Санкт-Петербургский Государственный Университет

Математико-механический факультет

Кафедра системного программирования

Реализация настраиваемого графического представления элемента на диаграмме в QReal

Курсовая работа студента 445 группы

Мордвинова Дмитрия Александровича

Научный руководитель

ст. пр. Брыксин Т.А.

Санкт-Петербург

2012

Оглавление

Введение

В настоящее время для разработки программного и аппаратного обеспечения довольно активно используется визуальное моделирование. Для автоматизации процесса разработки применяются CASE-средства, в которых разрабатываемая система представляется в виде модели на каком-либо визуальном языке программирования (UML[13, 14], IDEF[3], BPMN[6]). В целом данный подход упрощает процесс разработки и понимания системы, делает его более наглядным.

Однако, несмотря на эти преимущества и постоянное развитие этого подхода, разработчики в настоящее время довольно редко используют визуальное программирование. Существует мнение, что причиной этому является неудобство использования CASE-систем, что значитиельно снижает производительность труда.

В некоторых ситуациях производительность может быть повышена за счет использования предметно-ориентированного моделирования (DSM, Domain-Specific Modeling). Данный подход основывается на том факте, что чаще создание нового специального языка и решение поставленной задачи с его помощью можно осуществить быстрее, чем решать ту же задачу с помощью языков общего назначения. В данном случае, наглядность и удобство использования целевого редактора важнее, чем строгость и формальность, присущие языкам общего назначения.

QReal – metaCASE система, разрабатываемая на кафедре системного программирования математико-механического факультета СПбГУ. Основной ее идеей является генерация редакторов для конкретной предметной области. Сгенерированные редакторы работают на том же движке, что и метаредактор, таким образом, наследуя все «удобства» системы. Понятно, что для решения подобных задач требуется удобное и понятное инструментальное стредство, позволяющее быстро задавать и выполнять действия, типичные для данного редактора. Данная работа ведется применительно к QReal.

В проекте QReal уже существует ряд средств, ускоряющих и упрощающих работу проектировщика (редактор форм, распознование жестов мыши, визуальный отладчик и т.д.), однако остается открытой проблема быстрого и удобного редактирования свойств элементов. Если проектировщик хочет изменить какое-либо свойство, ему придется переключаться на редактор свойств, искать это свойство там и только после этого установить значение свойства.

Частично эта проблема была решена введением динамических меток на элементе. По отзывам команды разработчиков, это сделало процесс редактирования значения свойста намного более быстрым и удобным. Однако значение набирается на клавиатуре как строка, что не очень удобно, а также увеличивает вероятность возникновения ошибки. Например, значение перечислимого типа легче было бы выбрать из выпадающего списка, чем набирать его на клавиатуре. Все эти рассуждения указывают на необходимость возможности иметь элементы управления (виджеты) на элементе.

Вследствие реализации предметно-ориентированного подхода системой QReal, нельзя ограничиваться каким-либо предопределенным набором элементов управления. Необходимо реализовать

какое-либо средство создания пользовательских элементов управления. Это значительно расширит спектр возможностей системы.

Обзор существующих решений

В целом, системы визуального моделирования можно поделить на два класса - универсальные и предметно-ориентированные системы. Рассмотрим несколько типичных представителей из каждого класса и сделаем конкретные выводы применительно к теме ранней работы о каждом из них.

Visual Paradigm

Visual Paradigm[21] – это мощное кроссплатформенное UML CASE средство визуального моделирования. Оно поддерживает UML 2, генерацию кода, а также проводить reverse engeneering диаграмм из кода. Является одной из самых популярных CASE-систем. Обсуждаемая функциональность в нем реализована отчасти - разрешено менять имя классов и аттрибутов прямо на сцене. Аттрибуты добавляются из окон с крайне большим количеством содержимого (рис. 1), что крайне неудобно и затрудняет процесс разработки диаграммы. Точно так же дело обстоит с практически всеми редакторами UML. Улучшить положение невозможно, причина в специфике UML. Напомним, что UML относится к языкам общего назначения, а следовательно должен учитавать большое количество аспектов проектируемой системы.

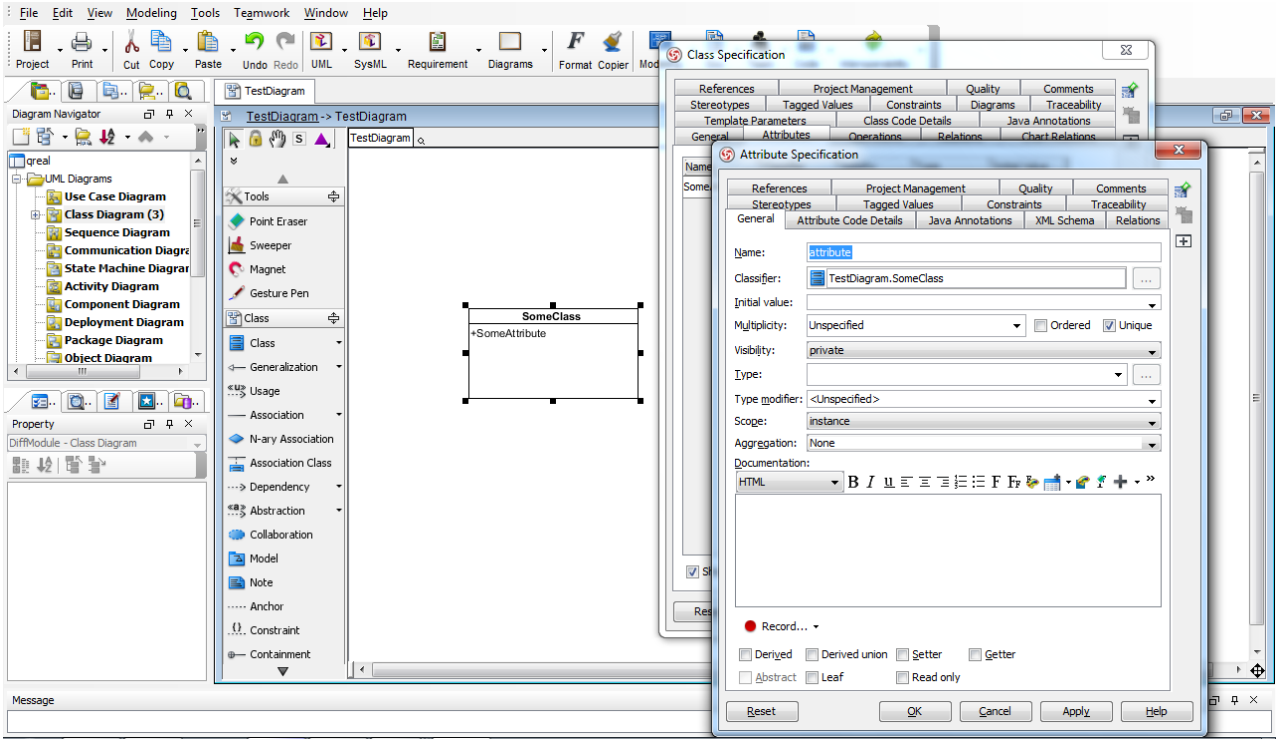

Рисунок 1. Пользовательский интерфейс системы Visual Paradigm

IBM Rational Rose

IBM Rational Rose[15] - популярное средство визуального моделирования, входит в состав пакета IBM Rational Suite и предназначен для моделирования программных систем с использованием широкого круга инструментальных средств и платформ. Являясь простым и мощным решением для визуальной разработки информационных систем любого класса, Rational Rose позволяет создавать, изменять и проверять корректность модели.

Rational Rose также относится к универсальным CASE-системам. За ненадобностью, интерактивных виджеты на диаграммах отсутствуют.

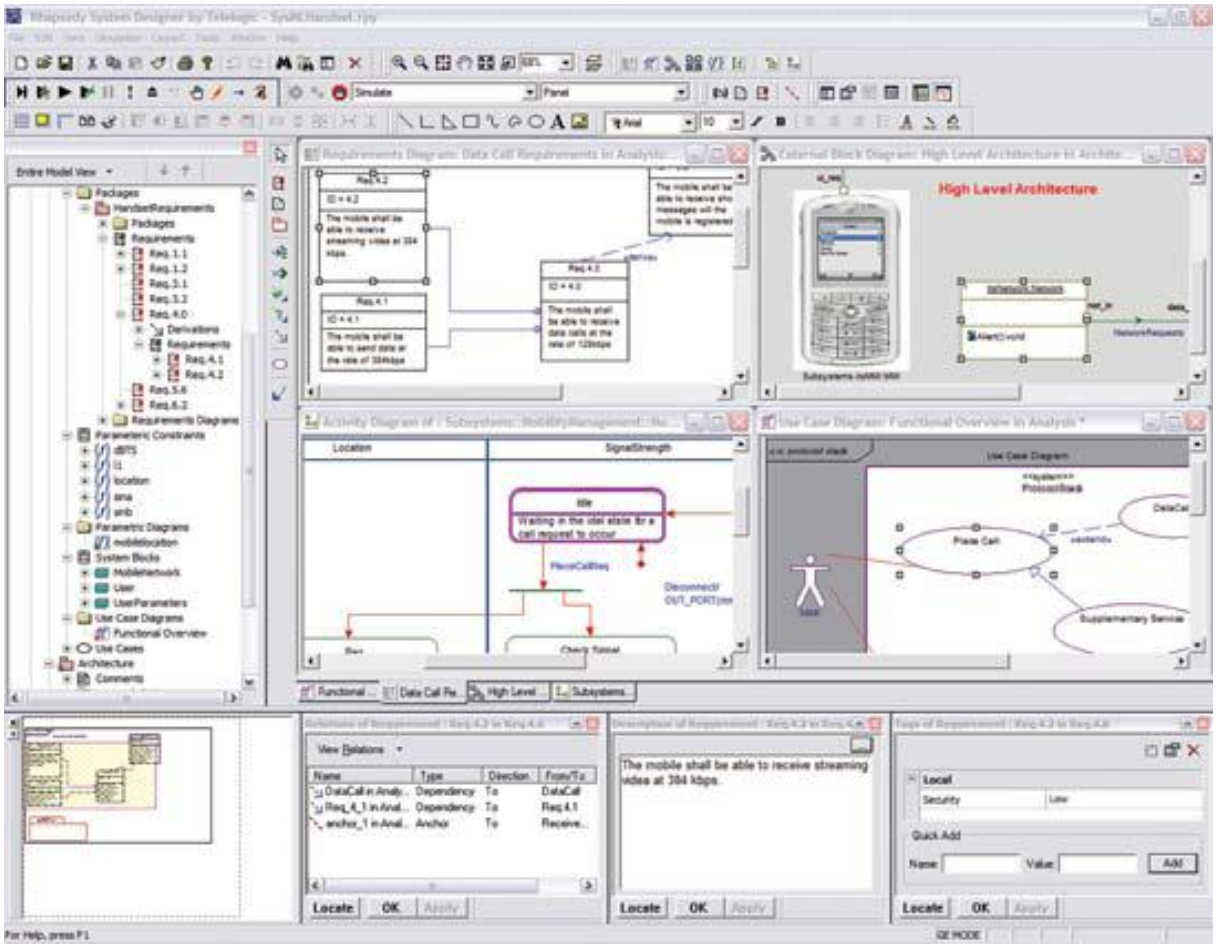

Рисунок 2. Пользовательсьский интерфейс системы IBM Rational Rose.

Таким образом, универсальные CASE-средства не нуждаются в большой интерактивности элемента. Решающим фактором здесь является строгость и формальность, следование стандартам, диаграммы громоздки, поэтому, если дополнять элемент еще чем-либо, даже повышающим скорость разработки, это может серьезно повредить читаемости диаграмм, что намного более важно.

Microsoft DSL Tools

Инструментарий Microsoft DSL Tools[11] является частью среды разработки Visual Studio и предназначен для создания встроенных в Visual Studio визуальных редакторов. Процесс создания нового редактора со- стоит из описания метамодели языка на специальном визуальном языке DSL Tools, генерации редактора и внесения в сгенерированный код ручных изменений для реализации дополнительной функциональности, такой как валидация моделей. Сохранение ручных изменений после перегенерации редактора осуществляется за счёт использования частичных классов (partial classes) .Net.

Внешний вид элемента позволяет варьировать стиль текста, “схлопывающиеся области” и изображения на элементе. Никакой речи об элементах управления нет. [1]

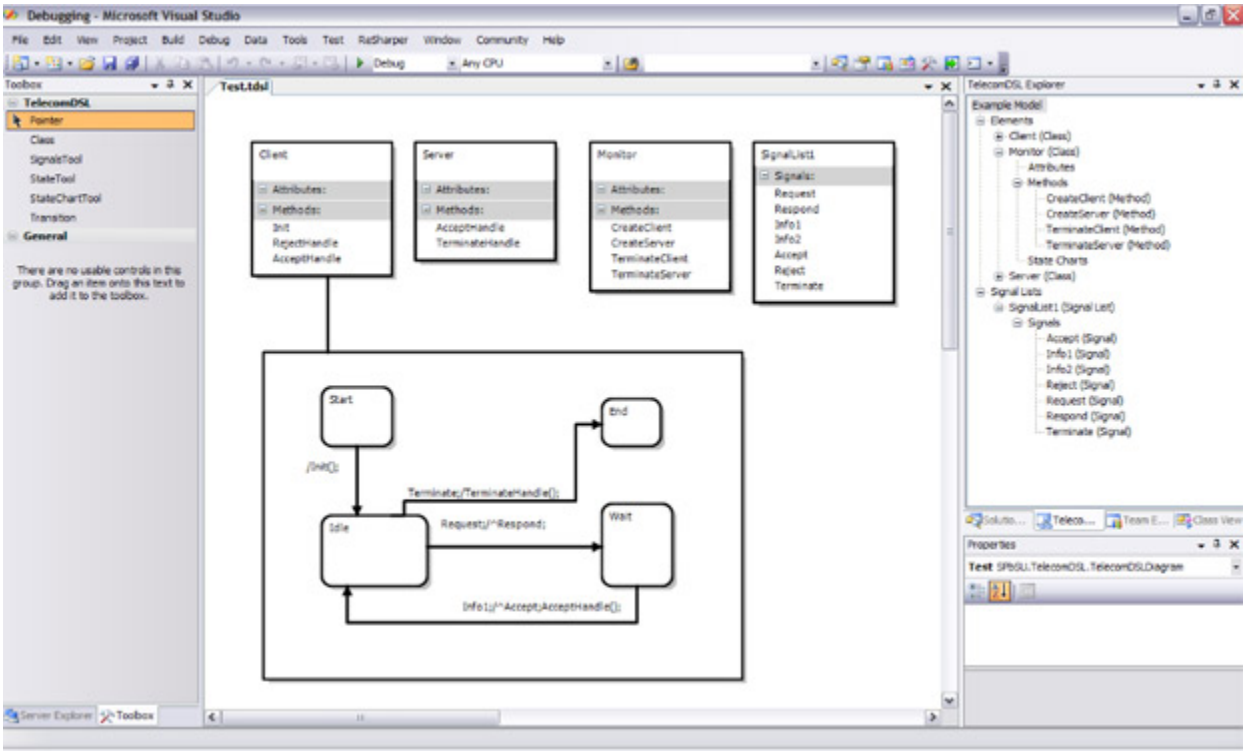

Рисунок 3. Редактор, созданный с помощью Microsoft DSL Tools.

Eclipse GMF

Технология Eclipse GMF (Graphical Modelling Framework[9]) создана на базе среды разработки с открытыми исходными кодами Eclipse. GMF предназначена для быстрой разработки графических редакторов, в основном, интегрируемых в Eclipse, и построена на двух широко используемых библиотеках — Eclipse Modeling Framework (EMF)[7] и Graphical Editing Framework (GEF)[8]. Архитектура построенного с помощью GMF пакета основана на шаблоне проектирования Model/View/Controller (MVC), где для создания моделей используется EMF, а для создания представлений и контроллеров используется GEF.

dВиджеты могут быть добавлены на элемент, но это требует большого объема кодирования. В основном, внешний вид элемента задается с использованием GEF, который позволяет лишь создать форму элемента с некоторыми эдементами интерактивности.

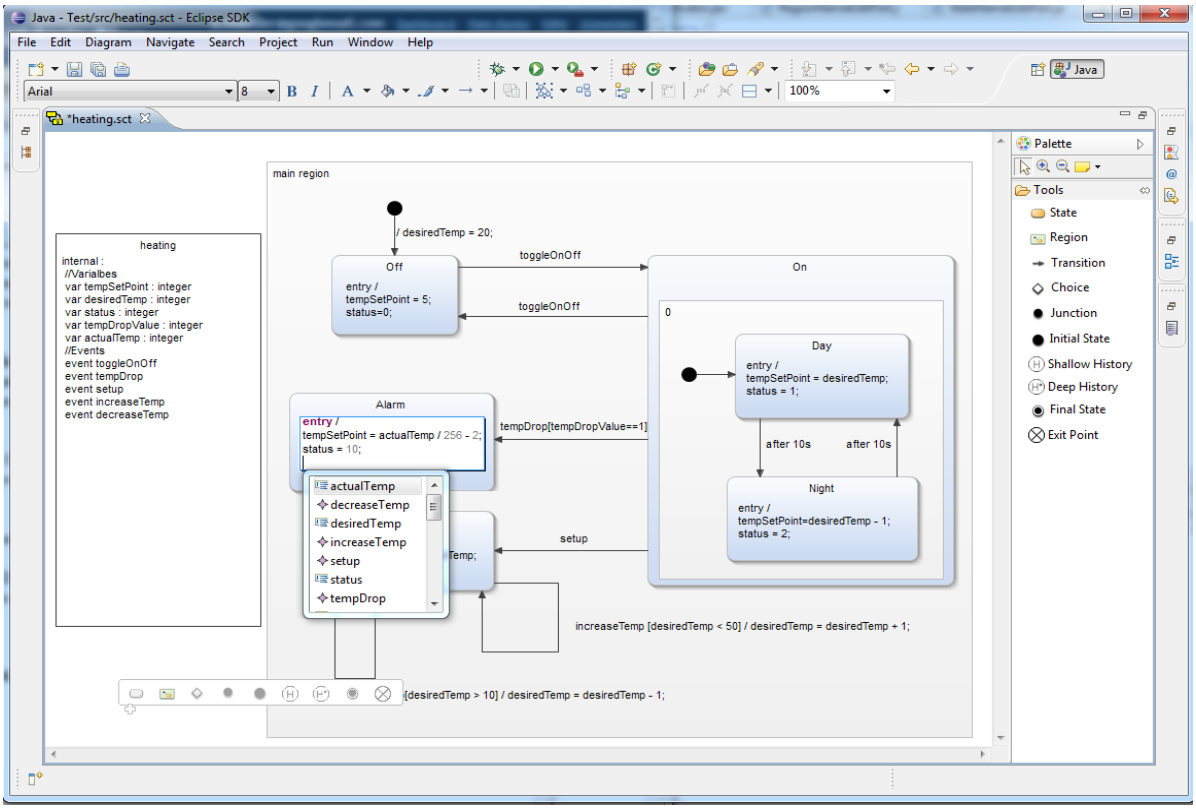

Рисунок 4. Редактор, соданный с помощью Eclipse GMF.

Таким образом, поддержка виджетов в предметно-ориентированных CASE-системах либо не реализована вовсе, либо неоправданно затруднена. Во многих случаях это может помешать созданию удобного редактора, что приоритетно для предметно-ориентированных систем.

Рассмотрим удачный с точки зрения данной работы пример.

NXT-G

Система NXT-G[10] - графическая среда программирования поведения роботов, входящая в комплект LEGO Mindstorms - конструктора для создания роботов. Элементы в ней обладают большой интерактивностью, при этом назначение всех элементов управления на элементе инуитивно понятны[12].

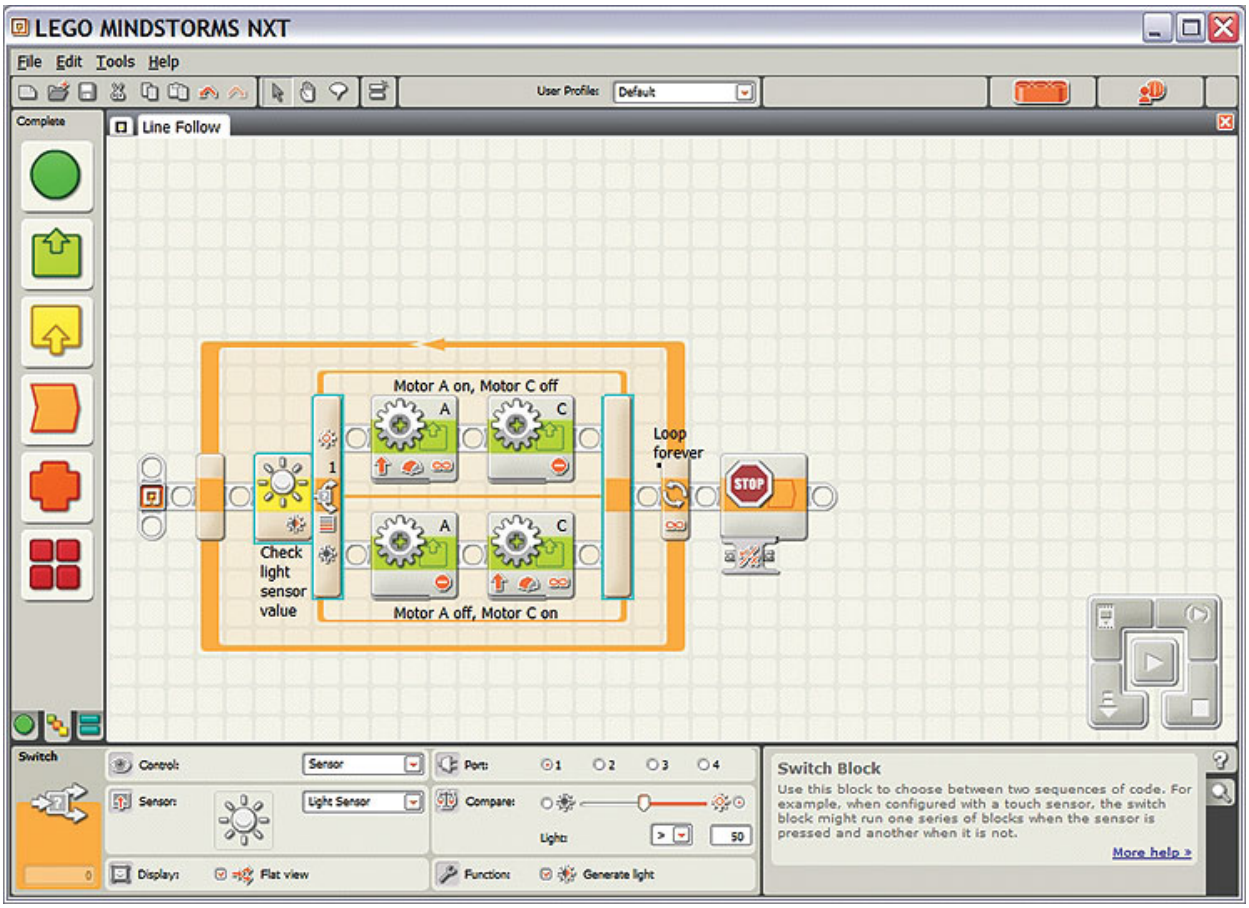

Рисунок 5. Пользовательский интерфейс системы NXT-G

Приоритетным на данный момет направлением развития системы QReal является проект QReal:Robots. Это среда обучения основам программирования и кибернетики. Подобно NXT-G, QReal:Robots может использоватья для программирования поведения роботов.

Другие технологии

Так как важной частью данной работы была адаптация и доработка частей существующих библиотек,проведем краткий экскурс по ним.

Qt Graphics View Framework

Система Graphics View Framework, появившаяся в Qt, начиная с версии 4.2 (на момент написания работы использовалась версия 4.7.4), пришла на смену графической системе, основанной на классе QCanvas. Она не только предоставляет средства по созданию, управлению и взаимодействию графическими элементами, но еще и реализует шаблон «модель-представление» для программ, работающих с двухмерной

графикой. Основу Graphics View Framework составляют три класса Qt Library, представленные на схеме (рис. 6).

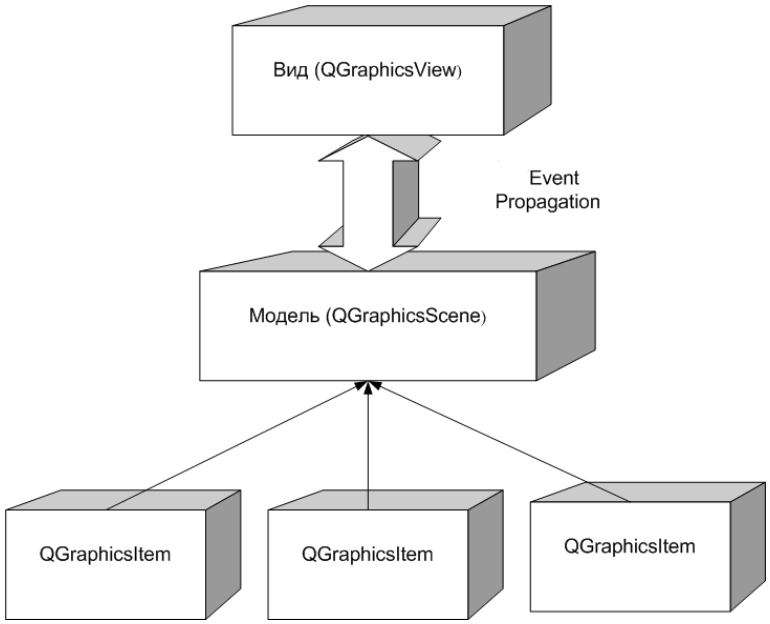

Рисунок 6. Общая схема Graphics View Framework.

Модель данных реализована с помощью объекта класса QGraphicsScene. Элементами модели данных являются графические примитивы (геометрические фигуры и растровые изображения). Все графические примитивы реализованы с помощью классов-потомков класса QGraphicsItem. Таким образом, объект класса QGraphicsScene можно рассматривать как контейнер для набора объектов классов-потомков QGraphicsItem. Для отображения модели, созданной в QGraphicsScene, служит объект класса QGraphicsView. Работая в системе Graphics View Framework, вы не рисуете изображение непосредственно в окне QGraphicsView (хотя в принципе можете это делать). Вместо этого вы управляете объектами, хранящимися в модели QGraphicsScene. Все изменения объектов модели автоматически отображаются в окне QGraphicsView. При этом вам не нужно заботиться о таких вещах как перерисовка изображения при изменении размеров окна. Поскольку объект класса QGraphicsView связан с моделью, он «знает», что нужно отображать в окне, и обновляет содержимое окна автоматически. Вторая важная задача, которую решает связка объектов QGraphicsView и QGraphicsScene – преобразование действий пользователя (таких, как щелчок мышью, перемещение курсора мыши над объектом или нажатие клавиши) в события модели. События модели могут быть переданы далее отдельным примитивам, формирующим модель. Эта система передачи событий между разными уровнями Graphics View Framework именуется в документации Qt термином event propagation

Встраивание виджетов

Начиная с Qt 4.4, система Graphics View Framework обогатилась еще одной весьма интересной

возможностью. Речь идет о встраивании виджетов в графическую сцену Встроенные виджеты не теряют своей функциональности. Благодаря механизму передачи событий Graphics View Framework встроенные виджеты реагируют на действия пользователя точно так же, как обычные (основанные на QWidget). Одновременно с этим встроенные виджеты обладают свойствами графических примитивов Graphics View Framework. Со встроенными виджетами можно выполнять те же геометрические преобразования, что и с остальными примитивами, для них также работает обнаружение коллизий и другие функции графической системы.

Элемент со встроенным виджетом поддерживает синхронизацию своих свойств с самим виджетом. Например, если прокси-виджет невидим или неактивен, вложенный виджет также будет невидим или неактивен, и наоборот. Синхронизация происходит в момент вложения виджета в прокси-виджет и поддерживается все то время, пока виджет вложен в него. Синхронизируются такие параметры, как активность, видимость, размер, позиция, направление компоновкаа, стиль, палитра, шрифт, форма курсора и т.д.

Как и обычные виджеты, прокси-виджеты могут иметь свою компоновку. Для этого используется класс QGraphicsLayout, чей API очень похож на API класса QLayout. Можно использовать существующие классы QGraphicsLinearLayout и QGraphicsGridLayout, а можно создать свою компоновку.

Для более подробного ознакомления с Qt Graphics View Framework см. документацию Qt [18].

Qt Property Browser Framework

Qt Property Browser Framework - еще один инструментарий в составе Qt SDK. Он предназначен для быстрого и удобного создания редактора свойств в приложении. Он включает в себя собственно виджеты-браузеры, менеджеры свойств, фабрики свойств и сами свойства. Реализованы две разновидности пользовательского интерфейса предоставляемых редакторов: древовидный (QtTreePropertyBrowser) и на основе group box`ов (QtGroupBoxPropertyBrowser) (рис. 7).

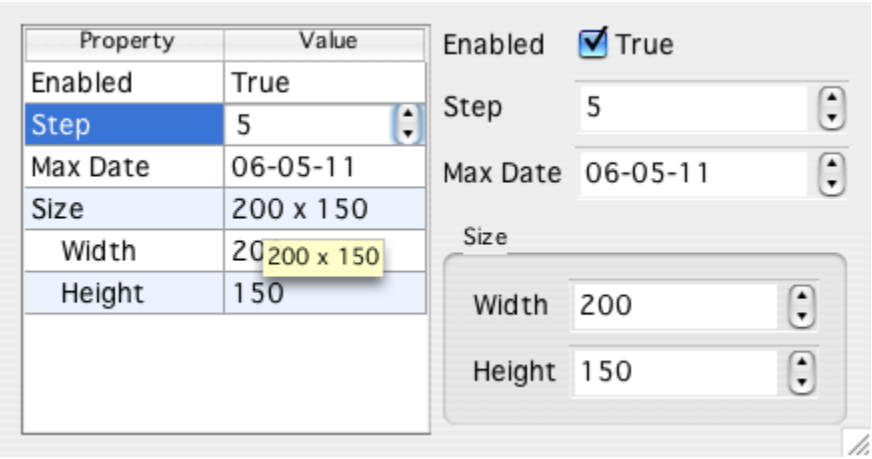

Рисунок 7. Виджеты-браузеры (слева - древовидный, справа - на основе group box`а)

В добавок, можно, используя класс QtAbstractPropertyBrowser, создать свой собственный браузер.

Пользовательские виджеты-редакторы создаются фабрикой, производной от класса QtAbstractEditorFactory.

Перед редактированием, свойства должны быть инкапсулированы в объекты класса QtProperty. Их экземпляры создаются и обслуживаются менеджерами, производными от класса QtAbstractPropertyManager.

Инструментарий предлагает менеджеры для наиболее часто используемых типов (например, QtIntPropertyManager). Сами свойства создаются вызовом метода addProperty() у менеджера. Например, метод addProperty() у класса QtIntPropertyManager возвращает заново созданное свойство типа QtIntProperty. Каждое свойство имеет общие характеристики, такие, как имя, всплывающий текст, текст для подсказки.Для установки и изменения значений и рамок свойств каждый менеджер имееет свой специфичный для типа интерфейс.

Свойства могут быть разбиты на категории с помощью класса QtGroupPropertyManager, а также иметь подсвойства (добавляются с помощью метода addSubProperty()).

Для связки менеджеров свойств с самим редактором используются фабрики. Фабрики производят редакторы. По аналогии с менеджерами, предлагаются часто используемые фабрики (например, QtSpinBoxFactory для целочисленных типов и QtEnumEditorFactory для перечислений, которые производят редакторы, основанные на виджетах QSpinBox и QComboBox соответственно). Связывание фабрики с менеджером гарантирует, что когда бы ни было создано свойство с помощью этого менеджера, оно будет редактироваться с использованием указанной фабрики.

Помимо явного указания типа свойства, менеджера и фабрики, инструментарий предлагает другой, более удобный подход. Он основан на типе QVariant. При этом подходе используются три класса:

- QtVariantProperty - наследует QtProperty, но имеет обогащенный интерфейс, позволяющий напрямую запрашивать и изменять значения и аттрибуты свойств.
- QtVariantPropertyManager - может быть использован для всех типов свойств, поддерживаемых классом QVariant.
- QtVariantEditorFactory - производит различные редакторы для типов, поддерживаемых классом QtVariantPropertyManager.

  Для более подробного ознакомления с Qt Property Browser Framework см. документацию[20].

QReal

Теперь рассмотрим, как описанние технологии применены в реализации системы QReal.

Глобально архитектуру системы QReal можно охарактеризовать как “модель-представление”[19]. Модель представлена репозиторием, представление можно разделить на несколько частей:

- Центральная сцена и мини-карта. Это ничто иное, как QGraphicsView с одной и той же сценой (как уже обсуждалось, это возможно, благодаря архитектуре “модель-представление” инструментарийа, следует лишь установить для центрального представления и мини-карты один и тот же экземпляр QGraphicsScene).
- Редакторы графической и логической моделей. Не представляют интереса для данной работы,

следовательно здесь обсуждаться не будут.

- Редактор свойств. Построен с использованием шаблона “модель-представление” с помощью Qt Property Browser Framework и введением еще одной модели, созданной специально для редактора свойств.

Далее речь пойдет об элементах. Базовые классы элементов на диаграмме находятся пакете “qrgui/umllib”. Их иерархия изображена на рисунке (рис. 8).

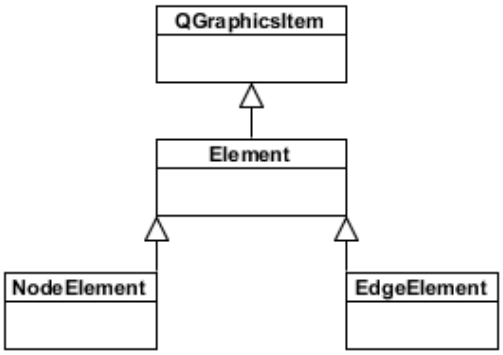

Рисунок 8. Старая иерархия базовых элементов в QReal

Класс Element является базовым для всех элементов на диаграмме. Он наследует класс QGraphicsItem, т.к. это дает возможность применять Graphics View Framework (также он наследует ElementRepoInterface для общения с репозиторием, но данного обсуждения это не касается, поэтому на диаграмме не указано). Element определяет простейшие свойства элементов (возможность быть выбранным, возможность передвигаться, всплывающие подсказки, курсор мыши при зависании над элементом, а также интерфейс, используемый репозиторием для сохранения информации об элементе).

Сам по себе Element не используется при создании конкретных элементов, зато он служит основой для двух других повсеместно используемых классов - NodeElement и EdgeElement. Оба этих класса являются базовыми для всех типов элементов на диаграмме и расширяются плагинами конкретного редактора.

Класс EdgeElement служит основой для всех ребер (стрелок и линий) на диаграмме (как известно, диаграмма в QReal - это граф со своими ребрами и узлами). Аспектов, подлежащих расширению плагинами относительно немного. Настраиваться могут только стиль (сплошной, пунктир и т.д.), цвет, толщина и форма стрелок ребра. Причина этому - поведение ребер во всех редакторах схоже и не нуждается в конкретизации.

С классом NodeElement дело обстоит намного сложнее. NodeElement - базовый класс для всех узлов на диаграмме. Если внешний вид и поведение для ребер предопределены, то здесь, как внешний вид, так и поведение могут быть совершенно разными и зависящими только от конкретного редактора. Аспектов для уточнения крайне много и перечислять их все не имеет смысла (так как для данной работы это не столь важно). Из основных можно выделить:

- порты (места, куда присоединяются ребра);
- внешний вид (размер, форма, графическое представление);

- поведение относительно дочерних элементов (может ли принимать, если да, то сколько, где и как располагать, как вести себя при изменении размера и т.д.)
- контектстные меню
- свойства

Конкретизация этих аспектов (так же, как и в EdgeElement) хранится в плагинах и происходит с помощью класса ElementImpl, который реализуют все элементы плагина. Для данной работы наиболее интересен аспект внешнего вида, конкретнее - графика на элементе.

На этапе метамоделирования пользователю дается возможность нарисовать прототип будущего элемента. Сделать это он может в редакторе форм, который открывается из редактора свойств элемента Node метаредактора при редактировании свойства shape. Форма - это комбинация графических примитивов (прямая, прямоугольник, еллипс, всевозможных кривых, пользовательских картинок, текста), редактируемых меток и портов. Сам редактор форм реализован также с помощью Graphics View Framework.

При сохрании редактируемая форма сериализуется в формате XML. Далее получившийся документ устанавливается в свойство shape элемента, сохраняется репозиторием, им же и выдается по необходимости..

При компиляции плагина нарисованные формы сохраняются на диск и используются системой при работе с разрабатываемым редактором. Прорисовкой формы на элементе занимается класс SdfRenderer. Его метод render() вызывается из перегруженного метода paint() класса NodeElement.

Таким образом, интерактивность элемента получается невеликой и ограничивается лишь редактируемыми метками. Они и используются для редактирования свойств прямо на элементе, что не всегда удобно.

Реализация

Теперь, когда имеется представление обо всех используемых технологиях и текущей реализации системы, изложим, что было предпринято для решения поставленной задачи и как это решение было реализовано.

В целом, работу можно поделить на две части:

1. создание редактора виджетов;
2. реализация поддержки виджетов в QReal.

Редактор виджетов

Как было отмечено, в модельно-ориентированном подходе одним из ключевых требований является быстрота и удобство редактирования. Как именно будет выглядеть желаемый редактор, заранее неизвестно, даже примерно. Поэтому нужно дать разработчику редактора возможность создавать свои виджеты.

Вследствие визуальной направленности среды QReal хотелось бы полностью отойти от текстового программирования. К счастью, подход к решению этой проблемы давно придуман и существует очень много его реализаций. Речь идет о визуальных конструкторах виджетов, которые реализованы практически в любом IDE. Нас бы полностью устроил редактор Qt Designer[2] (рис. 9). Он довольно удобен в использоваании, а также создает виджеты библиотеки Qt, что, конечно, больше всего подходит для системы QReal (она написана на Qt).

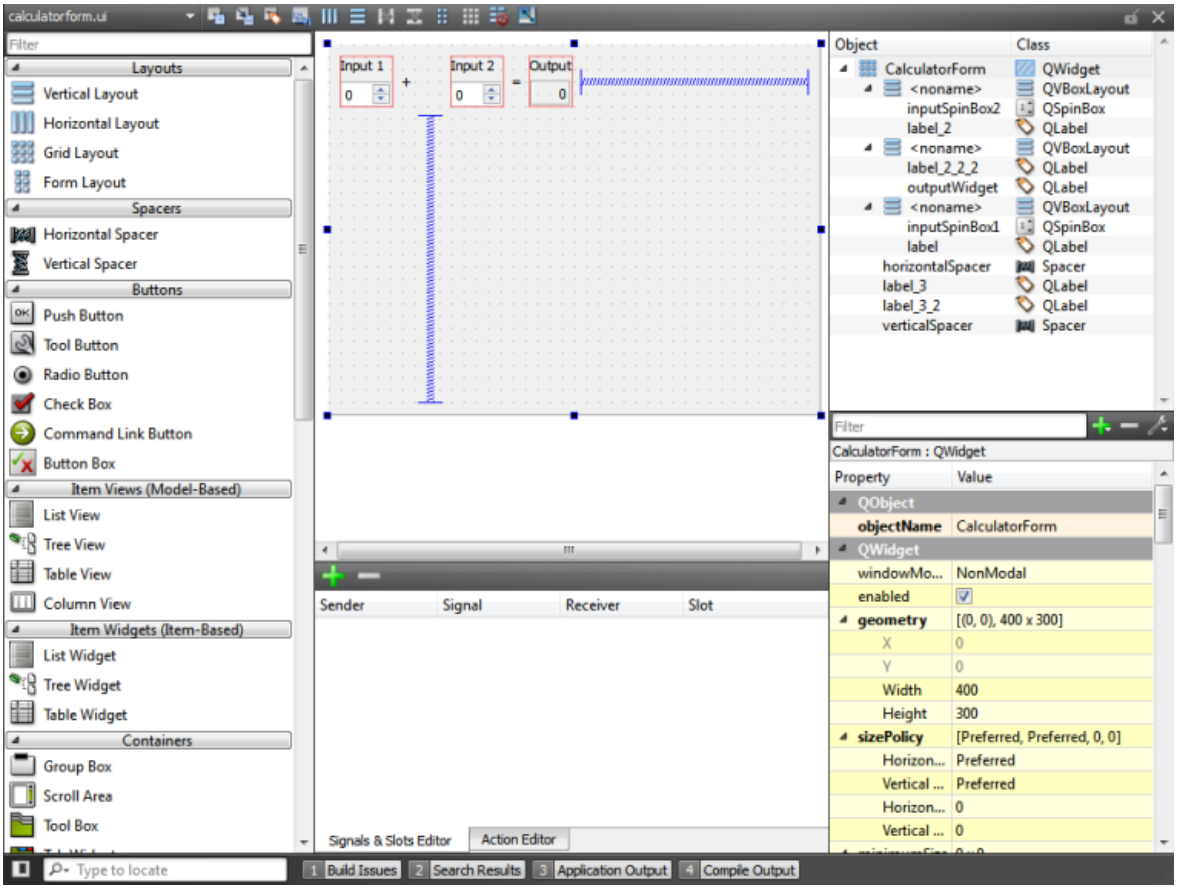

Рисунок 9. Пользовательский интерфейс Qt Designer.

К сожалению, помимо Qt Designer не существует открытого дизайнера, который мог бы подойти для QReal. Но и идея встраивания системы Qt Designer в QReal обладает множеством недостатков. Перечислим некоторые из них.

- Qt Designer довольно “тяжеловесный” продукт. Далеко не все его возможности в нашем случае нужны, поэтому не хотелось бы сильно увеличивать объем кода QReal.
- Могут понадобиться возможности, которых в Qt Designer нет (например, интеграция с уже существующим редактором форм QReal), реализация которых как дополнения может занять долгое время (вследствие большого количества незнакомого кода) или вообще невозможна (изначально другая направленность используемого продукта, неудачная для требуемых дополнений архитектура и т.п.).
- Qt Designer сильно привязан к IDE Qt Creator, отюда трудности интегрирования.
- Обычные минусы использования стороннего ПО (например, трудность поддерживать своими силами и, как следствие, зависимость от посторонней команды разработчиков).

В связи с описанными недостатками было принято создать свой редактор виджетов, похожий на Qt Designer, но реализующий ровно то, что нужно в QReal. Опишем реализованный редактор виджетов.

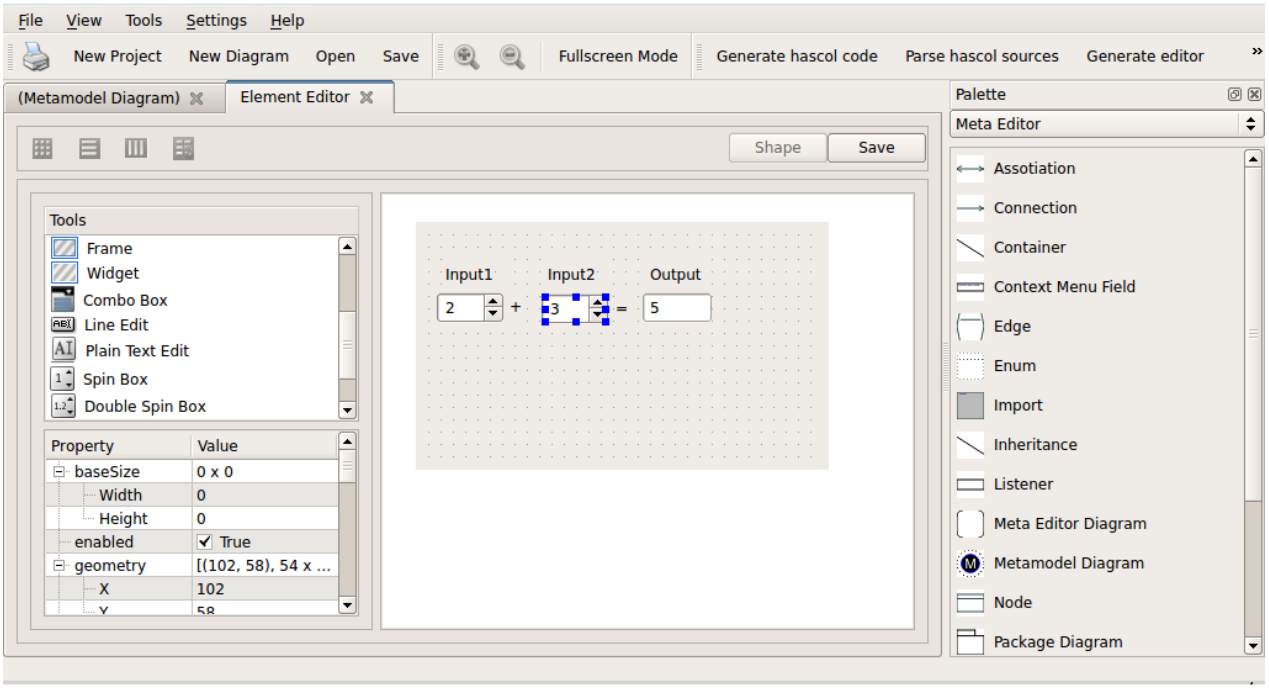

Рисунок 10. Пользовательский интерфейс редактора виджетов в QReal.

Редактор открывается тем же способом, что и старый редактор форм (редактор свойств элемента NodeElement метаредактора, в нем свойство template). Далее предлагается выбрать подход к созданию элемента: основанный на формах или на виджетах. Причина того, что сохранен старый функционал, даже при том, что новый намного мощнее и более общий, ясна: нужно сохранять совместимость с созданными в прошлом редакторами.

Как видно из рисунка 10, редактор открывется в отдельной вкладке. Интерфейс редактора виджетов можно разделить на четыре части:

- верхняя панель с кнопками выбора компоновкаов и командами;
- панель инструментов (примитивных виджетов);
- редактор свойств;
- сцена редактора.

Инструменты добавлятся на сцену и переупорядочиваются там перетаскиванием мышью. Все инструменты делятся на две части: обычные и контейнеры. В отличие от обычных, контейнеры позволяют добавлять на себя другие инструменты и устанавливать компоновкаы. Таким образом, множество инструментов на сцене представляет собой дерево с корнем в инструменте по умолчанию (Root).

К обычным иструментам относятся кнопки (PushButton, RadioButton, CheckBox), выпадающий список (ComboBox), строковый редактор (LineEdit), числовые редакторы (SpinBox, DoubleSpinBox) и т.п. К контейнерым относятся группирующий (GroupBox), рамка (Frame), прокручивающая область (ScrollArea), обычный виджет (Widget, просто серая область), редактор текста (RichTextBox) и метка (Label).

Каждый инструмент имеет свой набор свойств, значение каждого из которых может быть изменено

в редакторе свойств. Изменения моментально отображаются на сцене, равно, как и изменения свойств вне редактора (например, редактирование прямо на сцене) изменяется в редакторе свойств.

Инструменты-контейнеры позоляют устанавливать компоновку для расположения детей. Тип компоновки может быть одним из трех: сетка, вертикальное последовательное и горизонтальное последовательное. Для удобства, при добавлении на уже установленный компоновщик подсвечивается место, куда виджет будет добавлен при окончании операции перетаскивания (рис. 11).

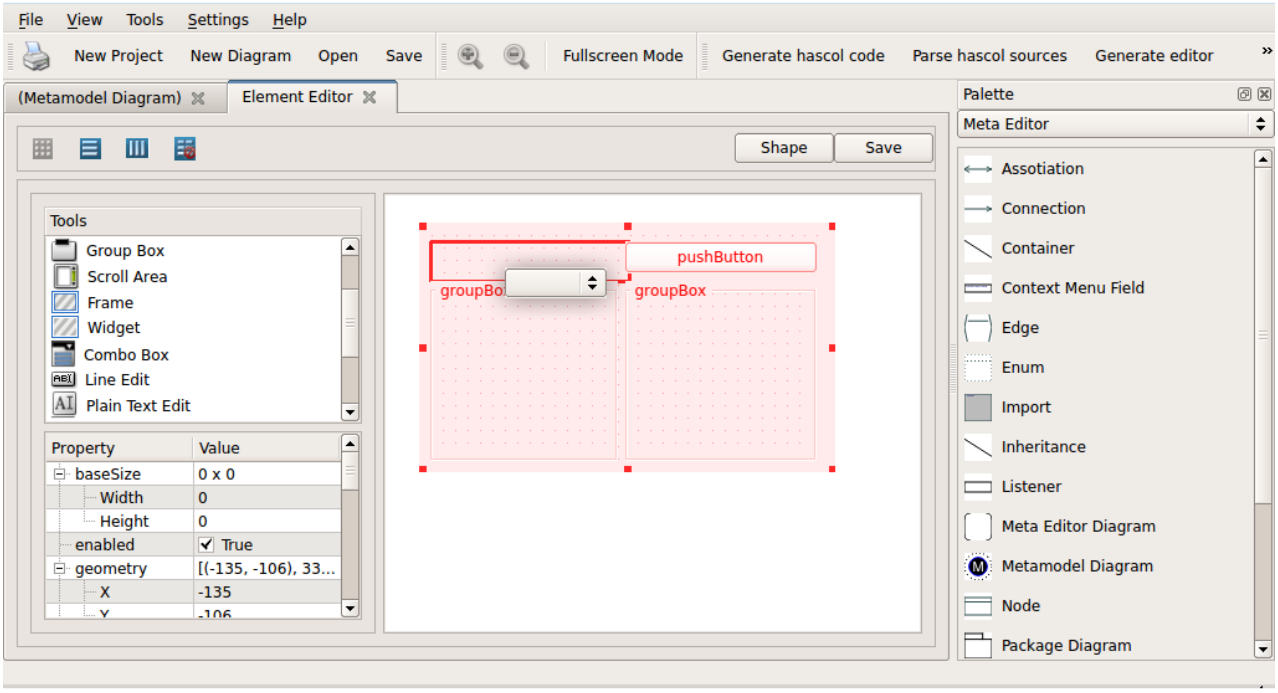

Рисунок 11. Подсветка места добавления при установленном сеточном компоновкае.

Редактор виджетов интегрирован с уже существующим редактором форм. Когда выбран корневой виджет, становится активной кнопка “shape” в правом углу верхней панели. При нажатии на нее открывается редактор форм. После редактирования формы и нажатия кнопки “Сохранить”, форма будет отображена на корневом виджете (рис. 12-14).

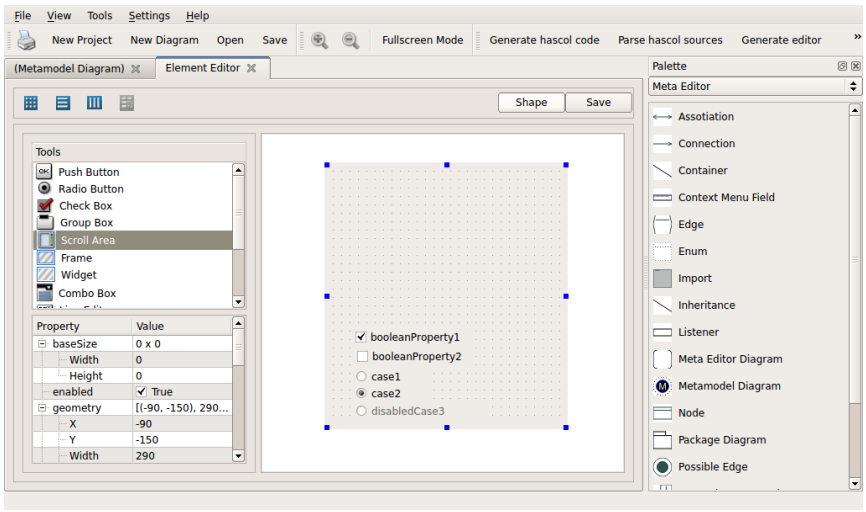

Рисунок 12. Редактируемый виджет.

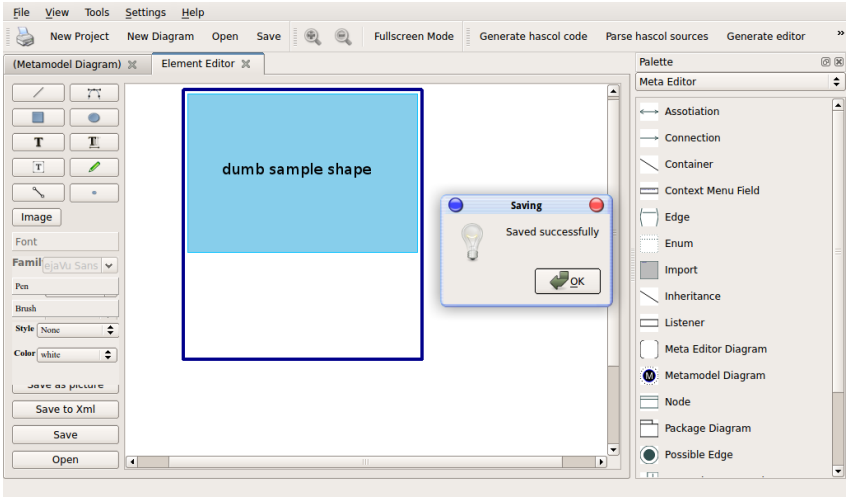

Рисунок 13. Редактируемая форма.

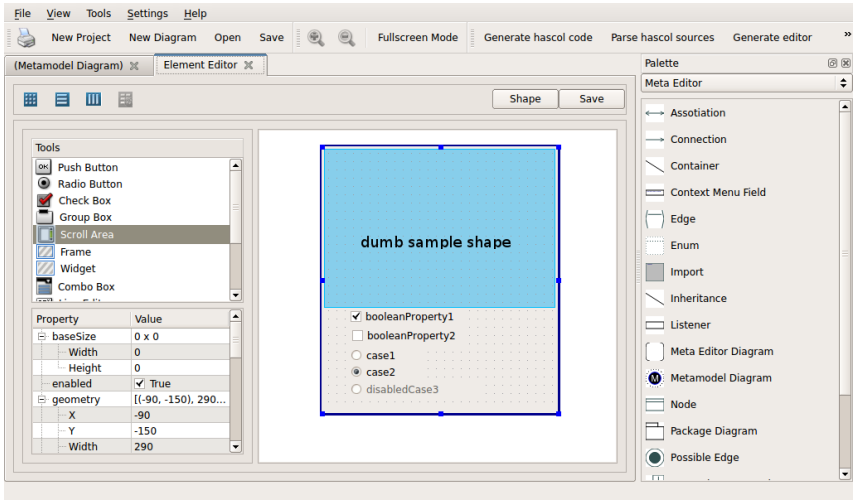

Рисунок 14. Результат - виджет с требуемой формой.

Готовый виджет можно сохранить нажатием кнопки “Save” в правом углу верхней панели. По аналогии с редактором форм, при условии, что редактор виджетов открыт из метаредактора, виджет сразу же свяжется с редактируемым в метаредакторе элементом.

Исходный код редактора форм был перемещен в пакет “elementEdit”. Это связано с тем, что теперь редактор внешнего вида теперь не просто редактор форм, а новый редактор, совмещающий в себе возможности редактора форм и редактора виджетов. Сам редактор виджетов находится также в пакете “elementEdit”. Архитектура модуля редактора виджетов представлена на рисунке 15.

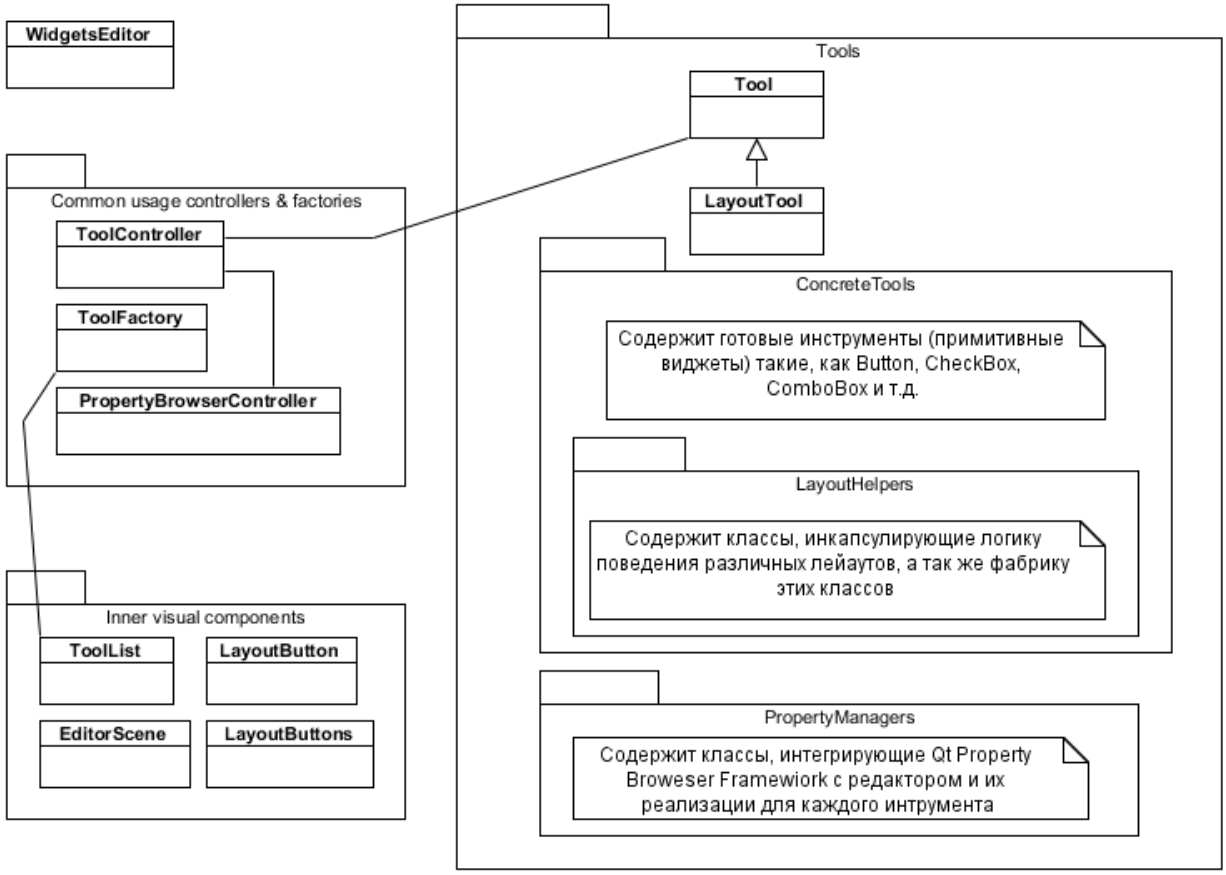

Рисунок 15. Общая архитектура редактора виджетов.

Использование редактора виджетов предполагает использование всего лишь одного класса WidgetsEditor. Существует два варианта создания редактора: один создает новый редактор, никак не связанный с окружающей средой, таким образом, годный только для сохранения будущего прототипа на диск. Второй принимает все нужные данные о редактируемой модели и сохраняет виджет в репозиторий.

Все компоненты, используемые внутри класса редактора, хранятся в подкаталоге “private”. Конкретнее, это всевозможные виджеты, используемые внутри редактора и контроллеры состояния элементов и редактора свойств. Основная часть редактора, инструменты, хранятся в подкаталоге “tools”.

- Класс ToolFactory инкапсулирует в себе создание инструментов, а также определяет их список. На данный момент этот список фиксирован, но, возможно, в будущем будет реализован механизм его

расширения с помощью плагинов. При реализации использован шаблон Singleton, чтобы фабрика была видна из любой точки модуля. При этом, даже в случае нескольких экземпляров одновременно работающих модулей, гарантируется корректное поведение фабрики.

- Класс ToolList представляет панель инструментов. Его задача - получить список инструментов из фабрики, отобразить его, а также обработать события перетаскивания инструмента на сцену.
- Классы LayoutButton и LayoutButtons представляют собой соответственно кнопку переключения компоновщика на верхней панели и фабрику этих кнопок. Фабрика узнает, какие компоновщики доступны в системе, генерирует соответствующие кнопки и управляет их состоянием.
- Класс EditorScene наследует QGraphicsScene, т.е. является моделью сцены для редактора. Добавляет функциональность по обработке горячих клавиш в состоянии, когда ничего не выбрано (ни один из элементов не в фокусе), а также генерирует иконки для операции drag & drop.
- Класс ToolController управляет состояниями “выбранности” элемента, уведомляет о началах и окончаниях операций перетаскивания мышью, устанавливает компоновщики, а так же уведомляет все компоненты об изменениях состояний элементов с помощью механизма сигналов и слотов. По сути, необходимо иметь единственный экземпляр этого класса, доступный каждому, т.е. применить шаблон проектирования Singleton. Но его реализация использует статическую память, разделяемую всеми экземплярами работающих редакторов. А так как в каждом редакторе множество состояний свое, то статическую память использовать нельзя. Следовательно ToolController должен быть инстанциируемым, при этом все элементы иметь ссылку на единственном экземпляре. Решение этой проблемы - фабрика элементов ToolFactory. Она скрывает детали создания элементов, при этом гарантируя, что при создании каждый элемент будет иметь ссылку на нужный контроллер.
- Класс PropertyBrowserController. Он имеет ссылку на браузер свойств и реагирует на изменение выбранного элемента, устанавливая необходимый менеджер свойств. Класс инстанциируется классом WidgetEditor (чтобы сообщить браузеру свойств об экземпляре контроллера), его использование инкапсулировано в классе ToolController.

Далее речь пойдет о каталоге tools. Как уже было сказано, в ней лежит исходный код инструментов и всевозможных дополнений к ним для корректного функционирования модуля. На рисунке 15, в пакете “tools”, показана общая архитектура классов в этой папке. Выделяются два класса: Tool и LayoutTool.

Класс Tool является базовым для всех инструментов. Он наследут класс QGraphicsProxyWidget, так как инструмент является виджетом. Зона ответственности этого класса следующая.

- Использование предлагаемых Qt флагов типа ItemIsSelectable (после установки этого флага разрешается выбирать элемент, при этом реализовывать ничего не нужно, так как это сделано внутри библиотеки), ItemIsMovable (возможность перемещать), ItemIsResizable (возможность менять размер) не дает результатов при использовании встроенных виджетов (об этом был подан пользовательский отчет об ошибке, но вскоре был переведен разрботчиками в состояние Out Of Scope) [16]. Поэтому необходимо было реализовать подобную функциональность (отсюда и необходимость в контроллере состояний ToolController)

- Предоставление интерфейса для класса ToolList, позволяющий получить иконку и название инструмента.
- Реализация части функциональности по сериализации и десериализации виджетов.
- Обработка события начала перетаскивания инструмента на сцене мышью (создание объекта Drag & Drop).

Класс LayoutTool наследует Tool. Он является базовым для всех инструментов-контейнеров. Название оправдано тем, что для каждого инструмента-контейнера имеется возможность установить компоновка. Его обязанности заключаются в следующем.

- Отслеживание изменения множества детей. В это понятие входит обработка окончания перетаскивания нового инструмента, а так же перехват сигнала, о том, что ребенок был перетащен в другое место либо удален.
- Реализация части функциональности по сериализации и десериализации виджетов.
- Контроль компоновок. Это довольно большая по объему реализации функциональность. Непосредственно класс LayoutTool инициализирует все необходимые компоненты, т.е. служит своеобразным контроллером для других компонент, обеспечивающих функционирования компоновщиков. К этим компонентам относятся следующие классы:
    - LayoutHelperBase. Этот класс является неотъемлимой частью класса LayoutTool. Он служит базой для всех классов, инкапсулирующих в себе логику конкретного компоновщика. Основное его назначение - предоставить для LayoutTool интерфейс сообщения обо всех действиях пользователя вне зависимости от выбранной компоновки, реализовать некоторые базовые элементы поведения, дополнить или исправить поведение компоновщиков библиотеки Qt (например, остановить сжатие виджета, если достигнут минимум по сжимаемой оси всех виджетов и подкомпоновок в нем), а также выполнить свою часть сереализации и десериализации. Интерфейс для LayoutTool предлагает оповестить о перемещении перетаскиваемого виджета с указанием новой позиции, что необходимо для подсветки текущей позиции в компоновке, метод прорисовки этой подсветки, оповещение о начале и окончании переноса виджета..
    - LinearLayoutHelper. Реализует LayoutHelperBase. Активируется при выборе вертикальной или горизонтальной компоновки из верхней панели редактора виджетов. Основную логику поведения виджетов в компоновке реализует класс QGraphicsLinearLayout. LinearLayoutHelper лишь при необходимости переупорядочивает виджеты в компоновкае и реализует отслеживание и подсветку позиции для линейного случая.
    - GridLayoutHelper. Этот класс выполняет аналогичные классу LinearLayoutHelper действия применитель к сеточной компоновке (QGraphicsGridLayout). Однако алгоритм отслеживания значительно сложнее по сравнению с линейным случаем. Например, если при линейной компоновкае заполнены все ячейки подряд, то в случае сетки могут оставаться “пустоты” (рис. 11), которые требуется специально отслеживать. Хуже,

QGraphicsGridLayout не предоставляет возможности добавить элемент в ряд или колонку перед всеми, что делает затруднительной обработку такого действия. В Qt Designer эта проблема решена добавлением сетки в линейный компоновка соответствующего направления, что, по мнению автора данной работы, является крайне странным и неоправданным решением, поскольку невозможно добавить элемент в какую-то отдельную ячейку перед всеми, что является наиболее частой потребностью пользователя. В редакторе виджетов системы QReal эта проблема решена созданием нового экземпляра компоновки со сдвигом элементов в соответствующем направлении.

- LayoutHelperFactory. Этот класс инкапсулирует функциональность по установке на инструмент какого-либо компоновщика.

Еще одной важной особенностью редактора виджетов является редактирование свойств инструмента. При реализации был использован Qt Property Browser Framework, конкретно - подход на основе QVariant. Основной класс - PropertyManagerBase, от него наследуются все остальные из этой папки. Он берет на себя задачу реализации основной части взаимодействий свойств инструментов с Qt Property Browser Framework. Для этого он оборачивает все объявленные свойства в класс Property, соединяет сигналы редактирования из редактора свойств с виртуальным слотом, переопределяемым конкретными менеджерами, и наоборот, позволяет сообщить редактору свойств об изменении значения свойства вне него. Все, что остается сделать конкретным менеджерам, унаследованным от базового - передать ему список свойств, обработать сигналы об изменении и оповещать редактор при изменении конкретного свойства вне этого редактора. Из определения наследования следует, что подкласс имеет все свойства родительского. Поэтому среди менеджеров свойств сохранена та же иерархия, что и среди самих инструментов. Выбором текущего менеджера свойств с преобразрованием его в термины Qt Property Browser Framework занимается связка ToolController-PropertyBrowserController.

Для интеграции с системой QReal редактор должен обладать возможность сериалиации и десериализации редактируемого виджета. Для формата хранения было решенио использовать XML. Каждая компонента редактора ответственна за сериализацию и десериализацию информации в своей зоне ответственности. Этапы сериализации:

1. Вызов метода generateXml() у класса WidgetEditor. Происходит по нажатию кнопки “Save”. Вызывается метод generateXml() с ссылкой на корневой тег генеруемого документа у корневого инструмента.
2. Далее процесс рекурсивен, выполняется спуск по дереву элементов. Каждый элемент пишет свою информацию в тег, вызавает родительское (в смысле наследования) переопределение метода “generateXml” (он виртуальный). На самом деле, метод generateXml не переопределен нигде, кроме как в Tool и LayoutTool. Это переопределение попросту не нужно (хотя возможно), так как все, что должен записать о себе конкретный инструмент - это свое имя (имя тега). Но имя можно получить из интерфейса для списка инструментов, там оно отображается вместе с иконкой. Таким образом, надобность в этом переопределении отпадает.

3. Как следует из пункта 2, в случае инструмента-контейнера вызавается сначала метод generateXml у класса LayoutTool, затем у класса Tool. В случае обычного инструмента сразу вызывается метод у класса Tool.
4. Класс LayoutTool дополняет аттрибуты тега информацией о компоновке, которая берется из фабрики компоновок. Затем производится обход детей и рекурсивно вызывается метод generateXml() у них. Механизм сохранения позиции детей в компоновке использует понятие приложенного свойства (attach property), когда к элементу приложено свойство какого-то другого элемента. В нашем случае этот элемент - компоновка. Приложенные свойства генерируются установленным компоновщиком.
5. Класс Tool вызывает метод generateXml() у соответствующего менеджера свойств.
6. Так как классу PropertyManagerBase уже известно обо всех свойствах, нет необходимости делать метод виртуальным. Менеджер обходит все свойства и добавляет для каждого дочерний тег “<property>”. Само значение свойства фактически может иметь разные типы, но обернуто в QVariant. Следовательно, возникает необходимость в сериализаторе класса QVariant. Он реализован в классе XmlUtils, так как, по сути, логически не привязан к сереализатору редактора и может быть использован другими компонентами в QReal.

Процесс десериализации является обратным процессу сериализации. Алгоритм таков:

1. Вызов статического метода deserializeWidget() у класса WidgetsEditor. Ясно, что для десериализации отдельного виджета нужен объект, обладающий информацией обо всех виджетах которые могут встретиться в процессе десериализации. Такой объект - ToolFactory, точнее единственный его экземпляр, который как раз можно получить из статического контекста.
2. Далее управление получает ToolFactory. Здесь рекурсивно воссоздается макет того редактора, который сереализовал желаемый виджет. Этот же код используется и для открытия сохраненного виджета в редакторе.
3. Далее, если текущий инструмент - инструмент-контейнер, устанавливаем ему соответствующую компоновку. Если у инструмента встретилось приложенное свойство, добавляем его на родителя в нужную позицию.
4. Остается только установить свойства. С помощью XmlUtils десериализуется значение свойства, затем у менеджера вызываются те же слоты, что и при редактировании из браузера свойств. При этом менеджер по реализованной в нем схеме сам установит на нужные инструменты нужные свойства.
5. Предыдущие шаги строили модель инструментов. Но инструмент - это не виджет, а вложенный виджет. Поэтому завершающий этап - перевести дерево вложенных виджетов в дерево обычных с конвертированием компоновок.

В итоге, с небольшой степенью детальности, было изложено устройсво редактора виджетов.

Интеграция с QReal

В первую очередь был реализован механизм связывания виджета с редактируемым в метаредакторе узловым элементом. При этом был изменен механим аналогичной функции редактора форм. Разница лишь в том, что теперь в свойство элемента пишется XML сериализованного виджета и, если создана, форма (в смысле редактора форм) этого виджета. При этом была сохранена полная совместимость со старым механизмом форм.

Далее оставалось лишь реализовать поддержку виджетов сценой представления. Как было показано, иерахия элементов в QReal основывается на классе QGraphicsItem. Он не похдодит для встраивания виджетов, для этого есть класс QGraphicsProxyWidget. Было предложено два подхода по устранению этой проблемы:

1. Композиция. На NodeElement добавляется элемент, содержащий встроенный виджет. При этом все события этого элемента передаются на обработку родителю. Плюс - класс NodeElement остается неизменным. Минус - громоздкость реализации, неоходимо учитывать все аспекты поведения элемента и при добавлении новой функциональности в класс NodeElement учитывать ее во внутреннем элементе.
2. Наследование. Предлагается изменить корень иерархии на QGraphicsProxyWidget. Плюс в естественности и относительной простоте поддержки. Проблема в том, что уже написано много кода, активно использующего всевозможные особенности графического элемента. Класс QGraphicsProxyWidget переопределяет эти особенности по-своему и это вызывает различные проблемы проблемы нетривиального характера, решение которых требует хорошего владения библиотекой Qt и хорошее знание кода системы QReal.

Было принято решение использовать второй подход. При этом возникли следующие трудности чисто технического характера:

- Изначально элементы вообще не появлялись на сцене. Связано с необходимостью по-другому устанавлявать размер. В результате этого все элементы имели нулевой размер, поэтому не отображались.
- Неверно обрабатывались события мыши. Элемент получал событие клика, мышью, но желаемый эффект (выбор, передвижение или изменение размера) не достигался. Связано с совершенно новой обработкой классом QGraphicsProxyWidget событий мыши. Решение потребовало кропотливого изучения исходного кода Qt, имеет большую техническую сложность и, следовательно, здесь обсуждаться не будет.
- Впервые появившийся на диаграмме элемент не отображался. Решение также требует хорошего знания используемых технологий, поэтому не обсуждается.
- Многие другие проблемы чисто технического характера.

Апробация

В качестве апробации было реализовано удобное радактирование свойств для системы QReal:Robots. Раньше все свойства редактировались с помощью динамических меток, что давало нежелательные результаты (рис. 16). Теперь свойства редактируются предназначенными для них элементами управления. На рисунке 17 показано редактирование свойства целого типа. При потере фокуса элемент управления исчезает, наглядность увеличивается.

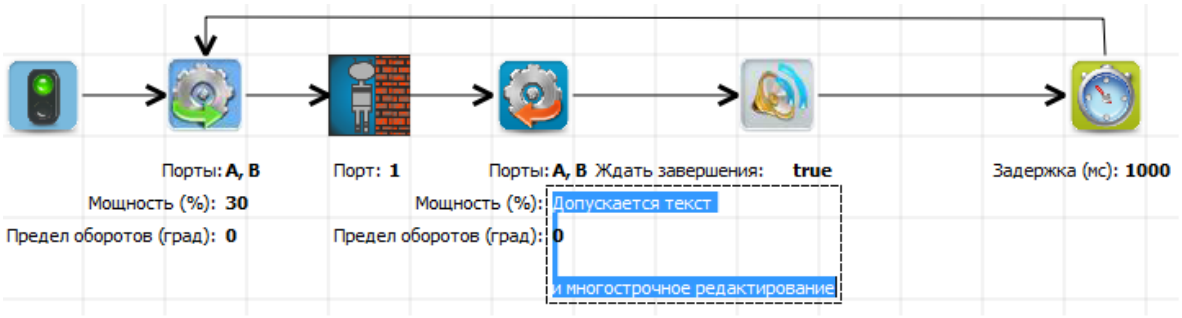

Рисунок 16. Некорректное поведение динамических меток до реализации этой работы

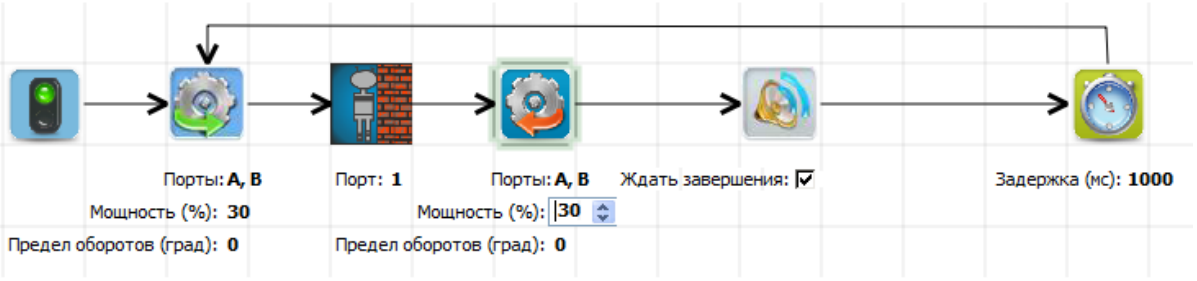

Рисунок 17. Редактор свойства прямо на элементе. Редактирование более удобно.

Заключение

Результаты

В результате данной работы в системе QReal была реализована поддержка виджетов на элементах. Данное нововведение может быть крайне полезным для создания небольших удобных редакторов, что является основным требованием к предметно-ориентированным системам визуального моделирования.

Быстрое и удобное создание виджетов для элемента дает преимущество системы QReal перед другими MetaCASE системами, такими как Microsoft DSL Tools, Eclipse GMF и т.д., так как в них поддержка виджетов затруднена или невозможна в общем.

Нововведение уже применено на практике. В проекте QReal:Robots реализовано редактирование свойств прямо на элементе с помощью соответствующих виджетов. Это делает процесс редактирования более удобным по сравнению с динамическими метками, используемыми раннее.

Дальнейшее развитие

Данная разработка открывает целый спектр новых возможностей. Улучшения могут пойти по двум путям: расширение возможностей и улучшение существующей функциональности. Из текущих планов можно выделить:

- Поддержка редактора форм и механизма отображения виджетов.
- Редактор форм для мобильных платформ. Можно рисовать и задавать переходы между состояниями формы прямо на сцене редактора в QReal.
- Улучшение юзабилити. Расширение метадодели с возможностью встраивать подсказки на элементы. Отлично подойдет для сложных и требующих специальному обучению для овладевания редакторов (например, редактор семантик переходов при визуальной отладке модели).

Список литературы

1. Краткий курс Microsoft DSL Tools, http://www.intuit.ru/department/se/vismodtp/13/
2. Руководство Qt Designer, http://doc.crossplatform.ru/qt/4.5.0/designer-manual.html
3. Семейство стандартов Integration DEFinition, http://www.idef.com/
4. Т.А.Брыксин, Ю.В.Литвинов, Технология визуального предметно-ориентированного проектирования и разработки ПО QReal
5. Терехов А.Н., Брыксин Т.А., Литвинов Ю.В., Смирнов К.К., Никандров Г.А., Иванов В.Ю., Такун Е.И., Архитектура среды визуального моделирования QReal
6. Business Process Modeling Notation. Final Notation Specification dtc/06-02-01, OMG, 2006. http://www.omg.org/spec/BPMN/1.2/PDF/
7. EMF, http://www.eclipse.org/modeling/emf/
8. GEF, http://www.eclipse.org/gef/
9. GMF, http://www.eclipse.org/modeling/gmp
10. LEGO Mindstorms, http://mindstorms.lego.com/en-us/Default.aspx
11. Microsoft DSL Tools sample, http://code.msdn.microsoft.com/windowsdesktop/Visualization-and-Modeling-313535db
12. NXT-G Quick Guide, http://www.legoengineering.com/index.php?option=com_content&view=article&id=124
13. OMG Unified Modeling Language (OMG UML), Infrastructure, Version 2.2, OMG, 2009. http://www.omg.org/spec/UML/2.2/Infrastructure/PDF/
14. OMG Unified Modeling Language (OMG UML), Superstructure, Version 2.2, OMG, 2009. http://www.omg.org/spec/UML/2.2/Superstructure/PDF/
15. Rational Rose, http://www-01.ibm.com/software/awdtools/developer/rose/
16. QGraphicsProxyWidget bugreport, https://bugreports.qt-project.org/browse/QTBUG-3136
17. QReal, https://github.com/qreal
18. Qt Graphics View Framework, http://qt-project.org/doc/qt-4.8/graphicsview.html
19. Qt Model View Programming, http://qt-project.org/doc/qt-4.8/model-view-programming.html
20. Qt Property Browser Framework, http://doc.qt.nokia.com/qq/qq18-propertybrowser.html
21. Visual Paradigm, http://www.visual-paradigm.com/