Санкт-Петербургский государственный университет

Фундаментальная информатика и информационные технологии
Математическое и программное обеспечение вычислительных машин,
комплексов и компьютерных сетей

Чистяков Сергей Юрьевич

Разработка унифицированной модели программирования для создания современных реактивных приложений

Магистерская диссертация

Научный руководитель:
Ст. преподаватель Салищев С. И.

Рецензент:
Директор по развитию Шолупов А. С.

Санкт-Петербург
2017

SAINT-PETERSBURG STATE UNIVERSITY

Fundamental Informatics and Information Technology
Mathematical and Software Support for Computers, Computer Systems and Networks

Sergey Chistyakov

Development of unified programming model for building modern reactive applications

Master's Thesis

Scientific supervisor:
Senior Lecturer S.I. Salishchev

Reviewer:
Director of Development A.S. Sholupov

Saint-Petersburg
2017

Содержание

1 Введение **5**

1.1 Диаграмма потоков данных 6

1.2 Программирование потоков данных 8

1.2.1 История 9

1.2.2 Основные определения 10

1.3 Реактивные приложения 11

1.3.1 Отзывчивость 12

1.3.2 Устойчивость 12

1.3.3 Гибкость 14

1.3.4 Основанность на сообщениях 14

1.4 Современное состояние проблемы разработки реактивных приложений 15

2 Проблема исследования **18**

3 Цели и задачи исследования **19**

4 Основная часть **20**

4.1 WRF платформа 20

4.2 RAML 20

4.2.1 Языки разметки графов 21

4.2.2 Общие сведения о RAML 27

4.2.3 Как работает RAML 29

4.2.4 Преимущества RAML 30

4.2.5 RAML vs CSharp Code 30

4.2.6 Структурные элементы WRF 31
4.2.7 Узел 32
4.2.8 Граф 46
4.3 Разработка простого приложения с использованием WRF . . 50
4.3.1 Задача 51
4.3.2 Проектирование 51
4.3.3 Реализация 52
4.3.4 Результаты 59
4.4 Приложение для Online шопинга разработанное с использованием WRF 60

5 Результаты **62**

6 Список литературы **63**

1 Введение

Организации работающие в разных сферах независимо друг от друга ищут шаблоны и подходы для создания программного обеспечения, которое выглядит и ведет себя одинаково. Эти системы являются более надежными, более устойчивыми, более гибкими и лучше приспособлены к современным требованиям.

Требования к приложениям резко изменились за последние годы. Всего несколько лет назад большое приложение могло содержать десятки серверов, секунды времени отклика, часы автономного обслуживания и гигабайты данных. Сегодня приложения разворачиваются на всем: от мобильных устройств до облачных кластеров, на которых работают тысячи многоядерных процессоров. Пользователи ожидают миллисекунды времени отклика и сто процентов времени безотказной работы, а данные измеряются в петабайтах. Вчерашние архитектурные решения, используемые для создания программного обеспечения, просто не удовлетворяют сегодняшним требованиям.

Реактивные системы более гибкие, слабосвязанные и масштабируемые. Это облегчает их разработку и адаптацию к изменениям. Они значительно более терпимы к ошибкам и сбоям, очень отзывчивы и предоставляют пользователям эффективную. интерактивную обратную связь.

Новые требования к приложениям повлекли за собой появление технологий позволяющих разрабатывать реактивные системы. Стали популярными интерфейсы программирования приложений для реактивного программирования такие как ReactiveX [1, 2, 9, 10, 11, 12, 13, 14], использующий наблюдаемые потоки и реализованный под огромное количество

языков, такие как C Sharp, Java, JavaScript, C++ и многие другие. Этот и похожие инструменты позволяют представить приложение как набор объектов, обменивающихся между собой данными, и превратить структуру приложения в асинхронный ориентированный граф данных. Однако, текстовое представление не дает прозрачного понимания о взаимодействии объектов в приложении и о её структуре в целом. Визуальные представление, в свою очередь, позволяют удобно и быстро конфигурировать и изменять графовую структуру приложения. Похожую задачу решает Windows Presentation Foundation, решение от Microsoft, ориентированная на создание клиентских приложений. Подробнее об этом решении можно почитать на официальном сайте MSDN или в книге Pro WPF 4.5 in C Sharp [4].

В данной работе описывается технология предназначенная для создания реактивных приложений Windows, и предоставляющая визуальный интерфейс, позволяющий создавать элементы реактивной системы для приложений, перетаскивая элементы управления из панели элементов и устанавливая свойства в окне «Свойства». Также данная технология предоставляет декларативную модель для описание реактивной системы благодаря использованию расширяемого языка разметки для реактивных приложений (XRAML).

1.1 Диаграмма потоков данных

Традиционно, программа моделируется как последовательность операций, происходящих в определенном порядке. Программа фокусируется на командах в соответствии с концепцией фон Неймана последователь-

ного программирования, когда данные обычно «находятся в состоянии покоя». Наоборот, реактивные программы используют парадигму асинхронного программирования, которая моделирует программу как ориентированный граф данных. Программирование потоков данных подчеркивает перемещение данных. Явно определенные входы и выходы соединяют операции, которые функционируют как черные ящики. Таким образом, языки потоков данных по сути своей параллельны и могут хорошо работать в больших децентрализованных системах.

Управление потоками данных может быть представлено с помощью диаграммы потока данных. Диаграмма потоков данных (DFD) представляет собой графическое изображение «потока» данных через информационную систему, моделируя ее аспекты процесса. DFD часто используется в качестве предварительного шага для создания структуры системы, не вдаваясь в детали, которые впоследствии могут быть разработаны. DFD можно также использовать для визуализации обработки данных (структурированный дизайн).

DFD показывает, какая информация будет вводиться и выводиться из системы, как данные будут продвигаться через систему и где будут храниться данные. Она не отображает информацию о времени процесса или информации о том, будут ли процессы работать последовательно или параллельно, в отличие от блок- схемы, которая также показывает эту информацию.

Преимущества этой диаграммы следующие:

- помогает в описании границ системы.
- выгодно для передачи существующих знаний о системе другим поль-

зователям и разработчикам

- представляет подробное представление компонентов системы
- может быть использована как часть файла системной документации
- поддерживает логику, лежащую в основе потока данных в системе

1.2 Программирование потоков данных

Программирование потоков данных - это одна из парадигм программирования, берущая за основу подход, при котором программа представляется в виде ориентированного графа потока данных между операциями, подобно диаграмме потока данных.

Традиционно, программа проектируется как набор операций, выполняющихся в специальном порядке(последовательное, процедурное или императивное программирование). Программа фокусируется на командах, опираясь на представление Джоном фон Нейманом последовательного программирования (sequential programming), где данные считаются "пассивными". С другой стороны, программирование потоков можно рассматривать как программирование алгоритмов движения данных и модели программы, которая представляет из себя набор соединений (connections). Явно определенные входы (inputs) и выходы (outputs) соединяют операции, которые функционируют как черные ящики (black boxes). Как только все входы заполнены данными, соответствующая операция может быть вычислена. Таким образом, языки программирования

потоков данных по сути своей являются языками параллельного программирования и могут быть использованы в больших, децентрализованных системах.

1.2.1 История

Самым старым языком программирования потоков данных (dataflow language) был BLOODY (BLOck DIagram), разработанный для определения импульсных систем. Функциональные элементы BLOODY и их соединения компилируются в единую цепь, которая обновляется каждый такт.

Другие языки программирования потоков данных были разработаны в первую очередь для того, чтобы сделать параллельное программирование более удобным. В 1966 году в своей статье The On-line Graphical Specification of Computer Procedures Берт Сазерленд (Bert Sutherland) описывает первый графический язык программирования потоков данных. Последующие языки обычно разрабатывались в больших лабораториях, имеющих в своем распоряже-нии суперкомпьютеры. Самым популярным был SISA, разработанный в Ливерморской национальной лаборатории (Lawrence Livermore National Laboratory). SISA похож на большинство statement-driven языков, однако переменным должно быть присвоено значение сразу. Это дает возможность компилятору определить входы и выходы. Существует множество ответвлений языка SISAL, включая SAC, Single Assignment C, который похож на популярный язык программирования C.

Более радикальную концепцию представляет язык программирования

Prograph, в котором программы создаются как визуальные графы. Разработка Prograph началась в Университете Акадия (Acadia University) в 1982 году как исследование в области языков программирования потоков данных. Объекты представлялись в виде шестиугольников, разделенных на 2 части: часть, содержащая поля с данными, и часть, содержащая методы для их обработки. Двойное нажатие на одну из частей открывало окно, отображающее более детализированную информацию данного объекта. Методы были представлены как набор иконок, каждая из которых содержала инструкцию или группу инструкций. В каждом методе поток данных изображался в виде линий в ориентированном графе. Данные проходили сверху вниз, через набор инструкций. Таким образом, Prograph - это комбинация объектно-ориентированных методологий и визуальной среды программирования. Потоки данных были предложены в качестве абстракции для обобщения поведения распределенных компонентов системы.

1.2.2 Основные определения

Программу, написанную на одном из языков программирования потоков данных, можно рассматривать как ориентированный граф. Вершина графа - узел, элемент, который производит обработку входных данных, преобразуя их в данные на выходе. Работа узла в течение периода активации считается единичным вычислением. Обмен данными между узлами происходит через порты - точки соединения дуг и узлов. Узел может быть связан с окружением только через порты. Иногда порты имеют уникальное в области соседних портов имя. Результат обработки узла

часто, но не обязательно, является функцией выходных данных, то есть, результат может меняться со временем. Вычислительная работа узла называется активацией. В состоянии активации узел считывает значения с входящих портов, производит вычисления, записывает данные в соответствующие выходные порты. Сами передаваемые данные называются токенами независимо от их типа. Далее токены передаются по дугам, что может вызвать активацию соответствующего соединенного узла. Обычно, на дуге запрещено иметь более одного токена, но в теории можно создавать модели с неограниченной емкостью. Также в некоторых моделях дуги могут сливаться и разветвляться. Все пути взаимодействия элементов явно задаются программистом. В случае конвейерной обработки (pipeline dataflow) элементы можно задать как последовательность единичных вычислений, которые производятся по очереди, при поступлении токенов на вход. Такая схема называется выполнением, управляемым данными (data-driven execution).

1.3 Реактивные приложения

Реактивные приложения

Для реактивных приложений существенны следующие характеристики:

- Отзывчивость
- Устойчивость
- Гибкость
- Основаны на сообщениях

1.3.1 Отзывчивость

Система отвечает своевременно, если это вообще возможно. Быстрота реагирования - краеугольный камень удобства использования и полезности, но более того, отзывчивость означает, что проблемы могут быть быстро обнаружены и эффективно решены. Адаптивные системы фокусируются на обеспечении быстрого и согласованного времени ответа и постоянного качества обслуживания. Это согласованное поведение, в свою очередь, упрощает обработку ошибок, повышает уверенность конечных пользователей и способствует дальнейшему взаимодействию.

1.3.2 Устойчивость

Система остается отзывчивой в случае сбоя. Отказы может случится в любом компоненте по отдельности, что изолирует компоненты друг от друга и гарантирует, что части системы могут выйти из строя и восстановиться без ущерба для системы в целом. Восстановление каждого компонента делегируется другому (внешнему) компоненту, а высокая доступность обеспечивается репликацией там, где это необходимо. Компонент не обременен обработкой своих ошибок. Таким образом устойчивость достигается репликацией, изоляцией и делегированием.

Репликация

Репликацией называется выполнение компонента одновременно в разных местах. Это может означать выполнение в разных потоках или пулах потоков, процессах, сетевых узлах или вычислительных центрах. Репликация обеспечивает масштабируемость, когда входящая рабочая нагруз-

ка распределяется по нескольким экземплярам компонента или устойчивость, когда входящая рабочая нагрузка реплицируется в несколько экземпляров, которые обрабатывают одни и те же запросы параллельно.

Изоляция

Изоляция может быть определена как разделение, как по времени, так и по пространству. Разделение по времени означает, что отправитель и получатель могут иметь независимые жизненные циклы - им не обязательно присутствовать одновременно для того, чтобы общение было возможным. Оно реализуется путем добавления асинхронных границ между компонентами, обменивающихся сообщениями. Разделение по пространству означает, что отправитель и получатель не должны запускаться в одном процессе.

Делегирование

Асинхронное делегирование задачи другому компоненту означает, что выполнение задачи будет выполняться в контексте этого другого компонента. Цель делегирования - передать ответственность за обработку задачи другому компоненту, чтобы делегирующий компонент мог выполнять другую обработку или, возможно, наблюдать за выполнением делегированной задачи в случае необходимости дополнительных действий, таких как обработка ошибки или отчет о прогрессе.

1.3.3 Гибкость

Система способна реагировать на изменяющуюся рабочую нагрузку. Реактивные системы могут реагировать на изменения данных, увеличивая или уменьшая ресурсы, выделенные для обслуживания этих данных. Это подразумевает задачи, которые не имеют ключевых точек, что распределять или реплицировать компоненты и распределять данные и обязанности между ними.

1.3.4 Основанность на сообщениях

Реактивные системы полагаются на асинхронный обмен сообщениями, чтобы обеспечить свободную связь между компонентами, изоляцию и прозрачность местоположения. Использование явного обмена сообщениями позволяет управлять нагрузкой, эластичностью и контролем потока путем формирования и мониторинга очередей сообщений в системе и применения обратного давления, когда это необходимо. Обмен сообщениями как средство коммуникации позволяет управлять сбоем с использованием одних и тех же конструкций и семантики по всему кластеру или внутри одного хоста. Связь без использования операций блокировок позволяет получателям потреблять только активные ресурсы , что приводит к снижению накладных расходов системы.

1.4 Современное состояние проблемы разработки реактивных приложений

На сегодняшний день существуют несколько технологий позволяющих создавать реактивные системы. В книге Reactive Application Development [1] объясняется как разрабатывать современные реактивные приложения с использованием стека Typesafe. Книга не только предоставляет архитектурный обзор систем в целом, но и описывает механизмы применения шаблонов, таких как CQRS, Event Sourcing, Microservices и многих других другое, механизмов создания моделей распределенных доменов для реактивных приложений и кластерных актерских систем для повышения гибкости и отказоустойчивости, а также как интегрировать реактивные системы с традиционными архитектурами. Typesafe Stack - это современная платформа для создания легко масштабируемых программных систем, включающая в себя язык программирования Scala, управляемое событиями промежуточное программное обеспечение Akka и веб-инфраструктуру Play вместе с мощным набором средств разработки. Как сообщает информационное агентство Marketwired (Торонто, Канада) компания Typesafe, поставляющая ведущую в мире реактивную платформу, 7 мая 2014 года объявила о дополнительных соглашениях в розничной торговле с клиентами Gilt, Tomax и Walmart. Статья утверждает, что индустрия розничной торговли переживает ренессанс в инфраструктуре приложений, поскольку разработчики поддерживают новые инициативы в области электронной коммерции и точек продаж. Play и Akka, соответственно, предоставляют каркас для веб-приложений. которые оптимизированы для современных требований к трафику в по-

пулярных розничных приложениях, которые должны обрабатывать множество одновременных пользователей и непредсказуемые всплески трафика. Инструмент Akka также доступен и для платформы .NET, однако сейчас популярность набирает другой инструмент реализующий Reactive Streams - Reactive Extensions. В книге Introduction to Rx: A step by step guide to the Reactive Extensions to .NET [2] представлены основные преимущества соответствующей технологии:

- **интегрированный** - LINQ интегрирован в язык C Sharp
- **объединяющий** - использование LINQ позволяет использовать имеющиеся навыки для запроса данных (LINQ to SQL, LINQ to XML или LINQ to objects). Можно рассматривать Rx как LINQ для событий. LINQ позволяет перейти от нескольких парадигм к общей. Например, инструмент Reactive Extensions позволяет перевести стандартное событие .NET, вызов асинхронного метода, задачу или, возможно, стороннее ПО промежуточного программного обеспечения в одну общую парадигму Rx
- **масштабируемый** - возможность расширить Rx с помощью собственных операторов запросов - методы расширения
- **декларативный** - LINQ позволяет рассматривать ваш код декларацию о том, что делает ваш код, и оставляет инструкции по реализации операторов
- **свободный** - функции LINQ, такие как методы расширения, синтаксис лямбда и синтаксис запросов, обеспечивают свободный API

для разработчиков. Запросы могут быть построены с помощью многочисленных операторов, а сами запросы могут быть составлены вместе для дальнейшего получения составных запросов

- **трансформируемый** - запросы могут преобразовывать свои данные из одного типа в другой

На сегодняшний день реактивное программирования часто освещается в литературе. Доказательство удобства и пригодности Reactive Extensions для разработки разнообразных современных приложений можно найти в книге Grokking ReactiveX [3]. В данный момент книга находится в стадии написания и ориентировочная дата ее публикации намечена на осень 2017 года. Данный факт еще раз доказывает интерес общественности к реактивному программированию. Авторы книги (Ivan Morgillo, Sasa Sekulic, Fabrizio Chignoli) обещают предоставить огромное количество примеров и решений, реализованных с помощью технологии Rx. Например использование реактивных расширений для работы с базами данных, облачными технологиями или операционной системой Android.

2 Проблема исследования

Основной исследовательской задачей, на рассмотрение которой направлена эта диссертация, является решение проблемы отсутствия инструмента для быстрого и гибкого создания реактивных приложений на языке программирования C Sharp. Разработка современных реактивных приложений требует от разработчиков создания и поддержания процессов, работающих долго и время от времени возвращающих промежуточные результаты. В .NET Framework есть события, с помощью событий один объект может вызвать метод второго объекта, передавая некоторую информацию, в тот момент времени, когда это необходимо. Также одним из способов обработки потоков данных могут служить Реактивные Расширения. Однако, при решении с использованием данного подхода сложной и объемной задачи, решение становатся запутанным и сложно потдерживаемым.

3 Цели и задачи исследования

- Анализ большого объема реактивных приложений
  - Написанных на C Sharp
    * с использованием Reactive Extensions
    * с использованием Events
    * с использованием других средств
  - Написанных на других языка программирования
- Анализ подходов к визуальному программированию, как к возможному решению проблемы возрастающей сложности поддержки и разработки больших реактивных приложений
- Разработка прототипов унифицированной модели программирования для создания реактивных приложений
- Анализ разработанных прототипов моделей программирования
- Разработка итоговой унифицированной модели программирования для создания реактивных приложений

Исходя из этих задач, цель этой исследовательской работы заключается в том, чтобы разработать инструмент позволяющий гибко и удобно разрабатывать сложные реактивные приложения.

4 Основная часть

В данной работе рассматривается разработанная в рамках данного исследования среда для создания реактивных приложений Windows - WRF (Windows Reactive Foundation). Более того представлены некоторые основные функции, которые необходимо понимать для создания приложений WRF.

4.1 WRF платформа

На сегодняшний день нет реального разделения между тем, как выглядит архитектура реактивного приложения и как приложение себя ведет. Современные реактивные приложения написанные на языке C Sharp имеют сложную и запутанную структуру которую непросто отлаживать, особенно если эта структура является еще и микро сервисной. В WRF элементы решения проектируются в RAML, тогда как поведение каждого элемента в отдельности может быть реализовано на языке C Sharp. Это позволяет отделить процесс конфигурации реактивного приложения от реализации его компонент.

4.2 RAML

Одним из основных преимуществ WRF является язык RAML, позволяющий описать граф компонентов реактивного приложения. RAML расшифровывается как Extensible Reactive Application Markup Language и представляет из себя простой и декларативный язык, основанный на XML.

4.2.1 Языки разметки графов

Визуальный подход к программированию более пригоден для конфигурации готовой структуры, чем для ее написания или надстройки, в отличии от традиционного текстового подхода. Совмещение двух подходов предоставляет более гибкую модель разработки приложений. Добавление полностью декларативного языка позволяет разработчику описать поведение программы и интеграцию компонентов без использования процедурного программирования. Использование такого языка для описания структурного графа приложения помогает разработчику разделить внутреннюю реализацию узла от самой структуры графа, что считается хорошим архитектурным принципом. Однако, несмотря на несколько ранних попыток определить стандарт, широко распространенный формат не согласован, и многие инструменты поддерживают только ограниченное число пользовательских форматов, которые обычно специфичны для области приложения.

DOT

DOT - это простой язык описания графов. В самом простом случае DOT может использоваться для описания неориентированного графа (Рис. 1). Ключевое слово graph используется для начала нового графа, а узлы - в фигурных скобках. Двойной дефис (-) используется для отображения отношений между узлами.

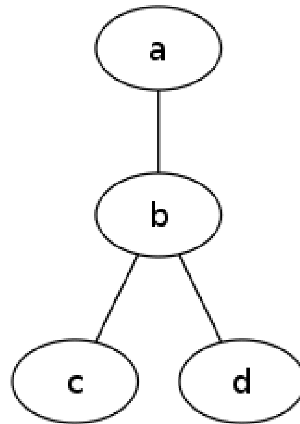

Рис. 1: Неориентированный граф

```
graph graphname {
    a -- b -- c;
    b -- d;
}
```

Подобно неориентированным графам, DOT может описывать ориентированные графы, такие как блок-схемы и деревья зависимостей (Рис. 2). Синтаксис такой же, как для неориентированных графов, за исключением того, что для начала графа используется ключевое слово digraph , а для отображения отношений между узлами используется стрелка (->).

```
digraph graphname {
    a -> b -> c;
    b -> d;
}
```

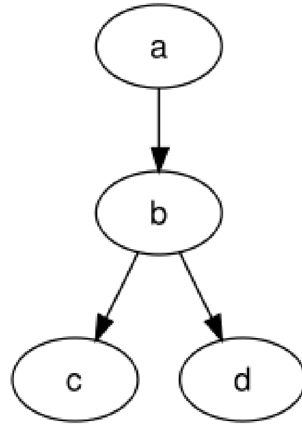

Рис. 2: Ориентированный граф

В файлах DOT к графам, узлам и ребрам могут применяться различные атрибуты. Эти атрибуты могут управлять такими аспектами, как цвет, форма и стили линий. Для узлов и ребер одна или несколько пар атрибут-значение помещаются в квадратных скобках ([]) после оператора и перед точкой с запятой (что не обязательно). Атрибуты графа (Рис. 3) задаются в виде прямых пар «атрибут-значение» под элементом графа. Несколько атрибутов разделяются запятой или несколькими наборами квадратных скобок. Атрибуты узла размещаются после оператора, содержащего только имя узла, и никаких отношений.

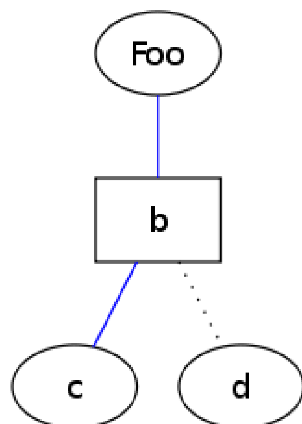

Рис. 3: Граф с атрибутами

```
graph graphname {
    a [label="Foo"];
    b [shape=box];
    a -- b -- c [color=blue];
    b -- d [style=dotted];
}
```

GraphML

GraphML - это простой в использовании формат файлов для описания графов. Он состоит из языкового ядра для описания структурных свойств графа и гибкого механизма расширения для добавления данных, специфичных для приложения. Его основные функции включают поддержку таких элементов, как

- направленные, неориентированные и смешанные графы

- гиперграфы
- иерархические графы
- графические представления
- ссылки на внешние данные
- данные атрибутов, зависящие от приложения
- легкие синтаксические анализаторы

В отличие от многих других форматов файлов для описания графов, GraphML не использует настраиваемый синтаксис. Вместо этого он основан на XML и, следовательно, идеально подходит в качестве общего знаменателя для всех видов генерации, архивирования или обработки графов.

Благодаря синтаксису XML, GraphML можно использовать в сочетании с другими форматами на основе XML. С одной стороны, собственный механизм расширения позволяет прикреплять метки <data> со сложным контентом (возможно, требуемым для соответствия другим моделям контента XML) элементам GraphML. Примерами таких сложных меток данных являются Масштабируемая векторная графика [W3Ca], описывающая появление узлов и ребер в чертеже. С другой стороны, GraphML может быть интегрирован в другие приложения, например, в сообщениях SOAP [W3Cb].

Граф обозначается узлом graph. Добавление узлов и ребер в существующий граф производится путем объявления вложенных узлов. Узел объявляется как node узел, а ребро как edge узел.

```
<Graph id = "G">
    <Node id = "n0" />
    <Node id = "n1" />
    ...
    <Node id = "n10" />
    <Edge source = "n0" target = "n2" />
    <Edge source = "n1" target = "n2" />
    ...
    <Edge source = "n8" target = "n10" />
</Graph>
```

В GraphML нет порядка, определенного для объявления элементов node и edge. Графы в GraphML могут одновременно содержать ориентированные и неориентированные ребра. Если, при объявлении ребра, направление не указано, применяется направление по умолчанию. Направление по умолчанию объявляется как атрибут XML edgedefault элемента graph . Двумя возможными значениями для этого XML-атрибута являются константы directed и undirected. Направление по умолчанию необходимо указать. Узлы в графе объявляются элементом node . Каждый узел имеет идентификатор, который должен быть уникальным во всем документе, то есть в документе не должно быть двух узлов с одним и тем же идентификатором. Идентификатор узла определяется идентификатором XML-атрибута.

Ребра в графе объявляются элементом edge. Каждое ребро должно определять свои две конечные точки с помощью source и target XML-атрибутов. Значение источника (target) - идентификатор узла в том же

документе.

4.2.2 Общие сведения о RAML

RAML (extensible **R**eactive **A**pplication **M**arkup **L**anguage) - это декларативный язык разметки. В рамках модели программирования .NET Framework RAML упрощает разработку реактивных приложений для платформы .NET Framework, путем предоставления гибкого инструмента описания графа взаимодействия компонентов системы. Благодаря RAML разработчик имеет возможность не только создавать реактивные приложения и конфигурировать их структуру используя разметку, но также отделять интерфейсы взаимодействия для каждого компонента приложения отдельно от логики выполнения с помощью файлов с выделенным кодом, присоединенных к разметке с помощью разделяемых классов. Файлы RAML в текстовой интерпретации представляют собой файлы XML, которые обычно имеют расширение .xraml. Файлы могут быть закодированы любой кодировкой XML, но в основном используется кодировка UTF-8.

В следующем примере показано, как создать узел на языке RAML. Этот пример предназначен для того, чтобы дать представление о метафоре разработки и конфигурации реактивных систем с использованием RAML:

```
<Node x:Class="CaptureWebCamNode"
        xmlns:x="http://schemas.wrf.com/raml"
        xmlns:system="clr-namespace:System;assembly=mscorlib">
    <OutBehaviorPort Id="Text"
                     Type="system:Int32"/>
</Node>
```

RAML, в общем случае, чувствителен к регистру. Такие элементы как объекты, свойства и имена атрибутов должны быть указаны с учетом регистра символов. Ключевые слова и примитивы языка RAML также чувствительны к регистру, в отличии от значения, для которых регистр учитывается не всегда. Чувствительность к регистру для значений будет зависеть от поведения преобразователя типа, связанного со свойством, которое принимает значение, или типом значения свойства. Например, свойства, которые работают с логическим типом, могут принимать значения false или False в качестве эквивалентных значений. Процессоры и сериализаторы RAML нормализуют существенные пробелы и игнорируют все несущественные.

Значение атрибута любого элемента должно быть установлено строкой. Базовая, собственная обработка того, как строки преобразуются в другие типы объектов или примитивы, основана на самом типе String, помимо собственной обработки для некоторых типов, таких как DateTime или Uri. Но многие типы разработанной системы или члены этих типов расширяют поведение обработки базового строкового атрибута таким образом, что экземпляры более сложных типов объектов могут быть ука-

заны как строки и атрибуты. Такая структура как тип атрибута Type у элемента OutBehaviorPort - это пример типа, который имеет преобразование типа, разрешенное для использования RAML. Благодаря конвертеру типов в Type, типы значения портов легче указать в RAML, потому что они могут быть указаны как атрибуты.

4.2.3 Как работает RAML

RAML - это декларативный язык в том смысле, что он определяет ЧТО и КАК хочет разработчик делать. Процессор RAML отвечает за часть КАК. Схема (Рис. 4) содержит информацию о процессе обработки RAML:

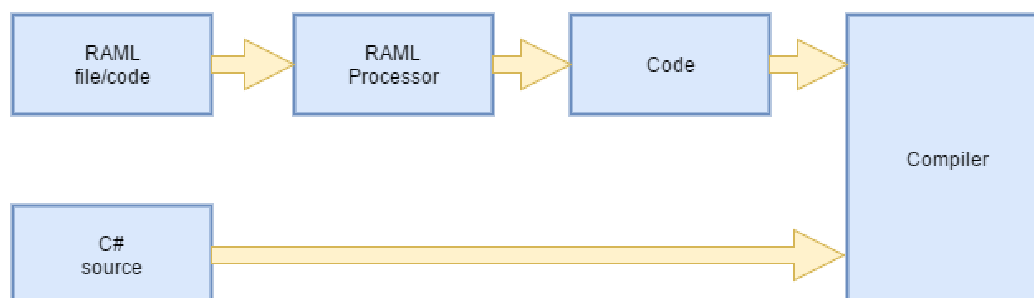

Рис. 4: Процесс обработки RAML

- Файл RAML интерпретируется процессором RAML
- Процессор RAML преобразует RAML во внутренний код, описывающий структуру реактивного элемента
- Внутренний код и код CSharp связываются между собой через разделяемые классы

4.2.4 Преимущества RAML

RAML предлагает естественный формат для моделирования и конфигурирования решения и предоставляет архитектору возможность проверить, что требуемые функциональные возможности системы реализуются компонентами. Это в свою очередь гарантирует приемлемость разрабатываемой системы. Благодаря RAML очень легко отделить взаимодействие объектов от их реализации. Код RAML очень прост в использовании, чтении и понимании. Граф компонента или приложения, описанный на языке RAML, полезен в первую очередь для разработчиков, так как он предоставляет им высокоуровневое архитектурное представление системы, которую они будут создавать. Это помогает разработчикам как можно раньше начать оформление плана реализации и приступить к назначению задач.

Еще одним преимуществом RAML является тот факт, что структурные элементы языка построены поверх Реактивных Расширений (Reactive Extensions (Rx)) для платформы .NET. Это позволяет разработчику отказатся от освоения нового API и использовать проверенную и современную библиотеку для асинхронного программирования наблюдаемых потоков.

4.2.5 RAML vs CSharp Code

Вы можете использовать RAML для создания, инициализации и установки свойств объектов. Одни и те же действия могут также выполняться с использованием программного кода. RAML - это еще один простой и простой способ разработки компонентов реактивной системы и ее кон-

фигурирования. Используя RAML разработчик сам решает, хочет ли он объявлять объекты в RAML или объявлять их с помощью кода. Ниже приведен простой пример, демонстрирующий использование языка разметки для объявления узла:

```
<Node x:Class="CaptureWebCamNode"
        xmlns:x="http://schemas.wrf.com/raml"
        xmlns:cv="clr-namespace:Emgu.CV;assembly=Emgu.CV.World">
    <OutPort Id="ImageMat"
                Type="cv:Mat"/>
</Node>
```

Этот фрагмент кода на языке RAML может быть сгенерирован в эквивалентное объявления узла на языке CSharp:

```
public partial class CaptureWebCamNode : Node
{
    public OutPort<Mat> ImageMat;

    protected sealed override void InitializeNode()
    {
        ImageMat = new OutPort<Mat>();
    }
}
```

4.2.6 Структурные элементы WRF

В этой главе будут описаны некоторые основные и важные структурные элементы приложений WRF. Основными элементами платформы

являются:

- Порт (InPort, OutPort)
- Узел (ValueNode, SumNode, WebCamNode)
- Граф

4.2.7 Узел

Структура пользовательского интерфейса WRF предлагает библиотеку элементов, которая ускоряет и упрощает разработку элементов реактивной системы. Все элементы узлы (Nodes) наследуются от WindowsReactiveFoundation.Nodes

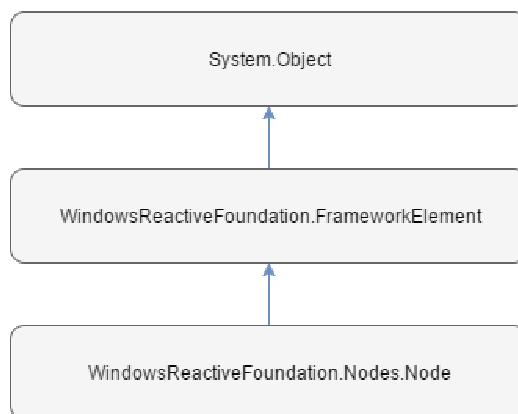

Рис. 5: Полная иерархия наследования элементов узлов

Узел может указывать разные логические местоположения для соединения. Логические местоположения называются «портами». Порты узла объявляются элементами порта как дочерние элементы соответствующих элементов узла. Элементы порта могут быть вложенными, т. е. Они

могут содержать сами портовые элементы. Каждый элемент порта должен иметь имя RAML-атрибута, которое является идентификатором для этого порта.

Порты могут быть разделены на две группы: входные и выходные.

Входные порты

Выходные порты предоставляют механизм для создания и отправки push-уведомлений. В зависимости от задачи разработчик может использовать различные типы входных портов:

- InPort
- InReplayPort
- InBehaviorPort

Все элементы порты наследуются от WindowsReactiveFoundation.Ports.InPortBase. Помимо этого, все элементы порты реализуют интерфейс IObservable<T> из пространства имен System.

InPort

InPort представляет из себя базовую реализацию абстрактного класса InPortBase.

InPort отдает только те значения, которые были произведены внешним источником после момента начала наблюдения за значениями самого входного порта.

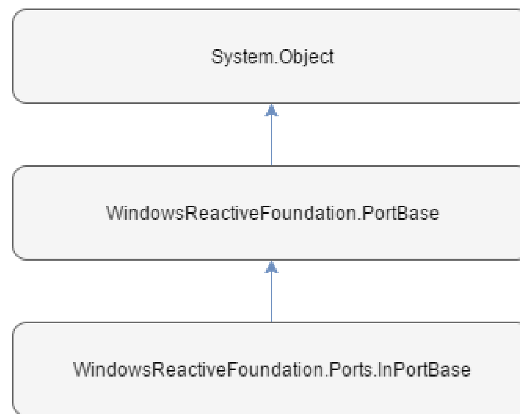

Рис. 6: Полная иерархия наследования элементов входных портов

InReplayPort

InReplayPort предоставляет возможность кэширования входных значений, а затем воспроизводит их для любых наблюдателей значений входного порта.

Конструктор по умолчанию InReplayPort<T> создаст экземпляр, который кэширует каждое значение. Во многих сценариях это может увеличить количество памяти используемое приложением. InReplayPort<T> имеет конструктор, использующий параметры истечения срока действия кэша входных значений. Используя этот параметр разработчик может устранить проблему чрезмерного использования памяти. Также данную проблему можно решить указав размер буфера в кэше. В этом примере создается объект IReplayPort<T> с размером буфера равным 4 и временным интервалом длительностью 50 миллисекунд. Таким образом кэш входного порта хранит не более четырех последних опубликованных до

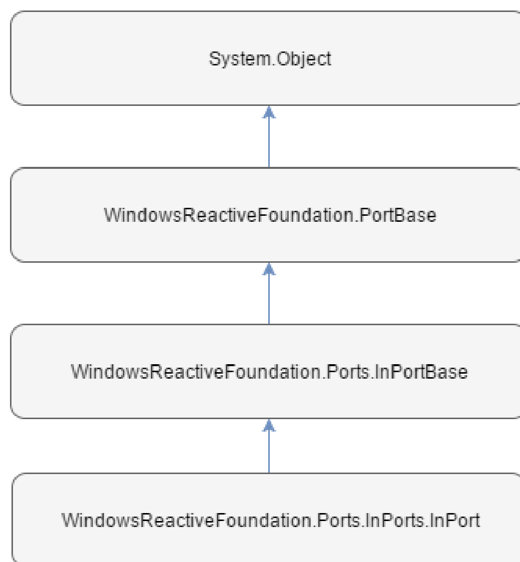

Рис. 7: Полная иерархия наследования элемента InPort

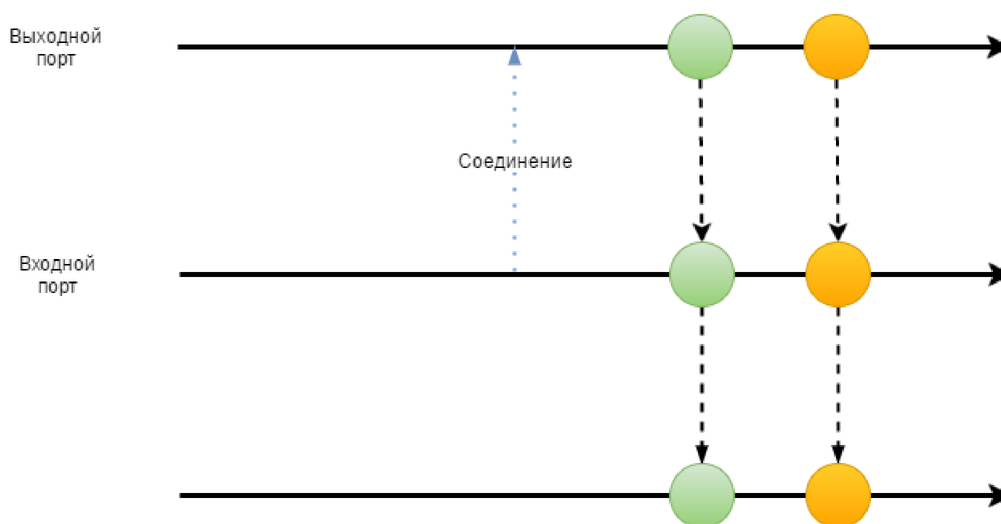

Рис. 8: Взаимодействие выходного порта, InPort и наблюдателя значений входного порта

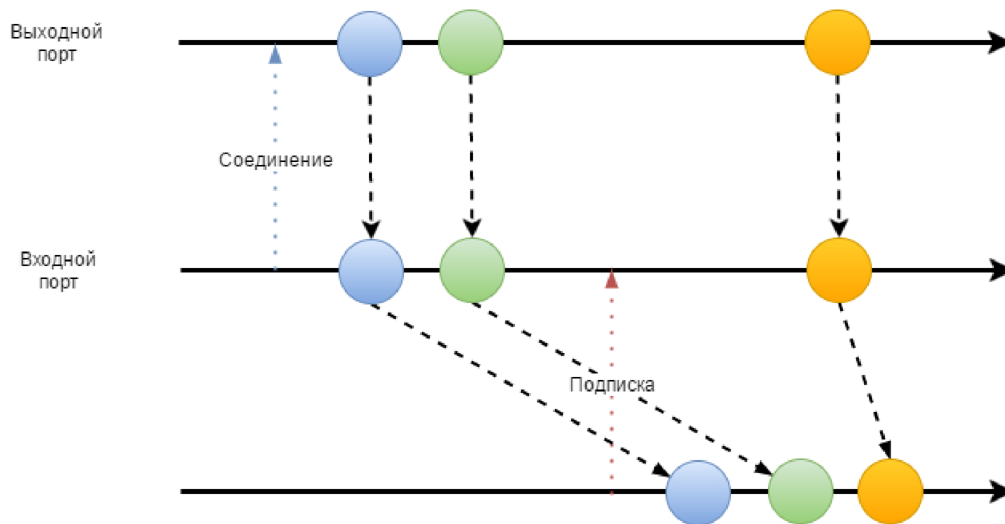

Рис. 9: Взаимодействие выходного порта, InReplayPort и наблюдателя значений входного порта

начала наблюдения значений внешнего источника и имеет длительность не более 50 миллисекунд:

```
...
<InReplayPort Id="IntValue"
              Type="system:Int32"
              BufferSize="4"
              Window="TimeSpan.FromMilliseconds(50)"/>
...
```

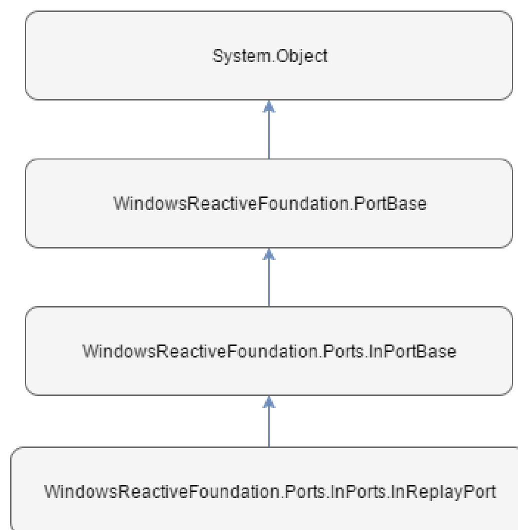

Рис. 10: Полная иерархия наследования элемента InReplayPort

```
...
protected override void InitializeNode()
{
    IntValue = new InReplayPort<Int32>(4,
                                       TimeSpan.FromMilliseconds(50));
}
...
```

InBehaviorPort

InBehaviorPort<T> похож на InReplayPort<T>, за исключением того, что он запоминает только последнее значение. InBehaviorPort<T> также требует, чтобы разработчик предоставил ему значение по умол-

чанию T. Это означает, что все наблюдатели значений входного порта при начале наблюдения получат значение немедленно. Стоит обратить внимание, что существует разница между InReplayPort<T> с размером буфера 1 и InBehaviorPort<T>. Для InBehaviorPort<T> требуется начальное значение.

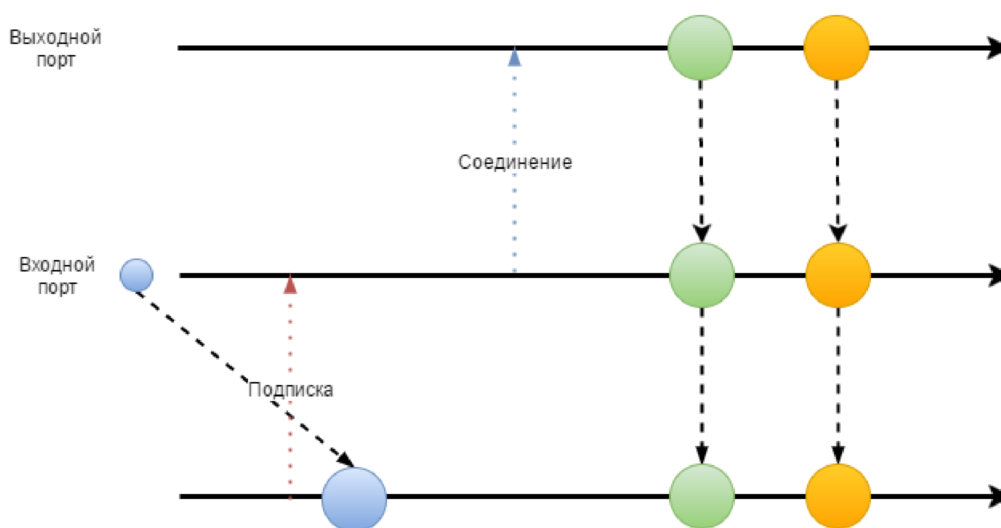

Рис. 11: Взаимодействие выходного порта, InBehaviorPort и наблюдателя значений входного порта

Выходные порты

Выходные порты предоставляют механизм для получения push-уведомлений. В зависимости от задачи разработчик может использовать различные типы выходных портов:

- OutPort
- OutReplayPort

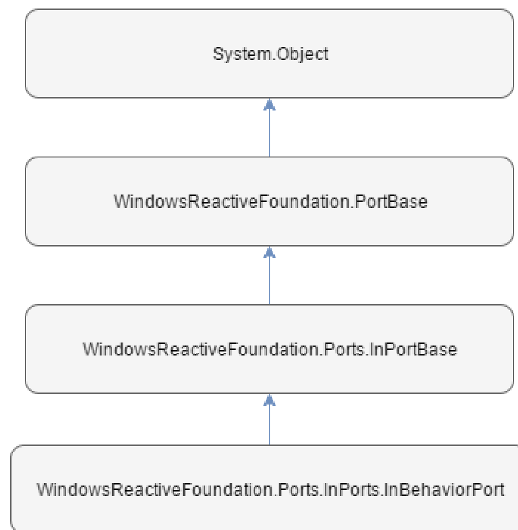

Рис. 12: Полная иерархия наследования элемента InBehaviorPort

- OutBehaviorPort

Все элементы порты наследуются от WindowsReactiveFoundation.Ports.OutPortBase. Помимо этого, все элементы порты реализуют интерфейс IObserver<T> из пространства имен System.

OutPort

OutPort представляет из себя базовую реализацию абстрактного класса OutPortBase.

OutPort отдает только те значения, которые были произведены внутренним источником после момента соединения его со входным портом.

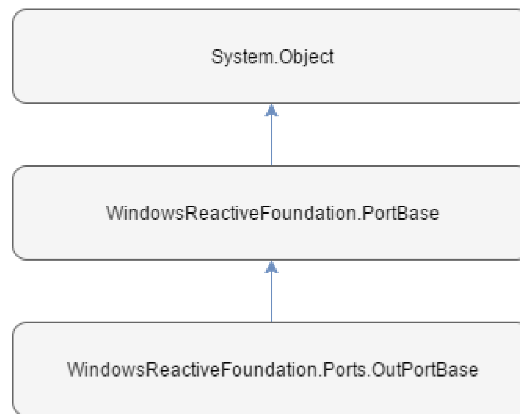

Рис. 13: Полная иерархия наследования элементов выходных портов

OutReplayPort

OutReplayPort предоставляет возможность кэширования значений, а затем воспроизводит их для любых поздних подсоединенных входных портов.

Конструктор по умолчанию OutReplayPort<T> создаст экземпляр, который кэширует каждое значение. Во многих сценариях это может увеличить количество памяти используемое приложением. OutReplayPort<T> предоставляет конструктор, позволяющий указать размер буфера в кэше. В этом примере создается объект OutReplayPort<T> с размером буфера 2 и поэтому получаем только два последних значения, опубликованные до подсоединения входных портов:

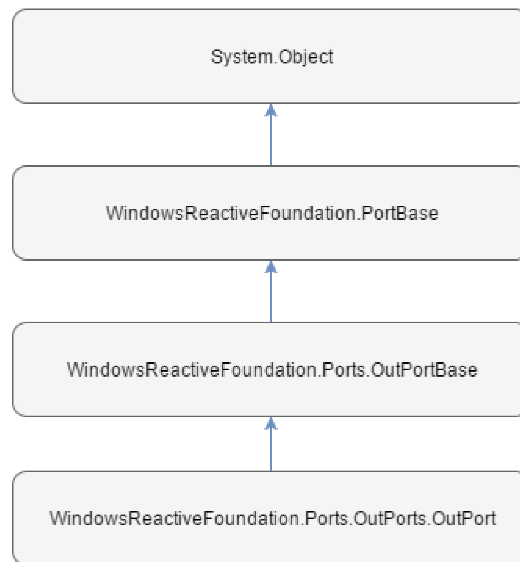

Рис. 14: Полная иерархия наследования элемента OutPort

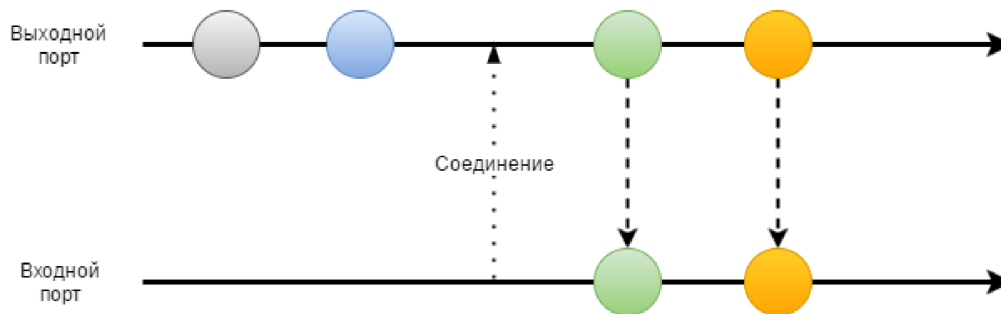

Рис. 15: Взаимодействие OutPort и входного порта

```
...
<OutReplayPort Id="Value"
               Type="system:Int32"
               BufferSize="2"/>
...
```

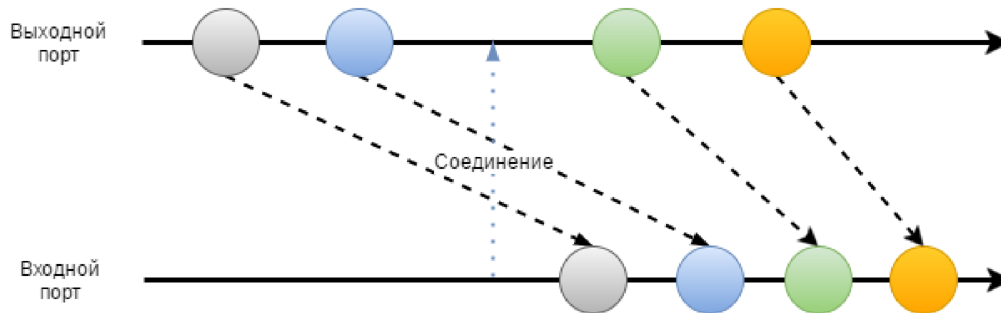

Рис. 16: Взаимодействие OutReplayPort и входного порта

```
...
protected override void InitializeNode()
{
    Value = new OutReplayPort<Int32>(2);
}
...
```

Другой вариант предотвращения бесконечного кэширования значений с помощью OutReplayPort<T> - предоставить временной интервал для хранения кэша. В этом примере вместо создания объекта OutReplayPort<T> с размером буфера мы указываем временной интервал, в котором действительны кэшированные значения.

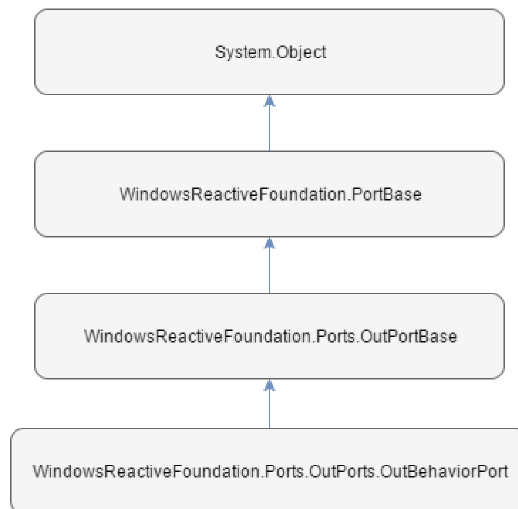

Рис. 17: Полная иерархия наследования элемента OutReplayPort

```
...
<OutReplayPort Id="Value"
               Type="system:Int32"
               Window="TimeSpan.FromSeconds(3)"/>
...
```

```
...
protected override void InitializeNode()
{
    Value = new OutReplayPort<int>(TimeSpan.FromSeconds(3));
}
...
```

OutBehaviorPort

OutBehaviorPort<T> схож с OutReplayPort<T>, за исключением того, что он хранит только последнее значение. OutBehaviorPort<T> также требует, чтобы разработчик предоставил ему значение по умолчанию T. Это означает, что все входные порты при присоединении получат значение немедленно. Стоит обратить внимание, что существует разница между OutReplayPort<T> с размером буфера 1 (обычно называемым «воспроизведение одного значения») и OutBehaviorPort<T>. Для OutBehaviorPort<T> требуется начальное значение.

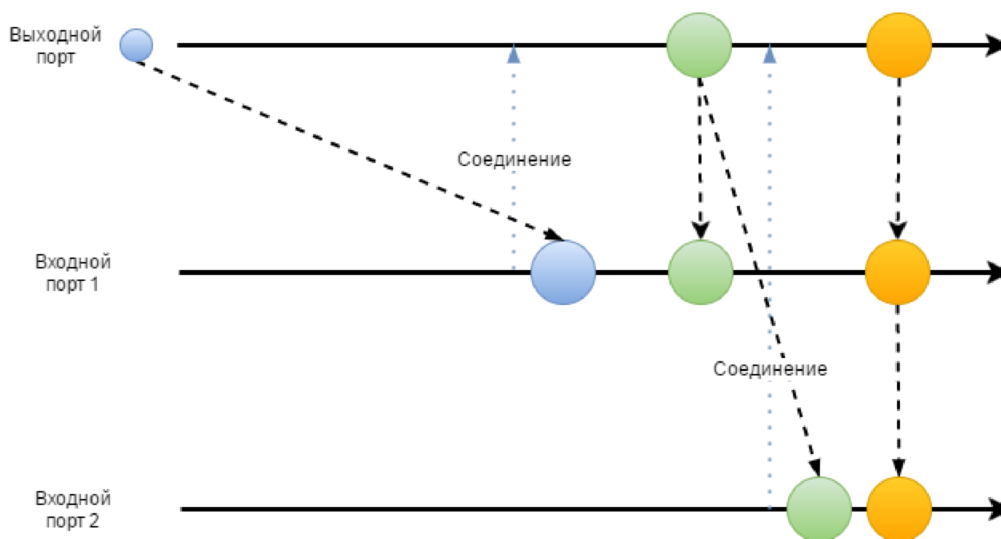

Рис. 18: Взаимодействие OutBehaviorPort и входных портов

OutBehaviorPort<T> схож со свойством класса, поскольку он всегда имеют значение и может предоставлять уведомления об изменениях.

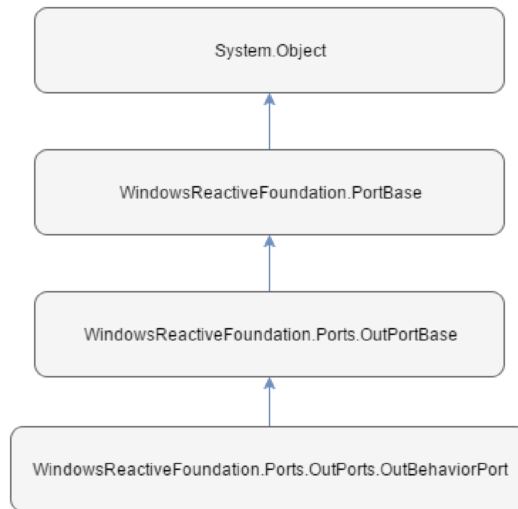

Рис. 19: Полная иерархия наследования элемента OutBehaviorPort

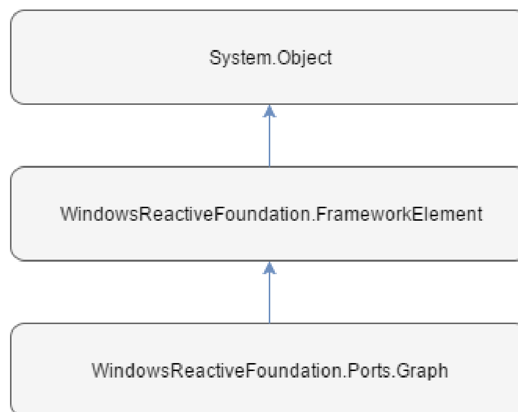

Рис. 20: Полная иерархия наследования элемента Graph

4.2.8 Граф

Граф объявляется с помощью элемента языка RAML Graph. Именно внутри этого элемента описывается структура самого графа, например объявляется узлы и ребра, соединяющие порты этих узлов.

```
<Graph x:Class="SomeGraph"
       xmlns:x="http://schemas.wrf.com/raml"
       xmlns:system="clr-namespace:System;assembly=mscorlib">
   ...
</Graph>
```

Вложенные узлы элемента Graph могут быть нескольких типов. Основные из них:

- Различные элементы узлы
- Ребра
- Порты

Узлы в графе

Каждый узел в графе объявляется элементом с соответствующим именем.

```
<Graph x:Class="SomeGraph"
       xmlns:x="http://schemas.wrf.com/raml"
       xmlns:system="clr-namespace:System;assembly=mscorlib">
  ...
  <SomeNode Id="Some node"/>
  ...
</Graph>
```

Каждый узел имеет идентификатор, который должен быть уникальным во всем документе, то есть в документе не должно быть двух узлов с одним и тем же идентификатором. Идентификатор узла определяется идентификатором XML-атрибута Id. Идентификатор нужен в первую очередь для ссылки на узел в документе.

Ребра

Ребра в графе объявляются элементом Edge. Каждое ребро должно определять свои две конечные точки с помощью RAML-атрибутов Source и Target. Значениями этих атрибутов должны быть идентификаторы портов узлов используемых в том же документе.

```
<Graph x:Class="SomeGraph"
       xmlns:x="http://schemas.wrf.com/raml"
       xmlns:system="clr-namespace:System;assembly=mscorlib">
   ...
   <SomeNode Id="FirstNode"/>
   <SomeNode Id="SecondNode"/>
   ...
   <Edge Source="FirstNode.SomeOutPort"
         Target="SecondNode.SomeInPort"/>
   ...
</Graph>
```

Так же ребра могут соединять порты узлов и порты графа, то есть значения атрибутов Source и Target могут принимать значения совпадающие с идентификаторами этих портов.

Порты

WRF поддерживает вложенные графы, т. е. графы, в которых узлы иерархически упорядочены. Иерархия выражается структурой документа RAML. Узел в документе RAML может представлять как простой узел, так и другой граф, который сам содержит узлы. Для того чтобы граф можно было использовать как узел другого графа, он должен иметь входные и выходные порты как и обычный узел.

```
<Graph x:Class="FirstGraph"
        xmlns:x="http://schemas.wrf.com/raml"
        xmlns:system="clr-namespace:System;assembly=mscorlib">
    ...
    <SomeNode Id="SomeNode"/>
    <OutPort Id="SomeOutPort"/>
    ...
</Graph>
```

```
<Graph x:Class="SecondGraph"
        xmlns:x="http://schemas.wrf.com/raml"
        xmlns:system="clr-namespace:System;assembly=mscorlib">
    ...
    <FirstGraph Id="SomeGraph"/>
    <SomeNode Id="SomeNode"/>
    ...
    <Edge Source="SomeGraph.SomeOutPort"
          Target="SomeNode.SomeInPort"/>
    ...
</Graph>
```

Внутри самого графа порты и порты узлов графа так же можно соединять как в коде на языке CSharp, так и на языке RAML. Для этого необходимо объявить ребро соединяющее эти порты.

```
<Graph x:Class="SomeGraph"
       xmlns:x="http://schemas.wrf.com/raml"
       xmlns:system="clr-namespace:System;assembly=mscorlib">
   ...
   <SomeNode Id="SomeNode"/>
   <OutPort Id="SomeOutPort"/>
   ...
   <Edge Source="SomeNode.SomeOut"
         Target="SomeOutPort"/>
   ...
</Graph>
```

Однако, как видно и из примера, ребра могут соединять только входные порты графа с входными портами узлов или выходные порты узлов с выходными портами графов. Конечно, входные и выходные порты графа могут быть соединены с источниками и используя язык CSharp код, но объявление ребра через RAML часто позволяет это сделать быстрее и нагляднее. Например в случае, когда необходимо просто представить порт узла в виде порта графа.

4.3 Разработка простого приложения с использванием WRF

В данной главе показан процесс создания простого игрового приложения имитирующего процесс игры двух игроков в настольный теннис.

4.3.1 Задача

Необходимо разработать приложения моделирующее игру настольный теннис, используя следующие ограничения:

- Наличие двух игроков
- Наличие судьи
- Поведение игроков моделируется
- Поведение судьи моделируется
- Наличие вывода текущего счета в формате целого числа, которое характеризует разницу очков игроков

4.3.2 Проектирование

Итоговое приложение должно состоять из трех компонентов:

- Первый игрок
- Второй игрок
- Судья

Взаимодействие этих компонентов изображено на соответствующей диаграмме коммуникации UML (Рис. 21).

Каждый компонент системы является реактивным, так как отвечает на воздействие со стороны внешнего объекта. Более того, его исполнение целиком и полностью зависит от внешних воздействий.

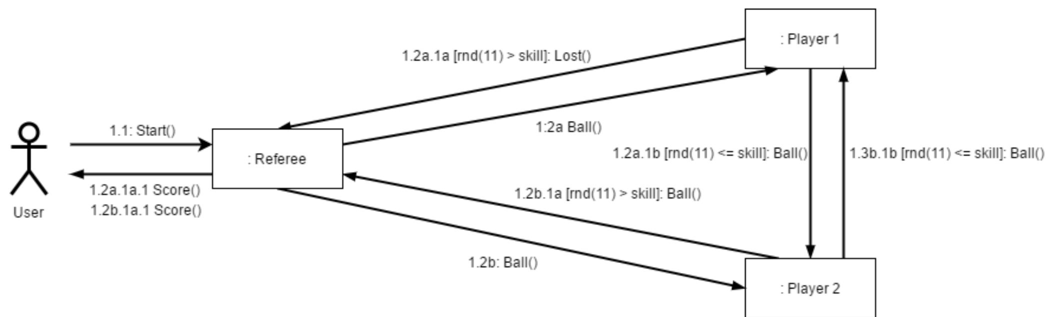

Рис. 21: Диаграмма коммуникации между компонентами приложения PingPongGame

4.3.3 Реализация

Для того чтобы создать WRF проект необходимы следующие действия:

- Нажать File > New > Project

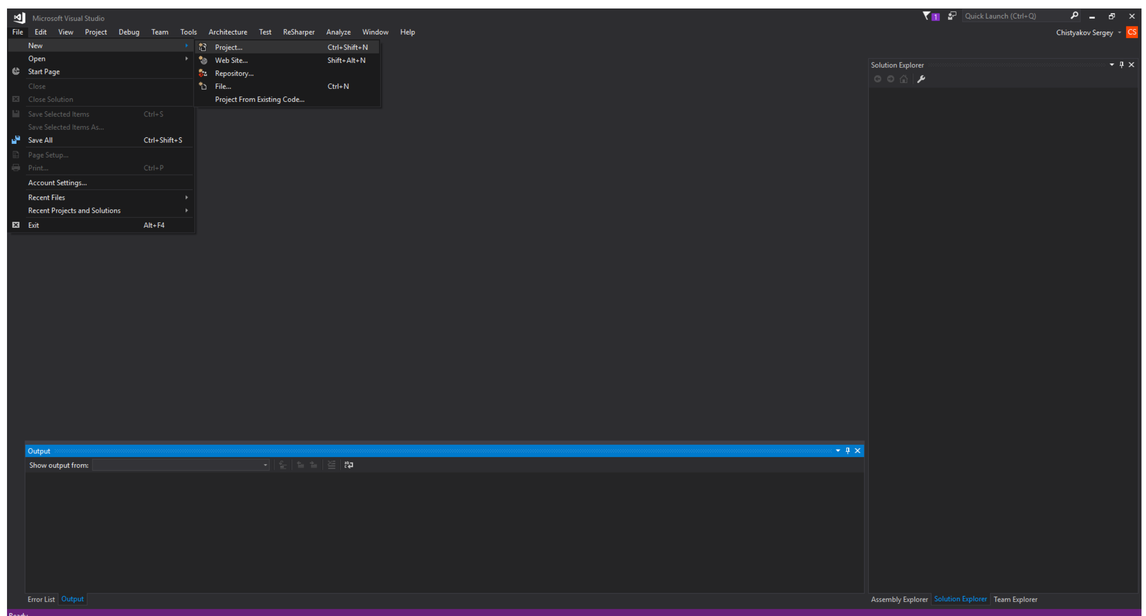

Рис. 22: Создание WRF приложения

- Выбрать раздел WRF и элемент WRF App

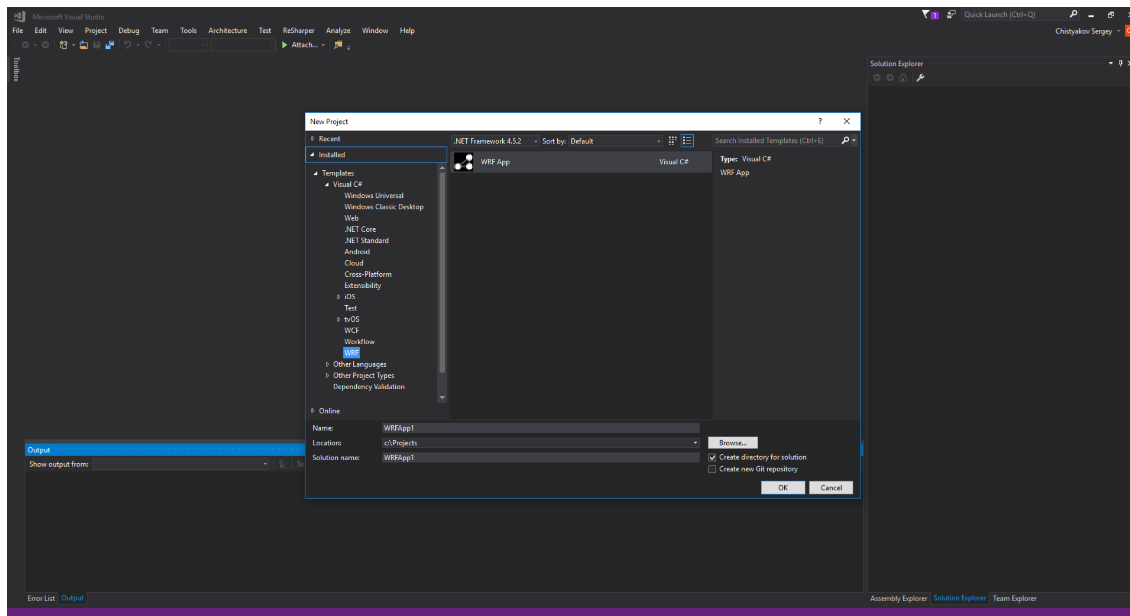

Рис. 23: Создание WRF приложения

- Выбрать имя для приложения и нажать кнопку OK

Для того чтобы разработать все желаемые компоненты приложения, необходимо создать два узла. Стоит отметить, что для обоих игроков будет использоваться один узел, но разные экземпляры в контексте графа приложения. Чтобы создать узел необходимы следующие действия:

- Нажать Add > New Item...

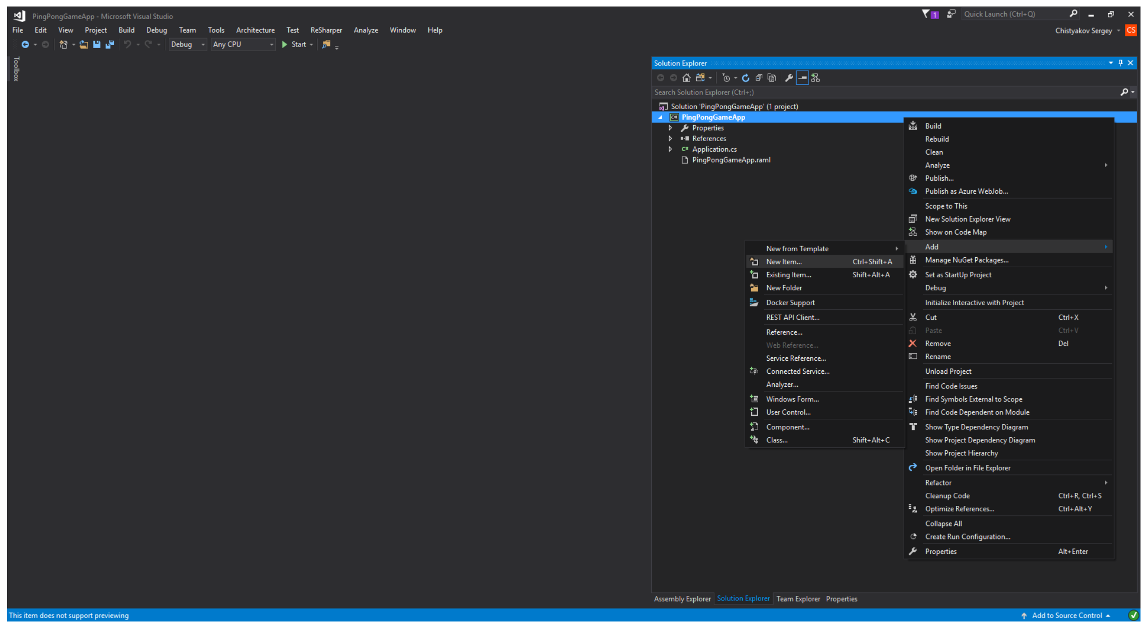

Рис. 24: Создание WRF узла

- Выбрать раздел WRF и элемент WRF Node

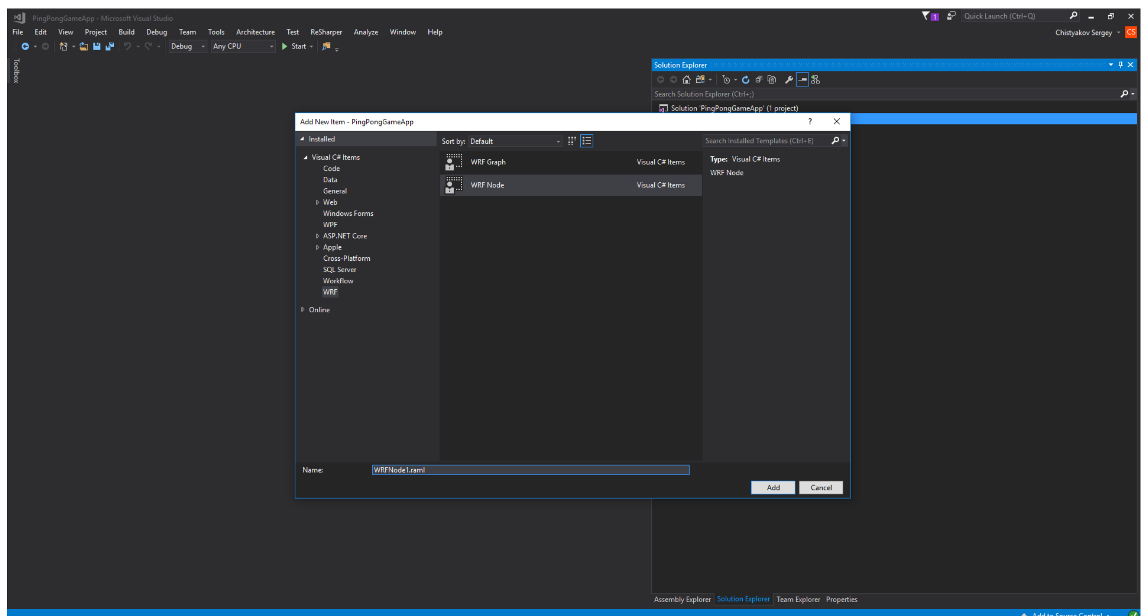

Рис. 25: Создание WRF узла

• Выбрать имя для узла и нажать кнопку OK

После добавления, разработчик может приступать к конфигурированию каждого узла используя средства языка RAML. Ниже приведен код на языке RAML описывающий структуру каждого из узлов.

```
<Node x:Class="PingPongPlayerNode"
      xmlns:x="http://schemas.wrf.com/raml"
      xmlns:system="clr-namespace:System;assembly=mscorlib"
      xmlns:reactive="clr-namespace:System.Reactive;
                      assembly=System.Reactive.Core">
   <InPort Id="In"
           Type="reactive:Unit"/>
   <OutPort Id="Out"
            Type="reactive:Unit"/>
   <InPort Id="GotBall"
           Type="system:Boolean"/>
   <OutPort Out="Lost"
            Type="reactive:Unit"/>
</Node>
```

```
<Node x:Class="PingPongRefereeNode"
        xmlns:x="http://schemas.wrf.com/raml"
        xmlns:system="clr-namespace:System;assembly=mscorlib"
        xmlns:reactive="clr-namespace:System.Reactive;
                          assembly=System.Reactive.Core">
    <OutBehaviorPort Id="GiveBall1"
                     Type="system:Boolean"
                     Value="true"/>
    <OutBehaviorPort Id="GiveBall2"
                     Type="system:Boolean"
                     Value="false"/>
    <OutBehaviorPort Id="Score"
                     Type="system:Int32"
                     Value="0"/>
    <InPort Id="Lost1"
            Type="reactive:Unit"/>
    <InPort Id="Lost2"
            Type="reactive:Unit"/>
</Node>
```

После добавления описания структуры двух компонент, можно приступать к разработке и конфигурации архитектуры графа приложения. Ниже приведен код на языке RAML описывающий структуру графа приложения.

```
<Graph x:Class="PingPongGameApp"
       xmlns:x="http://schemas.wrf.com/raml"
       xmlns:system="clr-namespace:System;assembly=mscorlib">
   <PingPongPlayerNode Id="Player1">
      <x:Arguments>
         <system:String>Player 1</system:String>
         <system:Int32>3</system:Int32>
      </x:Arguments>
   </PingPongPlayerNode>
   <PingPongPlayerNode Id="Player2">
      <x:Arguments>
         <system:String>Player 2</system:String>
         <system:Int32>6</system:Int32>
      </x:Arguments>
   </PingPongPlayerNode>
   <PingPongRefereeNode Id="Referee"/>
   <OutReplayPort Id="Score"
                  Name="GameScore"
                  BufferSize="1"/>
   <Edge Source="Player1.Out"
         Target="Player2.In" />
   <Edge Source="Player2.Out"
         Target="Player1.In"/>
   <Edge Source="Referee.GiveBall1"
         Target="Player1.GotBall"/>
   <Edge Source="Referee.GiveBall2"
         Target="Player2.GotBall"/>
   <Edge Source="Player1.Lost"
         Target="Referee.Lost1"/>
```

```
  <Edge Source="Player2.Lost"
        Target="Referee.Lost2"/>
  <Edge Source="Referee.Score"
        Target="Score"/>
</Graph>
```

В графе используется конструкция языка RAML, которая не была описана выше - Arguments. Часто бывает необходимо создавать узлы, у которых есть конструктор с параметрами или вызвать статический метод создания узла. Эти проблемы могут быть решены разработчиком с помощью атрибутов языка RAML x:Arguments и x:FactoryMethod. Атрибут x:Argument используется для указания аргументов для параметризированного конструктора или для объявления объекта фабричного метода. Атрибут x:FactoryMethod используется для указания фабричного метода, который может использоваться для инициализации узла. Кроме того, атрибут x: TypeArguments может использоваться для указания типа аргументов для конструктора универсального типа.

После получения окончательной архитектуры графа приложения и его компонентов, необходимо реализовать поведение каждого из них. WRF предоставляет удобные механизмы конфигурирования логики, которые в совокупности с Reactive Extensions API превращаются в серьезный инструмент разработки. Реализации поведений двух компонентов PingPongPlayerNode и PingPongRefereeNode представлены в приложениях к данной работе.

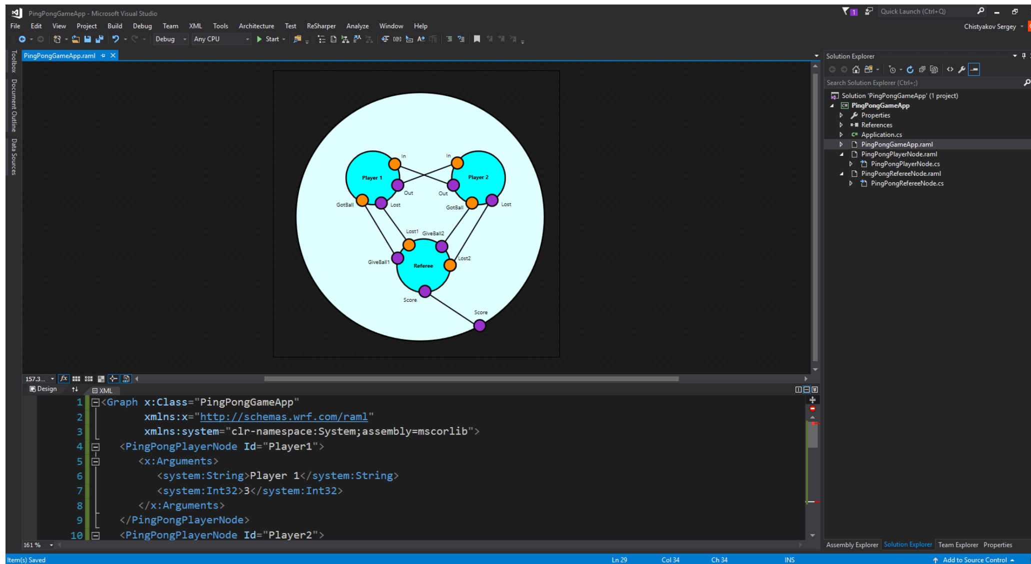

Рис. 26: Визуализация графа с помощью инструмента RAML Designer

4.3.4 Результаты

Ниже представлены результаты работы программы:

```
Current game score: 0
Current game score: -1
Current game score: -2
Current game score: -3
Current game score: -2
Current game score: -3
Current game score: -4
Current game score: -5
Current game score: -6
```

После запуска программы, отчетливо видно, что разработанное с помо-

щью платформы WRF решение удовлетворяет всем описанным требованиям. Стоит так же напомнить что при объявлении двух компонентов игроков были указаны разные аргументы характеризующие уровень их мастерства - 3 и 6. Этот факт объясняет разницу в счете, который представляет из себя разницу очков, относительно первого игрока. Данный пример приложения не показывает всех возможностей WRF, но описывает последовательность создания простого приложений с использованием основных инструментов платформы.

4.4 Приложение для Online шопинга разработанное с использванием WRF

В данной главе представлена реализация одного из компонентов приложения для шопинга Online. Компонент WebStore содержит три компонента, связанных с онлайн-шопингом: поисковая система, корзина покупок и аутентификация. Компонент «Поисковая система» позволяет осуществлять поиск или просмотр элементов. Компонент «Корзина» предоставляет механизм управления заказами. Компонент аутентификации позволяет клиентам создавать учетную запись, заходить или выходить из системы.

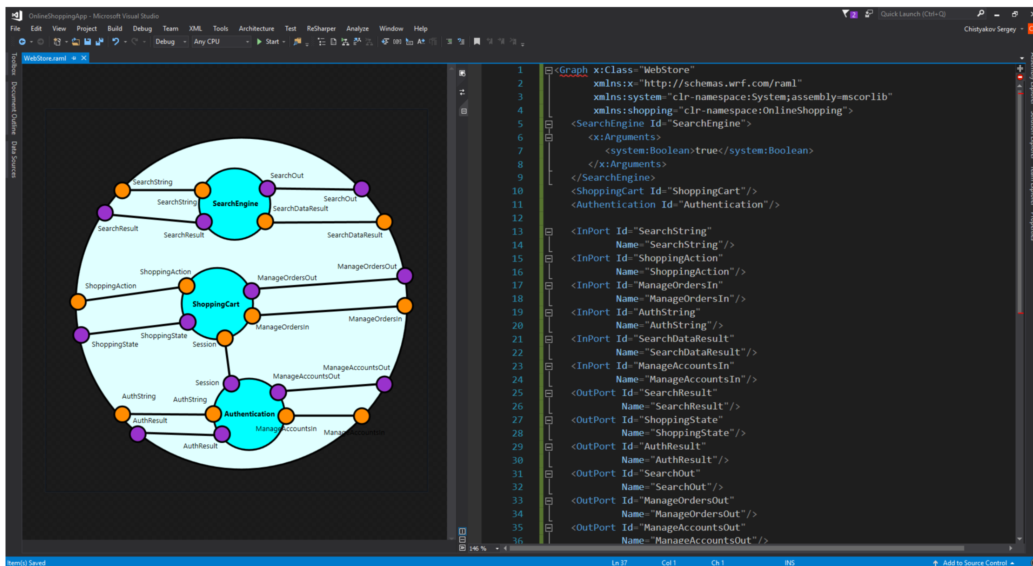

Рис. 27: Компонент WebStore

5 Результаты

- Проведен анализ большого объема реактивных приложений
- Проведен анализ подходов к визуальному программированию, как к возможному решению проблемы возрастающей сложности поддержки и разработки больших реактивных приложений
- Разработаны прототипы унифицированной модели программирования для создания реактивных приложений
- Разработана итоговая унифицированная модель программирования для создания реактивных приложений, включающая в себя
  - Декларативный язык разметки основанный на XML, который упрощает создание структурного графа реактивного приложения
  - Интеграцию с Visual Studio. Основные компоненты интеграции:
    - Визуальный конструктор
    - Поддержка расширения файлов
    - Расширенное выделение синтаксиса, функции завершения кода IntelliSense и интуитивную навигацию по коду в редакторе кода, созданном на основе платформы компилятора .NET («Roslyn»)
    - Различные спецификации

6 Список литературы

[1] Duncan K. DeVore, Sean Walsh, Brian Hanafee: Reactive Application Development (2014)

[2] Lee Campbell: Introduction to Rx: A step by step guide to the Reactive Extensions to .NET (2012)

[3] Ivan Morgillo, Sasa Sekulic, Fabrizio Chignoli: Grokking ReactiveX, (2016)

[4] Matthew MacDonald: Pro WPF 4.5 in C Sharp: Windows Presentation Foundation in .NET 4.5 (2012)

[5] Jonas Bonér, Co-Founder and CTO Lightbend, Inc.: Reactive Microservices Architecture (2016)

[6] Markus Eisele, Enterprise Advocate, Lightbend, Inc.: Developing Reactive Microservices (2016)

[7] Jeff Smith: Reactive Machine Learning Systems (2016)

[8] Julien Richard-Foy: Play Framework Essentials (2014)

[9] Andre Staltz: The introduction to Reactive Programming you've been missing (2014)

[10] The Couchbase documentation: Mastering Observables

[11] Russell Elledge: Reactive Programming in Java 8 With RxJava (2014)

[12] Tomasz Kowalczewski: 33rd Degree Reactive Java (2014)

[13] Bodil Stokke: What Every Hipster Should Know About Functional Reactive Programming (2014)

[14] Erik Meijer: Your Mouse is a Database (2012)

[15] McGraw, J.: The VAL Language: Description and Analysis (1982)

[16] Agha, G.: Actors: a Model of Concurrent Computation in Distributed Systems, Series in Artificial Intelligence (Jun 1985)

[17] Arvind, D.: IEEE Xplore - Dataflow architectures and multithreading. Annual review of computer science (1986)

[18] Browne, J., Hyder, S., Dongarra, J.: IEEE Xplore - Visual programming and debugging for parallel computing (1995)

[19] Cann, D.: Retire Fortran? A debate rekindled (1991)

[20] Dennis, J.B.: Data Flow Supercomputers. Computer 13(11), 48-56 (1980)

[21] Feo, J., Cann, D.: A report on the Sisal language project (1990)

[22] Halbwachs, N., Caspi, P., Raymond, P., Pilaud, D.: The synchronous data flow programming language LUSTRE. Proceedings of the IEEE 79(9), 1305-1320 (Sep 1991)

[23] Johnston, W., Hanna, J.: Advances in dataflow programming languages. ACM Computing Surveys (CSUR) (2004)

[24] Kahn, G.: The Semantics of a Simple Language for Parallel Programming. In Information Processing ґ Proceedings of the IFIP Congress (1974), pp. 471-475. 74: pp. 471-475 (1974)

[25] McGraw, J.: The VAL Language: Description and Analysis (1982)

Приложения

Приложение 1. PingPongPlayerNode

```
public partial class PingPongPlayerNode
{
    private readonly Random p_rnd;

    public PingPongPlayerNode(string _id, Int32 _skill)
    {
        InitializeComponent();
        p_rnd = new Random();
        In.Throttle(TimeSpan.FromMilliseconds(50)).Subscribe(_x
            =>
        {
            var n = p_rnd.Next(11);
            if (n > _skill)
                Lost.OnNext(Unit.Default);
            else
                Out.OnNext(Unit.Default);

        });
        GotBall
            .Where(_x => _x)
            .Select(_ => Unit.Default)
            .Subscribe(Out);
    }
}
```

Приложение 2. PingPongRefereeNode

```
public partial class PingPongRefereeNode
{
    public PingPongRefereeNode()
    {
        InitializeComponent();

        Lost1.Select(_ => -1)
            .Merge(Lost2.Select(_ => 1))
            .Scan(0, (_i, _x) => _i + _x)
            .Subscribe(Score);
        Lost1
            .Select(_x => true)
            .Subscribe(GiveBall1);
        Lost2
            .Select(_x => true)
            .Subscribe(GiveBall2);
    }
}
```