Санкт-Петербургский Государственный Университет

Математико-механический факультет
Кафедра системного программирования

Архитектура и прототипирование metaCASE-системы

Курсовая работа студента 345 группы
Тарана Кирилла Сергеевича

Научный руководитель ..... Т. А. Брыксин
ст. преподаватель /подпись/

Санкт-Петербург
2012

Содержание

1. Введение
    a. MetaCASE-системы и визуальное моделирование
2. Цели работы
3. Постановка задачи
    a. Необходимая функциональность
    b. Детали текущей реализации
4. Архитектура
5. Реализация
    a. Выбор инструментов и детали реализации
    b. Организация кода
6. Заключение
    a. Итог и перспективы
    b. Выводы
    c. Ссылки

1. Введение

> ***Архитектура*** *- это базовая организация системы, воплощенная в ее компонентах, их отношениях между собой и с окружением, а также принципы, определяющие проектирование и развитие системы.*
>
> ***Система*** *- это набор компонентов, объединенных для выполнения определенной функции или набора функций.*
>
> *[IEEE 1471]*

Эволюционирующий продукт неизбежно сталкивается с потребностью в реализации нового функционала, который может хорошо вписываться или совсем не вписываться в текущую архитектуру системы. В то же время с возрастом системы становится всё труднее модифицировать её архитектуру из-за увеличения связности различных её компонент, связанного с добавлением новых функций. Таким образом, *развитие системы затрудняет её дальнейшее развитие*.

С другой стороны, со временем ценность тех или иных характеристик системы меняется или переосмысливается, в том числе новый функционал может идти вразрез с подходами к разработке приложения, принятыми в момент зарождения системы.

Поэтому на некотором этапе становится целесообразно выбрать некоторое подмножество функций и характеристик системы (включая ещё не реализованные и не достигнутые) и закрепить за ним приоритет в развитии. Вполне возможно, что для достижения этих новых характеристик необходимо идти на кардинальные меры, к примеру перепроектировать систему заново, т.к. архитектура могла устареть.

MetaCASE-системы и визуальное моделирование

Для разработки программного обеспечения существует множество различных инструментов разработки (CASE-пакеты). Среди них встречаются и инструменты, использующие визуальное моделирование (такие программы как Rational Rose, Visual Paradigm, Borland Together).

Большинство из них позволяют пользоваться ограниченным, достаточно общим набором функций. В некоторых ситуациях (при необходимости реализовать какую-то совокупность проектов из одной предметной области) гораздо выгоднее использовать технологии, заточенные под конкретную область. Одним из возможных способов создавать такие предметно-ориентированные инструменты является использование metaCASE-систем с поддержкой визуальных языков моделирования. MetaCASE-системы — инструменты, предоставляющие возможность не только использовать готовые компоненты редакторов, но и создавать их самому; в данном случае речь идёт о возможности создавать предметно-ориентированные визуальные языки.

2. Цели работы

> ***Прототип программы*** *— макет (черновой, пробной версии) программы, обычно — с целью проверки пригодности предлагаемых для применения концепций, архитектурных и/или технологических решений, а также*

для представления программы заказчику на ранних стадиях процесса разработки.

Данная работа проводится в рамках проекта QReal, кроссплатформенного metaCASE-инструмента, основная идея которого — создание и использование предметно-ориентированных визуальных языков моделирования. Предполагается рассмотреть необходимые или перспективные для QReal функции, в соответствии с ними спроектировать новую систему и реализовать (частично) её прототип для проверки на практике принятых решений.

Цель работы — получение оптимальной архитектуры metaCASE-системы, подобной QReal. Эта архитектура должна по возможности быть более подходящей для реализации функционала QReal, как реализованного (т.е. обеспечивать меньшее количество ошибок), так и запланированного (т.е. ещё не реализованные функции QReal должны более стройно вписываться в новую архитектуру). Другими словами, необходимо развить архитектуру, альтернативную архитектуре QReal.

Дополнительно стоит рассмотреть возможные пути развития QReal, открывающиеся с новой архитектурой.

3. Постановка задачи

Необходимая функциональность

Функциональность разрабатываемого приложения должна повторять ключевую функциональность QReal и по возможности включать в себя запланированные, но ещё не реализованные функции (для примера, откат операций).

Одна из ключевых функций QReal — многопользовательская работа. На данный момент она осуществляется через синхронизацию с системами контроля версий, хотя есть возможность и работать с сетевым репозиторием.

С другой стороны, при создании инструмента для разработки необходимо задумываться об отказоустойчивости. Для этого в системе хранилище и редактор должны работать независимо в рамках работы одного человека.

Следует рассмотреть возможность улучшить и процесс редактирования. Варианты развития - продвинутое взаимодействие языков (импорт функций, доступность более чем одного метаязыка для описания новых языков), а так же автоматизация с помощью пользовательских команд.

Ещё одно направление развития — снижение ресурсоёмкости приложения, поскольку некоторые задачи (например, кодогенерция) могут ощутимо тормозить работу пользователей.

Детали текущей реализации

1. В текущей реализации у данных довольно большая изменяемость, т.е. элемент можно изменить из разных мест, одна операция может изменить сразу несколько элементов. Такие изменения трудно отследить, а значит и трудно реализовать такие функции как отмена действий, историй действий.

В качестве решения этой проблемы можно использовать т.н. чисто функциональные структуры данных. Суть такого подхода в использовании неизменяемых структур — при изменении которых строится новый экземпляр.

В простейшем случае новый экземпляр — полная копия структуры, но такие структуры можно реализовать и эффективно. Для этого при копировании нужно создавать объект, ссылающийся на прошлое состояние, и добавлять к нем список изменений.

При использовании таких структур — история и отмена действий реализуются

тривиально. В качестве истории достаточно использовать список состояний, а чтобы отменить действие, нужно удалить последнее состояние.

2. На данный момент в QReal сделан больший акцент на собственно редакторе, чем на внутреннем представлении диаграмм. В частности, диаграммы логически представляются простой таблицей “id-элемент”, где элементы бывают вершинами и рёбрами, а также содержат различную информацию. Этого достаточно для отображения и редактирования диаграмм, но для анализа диаграмм желательно наличие более сильных структур.

Пример — для обхода графа нужно выяснять, является ли текущий элемент вершиной или ребром, и в зависимости от этого либо обращаться к списку смежных рёбер, либо к списку смежных вершин элемента. Причём, для доступа к каждому последующему элементу необходимо обращаться к таблице и искать его по id.

То есть для обработки диаграмм необходимо анализировать каждый элемент по отдельности вместо того, чтобы взять нужную информацию из какой-то специальной структуры.

Таким образом, проблема в том, что текущие структуры представления слишком “слабые” (не предоставляют нужного количества информации) в то время, как хранимые в них данные, наоборот, слишком объёмные — хранят много различных своих аспектов.

Оптимальное решение такой ситуации — использование нескольких более сложных структур (граф, различные таблицы заранее вычисленных свойств вершин в нём) вместе с большей абстракцией от предметной области.

Текущая архитектура

Если рассматривать многопользовательский аспект QReal, то основной вариант обеспечения одновременной работы нескольких пользователей над одним проектом - синхронизация с помощью систем контроля версий. При этом также есть возможность использовать и сетевой репозиторий, реализованная с помощью RPC инструментария ICE. По тому же интерфейсу работают и независимые клиенты (вроде генераторов, не являющихся частью QReal).

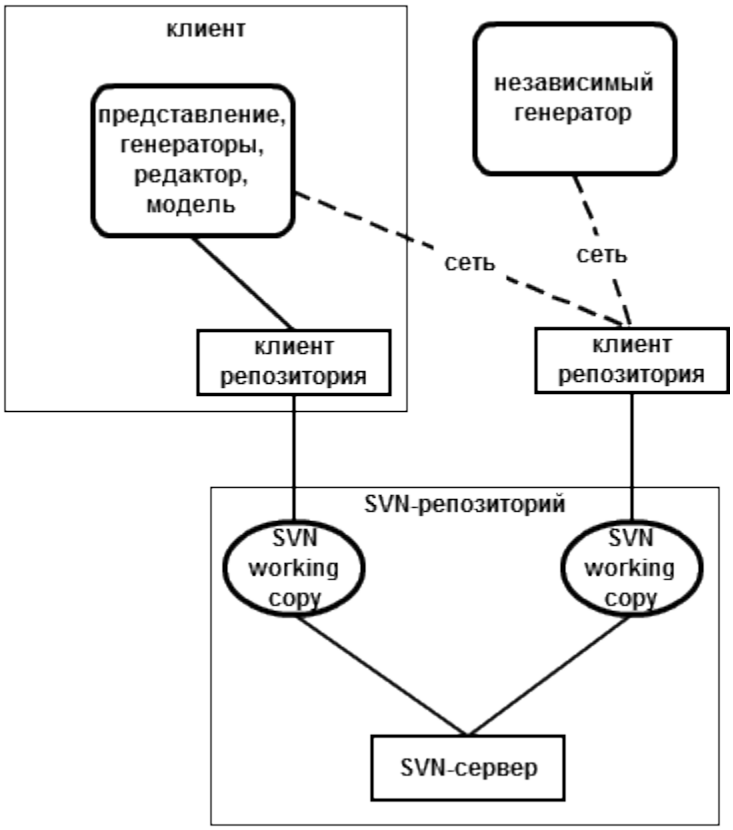

Рис.1 Упрощенная архитектура QReal.

Редактор работает в рамках того же компонента, что и клиент репозитория, при этом изменения записываются на диск только по команде пользователя. Поэтому ошибка, произошедшая в редакторе, может сбросить несохранённые данные, и они так и не будут синхронизированы с репозиторием системы контроля версий.

4. Архитектура разрабатываемого приложения

1. Одно из требований — хранилище должно сохранять свои данные даже в случае какой-либо ошибки в редакторе (независимость хранилища от редактора). Способ обеспечить это — запускать хранилище и редактор в разных процессах, а их общение устроить по одному из вариантов межпроцессного взаимодействия (обмен сообщениями, общая память, удалённый вызов процедур). Другое требование — возможность одновременной работы с одним репозиторием нескольких пользователей, а также удалённое хранение диаграмм. Это приводит к использованию сети. Оба эти требования удовлетворяются, если использовать клиент-серверную архитектуру, где репозиторий является частью сервера, а редактор выступает в роли клиента.

2. Способ снизить нагрузку на компьютер пользователя — перенести его тяжеловесные задачи на удалённый компьютер. Такой сервер будет оптимальнее расходовать свои ресурсы из-за многопользовательской работы, т.к. будет меньше времени простоя. Обеспечить же это можно, если выполнять всё редактирование на сервере.

3. Другой момент — собственно редактирование диаграмм. Можно осуществлять такое редактирование напрямую, т.е. изменять значения вершин в графе, но также можно составить собственный язык, предоставляющий операции для редактирования графа. С одной стороны, создание и использование языка усложняет работу. С другой стороны, мы в любом случае приходим к такой задаче, когда задаём интерфейс клиент-серверного взаимодействия (если оно осуществляется через обмен сообщений):

- либо мы в сообщении указываем изменения в графе - тогда для этого нужен специальный формат сообщений, т.е. язык, содержащий операторы вроде “add” и “delete”;
- либо мы помещаем в сообщение полный снимок нового графа, что непроизводительно.

Поэтому создание языка оправдывается хотя бы тем, что оно задаёт формат сообщений. Кроме этого у такого подхода есть такие преимущества, как:

- сообщения могут быть очень короткими (например, всего один оператор), но делать объёмные изменения в диаграмме — это даёт выигрыш в размере сообщений;
- тривиально реализуется CLI интерфейс (мало практической пользы для пользователя, но полезно для разработки);
- возможно расширение такого языка пользователями или сторонними разработчиками;

Последнее преимущество является наиболее весомым аргументом. Благодаря ему можно изменять многие свойства системы без изменения самой программы. В частности, предполагается ввести поддержку визуальных языков с помощью добавления специальных функций, интерпретирующих части диаграмм.

Недостатком такого решения могла бы стать сложность реализации, но он устраняется исполнением языка на основе существующего языка программирования, имеющего интерпретатор (такие языки называют EDSL — Embedded Domain Specific Language).

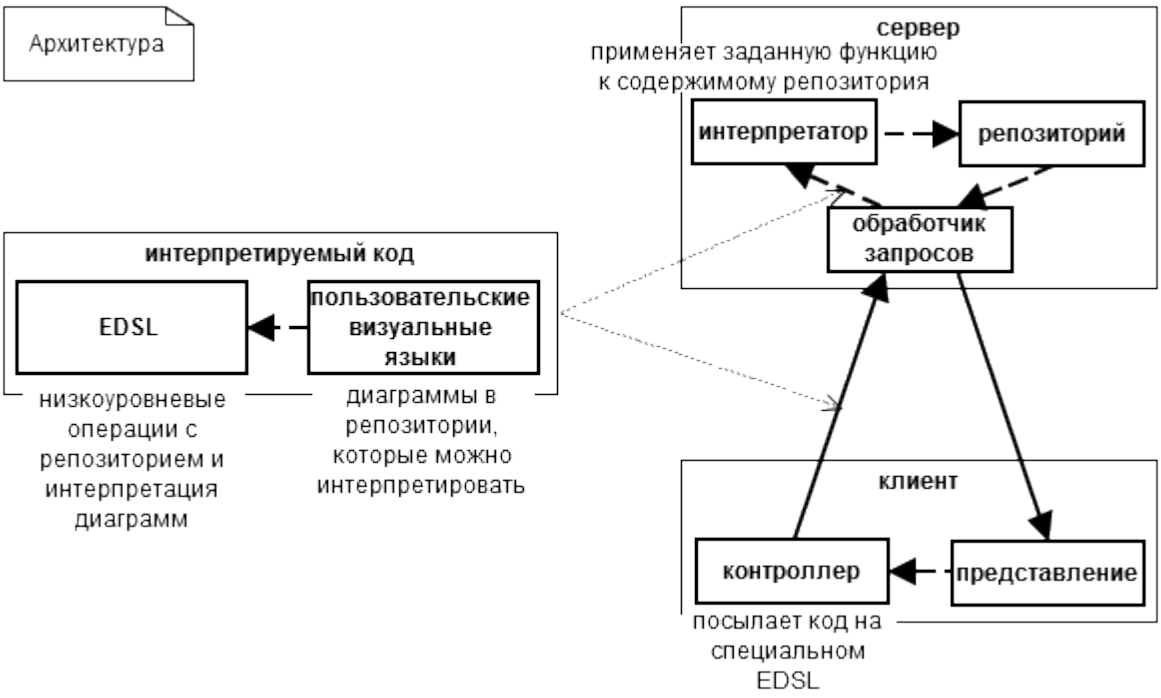

Рис.2 Архитектура системы.
Клиент посылает серверу сообщения, содержащие код преобразования на специальном интерпретируемом языке.

4. При создании EDSL желательно выбрать наиболее простой базис языка. Поскольку диаграммы представляют собой граф, надо решить, как его представить, при этом максимально просто.

Обычно граф представляется множеством вершин и рёбер между ними, но если обратить внимание на то, как обычно представляется дерево (вершина-родитель и множество вершин-детей), то становится ясно, что невзвешенные рёбра вполне можно представить отношением “родитель-ребёнок”. Отличие от дерева в такой реализации будет в том, что для получения одной компоненты связности графа надо разрешить указывать в качестве ребёнка любые уже существующие вершины.

Объявленный таким образом тип данных вместе с операциями для его обработки даёт подходящую структуру для работы с диаграммами. Но для различных целей необходимо реализовать дополнительные структуры:

- таблица “id-вершина” — нужна для быстрого произвольного доступа к вершинам графа, используется для хранения;
- список рёбер, матрица смежности — для алгоритмов на графе.

5. Другой важный примитив языка — атомарное действие, являющееся некоторым преобразованием графа с рядом проверок, выполняющихся до или после преобразования и отменяющих выполнение преобразования в случае их невыполнения.

Пример такого действия — добавление узла в граф должно проверять, существуют ли вершины, которые мы хотим связать вместе.

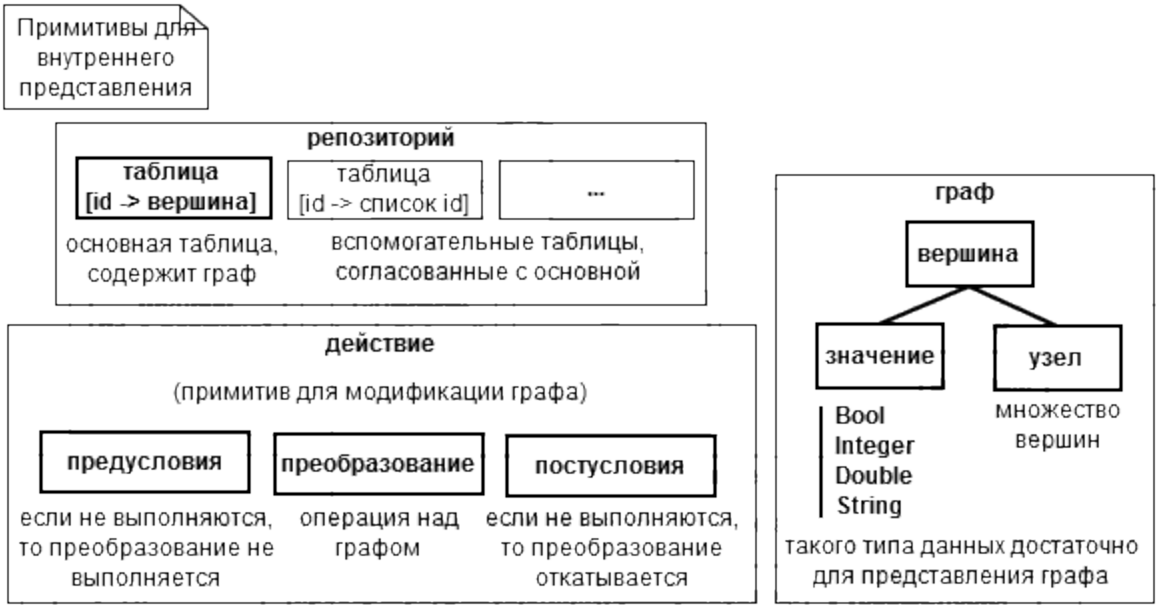

Рис.3 Внутреннее представление: репозиторий, собственный язык.

6. Репозиторий состоит из основной таблицы “id-вершина”, хранящей граф, и нескольких вспомогательных, хранящих другие аспекты графа (к примеру, родительские вершины).

Вместе с изменением графа кроме правок в основную таблицу вносятся согласованные изменения во вспомогательные; при этом вспомогательные таблицы не содержат уникальной информации, т.е. их содержимое может быть восстановлено по основной таблице.

Поддержка таких вспомогательных таблиц требует некоторого усложнения логики редактирования графа, но зато обеспечивает более быстрый доступ к различной информации.

5. Реализация

Выбор инструментов и детали реализации

В качестве языка реализации был выбран Haskell. Этому решению способствовало как наличие подходящих библиотек и инструментов, так и некоторые аспекты самого языка.

1. *Модуль-интерпретатор*.

Для Haskell есть полноценные интерпретаторы (GHCi, Hugs) вместе с библиотеками для работы с ними. Для создания модуля-интерпретатора CASE-системы взята библиотека HInt — модуль компилятора GHC с пользовательским API. Библиотека расположена в открытом репозитории Hackage: http://hackage.haskell.org/package/hint. Другими словами, данная библиотека позволяет внутри Haskell-кода вызывать функции для интерпретации Haskell-кода.

Пример использования библиотеки:

- `runInterpreter` *`interpreter`* — запускает процесс, использующий интерпретацию,
- `setImports` *`modules`* — импортирует указанные модули,
- `interpret` *`code`* `(as ::` *`Type`*`)` — возвращает значение указанного типа по строке, содержащей код.

Реализация EDSL выполнена в виде модулей, содержащих функции для работы с внутренним представлением и описание специального типа преобразования Modification.

Modification - тип синоним для функций вида:

```
type Modification = History -> Either Constraint World
```

т.е. функций, принимающих историю состояний (последнее состояние и есть текущее) и возвращающих либо ошибку проверки предусловий и постусловий, либо новое состояние.

Модуль-интерпретатор импортирует нужные модули и затем пытается интерпретировать поступающий к нему код как значения типа Modification, представляющего функции над диаграммой, которые могут завершиться ошибкой; после интерпретации в случае совпадения типа интерпретатор передаёт полученную функцию обратно серверу, который применяет её к репозиторию.

Примеры базовых функций языка:

- `insertValue :: Value  -> Action Id` — создание действия для вставки вершины-значения,
- `insertKnot  :: Set Id -> Action Id` — создание действия для вставки вершины-узла,
- `removeById  :: Id -> Action ()` — действие для удаления вершины,
- `apply  :: Action a -> World -> Either Constraint World` — пытается применить атомарное действие к текущему состоянию,

Функции-сокращения:

- `onHistory = wrap . apply` — применение функции сразу к истории,
- `bind      = onHistory . insertKnot . S.fromList`,
- `create    = onHistory . insertValue`,
- `int       = create . I`,
- `str       = create . S`,

Примеры значений типа Modification (т.е. это команды, поступающие от пользователя):

- `str "Hello, world!"` — вставка текстового элемента,
- `onHistory $ removeById 1` — удаление элемента с id = 1.

2. *Графический интерфейс*.

Графический интерфейс использует Gtk (а именно Gtk2Hs - библиотеку для Haskell, http://projects.haskell.org/gtk2hs/). Gtk — событийно-ориентированная библиотека. Это значит, что интерфейс описывается набором графических элементов; событий, которые могут с ними происходить (например, щелчок мыши по кнопке) и обработчиков событий — кода, который выполняется при возникновении некоторого события. Поскольку обработчики являются обыкновенными функциями, то при использовании функциональных языков появляется больше вариантов их использования (можно передать как аргумент функции, комбинировать с другими обработчиками и т.д.).

Таким образом, событийно-ориентированность Gtk хорошо согласуется с функциональной парадигмой, поэтому тут Haskell имеет преимущество перед другими высокоуровневыми языками.

Haskell имеет также и высокоуровневую библиотеку для работы с графикой Diagrams, предоставляющую функции рендеринга на окна Gtk и специальный язык описания. Веб-страница проекта: http://projects.haskell.org/diagrams/.

С помощью этого языка можно комбинировать изображения-примитивы (в данном случае это несколько вариантов изображения вершин) без использования низкоуровневых понятий (пиксель, точка и т.д.). В дополнение к библиотеке есть набор готовых алгоритмов по компоновке изображений (к примеру, вывод вершин графа в виде дерева реализован с помощью такого готового алгоритма).

Примеры языка:

- `f ~~ t # lw 2` — соединить линией толщиной 2 вершины f и t
- `circle 0.4 # lw 1 # (fc black)` — отрисовать чёрный круг радиусом 0.4

- ```
  strutY y
    ===
  strutX x ||| d ||| strutX x
    ===
  strutY y
  ```
  — обрамить элемент прозрачными разделителями нужного размера
- `withLabel = (<>) . (withKey Nothing) . text . show` — реализация функции, прикрепляющей к элементу его заголовок (withKey прикрепляет логический id к изображению)

Язык достаточно компактен и удобен для того, чтобы использовать его в качестве описания всех графических примитивов вместо использования изображений (к примеру .svg).

3. *Клиент-серверное взаимодействие*.

Клиент-серверное взаимодействие реализовано в виде RPC (удалённый вызов
  ```

процедур) с использованием инструментария Thrift (http://thrift.apache.org/). Он позволяет описывать сервисы (изолированные компоненты системы со строгим интерфейсом; имеют различные способы доступа, в т.ч. по сети) единственным образом для многих языков и платформ. При этом неважно, где будет располагаться компонент, реализующий сервис, как будет осуществляться доступ к нему, на каком языке он будет написан.

Пример описания сервиса:

```
struct Response {
    1: optional map<i32, Vertex> result,
    2: optional string comment
}
service World {
    Response interpret(1:string code)
}
```

По такому описанию Thrift для конкретного языка генерирует интерфейсы и код для взаимодействия с сервисом. После генерации интерфейса надо его реализовать в коде сервера:

```
instance World_Iface Server where
      interpret _        Nothing    = fail "No input."
      interpret server (Just code) = do ...
```

После этого в коде клиента можно вызвать интерпретацию на сервере: `interpret world "bind [2,3,4]".` bind [2,3,4] свяжет вершины 2,3,4 в узел. Как результат — Thrift берёт на себя низкоуровневую работу по доставке сообщений между клиентом и сервером, а разработчик может вызывать методы сервера так, как будто он находится в том же процессе, что и клиент.

Организация кода

Код разбит на пакеты Cabal. Cabal — система сборки и пакетный менеджер для Haskell; осуществляет сборку, проверку зависимостей, доступ к репозиторию библиотек Hackage. Такие модули собираются по отдельности, при этом некоторые могут зависеть от других.

В данном случае, все пакеты зависят от common — пакета, разбитого на:

1. Ядро, содержащее примитивы для внутреннего представления данных (вершины, репозиторий, атомарное действие, история состояний) вместе со структурами для их обработки (пока что это дерево и список рёбер).
2. Просто общий код, к примеру обработку файлов конфигураций (используется и в клиенте, и в сервере).

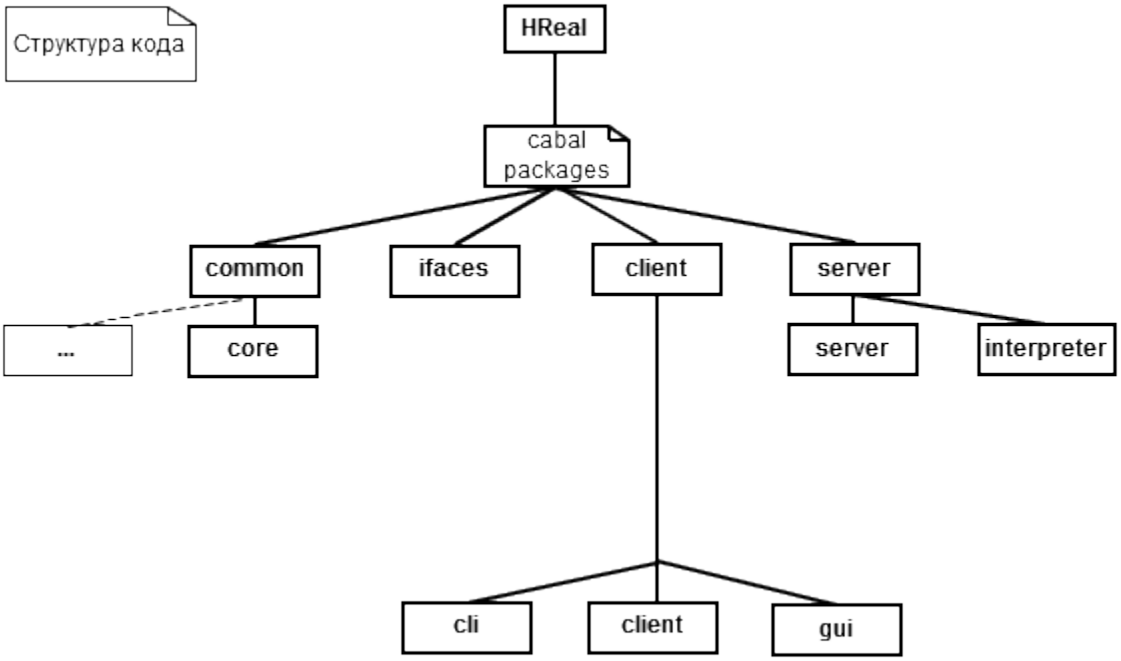

Рис.4 Структура кода.

Пакеты client и server в свою очередь зависят от пакета ifaces, содержащего код, сгенерированный с помощью Thrift, и вспомогательный код для запросов (конвертация данных в формат для сообщений и т.д.).

Пакет client разбит на несколько компонентов:

1. Общий клиентский код (чтение конфигурации, соединение с сервером).
2. Реализации CLI и GUI, вызываемые после инициализации клиента в зависимости от параметра, переданного исполняемому файлу.

Пакет server содержит:

1. Серверный код, обрабатывающий запросы к интерпретатору, применяющий интерпретированные преобразования и т.д.
2. Интерпретатор на основе HInt, подключающий модули EDSL и интерпретирующий присылаемый клиентом код.

6. Заключение

Итог и перспективы

В ходе работы были:

1. Спроектирована сетевая metaCASE-система с возможностями:
    a. многопользовательской работы;
    b. удалённого хранения пользовательских данных;
    c. возврата пользовательских данных к предыдущим состояниям;
    d. расширения встроенного языка сторонними разработчиками;
    e. реализации нескольких клиентов.
2. Реализован прототип этой системы, включающий:
    a. простую версию языка на основе Haskell, содержащую такие объекты, как
        i. графы, его вершины и узлы;
        ii. атомарные действия с предусловиями и постусловиями;
        iii. функции для редактирования графа;
    b. сервер, способный хранить и преобразовывать диаграмму и её историю состояний;
    c. интерпретатор в составе сервера для преобразования диаграмм;
    d. CLI версию клиента, способную загружать, отображать и редактировать диаграмму;
    e. GUI версию клиента, способную загружать и отображать диаграмму.

Сам проект расположен на github: https://github.com/kirillt/hreal

В дальнейшем планируется дополнить этот прототип:

1. Полноценным графическим редактором диаграмм.
2. Базовыми визуальными языками (наподобие UML).
3. Улучшить встроенный язык для поддержки фильтрации элементов диаграммы с целью раздельного редактирования.

Выводы

В результате, новая система стала лучше по некоторым параметрам:

функция или свойство системы	**текущая реализация в QReal**	**реализация в новой архитектуре**
многопользовательская работа	использует синхронизацию через SVN, есть возможность использовать сетевой репозиторий	синхронизация не требуется, т.к. данные всех пользователей хранятся в одном месте
управление историей изменений	не реализована, эффективная реализация требует разделения общих данных для хранения списка состояний, либо протоколировать и уметь обращать все операции	история доступна для просмотра, возможен последовательный откат изменений
возможность пользовательской автоматизации	с помощью переиспользования диаграмм, создания визуальных языков	*кроме этого*, пользователь может составлять сложные команды-скрипты, используя встроенный язык
возможность расширения системы	непосредственная реализация новых программных модулей	*кроме этого*, можно расширять систему с помощью расширения встроенного языка

Клиент-серверная архитектура является важной частью новой архитектуры, но относительно QReal она не является новой, т.к. использование сетевого репозитория было в старых версиях QReal.

С другой стороны, в новой архитектуре делается сильный акцент на использовании EDSL, с помощью которого предполагается реализовывать ту функциональность, которая реализована в QReal в коде самой системы.

Пример: диаграммы можно реализовать в виде подграфов особой структуры (или со специальными вершинами-метками), а их обработку (создание, переименование, наполнение корневыми вершинами) вынести в функции языка. Как следствие, код самой системы не изменяется, расширяется только язык, что является более гибко.