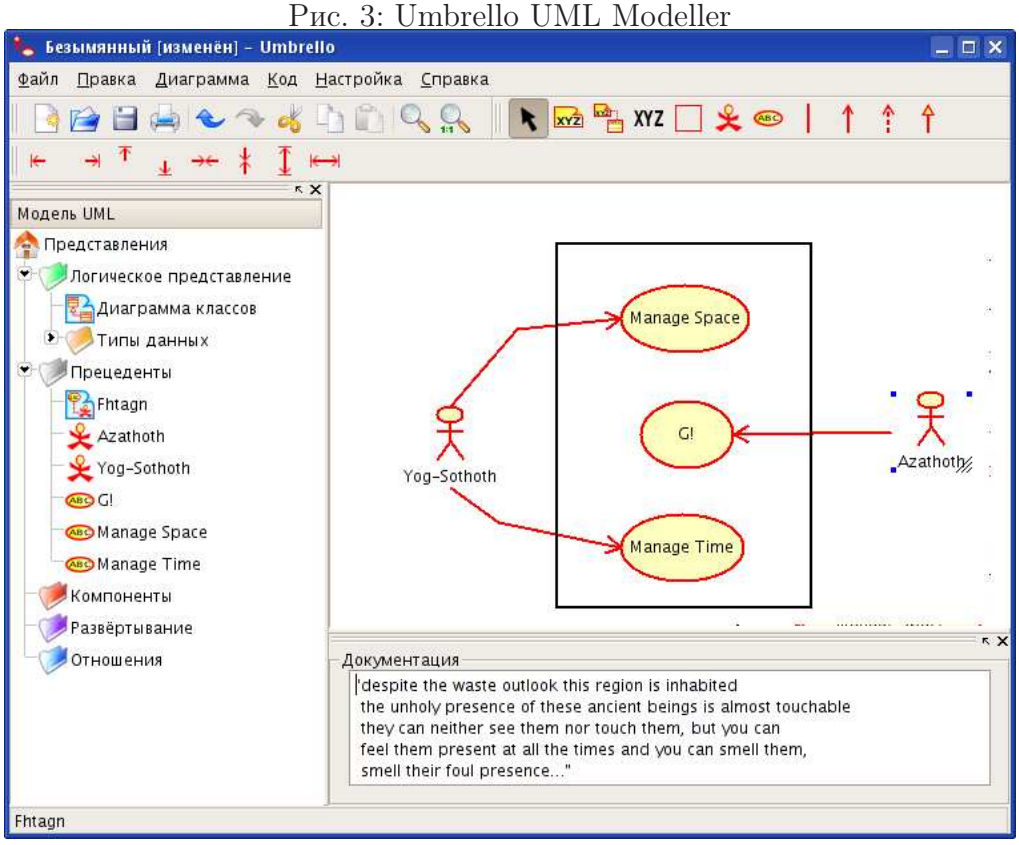
- Невозможность расширения метамодели. ArgoUML предоставляет только UML 1.4, реализованный при помощи кода на языке Java. Для создания новых сущностей и моделей необходимо использовать программиста, хорошо разбирающегося в проекте.

- При этом её исходный код еще более сложен, чем у BOUML. Более 1750 вручную написанных классов, из них около 1200 отвечают за метамодель UML. Несмотря на то, что метамодель UML 1.4 проще последних версий UML, описание его занимает больше места, чем у BOUML. Разобраться в этом коде и понять архитектуру системы очень сложно.

- Невозможность многопользовательской работы, файл сохраняется только локально, и одновременный доступ к нему множества пользователей невозможен.

### 3.1.3 Umbrello

Рис. 3: Umbrello UML Modeller




Umbrello — средство для моделирования UML из проекта KDE. Оно обладает интерфейсом, схожим с Rational Rose, и поэтому для многих

разработчиков она окажется достаточно привычной. Код этого средства самый удобночитаемый и структурированный среди всех вышеперечисленных утилит.

Основные недостатоки Umbrello:

- Как и у большинства UML-редакторов, метамодель реализована в C++-коде, поэтому исправлять её сложно. Тем не менее, хорошая и простая структура кода (немногим более 300 классов) позволяют разобраться в системе даже стороннему программисту.
- Невозможность работы в многопользовательском режиме.

### 3.1.4 Eclipse GMF

Eclipse GMF не является редактором UML — это средство для создания domain-specific language редакторов. В отличие от обычных редакторов UML, средство нацелено на создание собственных редакторов, поэтому их можно разрабатывать почти без написания кода. Однако средство не содержит реализации UML, поэтому использование его для разработки систем затруднено.

- Невозможность использовать элементы метамодели UML на своих диаграммах, невозможность их наследовать. Для объявления некоторой сущности необходимо описать её с нуля.
- Невозможность работы в многопользовательском режиме
- Полученные редакторы диаграмм привязаны к среде разработки Eclipse.

### 3.1.5 Итоги

Таким образом, работа, представленная в дипломе, является актуальной. В современных средствах UML весьма затруднено создание своих собственных графических моделей, а в фреймворках для создания редакторов затруднено переиспользование работы, в частности, невозможно использовать элементы метамодели UML в собственных проектах.

Также для большинства средств характерна невозможность многопользовательской работы, что затрудняет разработку крупных проектов.

Отличие предложенного нами решения состоит в попытке преодоления указанных сложностей.

## 3.2 История проекта

### 3.2.1 Переписывание графической библиотеки

Изначально планировалось переписать только графово-графическую библиотеку в существующей версии REAL/IT, оставив логику и репозиторий нетронутыми. На тот момент реализация библиотеки была совсем не очевидна, с большим количеством множественных наследований, и т.д. Стало понятно, что новая реализация графической библиотеки должна быть много-платформенной, поэтому было решено не пользоваться MFC, а разработать прослойку, которая позволила бы абстрагироваться от используемой оконной системы. Была предпринята попытка переписать всю библиотеку с использованием стандартных интерфейсов C++ STL, чтобы обеспечить возможность переносимости.

### 3.2.2 Добавление репозитория

В процессе переписывания стало ясно, что остальной код системы сильно завязан на текущую реализацию графово-графической библиотеки, и её смена повлечет за собой модификацию других компонентов системы. Поэтому было принято решение заняться также переписыванием репозитория, при этом учитывая недостатки предыдущего.

### 3.2.3 Отбрасывание прослойки совместимости

После реализации библиотеки в посредством прослойки стало ясно, что написанная таким образом система будет не в полной мере использовать возможности оконной библиотеки. Прослойка представляла собой обертку простейших графических примитивов, которые реализуются в несложных библиотеках вроде MFC, а вся более-менее серьезная функциональность была реализована с нуля руками. То есть при использовании библиотек, которые предоставляют не только примитивные функции рисования, но и различные геометрические преобразования и высокоуровневую графику происходит трата системных ресурсов. Также много времени программистов уходит на написание уже существующей функциональности. Поэтому было принято решение не использовать существующую реализацию графической библиотеки, а создать свою на основе имеющейся в библиотеке Qt модуля GraphicsView. Архитектура приложения к тому моменту называлась Model-View-Controller, хотя по сути ей и не являлась[1]. В реализованной архитектуре представ-

[1]Это очень частая ошибка, потому что в современных интерфейсах сущности контроллера и представления сильно взаимосвязаны, и граница между ними весьма размыта [4, 5, 6]. Программисты зачастую путают контроллер с другими сущностями: ≪Unfortunately, the popularity of the pattern has resulted in a number of faulty descriptions. In particular, the term "controller" has been used to mean different things in different contexts.≫[7].

ление имело доступ в репозиторий на чтение и на запись, а контроллером была названа сущность, которая управляла состоянием модели и представления, и не занималась изменением и обработкой данных. Вместо неё было решено использовать упрощенную схему взаимодействия Model-View вместе с дополнениями, которые используются для взаимодействия модели и представления в Qt — Model-View-Delegate. Это позволило использовать стандартные классы Qt для отображения большинства информации очень простым способом.

### 3.2.4 СУБД как репозиторий

Первая реализация репозитория представляла собой прослойку (реализованную Брыксиным Тимофеем) между логикой репозитория (реализованного Косякиным Антоном), и схемой взаимодействия модели и представления в Qt. Из-за того что на тот момент репозиторий еще не предоставлял достаточную функциональность, а также принципиальное несогласие с особенностями реализации породили свою идею репозитория. В отличие от предыдущей реализации репозитория, которая реализовывала функциональность базы данных с нуля, в новой реализации было принято решение использовать стандартную базу данных, что автоматически позволит использовать преимущества СУБД - надежность хранения, отлаженность, скорость работы. Таким образом мы также получаем независимость от хранилища данных, можно как использовать локальный файл, так и любую другую систему хранения (на сегодняшний день протестированы хранилища SQLite, MySQL, PostgreSQL и MS SQL, но теоретически должны работать и IBM DB2, Oracle, Sybase, Borland Interbase и стандартный ODBC).

# 4 Предлагаемое решение

## 4.1 Архитектура построения приложения

### 4.1.1 Model-View-Controller

Система интерактивного пользовательского интерфейса, реализуемая на основе этой архитектуры, содержит три четко выделенных уровня, так называемая MVC-триада [4]: модельный слой (model), инкапсулирующий представление данных и реализацию бизнес-логики приложения; слой презентации (view), определяющий правила формирования внешних форм, отображающих состояние приложения на экране; а также управляющий слой (controller), отвечающий за преобразование событий пользовательского интерфейса (нажатие кнопок клавиатуры, мыши и т.п.) в вызовы операций бизнес логики. Взаимодействие между этими слоями подчиняется жесткой системе ограничений, в соответствии с которой модельный слой не должен иметь прямых обращений к презентационному и управляющему слоям; слой презентации получает уведомления об изменении состояния приложения через механизм наблюдателя (observer); наконец, управляющий слой не может непосредственно модифицировать внешнюю форму и может воздействовать на состояние слоя презентации только опосредовано, через модификацию состояния модельного слоя.

Преимущества данного подхода достаточно очевидны:

- Пользовательский интерфейс меняется чаще, чем логика приложения. Если объединять логику и представление в одном объекте, то каждый раз при изменении пользовательского интерфейса придется изменять и логику. Такой подход с большой вероятностью приведет к ошибкам и потребует перетестирования логики приложения при изменении пользовательского интерфейса.

- Создание хорошего пользовательского интерфейса и эффективной системы хранения данных обычно требует различных программистских навыков. Поэтому желательно разделять создание разных частей в целях увеличения эффективности труда программистов.

- Иногда приложение представляет данные разными путями. К примеру, различные табличные данные можно представить и как таблицу со множеством чисел, и как диаграмму. В некоторых пользовательских интерфейсах возможно одновременное отображение одних и тех же данных в разном виде, причем при изменении данных в одном представлении все остальные должны соответственно обновляться.

### 4.1.2 Model-View

Сегодня, в приложении к современным системам поддержки разработки пользовательского интерфейса, использование MVC-архитектуры в основном сводится к выделению тех технологических преимуществ, которые вытекают из четкого концептуального отделения данных и логики приложения от определения форм внешнего интерфейса. В частности, этот принцип разделения слоев, принятый в MVC, можно увидеть в основе многочисленных сервер-ориентированных систем для создания веб-приложений (Struts, Spring Web MVC., Java Server Faces и др.). В самом деле, для Web приложений характерно четкое разделение представления (веб-страница, которую получает пользователь) и контроллера (который обрабатывает переходы по ссылкам). В приложениях, которые реализуются на базе полноценных фреймворков, разделение между представлением и контроллером зачастую гораздо более размыто[7]. Из-за этого возникает много путаницы с определением того, чем же всё таки является контроллер [6]. Так вышло и в предыдущей реализации проекта, которую возглавлял Косякин Антон: в ней сущность, названная контроллером не занималась изменением данных в модели, а весь контроль над чтением и изменением модели был отдан редакторам.

### 4.1.3 Model-View-Delegate

В отличие от паттерна Model-View-Controller, подход Model/View не включает полностью отдельный компонент для обработки команд пользователя. В целом, элемент view ответственен за представление данных пользователю и за обработку пользовательского ввода. Для обеспечения большей гибкости в процессе ввода данных, взаимодействие может осуществляться посредством делегатов. Эти компоненты предоставляют возможности ввода и, в некоторых представлениях, отвечают также за отрисовку данных [8].

### 4.1.4 Применение к CASE-пакету

В общем случае, реализацию CASE-пакета можно разделить на 3 основные составные части.

**Графово-графическая библиотека,** которая отвечает за отрисовку объектов, связей и т.д.

**Редакторы,** которые отвечают за изменение объектов, связей и т.д.

**Репозиторий,** который отвечает за хранение данных и предоставление их остальным модулям.

Таким образом, данные сущности достаточно очевидно отображаются в триаду MVC — репозиторий является моделью, хранилищем данных, графово-графическая библиотека с некоторыми элементами пользовательского интерфейса является представлением, а редактор является контроллером.

Однако в нашем случае было принятно решение интегрировать представление и контроллер в единую сущность. Это сильно упрощает решение задачи: в самом деле, в современных графических средах гораздо проще создать один объект, который будет сам обрабатывать и пользовательские события, и отрисовку. В традиционном паттерне MVC данный подход реализовать невозможно.

## 4.2 Выбор инструментария

В общем случае реализация более-менее сложных приложений является платформенно-зависимой задачей. Это связано со многими факторами.

- Рисование базовых графических примитивов очень сильно различается на каждой платформе. К примеру, на MacOS X и Windows (при использовании стандартных API) за рисование графических примитивов полностью отвечает ОС и её компоненты, в то время как в оконной системе X11, используемой в Unix, не существует единого API для рисования элементов пользовательского интерфейса, а приложение получает полный контроль за своей отрисовкой и обработкой событий.

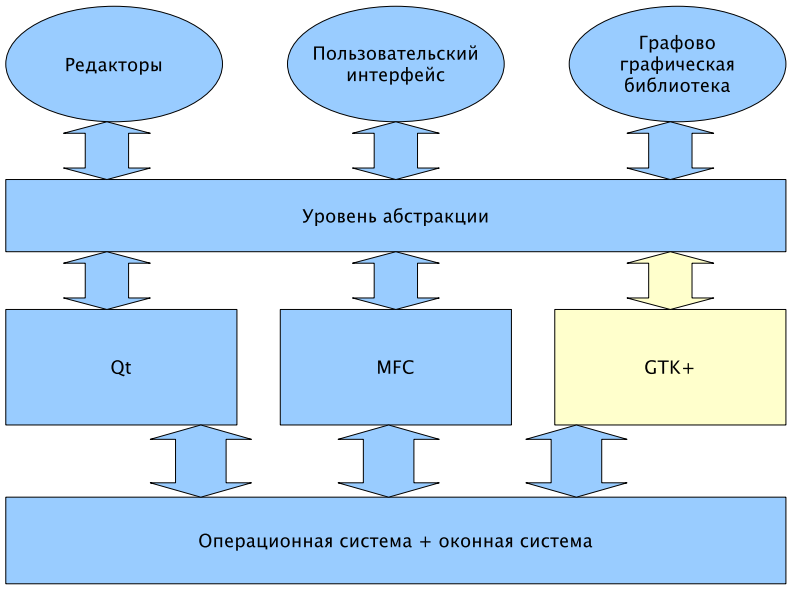
- Зачастую реализация базовых шаблонов различается на разных системах, к тому же иногда различные системы используют различные системы шаблонов.

- Библиотеки, осуществляющие, к примеру, сетевое взаимодействие или взаимодействие с базами данных имеют различный интерфейс у различных операционных систем.

Рассмотрим некоторые методы решения данных проблем.

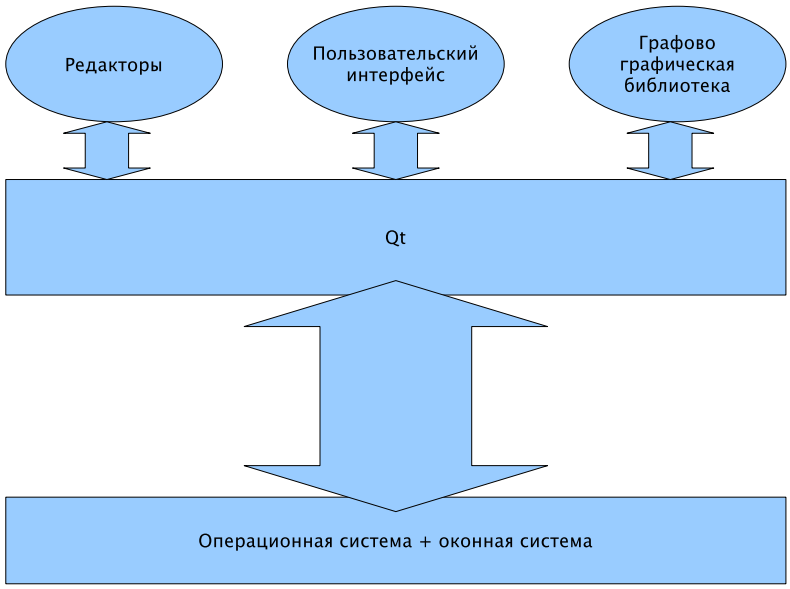
- Реализация уровня абстракций для сокрытия особенностей графического инструментария, особенностей сетевого взаимодействия. Такой подход изначально реализовывался на раннем этапе разработки, и сейчас до сих пор используется и описан в дипломных работах Косякина Антона и Бакалова Михаила. Этот подход, в принципе, достаточно верный, но реализация его сопряжена со значительными трудностями. Зачастую одни библиотеки, лежащие в основе уровня абстракции, позволяют осуществить желаемые действия проще чем другие, иногда действия осуществляются разными способами. Это особенно затрудняет разработку в

Рис. 4: Использование уровня абстракции




том случае, если в результате плохого дизайна уровень абстракции не логически абстрагирует возможности инструментария, а является простой оберткой вокруг него. Так и произошло в нашем случае на раннем этапе разработки. Изначально спроектированный как обертка над MFC, уровень абстракции был тяжело переносим на другие графические инструментарии. Некоторая низкоуровневая функциональность MFC была вынесена наверх. Её было проблематично реализовать при помощи библиотеки более высокого уровня Qt, поэтому долгое время порт Qt был сильно занижен в функционале и в стабильности работы, а порт на технологию Gtk+ вообще не был доведен до функционального состояния. Также проблемой является то, что зачастую уровень абстракции снижает возможности клиентов до общего знаменателя всех используемых графических библиотек, что очень нежелательно при использовании мощных библиотек. К примеру, в текущей реализации уровня абстракции Qt используется просто для рисования, как холст, несмотря на то, что в ней содержатся мощные средства, позволяющие создавать и управлять графическими объектами. При этом эта же функциональность реализована заново, но уже выше уровня абстракции, причем качество, надежность и стабильность этой реализации оставляет желать лучшего. Основной же аргумент против такого подхода — когда одна из библиотек не только мощная, но и кроссплатформенная сама по себе, то уровень абстракции не только усложняет код, но и теряет смысл, так как кроссплатформенность уже обеспечивается выбранным инструментарием.

Рис. 5: Использование кроссплатформенной библиотеки




- В последнее время для кроссплатформенных приложений всё чаще используются технологии Java и .NET. Несмотря на это, они не являются предпочтительными для графических приложений. Независимое научное исследование и полученный практический опыт эксплуатации показывают, что в Java зачастую нет необходимости, а предпочтительной является комбинация C++/Qt. Главными причинами этого являются более низкие производительность и эффективность использования памяти в Java (особенно при использовании инструментария Swing) при такой же обеспечиваемой продуктивности программирования. Во многих выполненных проектах начинающие программисты осваивали быстрее Java, более опытные и профессиональные разработчики (занимающиеся проектированием приложений и реализацией критичных участков программ) достигали быстрее лучших результатов с помощью C++ [9]. К тому же кроссплатформенность представляемая технологией .NET весьма ограничена, а Java с трудом запускается на старых вариантах Unix кроме Sun Solaris[3]. В частности, при использовании старых вариантов Linux (например, МСВС) могут возникать проблемы.

- И последний вариант - использование мощного кроссплатформенного инструментария для языка C++ с совместимостью на уровне исходных кодов. Причем именно библиотека Qt кажется наилучшим вариантом. Это динамично развивающаяся система к 4 версии стала не только набором кроссплатформенных графических библиотек, но и полноценным инструментарием для разра-

ботки приложений с переносимостью на уровне исходных кодов. Доступны мощные и удобные средства работы с графикой, базами данных, XML, сетью и другими технологиями. *Именно этот способ и был выбран за основу нашего подхода.* При этом сильно упрощается принципиальная схема приложения.

## 4.3 Выбор технологий

Пожалуй, стоит больше остановиться на выборе технологий. Почему в большинстве случаев стоит выбирать открытые технологии, а не придумывать и реализовывать свои? Если оставить в стороне вопрос о качестве получаемого кода (чем больше реализовывать самому, тем больше придется отлаживать), то существуют следущие преимущества.

Допустим, мы реализуем репозитарий со своим интерфейсом. Тогда программисту, который пишет генераторы или экспортеры, придется изучать новый интерфейс, или может быть даже его реализовывать (к примеру, если программист пишет на Java, а мы пишем реализацию интерфейса на C++). Если же мы будем использовать открытую и известную технологию хранения данных, допустим, будем хранить данные в обычной базе данных SQL, то программист сможет просто использовать стандартную библиотеку для доступа к данным, что гораздо быстрее и надежнее.

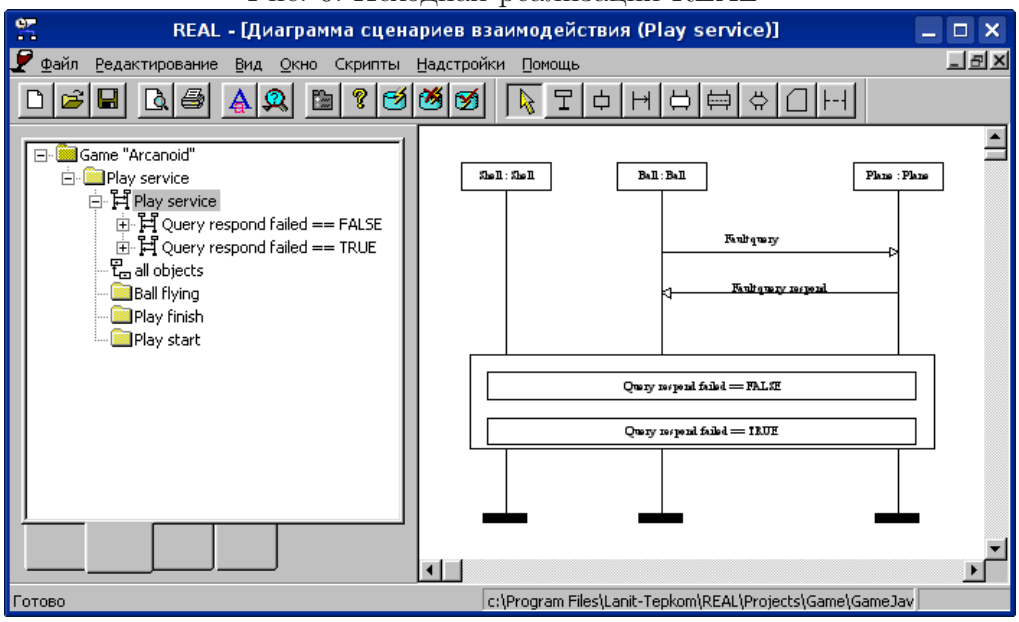
Именно поэтому в новой реализации была предпринята попытка использовать открытые технологии везде, где это возможно.

# 5 Практическая реализация

## 5.1 Пользовательский интерфейс

### 5.1.1 Интерфейсы современных средств разработки

Рис. 6: Исходная реализация REAL




На момент создания интерфейс CASE пакета REAL был весьма и весьма актуальный. Пользователь работает с диаграммой, при этом слева он видит структуру диаграммы в виде дерева. При необходимости просмотреть и изменить какое-либо свойство пользователь дважды щелкает на объекте и изменяет его свойство в отдельно открывающемся окне. Такой интерфейс был прост в реализации и был достаточно удобен в то время.

Однако, современные средства разработки (не только CASE средства) используют немного другой подход: они выносят большинство часто используемых средств в главное окно, что позволяет разработчику иметь быстрый доступ ко всем часто используемым функциям. Для редактирования отдельных элементов используются редакторы свойств, встроенные в основное окно. Иерархическая структура проекта обычно известна всякому разработчику более-менее крупных проектов, потому как все современные средства разработки её поддерживают.

Рассмотрим интерфейс какого-либо современного средства разработки. В качестве примера можно взять Eclipse. Мы видим, что рабочая область разделена на несколько частей. В центральной части и просходит сама работа — редактирование кода. Слева видна иерархическая структура проекта, справа — навигация по коду.

Рис. 7: Eclipse IDE

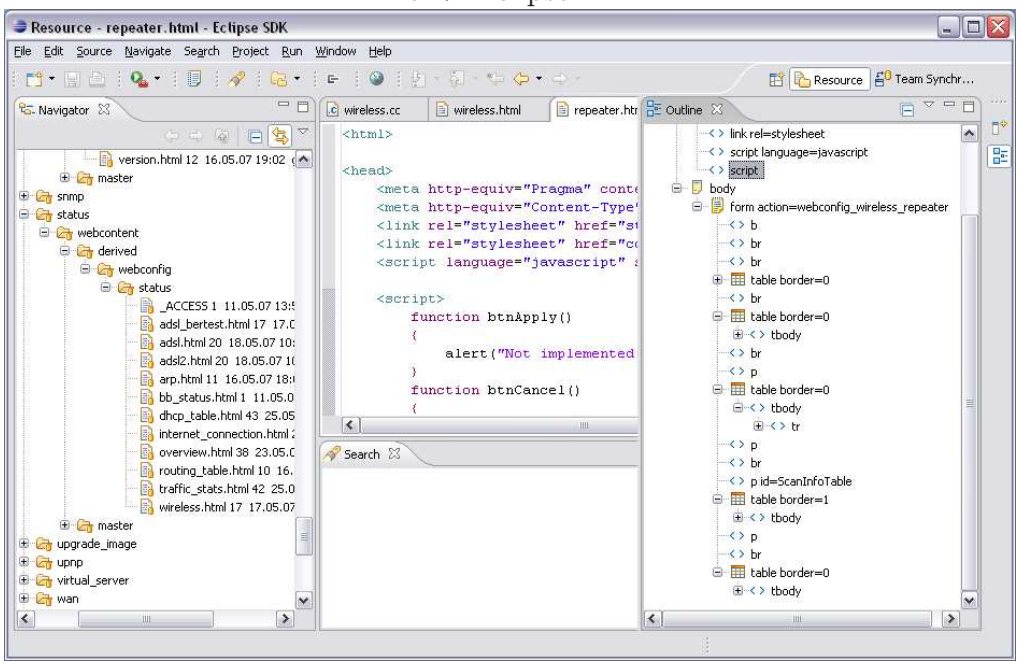



Рис. 8: Qt Designer

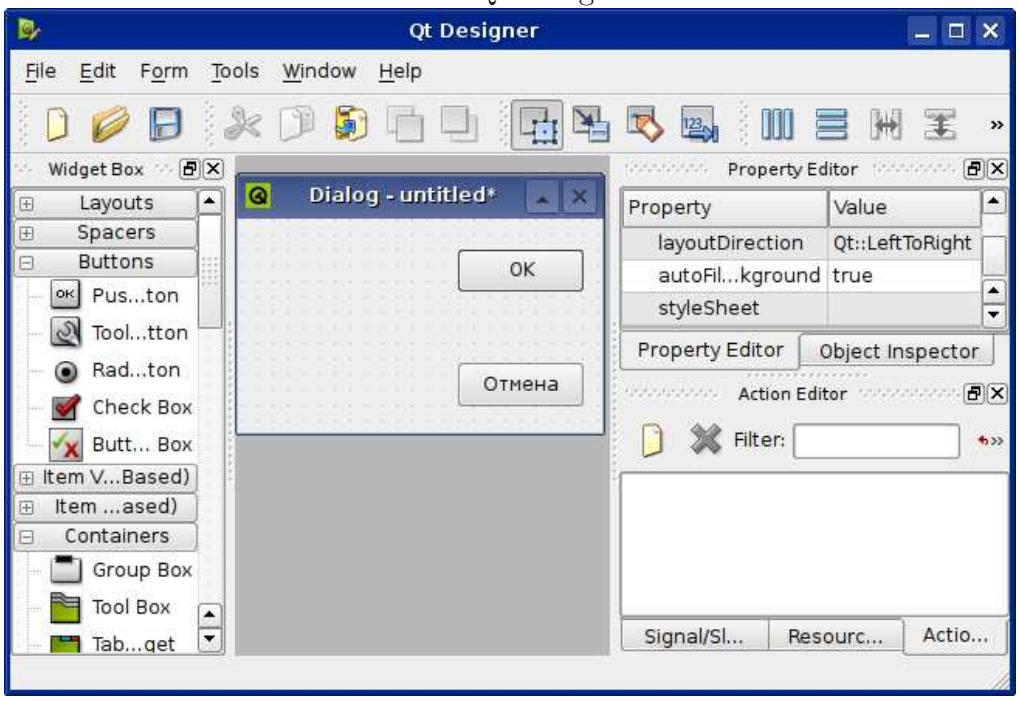
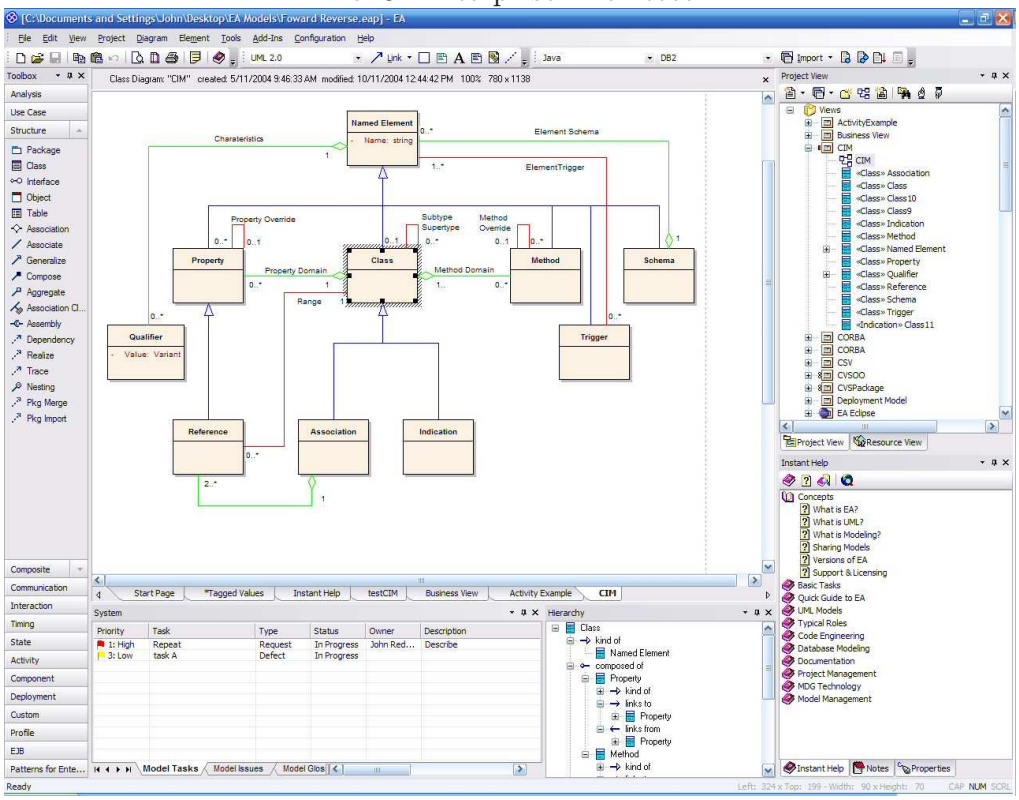


Рис. 9: Enterprise Architect



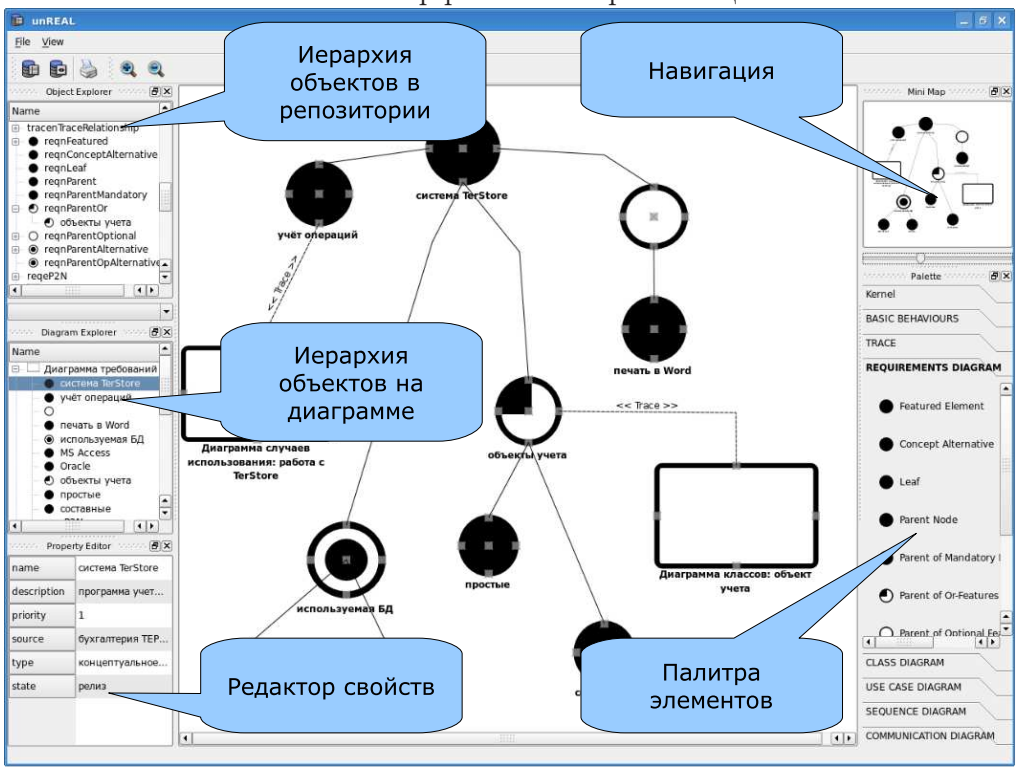

В средствах ускоренной разработки приложений (Rapid Application Development) можно обнаружить схожую концепцию. Например, в утилите Qt Designer, предназначенной для создания пользовательского интерфейса приложений, виден редактор свойств (property editor), который позволяет не открывать каждый раз новое окно для редактирования объекта, а позволяет изменять его параметры прямо в главном окне. Также есть палитра компонентов (widget box), с которой можно перетаскивать объекты мышью прямо на форму. Следует отметить, что это приложение является вполне типичным для средств RAD, и основные концепции удобства использования являются общими для всех приложений.

Также можно рассмотреть современные CASE-средства. К примеру, в Enterprise Architect мы видим всё те же элементы: навигацию по объектам, редактор свойств, палитру элементов, навигацию по диаграмме (ее можно считать аналогом навигации по коду в различных средах разработчика).

Таким образом, представляется логичным использовать современные наработки в области интерфейса пользователя и в нашем проекте. Тогда пользователю не придется переучиваться — он попадет в привычную среду, в которой объекты можно создавать перетаскиванием, свойства можно редактировать в том же окне и т.д.

### 5.1.2 Интерфейс новой реализации

Рис. 10: Интерфейс новой реализации.




Реализованное сейчас главное окно достаточно простое, но при этом оно содержит большинство привычных разработчикам элементов.

Иерархия объектов в репозитории. В данном окне содержатся все объекты в текущем проекте, распределенные по категориям. Тут же можно найти объекты, не представленные ни на одной диаграмме, и добавить их куда-либо.

Иерархия объектов на диаграмме. Верхний уровень — список всех диаграмм. Также здесь можно выбрать, какая диаграмма будет отображаться в главном окне.

Редактор свойств. Он предоставляет простой интерфейс для редактирования свойств и параметров выделенного объекта.

Навигация. В данном окошке отображается уменьшенная копия диаграммы для удобного ориентирования на больших диаграммах, причем степень увеличения можно регулировать ползунком.

Палитра элементов. Здесь можно найти все поддерживаемые элементы, рассортированные по категориям. С помощью мыши их

можно перетащить в дерево объектов и добавить на диаграмму.

Несмотря на то, что интерфейс предоставляет почти все важные возможности, главное окно спроектировано таким образом, что оно позволяет легко включать в него новые элементы интерфейса (например, можно добавить инструмент просмотра документации, или интерфейсы генераторов).

## 5.2 Представление: графическая библиотека, редактор

В старом REAL графово-графическая библиотека представляла собой надстройку над MFC, которая обеспечивала отображение и обработку событий от пользователя. Она занималась отрисовкой графических объектов и управляла их состоянием.

Когда автор диплома только начал работать над REAL, планировалось только лишь переписать графово-графическую часть с нуля. Эта библиотека базировалась на MFC, имела неявную структуру, множественные наследования. Была предпринята попытка исправить ситуацию и переписать библиотеку с нуля. Был реализован простой прототип с использованием библиотеки MFC, впоследствии планировалось интегрировать полученную библиотеку в существующий REAL, улучшив его интерфейс.

Примерно в это же время родилось требование кроссплатформенности — требовалось использовать REAL под открытыми системами, что на тот момент было невозможно. Поэтому, для облегчения дальнейшего переноса, решено было реализовать библиотеку так, чтобы её интерфейсы не содержали платформенно-зависимого кода. Эту обертку и реализовал Косякин Антон. Но, так как оригинальная реализация была заточена под MFC, фактически была реализована обертка некоторых классов MFC, отвечающих за рисование, с использованием стандартной C++ STL. В дальнейшем это привело к большим затруднениям — было весьма проблематично изменить обертку так, чтобы поддерживать другие библиотеки (для кроссплатформенности). К тому же, несмотря на реализацию классов при помощи STL, потребовались значительные усилия по переносу кода с MS Visual Studio на свободные компиляторы, для сборки под Unix. После того как перенос завершился, некоторое время разработка велась в другом направлении (писались редакторы и код для рисования элементов). При этом все изменения, которые вносились в обертку, приходилось заново портировать под открытые системы.

Из-за тяжелого положения, в котором всё время находился Unix-вариант пакета, было предложено реализовывать кроссплатформенность путем использования библиотеки Qt (подробнее об этом см. 4.2).

Резко возросла продуктивность, а также скорость работы продукта, при этом он работал и под Windows, и под Unix. Стали доступны всё новые возможности, которые предоставляет Qt для работы с графикой.

Среди нововведений — стало возможно использовать фреймворк Qt Graphics View, который предоставляет не только мощные средства отрисовки, но и управление элементами. Он предоставляет сцену, с возможностью управления и взаимодействия с большим количеством двумерных графических объектов, и элемент, представляющий собой вид на эту сцену, с возможностью поворота и масштабирования.

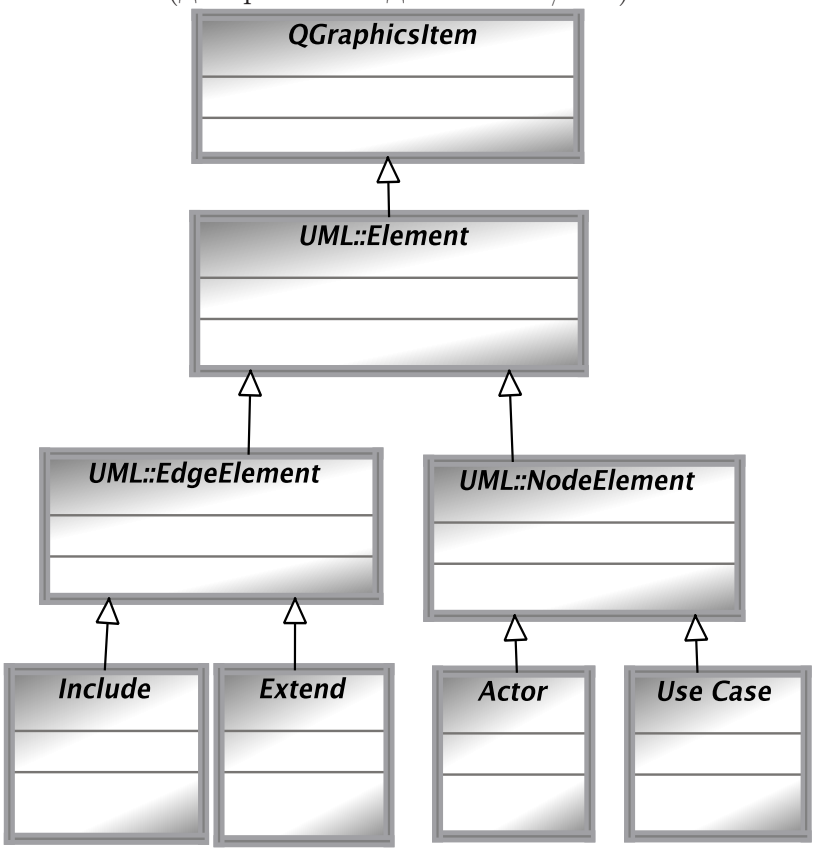
Данный фреймворк содержит архитектуру распределения событий, который позволяет осуществлять взаимодействие с элементами сцены. Элементы могут обрабатывать нажатия клавиш, щелчки мыши, а также могут отслеживать перемещения указателя. Фреймворк использует BSP-дерево для быстрого поиска элементов, поэтому он может визуализировать сцены из миллиона элементов в реальном времени.

Поэтому было решено для просмотра и редактирования диаграмм использовать представление, основанное на этом фреймворке. Так как в нашей реализации парадигме MVC представление и контроллер объединены, оказывается очень эффективным объединение рисования и взаимодействия с пользователем в одном фреймворке. Если говорить терминологией CASE средства, мы объединяем графово-графическую библиотеку (представление) и редактор (контроллер). Нету отдельной сущности, которая управляет изменением объектов, объекты сами читают данные из модели, реагируют на события пользователя и изменяют свои собственные данные при необходимости.

Реализация данных идей на практике осуществлена следущим образом (см. рис. 11). Все элементы, которые могут быть помещены на сцену в Graphics View Framework, должны наследоваться от базового класса под названием `QGraphicsItem`. Этот класс предоставляет различную базовую функциональность для всех графических элементов. От него мы наследуем класс `UML::Element`, в котором содержится базовая функциональность, которая позволяет элементам не только рисоваться и взаимодействовать с пользователем, но и взаимодействовать с моделью (репозиторием). Каждый элемент хранит в себе индекс, который он может использовать для доступа к своим данным в модели. Через этот индекс можно читать и устанавливать различные свойства. От этого класса наследуется набор базовых классов. `UML::NodeElement` представляет собой реализацию вершины графа диаграммы, а `UML::EdgeElement` — реализацию ребра графа диаграммы.

Реализация ребра диаграммы сама по себе достаточно сложная, однако для объявления новых видов ребер достаточно лишь указать тип линии (прерывистая, сплошная, точками и т.д.) и переопределить рисование стрелок. Следущий код создает прерывистую линию с надписью Trace над ней, и треугольной стрелкой на конце. Вся остальная функци-

Рис. 11: Графово-графическая библиотека
(диаграмма создана в Real/MV).




ональность реализуется базовым классом: линию можно перемещать, присоединять к вершинам, можно делать ломаную линию и т.д. Более того, её положение сохраняется в модели.

```
traceeTraceRelationshipClass::traceeTraceRelationshipClass()
{
    m_penStyle = Qt::DashLine;
    m_text = QString("Trace");
}

void traceeTraceRelationshipClass::drawStartArrow(QPainter *)
const
{
}

void traceeTraceRelationshipClass::drawEndArrow(QPainter *)
const
{
    QPolygonF triangle;
```

```
    triangle << QPointF(0,0)
             << QPointF(0,10) << QPointF(0,-10);
    painter->drawPolygon(triangle);
}
```

Вершина графа диаграммы реализует базовую графическую функциональность (возможность изменения размера элемента, и т.д.), а также логическую функциональность (работа с портами, взаимодействие с ребрами графа, и т.д.). Для реализации полноценного объекта в дочернем классе достаточно лишь переопределить метод рисования и указать, куда можно присоединять ребра, всё остальное реализуется в NodeElement. Полученный элемент будет автоматически сохранять свое положение, подключенные связи и т.д. в репозитории. Приведем небольшой пример реализации графического элемента — человечка из диаграммы случаев использования.

```
uscnActorClass::uscnActorClass()
{
    renderer.load(QString(":/shapes/uscnActorClass.svg"));
    m_contents.setWidth(40);
    m_contents.setHeight(100);
    pointPorts << QPointF(0.025, 0.45)
               << QPointF(0.775, 0.45);
    linePorts << QLineF(0.4, 0.5, 0.4, 0.66);
}
```

В конструкторе класса мы загружаем векторный графический файл, который будет рисовать человечка (в нашей системе графические данные задаются посредством SVG, однако никто не мешает разработчику генератора классов рисовать объекты самому, при помощи кода C++), далее устанавливаем изначальную ширину и высоту объекта и указываем, к каким точкам или отрезкам можно подсоединить линии.

```
void uscnActorClass::paint(QPainter *painter,
                 const QStyleOptionGraphicsItem *style,
                 QWidget *widget)
{
    renderer.render(painter, m_contents);

    QTextDocument d;
    d.setHtml(QString("<center>%1</center>")
       .arg(dataIndex.data(Qt::DisplayRole).toString()));

    d.setTextWidth(m_contents.width());
    d.drawContents(painter, m_contents);
```

```
    NodeElement::paint(painter, style, widget);
}
```

В процедуре рисования мы сначала рисуем загруженный SVG-файл. Затем мы используем индекс для получения данных из модели, и получаем строку с названием объекта. — это вызов метода `data()` объекта `dataIndex`. После этого мы форматируем эту строчку при помощи HTML, и затем рисуем этот простейший HTML-документ. После этого необходимо вызвать процедуру рисования от базового класса (она отвечает за рисование портов, рисование выделения текущего элемента и т.д.).

Данная система классов достаточно гибкая и может быть использована разными путями. В нашем проекте для графического задания объектов используется международный формат SVG [10], но это не единственный способ задания объектов. Для еще более гибкого контроля над рисуемыми элементами можно напрямую генерировать код C++, и использовать все возможности графической обработки Qt напрямую. В нашем проекте был выбран SVG для простого задания графической составляющей, а также в целях упрощения генерации кода. На самом деле, совершенно необязательно использовать генератор — данные классы вполне можно наследовать руками, причем полнота функциональности базовых классов обеспечивает простоту реализации дочерних. Такая гибкая система с простой, но мощной архитектурой обеспечивает возможность расширения системы при необходимости.

## 5.3 Модель: репозиторий

Однако самая важная и сложная в реализации составляющая CASE-пакета - это репозиторий. Именно он отвечает за хранение графовой структуры диаграмм. Плохо реализованный репозиторий может очень сильно замедлить работу CASE-средства, затруднит расширяемость. Кроме того, это одна из самых сложных частей CASE-пакета, поэтому именно тут наиболее вероятны ошибки программиста.

Как уже упоминалось в 3.2.4 на с. 12, до отделения проекта Real/MV от своего предшественника, репозиторий был реализован Косякиным Антоном с нуля, в качестве хранилища использовались файлы, а сам сервер репозитория был доступен только под Microsoft Windows. Тем не менее, была попытка использовать клиентскую часть репозитория (в ней заявлялась полная кроссплатформенность, что на деле не подтвердилось) в качестве основы нашего проекта. Как показала практика, было весьма проблематично использовать этот код, причем нормальной возможности протестировать его не было из-за невозможности установки сервера под отличные от Microsoft Windows операционные

системы. В результате, было принято решение писать полностью свой репозиторий.

Из-за своей сложности эта структура была одной из самых проблемных в написании. В нашем проекте для её разработки был выбран подход прототипирования — вместо разработки всей структуры сразу планировалось последовательно создавать серию рабочих прототипов, улучшая их, или даже переписывая при необходимости.

### 5.3.1 Выбор способа хранения данных

С самого начала основной проблемой был выбор подходящего способа хранения информации, который бы позволил не только хранить требуемые данные, но и обеспечивал бы сетевую прозрачность, многопользовательский доступ, возможность проводить транзакции, и т.д. Реализовывать такое хранилище полностью с нуля — достаточно проблематичное занятие, а отладить и протестировать все компоненты такого хранилища, принимая во внимание размер нашей команды, практически невозможно. К тому же, не хочется заново реализовывать функциональность, доступную (в гораздо более мощном и надежном виде) в других продуктах.

Поэтому было принято решение для хранения данных пользоваться реляционными базами данных. Они удовлетворяют всем требованиям, которые мы предъявляем к репозиторию:

- Большинство из них поддерживают доступ по сети, что позволяет установить один сервер, и использовать его целым предприятием. Реляционные клиент-серверные СУБД рассчитаны на обработку больших массивох данных большим количеством пользователей, поэтому даже небольшой сервер сможет без особых проблем обслуживать сотни пользователей.

- Поддержка механизма аутентикации пользователей: в СУБД для доступа к базе данных необходимо предоставить имя пользователя и пароль, причем в зависимости от пользователя возможно разграничения прав на редактирование данных.

- Для редактирования пользователей и прав доступа существуют различные утилиты, как от авторов СУБД, так и от сторонних производителей.

- Удобные возможности резервного копирования, переноса данных, транзакции, и многое другое.

В работе [11] рассматривается подобный подход к хранению диаграмм UML: «Several tools have been developed to support the modeling and management of UML diagrams. However, there may be some difficulties of diagram-retrieval and co-work among users, because the diagrams

are stored in files. To cope with the difficulties, we propose a database-supported methodology to model and manage the diagrams. In this methodology, the constituents of class diagram, use case diagram, and activity diagram are transformed in the form of relational tables. They can be stored in a database, and then retrieved from the database by using database queries. This database methodology provides the concurrent sharing and high reuse of diagrams». Но, в отличии от нашего подхода, в нем поддерживается только подмножество UML 1.4, и не поддерживаются расширяемость метамодели. Наш подход поддерживает расширяемость метамодели и нацелен на применение UML 2.x.

### 5.3.2 Представление данных внутри модели

Логически, набор диаграмм представляет собой графовую структуру — объекты могут быть связаны между собой, могут участвовать на разных диаграммах, диаграммы сами могут быть дочерними элементами каких-либо других диаграмм. В реляционных базах данных данные хранятся в плоских таблицах. Необходимо придумать подход, который позволит не только хранить данные, но и эффективно их получать и модифицировать.

Первая реализации репозитория, которую разработал и реализовал Брыксин Тимофей, была устроена следущим образом. Существовала основная таблица, которая читалась при подключении к репозиторию, в которой хранился список диаграмм (т.е. их названия). Каждая диаграмма представляла собой отдельную таблицу, на которой хранились название объекта, её свойства на диаграмме (в той реализации это были только координаты). Кроме того, существовали таблицы, в которых содержались свойства этих элементов. Связь этих таблиц осуществлялась не на уровне языка запросов, а на уровне приложения, причем связь осуществлялась по названию. Таким образом, произвольно именовать объекты было невозможно, для их переименовывания приходилось изменять значение несколько раз в разных таблицах, а переименовать диаграмму было и вовсе невозможно. Более того, так как изначально не была продумана система представления данных, то вся система взаимодействия представляла собой три отдельных модели, которые обменивались между собой сообщениями для обновления информации. Эта реализация была подвержена ошибкам, а её код был весьма трудно читать и сопровождать. Несмотря на всё вышесказанное, эта модель позволила нам протестировать элемент графического представления, а заодно и понять, как *не* стоит делать модели.

После этого за реализацию модели взялся автор данной дипломной работы. Была предпринята попытка исправления существующей схемы базы данных. Во-первых, была переписана клиентская часть, которая стала правильной единой моделью со множеством разных представлений. Это позволило упростить логику основного приложения, так

как не нужно было обеспечивать взаимодействие между разными моделями. Сама схема была нормализована, строковые идентификаторы были заменены на цифровые, что позволило решить проблемы с именованием элементов и диаграмм, работа репозитория ускорена, повышена надежность, но основные недостатки оставались — каждая новая диаграмма представляла собой отдельную таблицу, что затрудняло написание запросов SQL — их приходилось генерировать на лету. Для чтения диаграмм приходилось создавать несколько запросов, и читать разные таблицы по нескольку раз. Это почти не замедляло работу с репозиторием в локальной сети, но через глобальные сети это становилось достаточно ощутимым.

Стало ясно, что дизайн данной системы в корне неверен, поэтому была предпринята попытка переделать схему базы данных, чтобы уменьшить количество запросов и упростить клиентскую часть. В результате этого была создана схема, которая позволила упростить взаимодействие с базой данных. Сами запросы стали ощутимо сложнее, но количество запросов резко упало.

В результате этого, работа через глобальные сети стала практически неотличима от локальной. К примеру, автор данной работы использует для работы дома удаленный репозиторий, и на канале 256 kbps задержка практически неотличима от работы с локальным репозиторием.

Разработанная схема выглядит таким образом. Основная информация о поддерживаемых типах, вместе с уникальными идентификаторами типов хранится в таблице `metatable`. Все существующие в репозитории элементы перечислены в таблице `nametable` вместе со своими уникальными идентификаторами, типом и теми свойствами, которые являются общими для всех элементов (например, имени). Размещение таких свойств в этой таблице позволяет ускорить доступ, особенно для стандартных представлений. По этой таблице, используя аггрегирование и слияние с таблицей `metatable` можно получить список всех типов с количеством элементов данного типа в репозитории:

```
SELECT metatable.id, metatable.name,
       metatable.qualifiedName, COUNT(nametable.id)
  FROM metatable
LEFT JOIN nametable ON nametable.type=metatable.id
GROUP BY metatable.id;
```

В предыдущей реализации репозитория для выяснения числа потомков у элемента надо было создавать отдельный запрос, что заметно увеличивало время отклика, особенно при доступе через интернет.

Помимо этого, есть одна большая таблица `diagram`, которая содержит данные о потомках объекта, в частности, о содержимом диаграмм. В данной схеме не только диаграммы могут содержат внутри себя объекты. К примеру, пакеты тоже содержат классы в качестве дочерних

элементов. В таблице содержится идентификатор родительского элемента, идентификатор дочернего элемента и параметры, отвечающие за отображение объекта на данной диаграмме (координаты, размеры, конфигурацию ломаной линии, и т.д.). Таким образом, для получения всех элементов определенного типа вместе с именами и числом детей можно использовать следущий запрос:

```
SELECT nametable.id, nametable.name, nametable.type,
       nametable.qualifiedName, metatable.name,
       COUNT(diagram.diagram_id)
  FROM nametable
LEFT JOIN metatable ON metatable.id = nametable.type
LEFT JOIN diagram ON diagram.diagram_id = nametable.id
WHERE nametable.type = :type
GROUP BY nametable.id
ORDER BY diagram.diagram_id DESC;
```

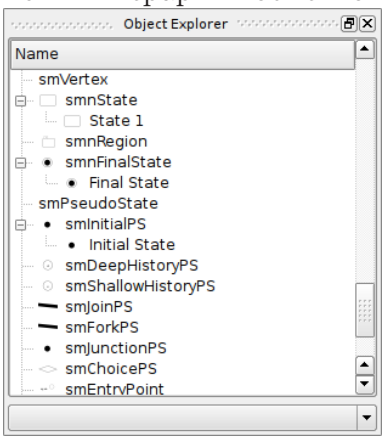
Однако, для нас наибольший интерес представляет то, как можно получить содержимое диаграммы. Это можно осуществить следущим образом:

```
SELECT diagram.el_id, nametable.name, nametable.type,
       diagram.x, diagram.y, diagram.cfg,
        nametable.qualifiedName, COUNT(children.el_id)
       FROM diagram
  LEFT JOIN nametable ON diagram.el_id = nametable.id
  LEFT JOIN diagram AS children
             ON children.diagram_id = diagram.el_id
  WHERE diagram.diagram_id = :type
  GROUP BY diagram.el_id ;
```

Помимо вышеуказанных таблиц, в базе данных содержатся таблицы с именами вида `el_123`. Они используются для хранения свойств, специфичных для каждого типа элементов. К сожалению, совместить эту информацию в одну таблицу (для упрощения обработки запросов) не представляется возможным по причине того, что разные элементы имеют разные свойства.

Таким образом мы видим, что при помощи созданной схемы легко можно построить дерево иерархии объектов, сократив при этом число запросов до минумума, что не только увеличивает скорость работы и сокращает нагрузку на сеть, но и упрощает код клиентской части репозитория. Точную схему базы данных (немного уменьшенную для увеличения читаемости) можно найти в приложении A на с. 36.

Рис. 12: Иерархия объектов.




### 5.3.3 Представление данных вне модели

Проблема заключается в том, что с точки зрения пользователя, объекты расположены в иерархической структуре — существует множество диаграмм, на которых содержатся элементы, которые в свою очередь содержат в качестве дочерних элементов другие, и т.д. Таким образом, одному элементу в графе может соответствовать несколько элементов в иерархической структуре. К примеру, один и тот же актер может присутствовать на нескольких диаграммах случаев использования. Таким образом, встаёт задача, как представить граф в виде дерева, причем максимально приближенного к пользовательскому восприятию.

В результате этого был выработан следущий подход (см. рис. 12). Весь репозиторий, к которому подключен пользователь, представляется в виде единого дерево. Листья самого верхнего уровня — это типы элементов, например диаграмма, актер, случай использования, класс, пакет и т.д. В качестве детей каждого листа содержатся все элементы данного типа в репозитории. Если в репозитории они содержат внутри себя объекты в качестве потомков, то и в дереве у них будут видны потомки.

При таком подходе один и тот же элемент может встречаться в дереве несколько раз, но поддерево, родителем которого он будет являться, всегда будет одно и то же. При обновлении модели (например, пользователь добавил к некоторому пакету еще один класс) необходимо удостовериться в том, что во всех других местах, где встречается этот элемент, его поддерево тоже обновилось.

### 5.3.4 Обработка данных.

Остановимся более подробно на реализации — как из плоских таблиц базы данных мы представляем данные иерархически. В 5.3.2 уже было рассказано, какие запросы в основном используются к СУБД для полу-

чения данных. Однако недостаточно просто делать каждый раз запрос при обращении к модели. Стандартные представления Qt обращаются к моделям достаточно часто, поэтому необходимо реализовать некий механизм кэширования, который смог бы ускорить работу и сократить нагрузку на сервер СУБД.

Обычно при реализации таких моделей в памяти строится дерево, структура которого и предоставляется наружному интерфейсу. В узлах данного дерева хранятся объекты, вместе со своими свойствами и параметрами. Однако при больших объемах данных такой подход неприемлем. В данном случае необходимо загружать данные по мере необходимости, т.е. когда их запрашивает пользователь (открывает диаграмму, раскрывает дерево и т.д.). Такой подход гораздо лучше предыдущего, и именно он был реализован изначально, однако с таким подходом возникает несколько проблем.

- Учитывая то, что в нашем дереве некоторые элементы могут встречаться несколько раз, при изменении одного элемента необходимо пройтись по всему дереву и обновить данные во узлах дерева, которые являются копией измененного. Приходится прилагать усилия для поддержания локальных данных в актуальном состоянии.

- Если пользователь уже получал данные об элементе недавно (допустим, просмотрел диаграмму с его участием) и перешел в другую сущность, содержащую этот объект как дочерний (например, на другую диаграмму с его участием), то актуальные данные будут запрашиваться еще раз, что замедляет работу.

- Также мы видим, что в данном случае возникает ненужно использование памяти, поскольку одни и те же данные хранятся в памяти несколько раз.

Поэтому был разработан другой подход. В дереве хранятся только уникальные идентификаторы элементов, а все данные хранятся в хэше. Хэш был выбран из-за того, что элементов может быть очень много, но к ним хочется иметь константное время доступа. В этом случае можно было бы использовать обычный вектор, который предоставляет константное время доступа, но в репозитории могут присутствовать элементы с самыми разными идентификаторами (как 1000, так и 100000), поэтому такой способ вызвал бы чрезмерное использование памяти. В среднем, хэш имеет время доступа и вставки элементов $O(1)$, и только в самом плохом случае получается $O(n)$. Самым лучшим вариантом было бы хранить все данные в хэше, но от дерева невозможно избавиться полностью по причине того, что объекту в модели необходимо знать своего родителя, что невозможно хранить в хэше, так как у объекта может быть несколько родителей (допустим, один объект может присутствовать на двух разных диаграммах).

Храня все данные в хэше, мы устраняем все вышеуказанные недостатки. Когда был реализован такой метод доступа к данным, по экспериментальным данным, при небольшом репозитории и/или при работе только с небольшиым числом диаграмм, количество запросов к базе данных снизилось в среднем в 4.5 раза. Самые большое ускорение получилось в случае доступа к репозиторию через интернет.

# 6 Выводы

Таким образом, был реализован прототип CASE-пакета, который позволяет определять редакторы при помощи специального описания на базе XML. В дальнешем это позволит расширить возможности и улучшить взаимодействие с пользователем.

В рамках данной работы были получены следующие результаты:

- Предложена идея описывать редакторы при помощи простого XML-описания, в дальнейшем конвертируя его в C++ для компиляции.
  - Схема такого описания был разработана и описана в дипломной работе Симоновой Александры и применена для описания набора редакторов UML 2.1 и редактора требований.
- Создана и реализована кроссплатформенная архитектура самого пакета, с разделением на отдельные модули с четко описанными интерфейсами.
- Разработан и реализован новый репозиторий для CASE-системы с поддержкой многопользовательности и сетевого доступа.
- Разработана и реализована новая графово-графическая библиотека, для нее создан набор базовых классов, который позволяет достаточно просто генерировать код редакторов по их описанию.
  - На его базе создан генератор, который по XML-описанию создаёт код на языке C++, который реализует функциональность редакторов. Он описан в дипломной работе Брыксина Тимофея.
- Разработан и реализован новый, современно выглядящий и кроссплатформенный интерфейс пользователя, что позволит эксплуатировать CASE-средство не только на операционных системах семейства Microsoft Windows, но и на Mac OS X и различных Unix-системах, включая Linux и FreeBSD.
- Данный подход апробирован на наборе редакторов UML 2.1 и редакторе требований, заданных при помощи XML-описания.

# A Схема базы данных репозитория

Здесь приведен код на SQL, который описывает внутреннюю структуру базы данных. Он был автоматически сгенерирован и содержить только описание диаграммы случаев использования (для простоты).

Данный код был создан при помощи генератора, написанного Брыксиным Тимофеем, из описания метамодели случаев использования, заданном посредством XML-описания из приложения B. Видно, как свойства переходят из XML-описания в SQL-схему.

```
CREATE TABLE nametable (
        id INTEGER PRIMARY KEY AUTO_INCREMENT NOT NULL,
        type MEDIUMINT NOT NULL,
        name VARCHAR(255),
        qualifiedName VARCHAR(255)
);

CREATE TABLE metatable (
        id INTEGER PRIMARY KEY AUTO_INCREMENT NOT NULL,
        name VARCHAR(255),
        qualifiedName VARCHAR(255)
);

CREATE TABLE diagram (
        diagram_id INTEGER NOT NULL,
        el_id MEDIUMINT NOT NULL,
        cfg VARCHAR(1000),
        x MEDIUMINT,
        y MEDIUMINT,
        isExpandable BOOL
);

INSERT INTO ‘metatable‘ (id, name, qualifiedName)
     VALUES (1, ’krnnElement’, ’Element’);
INSERT INTO ‘metatable‘ (id, name, qualifiedName)
     VALUES (2, ’krnnDiagram’, ’Diagram’);
INSERT INTO ‘metatable‘ (id, name, qualifiedName)
     VALUES (3, ’krnnNamedElement’, ’NamedElement’);
INSERT INTO ‘metatable‘ (id, name, qualifiedName)
     VALUES (4, ’krnnComment’, ’Comment’);
INSERT INTO ‘metatable‘ (id, name, qualifiedName)
     VALUES (5, ’krnnNamespace’, ’Namespace’);
INSERT INTO ‘metatable‘ (id, name, qualifiedName)
     VALUES (6, ’krnnPackage’, ’Package’);
INSERT INTO ‘metatable‘ (id, name, qualifiedName)
```

```
     VALUES (7, 'krnnPackageableElem', 'PackageableElement');
INSERT INTO `metatable` (id, name, qualifiedName)
     VALUES (8, 'krnnType', 'Type');
INSERT INTO `metatable` (id, name, qualifiedName)
     VALUES (9, 'krnnTypedElem', 'TypedElement');
INSERT INTO `metatable` (id, name, qualifiedName)
     VALUES (10, 'krnnRedefElement', 'RedefinableElement');
INSERT INTO `metatable` (id, name, qualifiedName)
     VALUES (11, 'krnnClassifier', 'Classifier');
INSERT INTO `metatable` (id, name, qualifiedName)
     VALUES (12, 'krnnProperty', 'Property');
INSERT INTO `metatable` (id, name, qualifiedName)
     VALUES (13, 'krnnOperation', 'Operation');
INSERT INTO `metatable` (id, name, qualifiedName)
     VALUES (14, 'krnnParameter', 'Parameter');
INSERT INTO `metatable` (id, name, qualifiedName)
     VALUES (15, 'krnnFeature', 'Feature');
INSERT INTO `metatable` (id, name, qualifiedName)
     VALUES (16, 'krnnBehavioralFeature', 'BehavioralFeature');
INSERT INTO `metatable` (id, name, qualifiedName)
     VALUES (17, 'krnnStructuralFeature', 'StructuralFeature');
INSERT INTO `metatable` (id, name, qualifiedName)
     VALUES (18, 'krneRelationship', 'Relationship');
INSERT INTO `metatable` (id, name, qualifiedName)
     VALUES (19, 'krneDirRelationship', 'DirectedRelationship');
INSERT INTO `metatable` (id, name, qualifiedName)
     VALUES (20, 'krneComLink', 'CommentLink');
INSERT INTO `metatable` (id, name, qualifiedName)
     VALUES (21, 'krneElementImport', 'ElementImport');
INSERT INTO `metatable` (id, name, qualifiedName)
     VALUES (22, 'krnePackageImport', 'PackageImport');
INSERT INTO `metatable` (id, name, qualifiedName)
     VALUES (23, 'krneGeneralization', 'Generalization');
INSERT INTO `metatable` (id, name, qualifiedName)
     VALUES (24, 'bbnBehavior', 'Behavour');
INSERT INTO `metatable` (id, name, qualifiedName)
     VALUES (25, 'bbnBehavioredClassifier', 'Behavour');
INSERT INTO `metatable` (id, name, qualifiedName)
     VALUES (26, 'bbnOpaqueBehavior', 'Behavour');
INSERT INTO `metatable` (id, name, qualifiedName)
     VALUES (27, 'bbnFunctionBehavior', 'Behavour');
INSERT INTO `metatable` (id, name, qualifiedName)
     VALUES (28, 'uscnActor', 'Actor');
INSERT INTO `metatable` (id, name, qualifiedName)
```

```
     VALUES (29, 'uscnUCClassifier', 'UCClassifier');
INSERT INTO `metatable` (id, name, qualifiedName)
     VALUES (30, 'uscnUseCase', 'UseCase');
INSERT INTO `metatable` (id, name, qualifiedName)
     VALUES (31, 'uscaExtend', 'Extend');
INSERT INTO `metatable` (id, name, qualifiedName)
     VALUES (32, 'uscaInclude', 'Include');

CREATE TABLE `el_1` (
        `id` mediumint NOT NULL,
        `name` VARCHAR(30),
        `state` VARCHAR(100)
);

CREATE TABLE `el_2` (
        `id` mediumint NOT NULL,
        `name` VARCHAR(30),
        `visibility` VARCHAR(100),
        `qualifiedName` VARCHAR(100),
        `state` VARCHAR(100)
);

CREATE TABLE `el_3` (
        `id` mediumint NOT NULL,
        `name` VARCHAR(30),
        `visibility` VARCHAR(100),
        `qualifiedName` VARCHAR(100),
        `state` VARCHAR(100)
);

CREATE TABLE `el_4` (
        `id` mediumint NOT NULL,
        `name` VARCHAR(30),
        `body` VARCHAR(100),
        `state` VARCHAR(100)
);

CREATE TABLE `el_5` (
        `id` mediumint NOT NULL,
        `name` VARCHAR(30),
        `visibility` VARCHAR(100),
        `qualifiedName` VARCHAR(100),
        `state` VARCHAR(100)
);
```

```
CREATE TABLE ‘el_6‘ (
        ‘id‘ mediumint NOT NULL,
        ‘name‘ VARCHAR(30),
        ‘visibility‘ VARCHAR(100),
        ‘qualifiedName‘ VARCHAR(100),
        ‘state‘ VARCHAR(100)
);

CREATE TABLE ‘el_7‘ (
        ‘id‘ mediumint NOT NULL,
        ‘name‘ VARCHAR(30),
        ‘visibility‘ VARCHAR(100),
        ‘qualifiedName‘ VARCHAR(100),
        ‘state‘ VARCHAR(100)
);

CREATE TABLE ‘el_8‘ (
        ‘id‘ mediumint NOT NULL,
        ‘name‘ VARCHAR(30),
        ‘visibility‘ VARCHAR(100),
        ‘qualifiedName‘ VARCHAR(100),
        ‘state‘ VARCHAR(100)
);

CREATE TABLE ‘el_9‘ (
        ‘id‘ mediumint NOT NULL,
        ‘name‘ VARCHAR(30),
        ‘type‘ INTEGER,
        ‘visibility‘ VARCHAR(100),
        ‘qualifiedName‘ VARCHAR(100),
        ‘state‘ VARCHAR(100)
);

CREATE TABLE ‘el_10‘ (
        ‘id‘ mediumint NOT NULL,
        ‘name‘ VARCHAR(30),
        ‘isLeaf‘ BOOL,
        ‘visibility‘ VARCHAR(100),
        ‘qualifiedName‘ VARCHAR(100),
        ‘state‘ VARCHAR(100)
);

CREATE TABLE ‘el_11‘ (
```

```
        `id` mediumint NOT NULL,
        `name` VARCHAR(30),
        `isAbstract` BOOL,
        `visibility` VARCHAR(100),
        `qualifiedName` VARCHAR(100),
        `state` VARCHAR(100),
        `isLeaf` BOOL
);

CREATE TABLE `el_12` (
        `id` mediumint NOT NULL,
        `name` VARCHAR(30),
        `isReadOnly` BOOL,
        `isStatic` BOOL,
        `isLeaf` BOOL,
        `visibility` VARCHAR(100),
        `qualifiedName` VARCHAR(100),
        `state` VARCHAR(100),
        `type` INTEGER
);

CREATE TABLE `el_13` (
        `id` mediumint NOT NULL,
        `name` VARCHAR(30),
        `isQuery` BOOL,
        `isOrdered` BOOL,
        `isUnique` BOOL,
        `lower` INTEGER,
        `upper` INTEGER UNSIGNED,
        `type` INTEGER,
        `isAbstract` BOOL,
        `isStatic` BOOL,
        `isLeaf` BOOL,
        `visibility` VARCHAR(100),
        `qualifiedName` VARCHAR(100),
        `state` VARCHAR(100)
);

CREATE TABLE `el_14` (
        `id` mediumint NOT NULL,
        `name` VARCHAR(30),
        `direction` VARCHAR(100),
        `type` INTEGER,
        `visibility` VARCHAR(100),
```

```
        `qualifiedName` VARCHAR(100),
        `state` VARCHAR(100)
);

CREATE TABLE `el_15` (
        `id` mediumint NOT NULL,
        `name` VARCHAR(30),
        `isStatic` BOOL,
        `isLeaf` BOOL,
        `visibility` VARCHAR(100),
        `qualifiedName` VARCHAR(100),
        `state` VARCHAR(100)
);

CREATE TABLE `el_16` (
        `id` mediumint NOT NULL,
        `name` VARCHAR(30),
        `isAbstract` BOOL,
        `isStatic` BOOL,
        `isLeaf` BOOL,
        `visibility` VARCHAR(100),
        `qualifiedName` VARCHAR(100),
        `state` VARCHAR(100)
);

CREATE TABLE `el_17` (
        `id` mediumint NOT NULL,
        `name` VARCHAR(30),
        `isReadOnly` BOOL,
        `isStatic` BOOL,
        `isLeaf` BOOL,
        `visibility` VARCHAR(100),
        `qualifiedName` VARCHAR(100),
        `state` VARCHAR(100),
        `type` INTEGER
);

CREATE TABLE `el_18` (
        `id` mediumint NOT NULL,
        `name` VARCHAR(30),
        `from` VARCHAR(100),
        `to` VARCHAR(100),
        `fromPort` VARCHAR(100),
        `toPort` VARCHAR(100),
```

```
        `state` VARCHAR(100)
);

CREATE TABLE `el_19` (
        `id` mediumint NOT NULL,
        `name` VARCHAR(30),
        `from` VARCHAR(100),
        `to` VARCHAR(100),
        `fromPort` VARCHAR(100),
        `toPort` VARCHAR(100),
        `state` VARCHAR(100)
);

CREATE TABLE `el_20` (
        `id` mediumint NOT NULL,
        `name` VARCHAR(30),
        `from` VARCHAR(100),
        `to` VARCHAR(100),
        `fromPort` VARCHAR(100),
        `toPort` VARCHAR(100)
);

CREATE TABLE `el_21` (
        `id` mediumint NOT NULL,
        `name` VARCHAR(30),
        `from` VARCHAR(100),
        `to` VARCHAR(100),
        `fromPort` VARCHAR(100),
        `toPort` VARCHAR(100),
        `alias` VARCHAR(100),
        `visibility` VARCHAR(100),
        `state` VARCHAR(100)
);

CREATE TABLE `el_22` (
        `id` mediumint NOT NULL,
        `name` VARCHAR(30),
        `from` VARCHAR(100),
        `to` VARCHAR(100),
        `fromPort` VARCHAR(100),
        `toPort` VARCHAR(100),
        `visibility` VARCHAR(100),
        `state` VARCHAR(100)
);
```

```
CREATE TABLE ‘el_23‘ (
        ‘id‘ mediumint NOT NULL,
        ‘name‘ VARCHAR(30),
        ‘from‘ VARCHAR(100),
        ‘to‘ VARCHAR(100),
        ‘fromPort‘ VARCHAR(100),
        ‘toPort‘ VARCHAR(100),
        ‘isSubstitutable‘ BOOL,
        ‘state‘ VARCHAR(100)
);

CREATE TABLE ‘el_24‘ (
        ‘id‘ mediumint NOT NULL,
        ‘name‘ VARCHAR(30)
);

CREATE TABLE ‘el_25‘ (
        ‘id‘ mediumint NOT NULL,
        ‘name‘ VARCHAR(30),
        ‘isAbstract‘ BOOL,
        ‘visibility‘ VARCHAR(100),
        ‘qualifiedName‘ VARCHAR(100),
        ‘state‘ VARCHAR(100),
        ‘isLeaf‘ BOOL
);

CREATE TABLE ‘el_26‘ (
        ‘id‘ mediumint NOT NULL,
        ‘name‘ VARCHAR(30),
        ‘body‘ VARCHAR(1000),
        ‘language‘ VARCHAR(1000)
);

CREATE TABLE ‘el_27‘ (
        ‘id‘ mediumint NOT NULL,
        ‘name‘ VARCHAR(30),
        ‘body‘ VARCHAR(1000),
        ‘language‘ VARCHAR(1000)
);

CREATE TABLE ‘el_28‘ (
        ‘id‘ mediumint NOT NULL,
        ‘name‘ VARCHAR(30),
```

```
        `isAbstract` BOOL,
        `visibility` VARCHAR(100),
        `qualifiedName` VARCHAR(100),
        `state` VARCHAR(100),
        `isLeaf` BOOL
);

CREATE TABLE `el_29` (
        `id` mediumint NOT NULL,
        `name` VARCHAR(30),
        `isAbstract` BOOL,
        `visibility` VARCHAR(100),
        `qualifiedName` VARCHAR(100),
        `state` VARCHAR(100),
        `isLeaf` BOOL
);

CREATE TABLE `el_30` (
        `id` mediumint NOT NULL,
        `name` VARCHAR(30),
        `isAbstract` BOOL,
        `visibility` VARCHAR(100),
        `qualifiedName` VARCHAR(100),
        `state` VARCHAR(100),
        `isLeaf` BOOL
);

CREATE TABLE `el_31` (
        `id` mediumint NOT NULL,
        `name` VARCHAR(30),
        `from` VARCHAR(100),
        `to` VARCHAR(100),
        `fromPort` VARCHAR(100),
        `toPort` VARCHAR(100),
        `visibility` VARCHAR(100),
        `qualifiedName` VARCHAR(100),
        `state` VARCHAR(100)
);

CREATE TABLE `el_32` (
        `id` mediumint NOT NULL,
        `name` VARCHAR(30),
        `from` VARCHAR(100),
        `to` VARCHAR(100),
```

```
        `fromPort` VARCHAR(100),
        `toPort` VARCHAR(100),
        `state` VARCHAR(100)
);
```

# В Пример XML-описания редактора

Данное XML-описание, схема которого придумана Симоновой Александрой, задаёт диаграмму случаев использования. Оно наследует некоторые базовые элементы метамодели UML, включает модули `kernel_metamodel` и `basicbehaviors_metamodel`, и наследуя от них некоторые сущности, задает элементы диаграммы.

```
<?xml version="1.0" encoding="utf-8" ?>
<metamodel xmlns="http://schema.real.com/schema/">
  <include>kernel_metamodel</include>
  <include>basicbehaviors_metamodel</include>
  <namespace>UML 2.1</namespace>
  <diagram name="USE CASE DIAGRAM">
    <node name="Actor" id="uscnActor">
      <graphics>
        <view>
          <svg_shape>
            <svg:svg width="40"
                    xmlns:xlink="http://www.w3.org/1999/xlink"
                    height="100" version="1.0">
              <svg:circle cx="16" cy="16" r="15" stroke="black"
                    stroke-width="2" fill="white"/>
              <svg:line x1="16" y1="31" x2="16" y2="66"
                    stroke="black" stroke-width="2" />
              <svg:line x1="1" y1="45" x2="31" y2="45"
                    stroke="black" stroke-width="2" />
              <svg:line x1="16" y1="66" x2="1" y2="89"
                    stroke="black" stroke-width="2" />
              <svg:line x1="16" y1="66" x2="31" y2="89"
                    stroke="black" stroke-width="2" />
            </svg:svg>
          </svg_shape>
          <repo_info>
            <html:html xmlns:html="http://www.w3.org/html/">
              <html:center>
                <html:text_from_repo name="name" />
              </html:center>
            </html:html>
          </repo_info>
          <ports>
            <point_port x="1" y="45" />
            <point_port x="31" y="45" />
            <line_port>
              <start startx="16" starty="50"/>
```

```
          <end endx="16" endy="66"/>
        </line_port>
      </ports>
    </view>
  </graphics>
  <logic>
    <generalizations>
      <generalization>
        <parent parent_id="bbnBehavioredClassifier"></parent>
      </generalization>
    </generalizations>
    <properties>
      <property name="name" type="string"></property>
    </properties>
    <containers>
      <container>
        <contains idref="uscnUseCase" role="use case"/>
      </container>
    </containers>
  </logic>
</node>
<node name="UCClassifier" id="uscnUCClassifier">
  <graphics>
    <view>
      <svg_shape>
        <svg:svg width="60"
                 xmlns:xlink="http://www.w3.org/1999/xlink"
                 height="110" version="1.0">
          <svg:rect x="1" y="1" width="58"
              style="stroke:#000000;fill:#f8f8f8;fill-opacity:0;"
              height="98"/>
        </svg:svg>
      </svg_shape>
      <repo_info>
        <html:html xmlns:html="http://www.w3.org/html/"
            x="5" y="5">
            <html:text_from_repo name="name"/>
        </html:html>
      </repo_info>
    </view>
  </graphics>
  <logic>
    <generalizations>
      <generalization>
```

```
        <parent parent_id="krnnClassifier"></parent>
      </generalization>
    </generalizations>
    <associations>
      <assoc_ref idref="uscaClassifier" />
    </associations>
    <containers>
      <container>
        <contains idref="uscnUseCase" role="use case" />
      </container>
    </containers>
    <properties>
      <property name="name" type="string"></property>
    </properties>
  </logic>
</node>
<node name="UseCase" id="uscnUseCase">
  <graphics>
    <view>
      <svg_shape>
        <svg:svg width="100"
             xmlns:xlink="http://www.w3.org/1999/xlink"
             height="60" version="1.0">
          <svg:ellipse cx="50" cy="25" rx="48" ry="24"
   style="fill:#ffffff;stroke:#000000;stroke-width:2" />
        </svg:svg>
      </svg_shape>
      <repo_info>
        <html:html xmlns:html="http://www.w3.org/html/"
          x="0" y="20">
          <html:center>
            <html:text_from_repo name="name"/>
          </html:center>
        </html:html>
      </repo_info>
      <ports>
        <point_port x="72" y="46" />
        <point_port x="28" y="4" />
        <point_port x="28" y="46" />
        <point_port x="72" y="4" />

        <point_port x="90" y="38" />
        <point_port x="10" y="12" />
        <point_port x="10" y="38" />
```

```
          <point_port x="90" y="12" />

          <point_port x="80" y="44" />
          <point_port x="20" y="6" />
          <point_port x="20" y="44" />
          <point_port x="80" y="6" />

          <point_port x="98" y="25" />
          <point_port x="2" y="25" />

          <point_port x="50" y="49" />
          <point_port x="50" y="1" />
        </ports>
      </view>
    </graphics>
    <logic>
      <generalizations>
        <generalization>
          <parent parent_id="bbnBehavioredClassifier"></parent>
        </generalization>
      </generalizations>
      <properties>
        <property name="name" type="string"></property>
      </properties>
      <associations>
        <assoc_ref idref="uscaExtend"/>
        <assoc_ref idref="uscaInclude"/>
      </associations>
      <containers>
        <container>
          <contains idref="uscnExtensionPoint"
                role="extension point" />
        </container>
        <container>
          <contained_by idref="uscnActor" role="actor"/>
        </container>
      </containers>
    </logic>
  </node>
  <edge name="Extend" id="uscaExtend">
    <graphics>
      <view>
        <line_type type="dashLine" />
        <repo_info>
```

```
          <html:html xmlns:html="http://www.w3.org/html/">
            <html:center>
              <html:text text="&lt;&lt;extend&gt;&gt;"/>
            </html:center>
          </html:html>
        </repo_info>
      </view>
    </graphics>
    <logic>
      <generalizations>
        <generalization>
          <parent parent_id="krnnNamedElement"></parent>
          <parent parent_id="krneDirRelationship"></parent>
        </generalization>
      </generalizations>
      <associations>
        <association id="uscaExtend" role="extend"
             multiplicity="true">
          <begin idref="uscnUseCase"/>
        </association>
        <association id="uscaExtend" role="extend"
             multiplicity="false" end_type="open_arrow">
          <end idref="uscnUseCase"/>
        </association>
      </associations>
    </logic>
  </edge>
  <edge name="Include" id="uscaInclude">
    <graphics>
      <view>
        <line_type type="dashLine" />
        <repo_info>
          <html:html xmlns:html="http://www.w3.org/html/">
            <html:center>
              <html:text text="&lt;&lt;include&gt;&gt;"/>
            </html:center>
          </html:html>
        </repo_info>
      </view>
    </graphics>
    <logic>
      <generalizations>
        <generalization>
          <parent parent_id="krneDirRelationship"></parent>
```

```
            <parent parent_id="krnnNamedElement"></parent>
          </generalization>
        </generalizations>
        <associations>
          <association id="ucsaInclude" role="include"
               multiplicity="true">
            <begin idref="uscnUseCase"/>
          </association>
          <association id="uscaInclude" role="include"
               multiplicity="false" end_type="open_arrow">
            <end idref="uscnUseCase"/>
          </association>
        </associations>
      </logic>
    </edge>
  </diagram>
</metamodel>
```

# Список литературы

[1] *Сэлик Б.* Unified Modeling Language, версия 2.0 // *IBM developerWorks.* — 2005. http://www.ibm.com/developerworks/ru/library/321_uml/index.html.

[2] *Iseger M.* Domain-specific modeling for generative software development // *IT Architect.* — 2005. http://www.itarchitect.co.uk/articles/display.asp?id=161.

[3] *Lutz M.* Programming Python. — O'Reilly & Associates Inc., 2001.

[4] *Китаев Е., Кузьмичев Д., Слепенков М.* Особенности реализации насыщенных пользовательских интерфейсов веб-приложений // *Препринт института прикладной математики им. М.В.Келдыша.* — 2006. http://www.keldysh.ru/papers/2006/prep75/prep2006_75.html.

[5] *Fowler M.* Patterns of Enterprise Application Architecture. — Addison-Wesley, 2003.

[6] What's a Controller, Anyway? http://c2.com/cgi/wiki?WhatsaControllerAnyway

[7] Model-View-Controller // *MSDN reference guide.* — 2007. http://msdn2.microsoft.com/en-us/library/ms978748.aspx.

[8] *Summerfield M.* Qt 4's Model/View Delegates // *Qtrac Ltd. set of training materials.* — 2006. http://www.ics.com/developers/papers/.

[9] *Dalheimer M. K.* A Comparison of Qt and Java for Large-Scale, Industrial-Strength GUI Development. — 2004. http://turing.iimas.unam.mx/~elena/PDI-Lic/qt-vs-java-whitepaper.pdf.

[10] Scalable Vector Graphics (SVG) Tiny 1.2 Specification. — 2006. http://www.w3.org/TR/2006/CR-SVGMobile12-20060810/.

[11] *Lee S.-D., Park H.-C.* Modeling and Management of UML Diagrams based on Relational Database // Modelling and Simulation Proceedings. — 2003. http://www.actapress.com/PaperInfo.aspx?PaperID=13132.