САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
Математико-механический факультет

Кафедра системного программирования

Технология сквозного проектирования программного обеспечения в среде QReal

Дипломная работа студентки 545 группы

Жуковой Беллы Юрьевны

Научный руководитель		д. ф.-м. н., проф. Терехов А.Н.
	/ подпись /	
Рецензент		ст. преп. каф. сист. прогр.
	/ подпись /	Литвинов Ю.В.
“Допустить к защите”		д. ф.-м. н., проф. Терехов А.Н.
заведующий кафедрой,	/ подпись /	

Санкт-Петербург
2010

SAINT PETERSBURG STATE UNIVERSITY
Mathematics & Mechanics Faculty

Software Engineering Chair

End-to-End Software Design Technology in QReal

by

Bella Zhukova

Master's thesis

Supervisor	Professor A. N. Terekhov
Reviewer	Senior Lect. Y. V. Litvinov
"Approved by" Head of Department	Professor A. N. Terekhov

Saint Petersburg
2010

Оглавление

Введение 4

1. Описание предметной области 9

2. Обзор существующих решений 15

3. Описание предлагаемого решения 17

4. Реализация 24

4.1. Что может указать пользователь 28

4.2. Что будет сохранено автоматически 30

5. Апробация технологии 37

6. Заключение 41

Литература 43

Введение

Тенденции развития современных информационных технологий приводят к постоянному возрастанию сложности информационных систем [6]. Современные крупные проекты характеризуются, как правило, следующими особенностями:

- сложностью описания (достаточно большое количество функций, процессов, элементов данных и сложное взаимодействие между ними), требующей тщательного моделирования и анализа данных и процессов;
- наличием совокупности тесно взаимодействующих компонентов (подсистем), имеющих свои локальные задачи и цели;
- отсутствием прямых аналогов, ограничивающим возможность использования каких-либо типовых проектных решений и прикладных систем;
- необходимостью интеграции существующих и вновь разрабатываемых приложений;
- функционированием в неоднородной среде на нескольких аппаратных платформах;
- разобщенностью и разнородностью отдельных групп разработчиков по уровню квалификации и сложившимся традициям использования тех или иных инструментальных средств.

Способами решения проблем, вызванных возрастающей сложностью разрабатываемых программных систем, в частности, являются частичная или полная автоматизация процесса разработки и использование языков описания системы более высокого уровня, чем стандартные языки программирования (такие как, например, Java и C#), автоматическая или полуавтоматическая

трансляция описания на языках более высокого уровня в описание на языках более низкого уровня, то есть трансформация программы из одного языка в другой [13]. Самые первые исследования по трансформации программ были связаны с проблемами разработки оптимизирующих компиляторов, но впоследствии выяснилось, что эти исследования и технологии, использующие их результаты, могут успешно применяться не только при разработке методов компиляции и оптимизации программ, но и во многих других областях разработки программного обеспечения (ПО). Применение систем трансформации программ способствует повышению производительности труда разработчиков ПО, обеспечивая возможность работы с ПО на более высоком уровне абстракции, повышая удобство сопровождения и возможности повторного использования разработанного ПО [10].

В настоящее время исследования по методам анализа и трансформации программ развиваются особенно интенсивно, на их основе разрабатываются и успешно внедряются новые прогрессивные технологии разработки и верификации ПО [2].

С ростом возможностей вычислительной техники растет объем задач как таковых. Для обсуждения концепций их дизайна между разработчиками очень часто используется UML (Unified Modeling Language) [4]. Проблема заключается в том, что ввиду структурной и функциональной сложности моделируемой системы соответствующие графические представления получаются громоздкими и в результате трудно читаемыми. Кроме того, при построении систем определенного уровня сложности избежать ошибок в принципе невозможно, а возрастающая сложность предоставляет все больше шансов для их совершения, при этом затрудняя быстрое их обнаружение. Нередко получается, что определены узлы, в которых нет надобности, или же образовались неиспользуемые фрагменты.

Конечная цель проекта — создание CASE-средства, удобного для пользователя, но ограничивающего в разумных пределах его степени свободы. Ограничения должны предотвращать возможность возникновения ошибки с точки зрения проектирования и требований конечного языка, чтобы сгенерированный в итоге код был исполняемым. Следовательно, необходимо создать набор ограничений. Это облегчит этап проектирования, так как будет указывать на возможные упущения, и облегчит этап сопровождения, потому что позволит контролировать взаимосвязи между различными частями программы, как структурными, так и теми, которые касаются описания поведения смоделированного продукта. Кроме того, это даст большую уверенность в том, что автоматически сгенерированный код будет исполняемым.

Целью данной дипломной работы является разработка способа объединения диаграмм, представляющих моделируемую область с разных точек зрения, в цельное описание. В качестве решения были предложены технология, позволяющая объединить такие графические нотации путем создания нового типа логической связи, а также способ задания таких логических связей, которые будут соединять элементы диаграмм, описывающих моделируемую систему, и перемещения по ним. Наличие такой технологии упростит процессы разработки и сопровождения, так как поможет ускорить изучение логики системы.

Для достижения поставленной цели было необходимо решить следующие задачи:

- изучить спецификацию графической нотации;
- выяснить, какие логические связи между выбранными диаграммами и в рамках каждой из них в отдельности надо добавить, чтобы описание стало цельным и удобным для пользователя;

- программно реализовать созданную технологию для CASE-системы с открытым исходным кодом QReal (то есть реализовать возможность описания новой логической связи и перемещения по ней);
- провести апробацию разработанного варианта технологии.

Так как конечным результатом процессов разработки и сопровождения должен быть исполняемый код системы на целевом языке, то в список задач было добавлено:

- выделить требования к моделям на соответствие графической нотации;
- разработать средства верификации для проверки предоставленных пользователем графических моделей, чтобы на вход генератору поступала корректная с точки зрения синтаксиса языка модель;
- создать генератор кода по диаграммам.

В качестве графической нотации использовался язык UML [42, 41], в качестве целевого языка для генератора — Java.

Предметом исследования являются подходы и алгоритмы создания графических моделей системы, CASE-средства и предлагаемые ими способы анализа и трансформации программ.

Объектом исследования являются графические модели логики проектируемой системы.

Структура работы

Во введении обосновывается актуальность темы исследования, постановка задачи, формулируются цели и задачи работы.

В первой главе приведено описание предметной области, рассказано про визуальное моделирование, стандарт UML, чьи графические нотации

рассматривает технология, инструментальное средство QReal, в рамках которого проводилась реализация технологии.

Вторая глава содержит краткие описания графических представлений, рассматриваемых технологией, предложен способ их структуризации и, как следствие из нее, перечислены возможные дополнительные по отношению к стандарту UML логические связи между ними, с помощью которых модель описанной системы становится цельной.

Третья глава содержит обзор существующих решений.

В четвертой главе описаны сама технология (перечислены введенные дополнительные логические связи), способ ее реализации в среде QReal (то есть добавления новой логической связи в уже существующее описание графической нотации) и возможности, предоставляемые QReal для навигации по этим логическим связям.

В пятой главе содержится описание проведенной в ходе работы апробации.

В шестой главе указаны полученные в ходе работы результаты и перечислены пути возможных дальнейших исследований в этой области.

Также приведен список использованной в ходе работы литературы.

1. Описание предметной области

В последнее время все больше интереса проявляется ко всем аспектам, связанным с разработкой сложных программных приложений. Существует высокий спрос на разработку и верификацию методов и подходов, позволяющих автоматизировать создание сложных программных систем, потому что систематическое использование таких методов позволяет значительно улучшить качество, сократить стоимость и время разработки системы.

Хорошие визуальные модели системы позволяют наладить взаимопонимание между заказчиками, пользователями и командой разработчиков. Они обеспечивают ясность представления, в частности касательно выбранной архитектуры проекта, и позволяют лучше понять разрабатываемую систему. С развитием науки сложность разрабатываемых систем постоянно увеличивается, и поэтому возрастает необходимость в кратких, но в то же время ясных и выразительных методах моделирования. Язык моделирования, как правило, включает в себя:

- элементы модели — фундаментальные концепции моделирования и их семантику;
- нотацию — визуальное предоставление элементов моделирования;
- принципы использования — правила применения.

Построение визуальных моделей позволяет решить сразу несколько типичных проблем. Во-первых, оно позволяет облегчить работу со сложными и системами и проектами. Как правило, сложность и объем программных систем возрастает по мере создания новых версий. И в какой-то момент наступает "эффект критической массы", когда дальнейшее развитие системы становится крайне затруднительным, потому что она становится чересчур большой и

детализированной. Во-вторых, визуальные модели позволяют содержательно организовать общение между заказчиками и разработчиками. Очень часто оказывается, что заказчик не может сразу четко сформулировать проблему или описать все, что он хочет в результате от системы. И даже если на начальном этапе работы над проектом системы заказчик думает, что точно знает, чего хочет, то, как правило (и об этом свидетельствует богатый опыт), в ходе выполнения проекта его требования изменяются. Кроме того, динамика бизнеса объективно заставляет менять требования к разрабатываемой (или поддерживаемой) системе.

Визуальное моделирование не является методом, способным решить все проблемы, однако при должном использовании оно может существенно облегчить достижение таких целей, как:

- повышение качества программного продукта;
- сокращение стоимости проекта;
- поставка системы в запланированные сроки.

Технология визуального проектирования применяется уже довольно долго. Основная причина ее появления состоит в том, что языки программирования не обеспечивают нужный уровень абстракции, способный облегчить процесс проектирования. В настоящее время унифицированный язык моделирования (UML — Unified Modeling Language) является одним из наиболее популярных инструментов в сфере разработки объектно-ориентированных систем. Он представляет собой открытый стандарт, находящийся под управлением группы OMG (Object Management Group), открытого консорциума компаний. UML появился в результате процесса унификации множества объектно-ориентированных языков графического моделирования крайне популярных в конце 1980-х и начале 1990-х годов. [5]

Обычно выделяют три режима использования UML: режим эскиза, режим проектирования и режим языка программирования. В режиме эскиза разработчики используют UML для обмена информацией о различных аспектах системы. UML как средство проектирования нацелен на полноту [14]. То есть подразумевается, что проектировщик будет создавать как можно более полную и подробную модель, чтобы программисту, выполняющему кодирование, не надо было задумываться над архитектурой. По полным UML-моделям, описывающим всю систему, есть возможность автоматически создавать программы. Это и есть режим использования UML в качестве языка программирования.

В UML основная единица спецификации системы — это модель. Она представляет собой определенный взгляд на предметную область. Но так как UML — язык моделирования и должен быть понятен как человеку, так и машине, в нем существует понятие диаграммы. Если UML-модель может быть сложной и малодоступной для понимания целиком одним человеком, то диаграмма всегда является некоторым «осознаваемым» срезом модели, который можно охватить одним взглядом. Обычно одной UML-модели соответствует множество UML-диаграмм, представляющих эту модель с различных точек зрения в понятном человеку виде. Существует тринадцать типов UML-диаграмм, каждый из которых предназначен для своей определенной цели и может содержать определенный набор классов элементов модели: диаграммы классов, объектов, пакетов, деятельности, вариантов использования, последовательностей, состояний и другие. [13]

Важнейшим элементом в архитектуре UML является понятие метамодели, или модели более высокого уровня. Можно считать, что конкретная система сущностей предметной области представляет собой модель нулевого уровня. Модель, описывающая классы этих объектов, их ассоциации и поведение, представляет собой модель первого уровня, собственно, она и называется

моделью. Ее структура обычно проще, чем структура системы. Модель, описывающая классы элементов различных моделей первого уровня (классы, ассоциации, атрибуты, операции, состояния и т.д.), называется моделью второго уровня, или *метамоделью*. Она задает язык описания моделей. В архитектуре UML определяется наличие *метаметамодели*, или модели третьего уровня, описывающей структуру различных метамоделей. Метаметамодель — это язык для создания метамоделей. Ее структура уже очень простая и не поддается дальнейшему упрощению (то есть модель четвертого уровня, или метаметаметамодель, будет иметь ту же структуру, что и модель третьего уровня, или метаметамодель). Модели всех уровней описаны средствами UML в спецификации языка.

Очень важным свойством UML является его расширяемость. UML задумывался как язык моделирования, применимый в самых различных областях, и, поскольку на этапе проектирования невозможно было предусмотреть все области применимости UML, в язык были заложены механизмы расширения. С помощью этих механизмов пользователь может определять свои диалекты языка, вводить новые элементы. Существует три основных типа конструкций для расширения:

- Ограничения (constraints) — расширяют семантику конструкций UML, позволяя создавать новые и отменять существующие правила.
- Стереотипы (stereotypes) — расширяют словарь UML, позволяя на основе существующих элементов языка создавать новые виды строительных блоков, аналогичные существующим, но специфичные для данной задачи, то есть ориентированные для решения конкретной проблемы.
- Теговые (помеченные) значения (tagged values) — расширяют свойства основных конструкций UML, позволяя включать дополнительную информацию в спецификацию элемента.

Совместно эти три механизма расширения языка позволяют модифицировать его в соответствии с потребностями проекта или особенностями технологии разработки. Набор ограничений, теговых значений и стереотипов, предназначенный для некоторой предметной области, является *профилем*. [21]

QReal — мета-CASE-средство, один из вариантов настройки которого может быть спозиционирован как конкретное CASE-средство, поддерживающее диаграммы UML. Оно представляет собой набор взаимосвязанных графических редакторов, репозиторий, а также набор генераторов, объединенных в интегрированную среду разработки. QReal обеспечивает одновременную работу многих пользователей, работающих над одним проектом, причем она может выполняться на разных платформах. Кроме того, есть возможность добавления поддержки новых языков, что позволит повысить эффективность разработки.

Языковые средства включают в качестве основного языка спецификаций язык UML, а в качестве целевого можно выбрать, например, Java или C#. Также есть возможность сериализации модели в XMI-формат для последующего экспорта в другие программные средства.

Методологические средства основаны на идее описания программной системы с помощью семейства спецификаций, связанных различными отношениями. Семейство спецификаций эволюционирует в процессе работы над проектом, в него добавляются новые спецификации и модифицируются существующие, но в любом случае должны сохраняться свойства правильности перехода по отношениям между спецификациями, что позволяет повысить степень уверенности в правильности итоговой программы и существенно облегчить и сделать более управляемым процесс внесения изменений в программу [12].

Инструментальные средства системы включают средства для добавления спецификаций в качестве плагина, средства для редактирования спецификаций, инструментальные средства целевого языка программирования и набор инструментов, предназначенных для трансляции спецификаций в целевой язык (в частности, исполняемый программный код или же в XMI-формат для дальнейшего экспорта).

2. Обзор существующих решений

В таблице 2.1 приведен обзор существующих UML CASE-средств. Исследовалось наличие в них дополнительных относительно UML стандарта логических связей и возможностей генерации кода по диаграммам в язык Java.

Таблица 2.1. UML CASE-средства.

Название	Связи между диаграммами	Генерация	
		по диаграммам	кода на Java
Visual Paradigm for UML	Нет	Class Diagram	Есть
ArgoUML	Нет	Class Diagram	Есть
Microsoft Visio	Нет	Нет	Нет
Dia	Нет, с элементом можно связать любую диаграмму	Class Diagram	Есть
Enterprise Architect	Да, довольно сложная, включающая в себя много диаграмм, не все из которых входят в стандарт UML	Class Diagram	Есть
UModel	Нет	Class Diagram	Есть
Poseidon	Нет	Class Diagram	Есть
Borland	Нет	Class Diagram	Есть

Together			
BoUML	Class Method и Activity Diagram	Class Diagram	Есть
IBM Rhapsody		Не имеет в своей палитре Activity Diagram	Есть
UML Studio	Нет	Class Diagram	Есть
Eclipse UML2 Tools	Нет	Class Diagram	Есть
Sybase PowerDesigner	Нет	Class Diagram	Есть
Real	Class Method и SDL	Не имеет в своей палитре Activity Diagram	Есть
Fujaba Life	Class Method и Story Diagram	Class Method и Story Diagram	Есть
UniMod	Class Method и State Machine Diagram	Class Diagram и State Machine Diagram	Нет

Были использованы материалы ресурсов [15, 19, 20, 22, 24, 25, 26, 27, 28, 32, 33, 35, 43, 44, 45, 46, 47].

3. Описание предлагаемого решения

Одним из основных путей использования компьютера является облегчение труда человека в разных областях (математические вычисления, создание документаций, эмуляция программ или устройств и много других). Но для этого изначально надо построить модель предметной области, то есть перевести задачу из области реального мира в область, которая реализована в целевой вычислительной технике, со всеми ее ограничениями и возникающими в результате работы артефактами. Окружающая реальность существует на основе своей собственной логической структуры, в то время как машинная реализация основана на другой структуре вследствие ограниченных аппаратной реализацией возможностей. Таким образом, разработчик оказывается механизмом перехода от одной структуры к другой. В качестве инструмента используются диаграммы.

UML построен на основе двух категорий: сущность и связь, это не философские, это внутренние категории UML. Многообразие реальности, построенной на совершенно другой архитектуре категорий, с помощью только этих двух не описать, таким образом, возникает определенное противоречие между целями и существующими средствами. В UML его снимают наличием нескольких видов диаграмм (в UML 2.2 их тринадцать). Следовательно, на данном этапе развития технологии проверить, правильно ли система описана с помощью диаграмм или в процессе моделирования были упущены из виду какие-то аспекты, существующие в реальности, не представляется возможным [47]. Таким образом, необходимо проверить предоставленную пользователем модель на целостность и соответствие описания элементов диаграмм спецификации UML.

Из возможных в стандарте UML диаграмм были выбраны три, как самые основные. Это диаграмма вариантов использования (use case diagram), диаграмма деятельности (activity diagram) и диаграмма классов (class diagram), что позволило логически поместить модель проектируемой системы в трехмерное пространство, осями которого являются: функциональность, поведение, структура.

На ранних этапах проектирования программного продукта используются диаграммы вариантов использования (use case diagram). Они описывают функциональное назначение системы или то, что система должна делать. Разработка диаграммы в дальнейшем помогает достижению следующих целей:

- определить общие границы и контекст моделируемой предметной области;
- сформулировать общие требования к функциональному поведению проектируемой системы;
- разработать исходную концептуальную модель системы для ее последующей детализации в форме логических и физических моделей;
- подготовить исходную документацию для взаимодействия разработчиков системы с ее заказчиками и пользователями.

Суть диаграммы вариантов использования (иногда она называется диаграммой прецедентов) состоит в том, что проектируемая система представляется в виде множества сущностей или экторов (элемент Actor), взаимодействующих с системой с помощью вариантов использования. При этом эктором называется любая сущность, взаимодействующая с системой извне. Это может быть человек, техническое устройство, программа или любая другая система, которая может служить источником воздействия на моделируемую систему так, как определит сам разработчик. Вариант использования служит для описания сервисов, которые система предоставляет эктору. Логика работы каждого прецедента должна быть описана с помощью других моделей UML

и/или текстового документа, определяющего, что должна делать система, когда эктор инициирует прецедент использования. Данная диаграмма — механизм, позволяющий описать систему с точки зрения потенциальных пользователей. [8, 40, 36]

Для описания поведения обычно используются следующие диаграммы:

- деятельности (activity diagram);
- последовательностей (sequence diagram);
- автоматов (state machine diagram).

Диаграмма деятельности близка к блок-схемам (позволяющим описывать алгоритмы или процессы, изображая шаги в виде блоков, каждый из которых соответствует выполнению одного или нескольких действий) и предоставляет возможности для описания параллельных процессов. Она единственная позволяет точно специфицировать поведение, в ней есть элементы (actions, pins, signals, etc.), позволяющие уточнять элементы, обладающие поведением, такие как метод класса или результат перехода из одного состояния в другое на диаграмме состояний [22]. На диаграмме последовательностей основное внимание уделяется временной упорядоченности событий, на ней изображают множество объектов и посланные или принятые ими сообщения. Диаграмма состояний показывает автомат, содержащий состояния, переходы, события и действия, основное внимание в них уделяется порядку возникновения событий, связанных с объектом. Таким образом, диаграмма деятельности делает упор на описание самого алгоритма, а две другие диаграммы — на жизненный путь объектов. Так как была необходимость описания именно алгоритма реализации функциональности, была выбрана диаграмма деятельности.

На ней определяется последовательность шагов в ходе выполнения некоторой задачи. Эта последовательность может быть спецификацией прецедента использования, кроме того, она может быть частью функциональных свойств, которые можно многократно переиспользовать в

различных частях системы [1]. Ее часто применяют для визуализации особенностей реализации операций классов, когда необходимо представить алгоритмы их выполнения. Диаграммы деятельности позволяют реализовывать в языке UML особенности процедурного, синхронного и асинхронного управления. Кроме того, они хорошо подходят для проектирования системного уровня ПО для встроенных систем [23]. Основными элементами диаграммы являются действия (actions), связанные между собой переходами. [8, 40, 29, 18]

Для визуализации статической модели преимущественно используется диаграмма классов (class diagram). Она позволяет описывать структуры данных и отношения между ними, а также идентифицирует операции над этими данными. Модели классов определяют структуры, отражающие внутреннее состояние системы. Они идентифицируют классы и их атрибуты, включая отношения. Кроме того, они определяют операции, необходимые для реализации требований динамического поведения системы, зафиксированных в прецедентах использования. Классы, реализованные на конкретном языке программирования, отражают статическую структуру и динамическое поведение приложения. [8, 40, 20]

Таким образом, выбрав эти три диаграммы, мы получили возможность описать систему с точки зрения функциональности и статической структуры, а также специфицировать реализацию операций элементов статической структуры.

От того, как эти три вида диаграмм будут организованы, зависит определение корректности.

Рассмотрим три возможных типа организации структуры.

- Иерархия (многоуровневая форма организации объектов со строгой соотнесенностью объектов нижнего уровня определенному объекту верхнего уровня).

Иерархия диаграмм, каждая из которых представляет собой граф. Следовательно, для такого типа описания структуры корректность — условия на графы.

- Одноуровневые независимые описания.

Объекты расположены на одном уровне, но описывают изучаемую область с разных сторон. В нашем случае это можно представить как описание реальности с разных независимых точек зрения. Так как они независимы, то проверить их можем только каждую в отдельности (как граф), а для проверки их в совокупности необходимо разработать способ описания связей между ними.

- Смешанный.

То есть присутствуют черты и иерархии и одноуровневых независимых описаний.

Учитывая специфику самого стандарта UML (разные виды диаграммы создавались специально для рассмотрения моделируемой системы с разных точек зрения) и человеческого мышления (стремление структурировать информацию), наиболее удачным был признан третий тип, так как он объединяет в себе и иерархию, и описание области с разных позиций.

На самом высоком уровне иерархии расположена диаграмма прецедентов. Для каждого прецедента, определенного на ней, должны быть указаны методы, определенные пользователем на диаграмме классов, которые являются реализацией заявленной в прецеденте функциональности. Таким образом, получается, что на втором уровне иерархии расположена диаграмма классов. Для метода необходимо представить алгоритм его выполнения. Это можно осуществить с помощью диаграммы деятельности, что ставит ее на нижнюю ступень иерархии. Такой способ структуризации определяет, в частности, и связи между диаграммами, которые в самом стандарте UML независимы, что позволяет сформулировать требования на целостность описания системы. Под

целостностью понимается наличие всех необходимых для генерации кода диаграмм, с указанными логическими связями, удовлетворяющими ниже определенным требованиям. Из этого следует, что по диаграммам может быть автоматически создан код, а это означает, что все необходимые для создания кода части системы были описаны. Кроме того, подобная структура дает возможность проецировать любой элемент графического представления на любую из трех осей, что оказывается полезным при изучении структуры системы для дальнейшей разработки или сопровождения.

Под проверкой каждой диаграммы «как граф» подразумевается соответствие всех ее элементов требованиям, указанным в описании стандарта UML, то есть применение статического метода верификации [7, 16]. Так как QReal поддерживает нотацию UML не в полной мере, то необходимо поддерживать соответствие не всем обозначенным там требованиям, а только тем, которые распространяются на поддерживаемое нашим средством подмножество.

Требования к целостности описания системы вытекают из структуры организации диаграмм и получаются следующими:

- для каждого прецедента на диаграмме случаев использования должен быть указан метод, объявленный на диаграмме классов (это не критическое требование, но иначе заявленная функциональность не будет реализована);
- для каждого метода, объявленного на диаграмме классов, указана ровно одна диаграмма деятельности, которая является описанием алгоритма его работы (требование не критическое, но иначе такой метод будет пустым);
- на диаграммах деятельности в качестве объектов используются только экземпляры элементов диаграммы классов (доступные для этой

диаграммы как реализации алгоритма действия), поля и методы определенного класса и только с соответствующим уровнем доступа.

Последнее требование действует не во всех формальных семантиках диаграммы активностей. В [30, 31], например, подразумевается, что эктор владеет прецедентом, который, в свою очередь, обладает всеми атрибутами, содержащими данные, относящиеся к этому прецеденту. Но в силу того, что в описанной выше архитектуре диаграмма деятельностей не самостоятельна, а существует только для описания алгоритмов методов классов в объектно-ориентированном подходе, то, следовательно, она имеет доступ только к тем атрибутам и методам, которые доступны классу, чей метод она реализует.

Наличие требований, полученных из спецификации самого UML, связано с тем, что на данный момент не разработана технология задания и реализации проверки ограничений для графических нотаций, а использование генератором в качестве входных данных некорректной модели может привести к аварийному завершению работы программы.

4. Реализация

Есть три типа диаграмм. По каждому элементу на них нужно иметь возможность узнать, с какими другими элементами и диаграммами он связан, то есть какие из них он использует и где используется.

Для этого был введен новый тип связи — провязка.

Определение: Под *элементами диаграммы А* понимаются элементы, которые могут присутствовать на такого типа диаграмме в соответствии со стандартом UML, а также элемент, представляющий диаграмму.

Например, элементами диаграммы классов являются Class, Class Field, Class Method, Interface, Generalization и другие элементы, которые по стандарту могут быть на диаграмме классов, а также нововведенный элемент Class Diagram, представляющий из себя описание графического и логического (список элементов, которые могут быть на данного типа диаграмме) представлений диаграммы.

Определение: *Провязкой* называется бинарная логическая связь между двумя несовпадающими элементами, описывающая отношение иерархии между ними.

Например, алгоритм метода класса (Class Method) может быть описан с помощью диаграммы деятельностей (Activity Diagram).

Определение: Элемент А *использует* элемент В, если для реализации А необходимо знать В.

Например, если в теле действия (Action) содержится

int i = getCount();

то в список элементов, которые используются этим действием, будут добавлены переменная i и использование метода getCount(). Кроме того, в

список элементов, которые используют метод getCount(), автоматически будет добавлен этот элемент Action.

Топология провязки изображена на рис. 4.1.

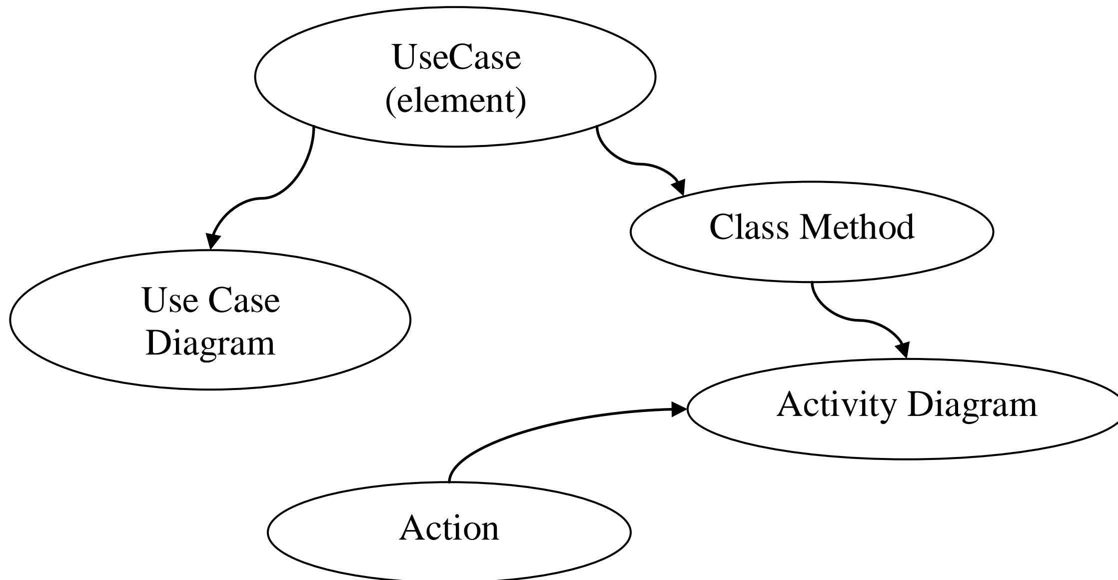

Рис. 4.1

Связи UseCase (element) → Class Method, Class Method → Activity Diagram возникли из структуры, которая была выбрана для организации UML диаграмм и вытекающего из нее понятия целостности модели системы. Возможность провязывания UseCase (element) → Use Case Diagram и Action → Activity Diagram была добавлена для облегчения восприятия диаграммы. При наличии таких связей есть возможность создания иерархии диаграмм. Нижние уровни уточняют соответствующие элементы верхних. Например, иногда возникает необходимость представить на диаграмме деятельности некоторое сложное действие, которое, в свою очередь, состоит из нескольких более простых действий. Такое действие является графом деятельности и обозначается специальной пиктограммой в правом нижнем углу. Подобная конструкция

может применяться к любому элементу языка UML, который поддерживает вложенность своей структуры.

Для взаимодействия с пользователем было создано всплывающее меню со следующими возможностями:

- add connection (добавить провязку);
- disconnect (удалить провязку);
- add usage (добавить использование (рис. 4.2));
- delete usage (удалить использование);
- go to.

Для просмотра полных списков, а также перемещения по связям на диаграмму, содержащую соответствующий элемент, в меню был добавлен пункт go to, содержащий следующие подпункты:

- backward connection (список элементов, которые создали провязку с данным элементом);
- forward connection (список элементов, с которыми данный элемент создал провязки);
- uses (список элементов, использующихся в данном элементе (рис. 4.3));
- used in (список элементов, использующих данный элемент).

Каждый из них так же разбивается на подпункты:

- by itself;
- with all children.

Таким образом, для каждого элемента есть возможность посмотреть, что он использует/где он используется сам по себе (by itself) или вместе со своими потомками (with all children), то есть показываются объединенные списки самого элемента и всех таких, которые являются его потомками в представлении диаграммы в виде графа (например, класс и его поля и методы).

Так как провязка возникла для реализации иерархии, в данном случае она тоже будет рассматриваться как отношение «родитель-потомок».

Эти списки частично заполняются пользователем в процессе создания модели системы (с помощью *add connection, add usage, disconnect, delete usage*), а частично автоматически: например, если пользователь создал для метода класса провязку с диаграммой деятельности (что влечет за собой добавление этой диаграммы в список *forward connections* данного метода), то нет необходимости вынуждать его указывать также, что она провязана с методом (то есть добавлять метод в список *backward connections* этой диаграммы), это действие выполнится автоматически.

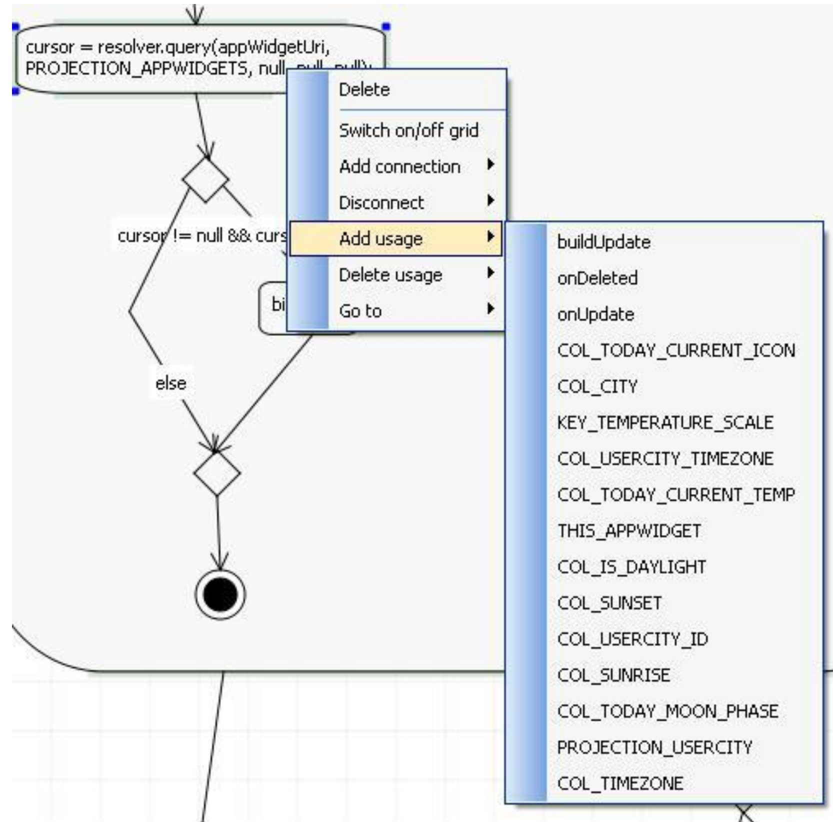

Рис. 4.2. Добавление использования.

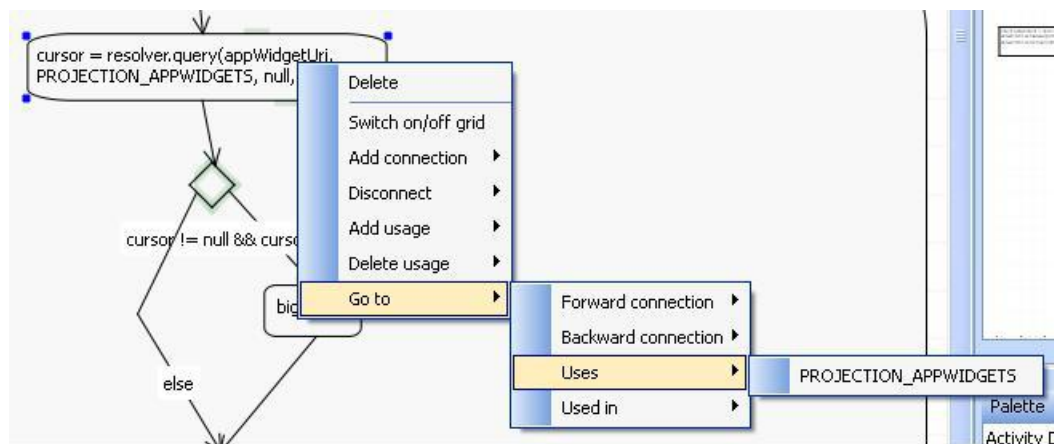

Рис. 4.3. Просмотр элементов, используемых данным.

4.1. Что может указать пользователь

В таблице 4.1, в графе «Провязать с» содержатся типы элементов, с которыми данный элемент может быть провязан. Таким образом, список *add connection* содержит ссылки на все существующие в модели экземпляры данных типов, и при выборе одного из них создается провязка между элементом, на котором было вызвано меню, и выбранным. В процессе моделирования может возникнуть необходимость в создании нового элемента, с которым надо провязать данный. Например, для метода класса необходимо создать диаграмму деятельности, содержащую описание его реализации. То есть надо совершить два действия: создать новую диаграмму и провязать с ней метод. Для таких ситуаций в список *add connections* добавляются ссылки на новые элементы. Они перечислены в графе «Может создать новую провязку с».

Определение: Под *Imported Class* подразумевается каждый класс из списка elementImport, являющийся одним из свойств элемента Class и отвечающий за то, какие классы и пакеты данный класс импортирует. Он не имеет графического представления, но включен в список элементов, потому что поля и методы класса могут использовать поля и методы импортированных классов,

что может оказаться важным в процессе разработки системы или ее сопровождении.

Таблица 4.1. Логические связи, которые пользователь может указать вручную.

Элемент	**Может использовать**	**Провязать с**	**Может создать новую провязку с**
Class Diagram			
Class			*Activity Diagrams* для всех его методов, которые еще не были реализованы.
Class Method		Activity Diagram, UseCase (element)	Activity Diagram, UseCase (element)
Class Field, Associations (directed, not directed)	Class Method, Class Field, Imported Class, Interface		
Activity Diagram			
Activity Diagram		Class Method, UseCase (element)	UseCase (element)
Action	Class Method, Class Field, Imported Class, Interface	Activity Diagram	Activity Diagram
DecisionNode	Class Method, Class Field, Imported Class, Interface		

MergeNode		DecisionNode	
Use Case Diagram			
UseCase (element)		Class Method, Activity Diagram, Use Case Diagram	Activity Diagram, Use Case Diagram

Примечания:

Association является просто особой формой представления Class Field, поэтому везде, где написано Class Field, подразумевается так же и Association.

Остальные элементы не рассматриваются в силу того, что пользователь ничего для них не может указать. Imported Class и Interface выделены, так как они могут быть указаны пользователем в качестве используемых.

4.2. Что будет сохранено автоматически

Списки *uses*, *forward connections* и *backward connections* будут изменены автоматически, основываясь на введенных пользователем данных (см. таблицу 4.2).

Определение: Рассмотрим две диаграммы с именами First и Second. Second называется *производной* от First (First *породила* Second), если на First диаграмме есть элемент Action (для Activity Diagram) или UseCase (для Use Case Diagram), в списке forward connections которого содержится Second диаграмма (которая является реализацией алгоритма работы этого Action или более подробным описанием логики прецедента UseCase). Говорим также, что Action *породил* Activity Diagram и UseCase (element) *породил* Use Case Diagram.

Таблица 4.2. Логические связи, которые буду сохранены автоматически.

	Uses	**Forward Connections**	**Backward Connections**
Class Diagram			
Class	*Class Method, Class Field, Imported Class, Interface*, которые используются (списки *uses*) его полями и методами.	Все те, что есть у его полей и методов.	Все те, что есть у его полей и методов.
Class Method	Элементы (*Class Method, Class Field, Imported Class, Interface*), которые использует (список *uses*) реализующая его *Activity Diagram* (и все ее производные *Activity Diagram*).	Activity Diagram	UseCase (element)
Class Field, Associations (directed, not			

directed)			
Activity Diagram			
Activity Diagram	Элементы (*Class Method, Class Field, Imported Class, Interface*) из списков *uses* элементов, находящихся на ней и на производных от нее *Activity Diagram*. Все производные от нее *Activity Diagram*.		Class Method, Action (из *forward connections*). *UseCase (element)* из списка *backward connections* для *Class Method*.
Activity	Элементы (*Class Method, Class Field, Imported Class, Interface*) из списков *uses* элементов, находящихся на ней и на производных от ее *Action* элементов *Activity Diagram*.		

Action			
DecisionNode			Merge Node
MergeNode		Decision Node	
Use Case Diagram			
Use Case Diagram	Все то, что используют прецеденты (*UseCase (element)*), находящиеся на ней (списки *uses*). *Use Case Diagram*, производные от этой.		UseCase (element)
UseCase (element)	*Class Method, Class Field, Imported Class, Interface*, которые используются при реализации этого прецедента, то есть находятся в списке *uses Activity Diagram*, провязанной с этим прецедентом, и ее производных,	*Class Method*, *Use Case Diagram*. *Activity Diagram* из списков *forward connections* для *Class Method*.	

	а также сама реализующая *Activity Diagram* со своими производными.		
Actor	Все те элементы, которые используются прецедентами (*UseCase (element)*), с которыми *Actor* соединен линками на диаграмме.		

Примечания:

Порядок построения списков *uses*, содержащих только элементы диаграммы классов (*Class Field, Class Method, Imported Class, Interface*) изображен на рис. 4.4.

Потом в каждый из них дописываются:

- *Class*

Те классы, поля или методы которых уже есть в списке.

- *Activity Diagram*

Такие диаграммы деятельности, которые являются реализацией включенных в список Class Method, а также все производные от них Activity Diagram.

Так как очевидно, что из использования самой диаграммы следует использование всех находящихся на ней элементов (*Action, Decision Node, Merge Node, Activity* на данный момент), то нет смысла включать их в список.

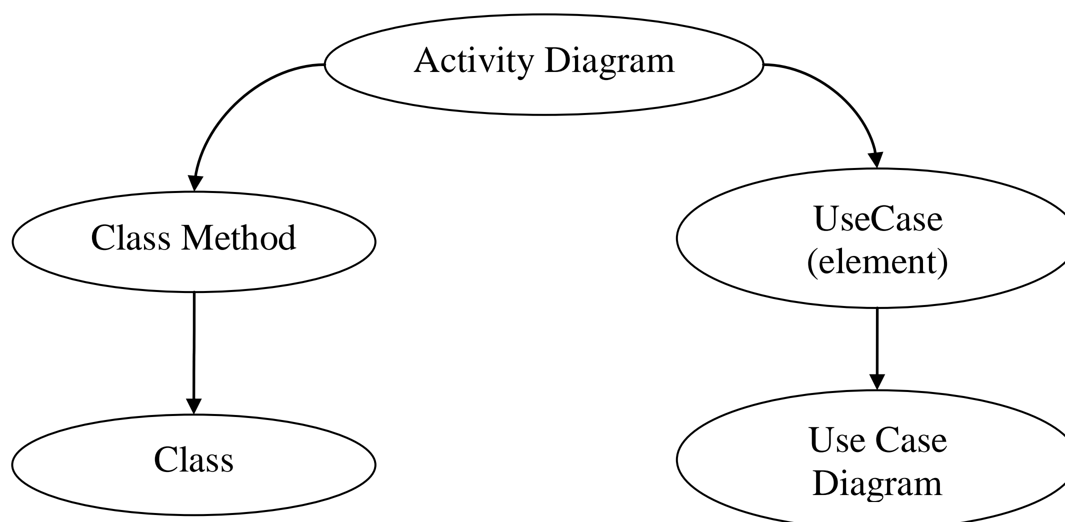

Рис. 4.4

Use Case Diagram и *UseCase (element)* для элементов Class Diagram и Activity Diagram не присутствуют в этом списке, так как они не могут быть использованы, они являются просто объявлениями функциональности.

Списки *used in* заполняются исходя из заполненных списков *uses*: если элементу A добавилось использование элемента B, то в список *used in* элемента B добавляется элемент A.

Аналогично реализовано добавление провязки и удаление использования и провязки.

Таким образом, из объединения списков *uses*, *used in*, *backward connections* и *forward connections* можно получить проекцию каждого элемента на любую из

трех осей. Можно проследить за любой проекцией, как до уровня диаграмм, так и до уровня элементов.

5. Апробация технологии

Для проверки предлагаемого решения, реализованного в QReal, а также определенных в ходе работы правил для верификации и генератора кода было выбрано приложение для мобильного телефона под управлением операционной системы Android: виджет, показывающий прогноз погоды на неделю (данные берутся с сайта), а также текущее время, дату, день недели и город, для которого отображается прогноз погоды.

В ходе проектирования была выявлена необходимость создания возможности провязки для таких элементов как Action и UseCase (element), то есть таких, которые поддерживают вложенность своей структуры. После добавления этих связей, а также добавления новых возможностей в саму среду, процесс создания модели стал заметно удобнее, хотя, с моей точки зрения, для многих программистов было бы привычней писать код для методов классов самостоятельно. С другой стороны, при сопровождении или изучении чужого кода разработанная технология помогает выявить зависимости между элементами программы. Таким образом, при наличии генератора диаграмм по исходному коду задача внесения изменений в код уже существующей, но незнакомой для программиста, системы решалась бы быстрее, чем при изучении программистом непосредственно кода.

Логика приложения для мобильного телефона была спроектирована на UML. Диаграмма прецедентов изображена на рис. 5.1.

Поскольку система нетривиальная, диаграмма классов содержит пятнадцать классов, а суммарное число диаграмм активностей существенно превосходит

двадцать. В тексте дипломной работы приведены только один класс (рис. 5.2) и диаграммы активностей для одного из его методов (рис. 5.3, 5.4, 5.5).

Необходимые провязки (например, между методом и диаграммой деятельности, то есть включение диаграммы в список метода *forward connections*) были созданы вручную, другие были добавлены автоматически (в данном случае, методы был автоматически добавлен в список диаграммы *backward connections*).

Была проведена серия экспериментов (снизу вверх и сверху вниз в иерархии зависимостей), что списки составляются корректно и своевременно изменяются.

Таким образом, в любой момент времени по элементу можно посмотреть, какие элементы будут задеты при удалении или изменении данного.

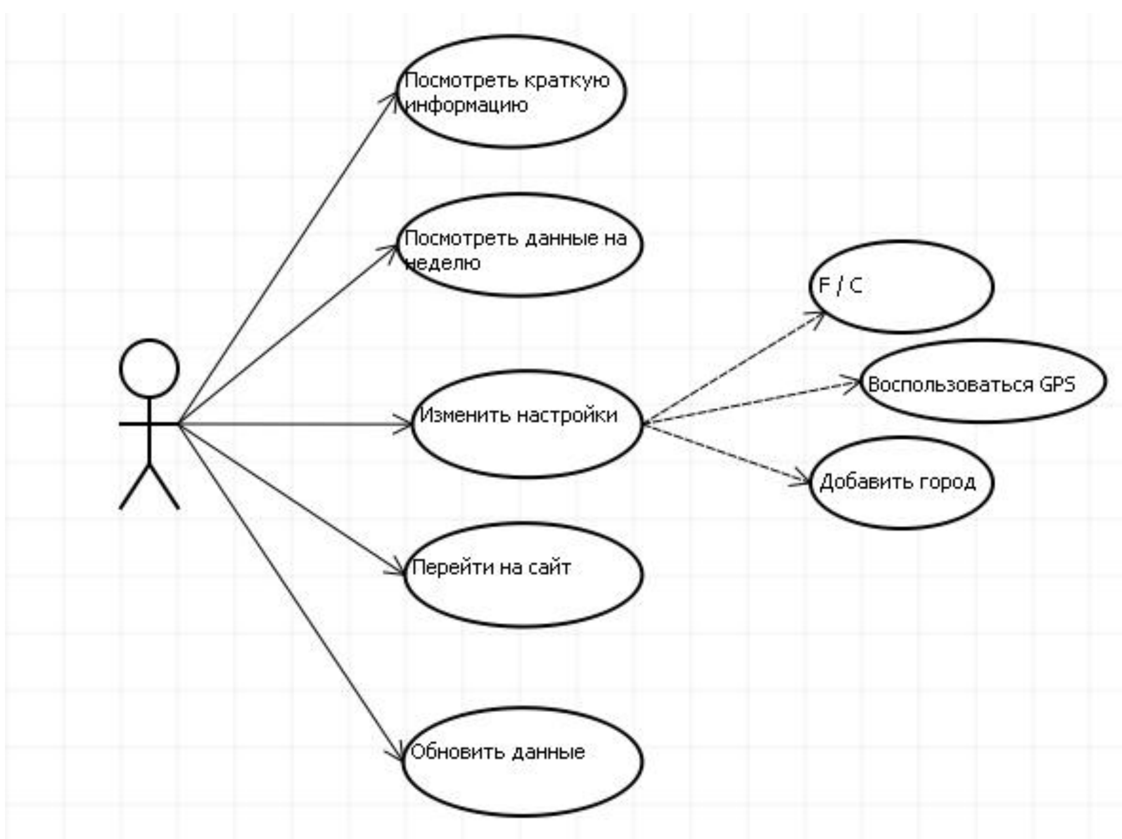

Рис. 5.1. Диаграмма прецедентов.

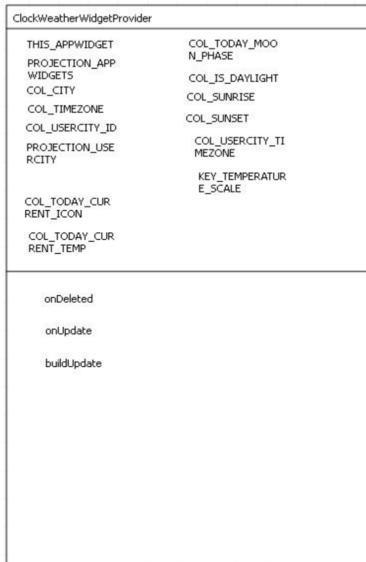

Рис. 5.2. Класс ClockWeatherWidgetProvider

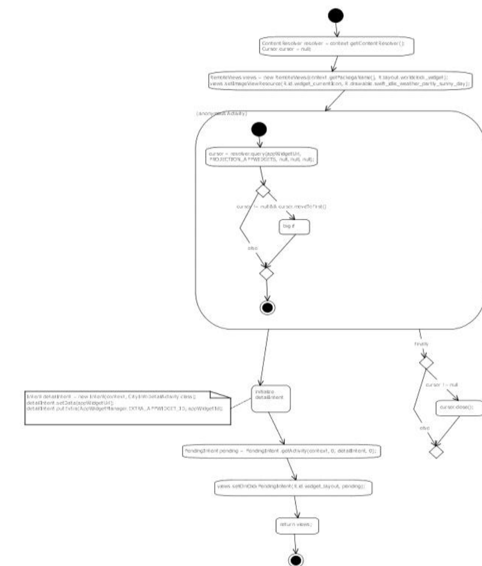

Рис. 5.3. Диаграмма деятельности. Метод buildUpdate

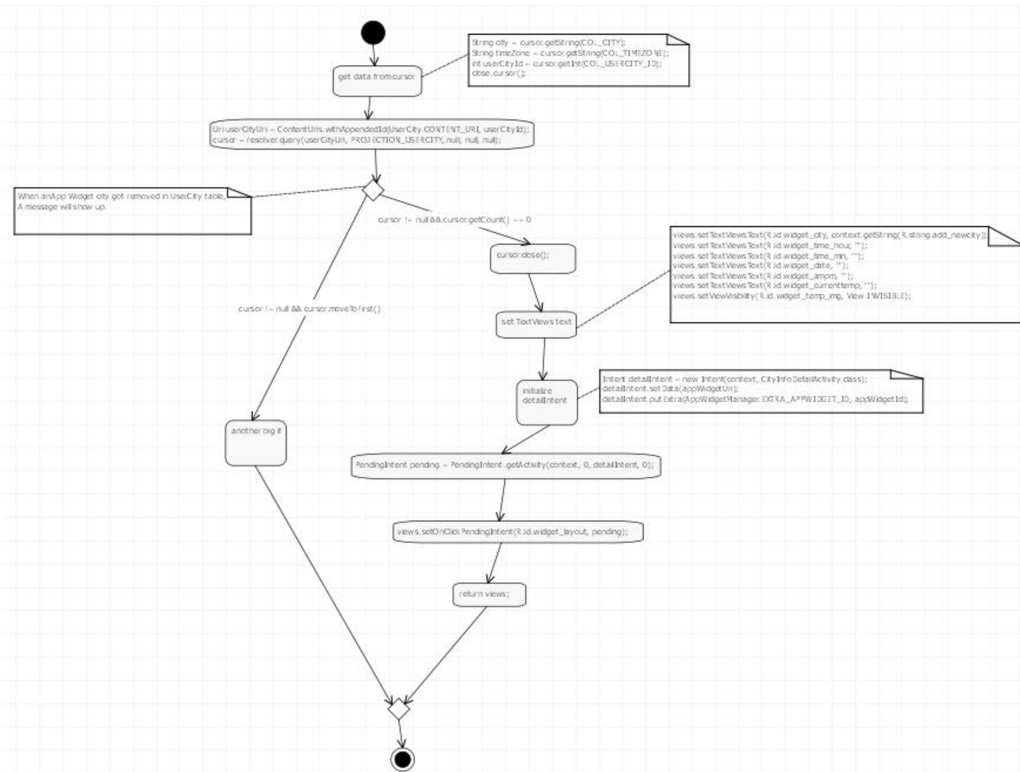

Рис. 5.4. Диаграмма деятельности. Описание одного из Action в методе buildUpdate.

Рис. 5.5. Диаграмма деятельности. Описание одного из Action в методе buildUpdate.

6. Заключение

Основные результаты

В работе получены следующие результаты

- изучена спецификация UML и выбраны основные, на мой взгляд, диаграммы (Activity Diagram, Class Diagram, Use Case Diagram);
- описаны метамодели выбранных диаграмм UML в среде QReal;
- сформулированы ограничения на создаваемые в QReal диаграммы;
- реализована их верификация;
- проведен анализ логических связей между элементами диаграмм, добавлен новый неграфический вид связи, которая призвана облегчить процесс изучения уже созданных частей системы;
- в среде QReal добавлены возможности для задания и визуализации подобных связей, а также организации переходов по ним;
- реализована генерация кода по диаграммам;
- проведена апробация технологии, верификации и генератора кода на приложении (виджете для мобильного телефона с операционной системой Android).

Дальнейшие исследования

- Доработка генератора до более полного соответствия спецификации UML.
- Автоматическое дополнение вводимого пользователем кода (аналоги, например, Autocomplete в IntelliJ Idea, IntelliSense в Microsoft Visual Studio).

- Реализация добавления и удаления элемента в список используемых в автоматическом режиме, путем анализа введенного пользователем кода.
- Разработка технологии и реализация возможности определения формальных методов верификации графических моделей в QReal для того, чтобы иметь возможность проверять, например, что Class не является потомком самого себя (явно или неявно) или UseCase (element) не включает самого себя (явно или неявно).
- Просмотр возможных путей в графе диаграммы деятельности. Выявление недостижимых, неиспользуемых элементов. Предоставление пользователю возможности просматривать пути.
- Создание средств отладки на уровне диарамм.
- Расширение XML-описания графических нотаций для QReal с целью создания возможности описывать в нем ограничения, накладываемые на логику элементов.
- Возможность создания профилей UML в QReal (технология и реализация).
- Семантический анализ графических представлений.
- Реализация reverse engineering, то есть генерации UML диаграмм по исходному коду системы.

Литература

1. Бок К. UML 2: модель деятельности и модель действий. URL: http://www.osp.ru/text/print/302/183924.html.
2. Гайсарян С.С., Иванников В.П.,Аветисян А.И. Анализ и трансформация программ. URL: http://www.ict.edu.ru/ft/005642/62319e1-st06.pdf.
3. Гуров В., Нарвский А., Шалыто А. Исполняемый UML из России. PC Week/RE. 2005. № 26. С. 18, 19.
4. Кознов Д.В., Перегудов А.Ф. «Человеческие» особенности использования UML. Системное программирование. Вып. 1: СПб.: Изд-во СПбГУ, 2005. С. 4-17.
5. Кознов Д.В., Перегудов А.Ф., Романовский К.Ю., Кашин А.А, Тимофеев А.Е. Опыт использования UML при создании технической документации. Системное программирование. Вып. 1: СПб.: Изд-во СПбГУ, 2005. С. 18-35.
6. Кузнецов М.Б. Трансформация UML-моделей и ее применение в технологии MDA. URL: http://www.citforum.ru/SE/project/uml_mda/.
7. Кулямин В.В. Методы верификации программного обеспечения. Статьи конкурса по ИТС.
8. Мацяшек Л.А. Анализ и проектирование информационных систем с помощью UML 2.0. 3-е издание. М., СПб., Киев: Вильямс, 2008.
9. Методология и CASE-средство RTST++. UML: http://real.tepkom.ru/Report_Methodology.php.
10. Наганов М.В. Автоматизированная генерация кода для интеграции систем управления телекоммуникациями. URL: http://www.math.spbu.ru/D21223251/AUTO/Наганов М.В.pdf.

11. Управляющие конструкции языка Java. URL: http://ru.sun.com/research/materials/Monakhov_Java/Chapter5.pdf.

12. Чунихин О.Ю. Транслятор UML-спецификаций в исполняемый код. Сибирский журнал индустриальной математики. 2004. №2 (18). С. 133-147.

13. Чунихин О.Ю. Формальная модель машин состояний UML. Сибирский журнал индустриальной математики. 2004. №1 (17). С. 152-165.

14. Фаулер М. UML Основы. 3-е издание. СПб.: Символ, 2006.

15. ArgoUML. URL: http://argouml.tigris.org/.

16. Baldan P., Corradini A., Gadducci F. Specifying and Verifying UML Activity Diagrams via Graph Transformation. URL: http://www.springerlink.com/content/vdayh642tcvapr95/.

17. Barra E., Gnova G., Llorens J. An Approach to Aspect Modeling with UML 2.0 5th International Workshop on Aspect Oriented Modeling, 2004.

18. Borger E., Cavarra A., Riccobene E. An ASM Semantics for UML Activity Diagrams. AMAST 2000, LNCS 1816, pp. 293-308, 2000.

19. Borland Together. URL: http://www.borland.com/us/products/together/index.html.

20. BoUML. URL: http://bouml.free.fr/.

21. Cepa V., Kloppenburg S. Representing Explicit Attributes in UML. URL: http://dawis2.icb.uni-due.de/events/AOM_MODELS2005/Cepa.pdf.

22. Chaves R. Full Code Generation from UML Class, State and Activity Diagrams. URL: http://abstratt.com/blog/2007/06/01/full-code-generation-in-uml-from-the-class-state-and-activity-diagrams/

23. Chen M., Mishra P., Kalita D. Coverage-driven Automatic Test Generation for UML Activity Diagrams. Orlando. GLSVLSI’08, May 4-6.

24. Dia. URL: http://dia-installer.de/index_en.html.

25. Dia. URL: http://live.gnome.org/Dia.

26. Eclipse UML2 Tools. URL: http://wiki.eclipse.org/MDT-UML2Tools.

27. Enterprise Architect. URL: http://www.sparxsystems.com/.

28. Enterprise Architect. URL: http://www.sparxsystems.com/uml_tool_guide/code_engineering/code_generation___activity_dia.htm.

29. Eshuis R., Wieringa R. An Execution Algorithm for UML Activity Graphs. Proc. Forth Int'l Conf. Unified Modeling Language (UML), M. Gogolla and C. Kobryn, eds., 2001.

30. Eshuis R., Wieringa R. Tool Support for Verifying UML Activity Diagrams. IEEE Transactions on Software Engineering, vol. 30, No. 7, July 2004.

31. Eshuis R., Wieringa R. Verification Support for Workflow Design with UML Activity Graphs. Proc. Int'l Conf. Software Eng. (ICSE 2002), pp. 166-176, 2002.

32. Fischer T., Niere J., Torinski L., Zundorf A. Story Diagrams: A New Graph Rewrite Language Based on the Unified Modeling Language and Java. London, Springer-Verlag, 1998.

33. Fujaba. URL: http://www.fujaba.de/.

34. Glinz M., Berner S., Joos S. A Classification of Stereotypes for Object-Oriented Modeling Languages. UML '99 — The Unified Modeling Language: Beyond the Standart, LNCS, 1723:249-264, 1999.

35. IBM Rhapsody. URL: http://publib.boulder.ibm.com/infocenter/rsdp/v1r0m0/topic/com.ibm.help.download.rhapsody.doc/pdf75/tutorialj.pdf.

36. King J.J. Models of Beans? Java and UML Working Together… ODTUG 2004.

37. Martin R.C. UML for Java Programmers. New Jersey: Pearson Education, 2003.

38. Meyer M., Wendehals L. Teaching Object-Oriented Concepts with Eclipse. OOPSLA’04 Eclipse Technology eXchange Workshop, Oct. 24-28, 2004, Vancouver, British Columbia, Canada.

39. Milicev D. Model-Driven Development with Executable UML. Indianapolis: Wiley, 2009.

40. Miller G. What’s New in UML 2.0? Borland, Dec. 2003.

41. OMG Unified Modeling Language (OMG UML), Infrastructure. Version 2.2. URL: http://www.omg.org/spec/UML/2.2/.

42. OMG Unified Modeling Language (OMG UML), Superstructure. Version 2.2. URL: http://www.omg.org/spec/UML/2.2/.

43. Poseidon. URL: http://www.gentleware.com/.

44. Sybase PowerDesigner. URL: http://www.sybase.com/detail?id=1038605.

45. UML Studio. URL: http://www.pragsoft.com/prod_umls.html.

46. UModel. URL: http://www.altova.com/umodel.html.

47. UniMod. URL: http://unimod.sourceforge.net/intro.html.

48. Voigt H., Guildali B., Engels G. Quality Plans for Measuring Testability of Models. Department of Computer Science, Software Quality Lab. University of Paderborn, Germany.

49. Yu. L., France R.B., Ray I., Lano K. A Light-Weight Static Approach to Analyzing UML Behavioral Properties. Proc. of 12th IEEE International Conference on Engineering Complex Computer Systems (ICECCS 2007), pp. 56-63, 2007.