Министерство образования Республики Беларусь

Учреждение образования
БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

УДК 004.652.43 + 004.414.23

Сафонов
Анатолий Анатольевич

Реактивное и функциональное программирование для обработки данных

ДИССЕРТАЦИЯ

на соискание степени магистра технических наук
по специальности 1-40 80 04 «Математическое моделирование, численные методы и комплексы программ»

Научный руководитель
Ганжа Виктор Александрович
кандидат физико-математических наук, доцент

Минск 2016

Нормоконтроль

_____

_____

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

В последние годы требования к приложениям значительно изменились. С ростом объёмов информации, требуются новые способы их обработки. Пользователям требуется быстрый доступ к новой информации и возможность получения данных в реальном времени. Это стимулирует разработчиков создавать отзывчивые интерфейсы и модели для обработки. Десятки серверов, время отклика в несколько секунд, оффлайновое обслуживание, которое могло длиться часами, гигабайты данных — такими были большие приложения несколько лет назад. Сегодня же приложения работают абсолютно на всём, начиная с простых мобильных телефонов и заканчивая кластерами из тысячи процессоров. Пользователи ожидают миллисекундного времени отклика и стопроцентного аптайма, в то время как данные выросли до петабайтов.

Новые требования требуют новых решений. Раньше делался акцент на повышение мощности аппаратной части системы. Масштабирование достигалось засчёт покупки более производительных серверов и использования многопоточности. Для добавления новых серверов приходилось применять комплексные, неэффективные и дорогие проприетарные решения.

Однако прогресс не стоит на месте. Архитектура приложений изменяется в соответствии с новыми требованиями. Новые архитектуру позволяют разработчикам создавать событийно-ориентированные, масштабируемые, отказоустойчивые и отзывчивые приложения — приложения, работающие в реальном времени и обеспечивающие хорошее время реакции, основанные на масштабируемом и отказоустойчивом стеке и которые легко развернуть на многоядерных и облачных архитектурах. Эти особенности критически важны для реактивности.

Стремясь к упрощению разработки сложных систем, было создано множество парадигм программирования, каждая из созданных парадигм занимает свою нишу. Все парадигмы разделяются на две большие группы:

– императивные;
– декларативные.

Также существует множество парадигм, которые имеют черты как императивных так и декларативных систем. Одна из них — Functional Reactive Programming (FRP). Она объединяет черты функционального и реактивного программирования.

Функциональное программирование предполагает обходиться вычислением результатов функций от исходных данных и результатов других функций, и не предполагает явного хранения состояния программы. Как известно, функциональный подход к программированию имеет свою специфику: в нём

мы преобразовываем данные, а не меняем их. Соответственно, не предполагает оно и изменяемость этого состояния. Но это накладывает свои ограничения, например, при создании программ активно взаимодействующих с пользователем. В императивном языке намного проще реализовать такое поведение, ведь мы можем реагировать на какие-либо события «в реальном времени», в то время как в чистых функциональных языках нам придётся откладывать общение с системой до самого конца. Реактивное программирование ориентируется на потоки данных и распространение изменений. Функциональное реактивное программирование объединяет принципы функционального программирования и реактивного программирование:

– преобразование данных;
– распространение изменений.

Приложения, разработанные на основе этой архитектуры, называются реактивными приложениями. Словарь Merriam Webster даёт определение реактивному как «готовому реагировать на внешние события», что означает что компоненты всё время активны и всегда готовы получать сообщения. Это определение раскрывает суть реактивных приложений, фокусируясь на системах, которые:

– реагируют на события;
– реагируют на повышение нагрузки;
– реагируют на сбои;
– реагируют на пользователей.

Каждая из этих характеристик существенна для реактивного приложения. Все они зависят друг от друга, но не как ярусы стандартной многоуровневой архитектуры. Напротив, они описывают свойства, применимые на всём стеке технологий.

В данной работе исследуются и реализуются некоторые механизмы реляционной алгебры, реализованные в соответствии с концепциями реактивного и функционального программирования.

В результате была разработана библиотека классов для платформы .NET, написанная на языках программирования C# и F#, пригодная для решения практических задач в коммерческих проектах. На данный момент в библиотеке реализованы структуры хранения данных и операции реляционной алгебры.

ГЛАВА 1
ОБЗОР ПРЕДМЕТНОЙ ОБЛАСТИ

В данном разделе будет произведён обзор предметной области задачи, решаемой в рамках диссертации; рассмотрены вопросы о сущности функционального и реактивного программирования и принципе их работы; также будут приведены существующие системы.

1 Функциональное программирование

Программы написанные на императивных языках программирования, таких как C#, Java или C++, в своей работе опираются на изменение значений набора переменных, называемого состоянием. Если мы пренебрежём операциями ввода-вывода и вероятностью того, что программы будет работать постоянно, то мы можем перейти к следующей абстракции. Первоначально состояние σ, представляющее собой входные данные для программы, а после завершения её исполнения — новое значение σ', представляющее результаты [1]. Выполнение отдельных операторов сводится к изменению ними состояния, которое последовательно проходит через конечное число значений:

$$\sigma = \sigma_0 \to \sigma_1 \to \sigma_2 \to ... \to \sigma_n = \sigma' \tag{1.1}$$

Например, в программе сортировки состояние первоначально включает в себя массив значений, а после того, как программа завершается, состояние модифицируется таким образом, что эти значения становятся упорядоченными, в то время как промежуточные состояния представляют собой ход достижения данной цели.

Состояние обычно изменяется с помощью операторов присваивания, часто записываемых в виде $v = E$ или $v := E$, где v — переменная, а E — некоторое выражение. Последовательность выполнения таких операторов задаётся в тексте программы их размещением друг за другом (при этом часто в качестве разделителя применяется точка с запятой). С помощью составных операторов, таких как if и while, можно выполнять операторы в зависимости от условия или циклически, часто полагаясь на другие свойства текущего состояния. В результате программа превращается в набор инструкций по изменению состояния, и поэтому данный стиль программирования часто называется императивным или процедурным. Соответственно, традиционные языки программирования, поддерживающие такой стиль, также известны как императивные или процедурные языки.

Функциональное программирование радикально отличается от этой модели. По существу, функциональная программа представляет собой просто выражение, а выполнение программы процесс его вычисления. В общих чертах мы можем понять, как это возможно, используя следующие рассуждения. Предположим, что императивная программа (вся целиком) детерминирована, т.е. выход полностью определяется входом; мы можем сказать, что конечное состояние или тот его фрагмент, который нас интересует, являются функцией начального состояния, например $\sigma' = f(\sigma)$. В функциональном программировании эта точка зрения имеет особое значение: программа – это выражение, которое соответствует математической функции f. Функциональные языки поддерживают создание таких выражений за счет того, что позволяют использовать мощные функциональные конструкции [1].

Функциональное программирование может противопоставляться императивному как с хорошей, так и в плохой стороны. К недостаткам ФП можно отнести то, что функциональные программы не используют переменные — то есть не имеют состояния. Соответственно, они не могут использовать присваивание, поскольку нечему присваивать. Кроме того, идея последовательного выполнения операторов также бессмысленна, поскольку первый оператор не имеет никакого влияния на второй, так как нет никакого состояния, передаваемого между ними. К достоинствам функционального подхода можно отнести то, что функциональные программы могут использовать функции более изящным способом. Функции могут рассматриваться точно так же, как и более простые объекты, такие как целые числа: они могут передаваться в другие функции как аргументы и возвращаться в качестве результатов, а также применяться в вычислениях. Вместо последовательного выполнения операторов и использования циклов, функциональные языки программирования предлагают рекурсивные функции, т.е. функции, определённые в терминах самих себя. Большинство традиционных языков программирования обеспечивают весьма скудные возможности в этих областях. Язык C имеет некоторые ограниченные возможности работы с функциями при помощи указателей, но не позволяет создавать новые функции динамически. В языке C# существуют более широкие способы работы с функциями:

– делегаты;
– лямбда-функции;
– анонимные типы;

На первый взгляд, язык без переменных или возможности последовательного выполнения инструкций кажется совершенно непрактичным. Многие свойства императивных языков программирования развились в процессе абстрагирования от типового компьютерного оборудования, от машинного ко-

да к ассемблерам, затем к макроассемблерам, языку C++ и так далее. Нет оснований утверждать, что такие языки представляют собой наиболее удобный способ взаимодействия человека и машины. В самом деле, последнее слово в развитии компьютерных архитектур еще не сказано, и компьютеры должны служить нашим нуждам, а не наоборот. Вероятно, что правильный подход не в том, чтобы начать с оборудования и продвигаться вверх, а наоборот – начать работу с языка программирования, как средства для описания алгоритмов, и затем двигаться вниз к оборудованию [2]. В действительности, данная тенденция может быть обнаружена и в традиционных языках программирования. Даже C++ позволяет записывать арифметические выражения обычным способом. Программист не обеспокоен задачей линеаризации вычисления подвыражений и выделения памяти для хранения промежуточных результатов.

2 Реактивное программирование

Реактивное программирование — парадигма программирования, ориентированная на потоки данных и распространение изменений. Это означает, что должна существовать возможность легко выражать статические и динамические потоки данных, а также то, что нижележащая модель исполнения должна автоматически распространять изменения благодаря потоку данных.

К примеру, в императивном программировании присваивание $a := b + c$ будет означать, что переменной a будет присвоен результат выполнения операции $b+c$, используя текущие (на момент вычисления) значения переменных. Позже значения переменных b и c могут быть изменены без какого-либо влияния на значение переменной a. В реактивном же программировании значение a будет автоматически пересчитано, основываясь на новых значениях.

Системы построенные с помощью реактивного программирования обладают такими свойтвами 1.1:

- событийно-ориентированость (Event-driven);
- масштабируемость (Scalable);
- отказоустойчивость (Resilient);
- отзывчивость (Responsive);

Это обеспечивает комфортное взаимодействие с пользователем, дающее ощущение реального времени и поддерживаемое самовосстанавливающимся масштабируемым прикладным стеком, готовым к развертыванию в многоядерных и облачных окружениях. Каждая из четырех характеристик реактивной архитектуры применяется ко всему технологическому стеку, что отличает их от звеньев в многоуровневых архитектурах. Рассмотрим их немного подробней.

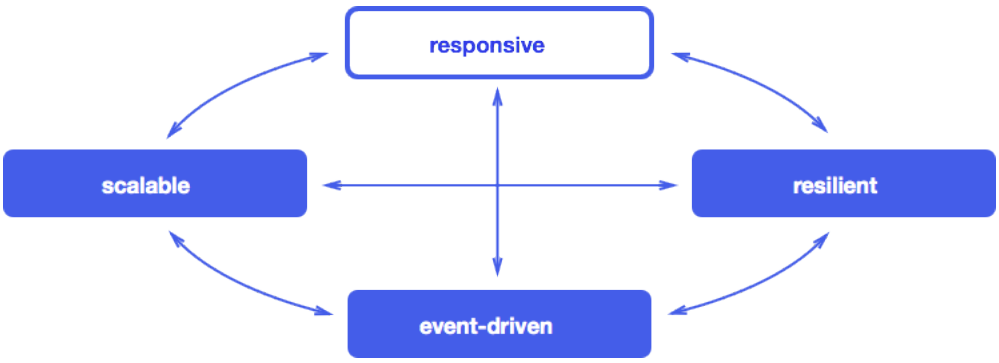

Рисунок *1.1* — Принципы реактивного программирования

2.1 Event-driven. Событийно-ориентированные приложения предполагают асинхронные коммуникации компонент и реализуют их слабую связанность (loosely coupled design): отправитель и получатель сообщения не нуждаются в сведениях ни друг о друге, ни о способе передачи сообщения, что позволяет им сконцентрироваться на содержании коммуникаций. Кроме того, что слабосвязанные компоненты значительно улучшают сопровождаемость, расширяемость и эволюционирование системы, асинхронность и неблокирующий характер их взаимодействия позволяют также освободить значительную часть ресурсов, снизить время оклика и обеспечить большую пропускную способность по сравнению с традиционными приложениями. Именно благодаря событийно-ориентированной природе возможны остальные черты реактивной архитектуры.

В событийно-ориентированном приложении компоненты взаимодействуют друг с другом путём отправки и получения сообщений — дискретных частей информации, описывающих факты. Эти сообщения отправляются и принимаются в асинхронном и неблокирующем режиме. Модели построенные на событийно-ориентированной модели более склонны к push-модели, нежели чем к pull или poll. Т.е. они проталкивают данные к своим потребителям, когда данные становятся доступными, вместо того чтобы впустую тратить ресурсы, постоянно запрашивая или ожидая данные.

– Асинхронная передача сообщений означает, что приложение по своей природе обладает высокой степенью конкуренции и может без изменений работать на многоядерной архитектуре. Любое ядро CPU может обработать любое сообщение, что даёт большие возможности параллелизации.

– Неблокирование означает способность продолжать работать, чтобы приложение было отзывчивым всё время, даже в условиях сбоя или пиковой нагрузки. Для этого все необходимые для обеспечения отзывчивости ресурсы, например CPU, память и сеть, не должно быть монополизированы. Это приведёт к более низкой латентности, большей пропускной способности и лучшей

масштабируемости;

Традиционные серверные архитектуры используют общедоступное изменяемое состояние и блокирующие операции на одном потоке. Это вносит трудности при масштабировании системы. Общедоступное изменяемое состояние требует синхронизации, что привносит сложность и недетерменированность, делая код трудным для понимания и сопровождения. Переключение потока в спящий режим расходует ограниченные ресурсы, а пробуждение стоит дорого.

Разделяя генерацию событий и их обработку, мы позволяем платформе самой позаботиться о деталях синхронизации и диспетчеризации событий между потоками, в то время как мы сами концентрируемся на более высокоуровневых абстракциях и бизнес-логике. Мы думаем о том, откуда и куда пересылаются события, и о том, как компоненты взаимодействуют между собой, вместо того чтобы копаться с низкоуровневыми примитивами вроде потоков или блокировок.

Событийно-ориентированные модели обеспечивают слабую связанность между компонентами и подсистемами. Такая связанность, как мы увидим позже, является одним из необходимых условий масштабируемости и отказоустойчивости. Без сложных и сильных зависимостей между компонентами расширение системы требует минимальных усилий.

Когда от приложения требуется высокая производительность и хорошая масштабируемость, трудно предусмотреть, где могут возникнуть узкие места. Поэтому очень важно, чтобы всё решение было асинхронным и неблокирующим. Для типичного приложения это означает, что архитектура должна быть полностью событийно-ориентированной, начиная с запросов пользователей через графический интерфейс и обработки запросов в веб-слое и заканчивая сервисами, кэшем и базой данных. Если хотя бы один из этих слоёв не будет отвечать этому требованию — будет делать блокирующие запросы в БД, использовать общедоступное изменяемое состояние, вызывать дорогие синхронные операции — то весь стек заглохнет и пользователи будут страдать из-за возросших задержек и упавшей масштабируемости.

2.2 Scalable. Масштабируемость в контексте реактивного программирования — это реакция системы на изменение нагрузки, т.е. эластичность, достигаемая возможностью добавления или освобождения вычислительных узлов по мере необходимости. Благодаря низкой связанности, асинхронному обмену сообщениями и независимости от размещения компонент (location transparency), способ развертывания и топология приложения становятся решением времени развертывания и предметом конфигурации и адаптивных ал-

горитмов, реагирующих на нагрузку. Таким образом, вычислительная сеть становится частью приложения, изначально имеющего явную распределенную природу.

Событийно-ориентированная система, базирующаяся на асинхронной передаче сообщений, является основой масштабируемости. Слабая связанность и локационная независимость компонентов и подсистем позволяют разворачивать систему на множестве узлов, оставаясь в пределах той же самой программной модели с той же семантикой. При добавлении новых узлов возрастает пропускная способность системы. В терминах реализации не должно быть никакой разницы между развертыванием системы на большем количестве ядер или большем количестве узлов кластера или центра обработки данных. Топология приложения становится проблемой конфигурации и/или адаптивных алгоритмов времени выполнения, следящих за нагрузкой на систему. Это то, что мы называем локационной прозрачностью [3].

Важно понимать, что цель — не изобрести прозрачные распределённые вычисления, распределённые объекты или RPC-коммуникации — это уже пытались сделать раньше и эта затея провалилась. Вместо этого мы должны охватить сеть, представив её прямым образом в программной модели через механизм асинхронных сообщений. Настоящая масштабируемость естественным образом полагается на распределённые вычисления и их межузловое взаимодействие, что означает обход сети, который по своей сути является ненадёжным. Поэтому важно учесть ограничения, компромиссы и сценарии исключительных ситуаций явно в программной модели вместо того, чтобы прятать их за ширмой дырявых абстракций, которые якобы пытаются «упростить» вещи. Как следствие, в равной степени важно обеспечить себя программными инструментами, содержащими в себе строительные блоки для решения типичных проблем, которые могут возникнуть в распределённом окружении — вроде механизмов для достижения консенсуса или интерфейсов обмена сообщениями, которые обладают высоким уровнем надёжности.

2.3 Resilent. Отказоустойчивость реактивной архитектуры тоже становится частью дизайна, и это значительно отличает ее от традиционных подходов к обеспечению непрерывной доступности системы путем резервирования серверов и перехвата управления при отказе. Устойчивость такой системы достигается ее способностью корректно реагировать на сбои отдельных компонент, изолировать эти сбои, сохраняя их контекст в виде вызвавших их сообщений, и передавать эти сообщения другому компоненту, способному принять решения о том, как следует обрабатывать ошибку. Такой подход позволяет сохранить чистой бизнес-логику приложения, отделив от нее логику обработки

сбоев, которая формулируется в явном декларативном виде для регистрации, изолирования, и обработки сбоев средствами самой системы. Для построения таких самовосстанавливающихся систем компоненты упорядочиваются иерархически, и проблема эскалируется до того уровня, который способен ее решить.

Чтобы управлять сбоями, нам нужен способ изолировать их, так чтобы они не распространялись на другие работоспособные компоненты, и вести за ними наблюдение из безопасного места вне контекста, в котором могут происходить сбои. Один способ, который приходит на ум, — это переборки, разделяющие систему на отсеки таким образом, что если затапливается один из отсеков (выходит из строя), то это никак не влияет на другие отсеки. Это предотвращает классическую проблему каскадных сбоев и позволяет решать проблемы изолированно.

Событийно-ориентированная модель, которая даёт масштабируемость, также предоставляет необходимые примитивы для решения проблемы отказоустойчивости. Слабая связанность в событийно-ориентированной модели снабжает нас полностью изолированными компонентами, в которых сбои инкапсулируются в сообщения вместе с необходимыми деталями и пересылаются другим компонентам, которые в свою очередь анализируют ошибки и решают, как реагировать на них.

Такой подход создаёт систему, в которой:

– бизнес-логика остаётся чистой, отделённой от обработки ошибок;

– сбои моделируются явно, чтобы разбиение на отсеки, наблюдение, управление и конфигурация задавались декларативно;

– система может «лечить» себя и восстанавливаться автоматически;

Лучше всего, если отсеки организуются иерархическим образом, подобно большой корпорации, где проблемы поднимаются до уровня, имеющего достаточно власти, чтобы принять соответствующие меры.

Мощь данной модели в том, что она чисто событийно-ориентированная — она основана на реактивных компонентах и асинхронных событиях, и поэтому обладает локационной прозрачностью. На практике это означает, что её семантика не зависит от того, работает ли она на локальном сервере или в распределённом окружении.

2.4 Responsive. И наконец отзывчивость — это способность системы реагировать на пользовательское воздействие независимо от нагрузки и сбоев, такие приложения вовлекают пользователя во взаимодействие, создают ощущение тесной связи с системой и достаточной оснащенности для выполнения текущих задач. Отзывчивость актуальна не только в системах реального вре-

мени, но и необходима для широкого круга приложений. Более того, система, неспособная к быстрому отклику даже в момент сбоя, не может считаться отказоустойчивой. Отзывчивость достигается применением наблюдаемых моделей (observable models), потоков событий (event streams) и клиентов с состоянием (stateful clients). Наблюдаемые модели генерируют события при изменении своего состояния и обеспечивают взаимодействие реального времени между пользователями и системами, а потоки событий предоставляют абстракцию, на которой построено это взаимодействие путем неблокирующих асинхронных трансформаций и коммуникаций.

Реактивные приложения используют наблюдаемые модели, потоки событий и клиенты с состоянием.

Наблюдаемые модели позволяют другим системам получать события, когда их состояние изменяется. Это обеспечивает связь в реальном времени между пользователями и системами. Например, когда несколько пользователей работают одновременно над одной и той же моделью, изменения могут реактивно синхронизироваться между ними, избавляя от необходимости блокировки модели.

Потоки событий образуют базовую абстракцию, на которой строятся такие связи. Сохраняя их реактивными, мы избегаем блокировок и позволяем преобразованиям и коммуникациям быть асинхронными и неблокирующими.

Реактивные приложения должны иметь понятие о порядках алгоритмов [4], чтобы быть уверенным, что время отклика на события не превышает O(1) или, как минимум, O(log n) независимо от нагрузки. Может быть включён коэффициент масштабирования, но он не должен зависеть от количества клиентов, сессий, продуктов или сделок.

Вот несколько стратегий, которые помогут сохранить латентность независимой от профиля нагрузки:

– В случае взрывного трафика реактивные приложения должны амортизировать затраты на дорогие операции, такие как ввод-вывод или конкурентный обмен данными, применяя batching с пониманием и учитыванием специфики низлежащих ресурсов.

– Очереди должны быть ограничены с учётом интенсивности потока, длины очередей при данных требованиях на время отклика должны определяться согласно закону Литтла [5].

– Системы должны находиться в состоянии постоянного мониторинга и иметь адекватный запас прочности.

– В случае сбоев активируются автоматические выключатели и запускаются запасные стратегии обработки [6].

В качестве примера рассмотрим отзывчивое веб-приложение с «богаты-

ми» клиентами (браузер, мобильное приложение), чтобы предоставить пользователю качественный опыт взаимодействия. Это приложение исполняет логику и хранит состояние на стороне клиента, в котором наблюдаемые модели предоставляют механизм обновления пользовательского интерфейса при изменении данных в реальном времени. Технологии вроде WebSockets или Server-Sent Events позволяют пользовательскому интерфейсу соединяться напрямую с потоком событий, так что система целиком становится событийно-ориентированной, начиная с back-end слоя и заканчивая клиентом. Это позволяет реактивным приложениям проталкивать события браузеру и мобильным приложениям посредством асинхронной и неблокирующей передачи данных, сохраняя масштабируемость и отказоустойчивость.

Таким образом, реактивные приложения представляют собой сбалансированный подход к решению широкого спектра задач современной разработки ПО. Построенные на событийно-ориентированном основании, они предоставляют средства, необходимые для гарантий масштабируемости и отказоустойчивости и поддерживают полнофункциональное отзывчивое пользовательское взаимодействие [7].

ГЛАВА 2
ИСПОЛЬЗУЕМЫЕ ТЕХНОЛОГИИ

Выбор технологий является важным предварительным этапом разработки сложных информационных систем. Платформа и язык программирования, на котором будет реализована система, заслуживает большого внимания, так как исследования показали, что выбор языка программирования влияет на производительность труда программистов и качество создаваемого ими кода [8, с. 59].

Основываясь на опыте работы имеющихся программистов разрабатывать ПО целесообразно на платформе .NET. Приняв во внимание необходимость обеспечения доступности дальнейшей поддержки ПО, возможно, другой командой программистов, целесообразно не использовать малоизвестные и сложные языки программирования. С учетом этого фактора выбор языков программирования сужается до четырех официально поддерживаемых компанией Microsoft и имеющих изначальную поддержку в Visual Studio 2015: Visual C++/CLI, C#, Visual Basic .NET и F#. Необходимость использования низкоуровневых возможностей Visual C++/CLI в разрабатываемом ПО отсутствует, следовательно данный язык можно исключить из списка кандидатов. Visual Basic .NET уступает по удобству использования двум другим кандидатам из нашего списка. Оставшиеся два языка программирования C# и F# являются первостепенным, элегантными, мультипарадигменными языками программирования для платформы .NET. Для платформы .NET существуют библиотеки, которые позволяют упростить работу с системами построенными на событийной модели. В данном случая я буду использовать инструменты разработанные в Microsoft Research — Reactive Extensions. В отдельной главе будут описаны основные функции, которые берет на себя данная библиотека. Таким образом, с учетом вышеперечисленных факторов, целесообразно остановить выбор на следующих технологиях:

- операционная система Windows 10;
- платформа разработки .NET;
- язык программирования C#;
- библиотека для работы с событиями — Reactive Extensions.

Поддержка платформой .NET различных языков программирования позволяет использовать язык, который наиболее просто позволяет решить возникающую задачу. Разрабатываемое программное обеспечение в некоторой степени использует данное преимущество платформы. Далее проводится характеристика используемых технологий и языков программирования более подробно.

1 Программная платформа .NET

Программная платформа .NET является одной из реализаций стандарта ECMA-335 [9] и является современным инструментом создания клиентских и серверных приложений для операционной системы Windows. Платформа .NET — это один из компонентов системы Windows. Он позволяет создавать и использовать приложения нового поколения. Назначение платформы .NET:

– создание целостной объектно-ориентированной среды программирования допускающей различные варианты реализации: код может храниться и выполняться локально; выполняться локально, а распространяться через Интернет; или выполняться удаленно;

– предоставление среды выполнения кода, в которой число конфликтов при развертывании программного обеспечения и управлении версиями будет сведено к минимуму;

– обеспечение безопасности выполнения кода в среде - в том числе кода, созданного неизвестным разработчиком или разработчиком с частичным доверием;

– предоставление среды выполнения кода, позволяющей устранить проблемы, связанные с производительностью сред на основе сценариев или интерпретации;

– унификация работы разработчиков в совершенно разных приложениях: как в приложениях Windows, так и в веб-приложениях;

– использование промышленных стандартов во всех областях обмена данными и, как следствие, обеспечения совместимости кода, созданного в .NET Framework, с другими программами.

Первая общедоступная версия .NET Framework вышла в феврале 2002 года. С тех пор платформа активно эволюционировала и на данный момент было выпущено десять версии данного продукта. На данный момент номер последней версии .NET Framework — 4.5.3. Платформа .NET была призвана решить некоторые наболевшие проблемы, скопившиеся на момент её выхода, в средствах разработки приложений под Windows. Ниже перечислены некоторые из них [10, с. XIV – XVII]:

– сложность создания надежных приложений;

– сложность развертывания и управления версиями приложений и библиотек;

– сложность создания переносимого ПО;

– отсутствие единой целевой платформы для создателей компиляторов;

– проблемы с безопасным исполнением непроверенного кода;

– великое множество различных технологий и языков программирова-

ния, которые не совместимы между собой.

Многие из этих проблем были решены. Далее более подробно рассматривается внутреннее устройство .NET.

Основными составляющими компонентами .NET являются общая языковая исполняющая среда (Common Language Runtime) и стандартная библиотека классов (Framework Class Library). CLR представляет из себя виртуальную машину и набор сервисов обслуживающих исполнение программ написанных для .NET. Ниже приводится перечень задач, возлагаемых на CLR [11]:

– загрузка и исполнение управляемого кода;
– управление памятью при размещении объектов;
– изоляция памяти приложений;
– проверка безопасности кода;
– преобразование промежуточного языка в машинный код;
– доступ к расширенной информации от типах — метаданным;
– обработка исключений, включая межъязыковые исключения;
– взаимодействие между управляемым и неуправляемым кодом (в том числе и COM-объектами);
– поддержка сервисов для разработки (профилирование, отладка и т. д.).

Программы написанные для .NET представляют из себя набор типов взаимодействующих между собой. .NET имеет общую систему типов (Common Type System, CTS). Данная спецификация описывает определения и поведение типов создаваемых для .NET [12]. В частности в данной спецификации описаны возможные члены типов, механизмы сокрытия реализации, правила наследования, типы-значения и ссылочные типы, особенности параметрического полиморфизма и другие возможности предоставляемые CLI. Общая языковая спецификация (Common Language Specification, CLS) — подмножество общей системы типов. Это набор конструкций и ограничений, которые являются руководством для создателей библиотек и компиляторов в среде .NET Framework. Библиотеки, построенные в соответствии с CLS, могут быть использованы из любого языка программирования, поддерживающего CLS. Языки, соответствующие CLS (к их числу относятся языки C#, Visual Basic .NET, Visual C++/CLI), могут интегрироваться друг с другом. CLS — это основа межъязыкового взаимодействия в рамках платформы .NET [11].

Некоторые из возможностей, предоставляемых .NET: верификация кода, расширенная информация о типах во время исполнения, сборка мусора, безопасность типов, — невозможны без наличия подробных метаданных о типах из которых состоит исполняемая программа. Подробные метаданные о типах генерируются компиляторами и сохраняются в результирующих сборках. Сборка — это логическая группировка одного или нескольких управляемых

модулей или файлов ресурсов, является минимальной единицей с точки зрения повторного использования, безопасности и управлениями версиями [12, с. 6].

Одной из особенностей .NET, обеспечивающей переносимость программ без необходимости повторной компиляции, является представление исполняемого кода приложений на общем промежуточном языке (Common Intermediate Language, CIL). Промежуточный язык является бестиповым, стековым, объекто-ориентированным ассемблером [12, с. 16 – 17]. Данный язык очень удобен в качестве целевого языка для создателей компиляторов и средств автоматической проверки кода для платформы .NET, также язык довольно удобен для чтения людьми. Наличие промежуточного языка и необходимость создания производительных программ подразумевают наличие преобразования промежуточного кода в машинный код во время исполнения программы. Одним из компонентов общей языковой исполняющей среды, выполняющим данное преобразование, является компилятор времени исполнения (Just-in-time compiler) транслирующий промежуточный язык в машинные инструкции, специфические для архитектуры компьютера на котором исполняется программа.

Ручное управление памятью всегда являлось очень кропотливой и подверженной ошибкам работой. Ошибки в управлении памятью являются одними из наиболее сложных в устранении типами программных ошибок, также эти ошибки обычно приводят к непредсказуемому поведению программы, поэтому в .NET управление памятью происходит автоматически [12, с. 505 – 506]. Автоматическое управление памятью является механизмом поддержания иллюзии бесконечности памяти. Когда объект данных перестает быть нужным, занятая под него память автоматически освобождается и используется для построения новых объектов данных. Имеются различные методы реализации такого автоматического распределения памяти [13, с. 489]. В .NET для автоматического управления памятью используется механизм сборки мусора (garbage collection). Существуют различные алгоритмы сборки мусора со своими достоинствами и недостатками. В .NET используется алгоритм пометок (mark and sweep) в сочетании с различными оптимизациями, такими как, например, разбиение всех объектов по поколениям и использование различных куч для больших и малых объектов.

Ниже перечислены, без приведения подробностей, некоторые важные функции исполняемые общей языковой исполняющей средой:

– обеспечение многопоточного исполнения программы;
– поддержание модели памяти, принятой в CLR;
– поддержка двоичной сериализации;
– управление вводом и выводом;

– структурная обработка исключений;

– возможность размещения исполняющей среды внутри других процессов.

Как уже упоминалось выше, большую ценностью для .NET представляет библиотека стандартных классов — соответствующая CLS-спецификации объектно-ориентированная библиотека классов, интерфейсов и системы типов (типов-значений), которые включаются в состав платформы .NET. Эта библиотека обеспечивает доступ к функциональным возможностям системы и предназначена служить основой при разработке .NET-приложений, компонент, элементов управления [11].

2 Язык программирования C#

C# — объектно-ориентированный, типо-безопасный язык программирования общего назначения. Язык создавался с целью повысить продуктивность программистов. Для достижения этой цели в языке гармонично сочетаются простота, выразительность и производительность промежуточного кода, получаемого после компиляции. Главным архитектором и идеологом языка с первой версии является Андрес Хейлсберг (создатель Turbo Pascal и архитектор Delphi). Язык C# является платформенно нейтральным, но создавался для хорошей работы с .NET [14].

Язык имеет богатую поддержку парадигмы объекто-ориентированного программирования, включающую поддержку инкапсуляции, наследования и полиморфизма. Отличительными чертами C# с точки зрения ОО парадигмы являются:

– Унифицированная система типов. В C# сущность, содержащая данные и методы их обработки, называется типом. В C# все типы, являются ли они пользовательскими типами, или примитивами, такими как число, производны от одного базового класса.

– Классы и интерфейсы. В классической объекто-ориентированной парадигме существуют только классы. В C# дополнительно существуют и другие типы, например, интерфейсы. Интерфейс — это сущность напоминающая классы, но содержащая только определения членов. Конкретная реализация указанных членов интерфейса происходит в типах, реализующих данный интерфейс. В частности интерфейсы могут быть использованы при необходимости проведения множественного наследования (в отличие от языков C++ и Eiffel, C# не поддерживает множественное наследование классов).

– Свойства, методы и события. В чистой объекто-ориентированной парадигме все функции являются методами. В C# методы являются лишь одной

из возможных разновидностей членов типа, в C# типы также могут содержать свойства, события и другие члены. Свойство — это такая разновидность функций, которая инкапсулирует часть состояния объекта. Событие — это разновидность функций, которые реагируют на изменение состояния объекта [14].

В большинстве случаев C# обеспечивает безопасность типов в том смысле, что компилятор контролирует чтобы взаимодействие с экземпляром типа происходило согласно контракту, который он определяют. Например, компилятор C# не скомпилирует код который обращается со строками, как если бы они были целыми числами. Говоря более точно, C# поддерживает статическую типизацию, в том смысле что большинство ошибок типов обнаруживаются на стадии компиляции. За соблюдение более строгих правил безопасности типов следит исполняющая среда. Статическая типизация позволяет избавиться от широкого круга ошибок, возникающих из-за ошибок типов. Она делает написание и изменение программ более предсказуемыми и надежными, кроме того, статическая типизация позволяет существовать таким средствам как автоматическое дополнение кода и его предсказуемый статический анализ. Еще одним аспектом типизации в C# является её строгость. Строгая типизация означает, что правила типизации в языке очень «сильные». Например, язык не позволяет совершать вызов метода, принимающего целые числа, передавая в него вещественное число [14]. Такие требования спасают от некоторых ошибок.

C# полагается на автоматическое управление памятью со стороны исполняющей среды, предоставляя совсем немного средств для управления жизненным циклом объектов. Не смотря на это, в языке все же присутствует поддержка работы с указателями. Данная возможность предусмотрена для случаев, когда критически важна производительность приложения или необходимо обеспечить взаимодействие с неуправляемым кодом [14].

Как уже упоминалось C# не является платформенно зависимым языком. Благодаря усилиям компании Xamarin возможно писать программы на языке C# не только для операционных систем Microsoft, но и ряда других ОС. Существуют инструменты создания приложений на C# для серверных и мобильных платформ, например: iOS, Android, Linux и других.

Создатели языка C# не являются противниками привнесения в язык новых идей и возможностей, в отличии от создателей одного из конкурирующих языков. Каждая новая версия компилятора языка привносит различные полезные возможности, которые отчаются требованиям индустрии. Далее приводится краткий обзор развития языка.

Первая версия C# была похожа по своим возможностям на Java 1.4, несколько их расширяя: так, в C# имелись свойства (выглядящие в коде как

поля объекта, но на деле вызывающие при обращении к ним методы класса), индексаторы (подобные свойствам, но принимающие параметр как индекс массива), события, делегаты, циклы foreach, структуры, передаваемые по значению, автоматическое преобразование встроенных типов в объекты при необходимости (boxing), атрибуты, встроенные средства взаимодействия с неуправляемым кодом (DLL, COM) и прочее [15].

Версия .NET 2.0 привнесла много новых возможностей в сравнении с предыдущей версией, что отразилось и на языках под эту платформу. Проект спецификации C# 2.0 впервые был опубликован Microsoft в октябре 2003 года; в 2004 году выходили бета-версии (проект с кодовым названием Whidbey), C# 2.0 окончательно вышел 7 ноября 2005 года вместе с Visual Studio 2005 и .NET 2.0. Ниже перечислены новые возможности в версии 2.0

– Частичные типы (разделение реализации класса более чем на один файл).

– Обобщённые, или параметризованные типы (generics). В отличие от шаблонов C++, они поддерживают некоторые дополнительные возможности и работают на уровне виртуальной машины. Вместе с тем, параметрами обобщённого типа не могут быть выражения, они не могут быть полностью или частично специализированы, не поддерживают шаблонных параметров по умолчанию, от шаблонного параметра нельзя наследоваться.

– Новая форма итератора, позволяющая создавать сопрограммы с помощью ключевого слова **yield**, подобно Python и Ruby.

– Анонимные методы, обеспечивающие функциональность замыканий.

– Оператор ??: return obj1 ?? obj2; означает (в нотации C# 1.0) `return obj1!=null ? obj1 : obj2;`.

– Обнуляемые (nullable) типы-значения (обозначаемые вопросительным знаком, например, `int? i = null;`), представляющие собой те же самые типы-значения, способные принимать также значение null. Такие типы позволяют улучшить взаимодействие с базами данных через язык SQL.

– Поддержка 64-разрядных вычислений позволяет увеличить адресное пространство и использовать 64-разрядные примитивные типы данных [15].

Третья версия языка имела одно большое нововведение — Language Integrated Query (LINQ), для реализации которого в языке дополнительно появилось множество дополнительных возможностей. Ниже приведены некоторые из них:

– Ключевые слова **select**, **from**, **where**, позволяющие делать запросы из SQL, XML, коллекций и т. п.

– Инициализацию объекта вместе с его свойствами:

```
Customer c = new Customer(); c.Name = "James"; c.Age=30;
```

можно записать как

```
Customer c = new Customer { Name = "James", Age = 30 };
```

– Лямбда-выражения:

```
listOfFoo.Where(delegate(Foo x) { return x.size > 10; });
```

теперь можно записать как

```
listOfFoo.Where(x => x.size > 10);
```

– Деревья выражений — лямбда-выражения теперь могут быть представлены в виде структуры данных, доступной для обхода во время выполнения, тем самым позволяя транслировать строго типизированные C#-выражения в другие домены (например, выражения SQL).

– Вывод типов локальной переменной: `var x = "hello";` вместо `string x = "hello";`

– Безымянные типы: `var x = new { Name = "James"};`

– Методы-расширения — добавление метода в существующий класс с помощью ключевого слова **this** при первом параметре статической функции.

– Автоматические свойства: компилятор сгенерирует закрытое поле и соответствующие аксессор и мутатор для кода вида

```
public string Name { get; private set; }
```

C# 3.0 совместим с C# 2.0 по генерируемому MSIL-коду; улучшения в языке — чисто синтаксические и реализуются на этапе компиляции [15].

Visual Basic .NET 10.0 и C# 4.0 были выпущены в апреле 2010 года, одновременно с выпуском Visual Studio 2010. Новые возможности в версии 4.0:

– Возможность использования позднего связывания.

– Именованные и опциональные параметры.

– Новые возможности COM interop.

– Ковариантность и контрвариантность интерфейсов и делегатов.

– Контракты в коде (Code Contracts) [15].

В C# 5.0 было немного нововведений, но они носят большую практическую ценность. В новой версии появилась упрощенная поддержка выполнения асинхронных функций с помощью двух новых слов — **async** и **await**. Ключевым словом **async** помечаются методы и лямбда-выражения, которые внутри содержат ожидание выполнения асинхронных операций с помощью оператора **await**, который отвечает за преобразования кода метода во время компиляции.

На данный момент актуальная версия C# 6.0. Новые возможности в версии 6.0 [16]:

– инициализация свойств со значениями:

```
public int Value { get; set; } = 100500;
```

– интерполяция строк:

```
obj.FirstName = "Anatoly";
obj.LastName = "Safonov";
name = $"First name {obj.FirstName}, last name {obj.LastName}"
   ;
```

– свойства и методы можно определять через лямбда-выражения:

```
public string[] GetCountryList() => new string[] { "Russia", "
   Belarus", "Poland" };
```

– импорт статических классов:

```
using System.Math;
double powerValue = Pow(2, 3);
double roundedValue = Round(10.6);
```

Это можно использовать не только внутри класса, но и при выполнении метода;

– null-условный оператор:

```
string location = obj?.Location;
```

– nameof оператор:

```
Console.WriteLine(emp.Location);
Console.WriteLine(nameof(Employee.Location));
```

– await в catch и finally блоках:

```
public async Task StartAnalyzingData()
{
  try
  {
    // code
  }
  catch
  {
    await LogExceptionDetailsAsync();
  }
  finally
  {
    await CloseResourcesAsync();
  }
}
```

– фильтры исключений:

```
try
{
  //throw exception
```

```
}
catch (ArgumentNullException ex) if (ex.Source == "
   EmployeeCreation")
{
  //catch block
}
catch (InvalidOperationException ex) if (ex.InnerException !=
   null)
{
  //catch block
}
catch (Exception ex) if (ex.InnerException != null)
{
  //catch block
}
```

– инициализация Dictionary:

```
var country = new Dictionary<int, string>
{
  [0] = "Russia",
  [1] = "USA",
  [2] = "UK",
  [3] = "Japan"
};
```

Последняя версия языка доступна в Visual Studio 2015. Microsoft выпустила предварительную версию Visual studio 2015 и .Net 4.6 для разработчиков, в которой можно изучить нововведения. Существует специальная редакция — Community Edition. Её можно использовать для обучения бесплатно, сущетвуют только ограничения для коммерческих проектов.

3 Reactive Extensions

Реактивные расширения — звучит настолько круто, что напрашивается связь с реактивными самолетами, но как мы уже знаем, речь идет о реактивном программировании. Reactive происходит от слова react (реагировать), подразумевается, что система реагирует на изменения состояния.

Как правило, мы пишем код, в котором есть методы и функции, которые мы вызываем, получаем результат и его обрабатываем. Reactive Extensions в свою очередь позволяет создавать события и обработчики, которые будут реагировать на них. Таким образом, система будет состоять из последовательности событий, которые будут сообщать об изменении состояния и должным образом реагировать на них.

Rx состоит из двух базовых абстракций в пространстве имен System

начиная с .NET 4.0, а именно `System.IObserver` и `System.IObservable`. Как видно из названия, это реализация паттерна «наблюдатель» (Observer). В данной реализации `IObservable` выступает как субъект, и очевидно, что `IObserver` это наблюдатель, который может подписываться на изменения. В .NET уже есть реализация наблюдателя в виде событий (Events). Как уже упоминалось, Rx позволяет создавать последовательность событий, и само собой разумеется, что это можно сделать с помощью ивентов. Способы работы с Reactive Extensions и Events отличаются, но об этом немного позже.

IObserver — предоставляет механизм получения уведомлений. Интерфейс объявляет три метода:

– **void** `OnNext(T value)` — предоставляет следующий элемент в последовательности.

– **void** `OnError(Exception ex)` — позволяет передать `Exception` и адекватно его обработать. Подразумевается, что после этого сообщения последовательность заканчивается и наблюдателям больше не нужно следить за изменениями.

– **void** `OnCompleated()` — сообщается, что последовательность закончилась и больше не будет новых сообщений, не нужно их ожидать.

IObservable — производит уведомления и позволяет подписываться наблюдателям. Объявляет один метод:

`IDisposable Subscribe(IObserver observer)` — принимает объект наблюдателя (`IObserver`) параметром и подписывает его на сообщения. Обратите внимание, что метод возвращает `IDisposable`, с помощью чего можно потом вызывать метод `Dispose`, тем самым отписав и уничтожив наблюдателя.

Если мы захотим реализовать `IObservable`, то нужно будет помимо метода `Subscribe` также реализовать логику, которая может отправлять новые сообщения, ошибки или сообщать об окончании последовательности. Получается, что также нужно будет реализовать интерфейс `IObservable`, для таких целей можно использовать тип `Subject`. Но чтобы его использовать, нужно будет с Nuget установить дополнительную библиотеку (Install-Package Rx-Mail), которая также предоставляет дополнительные расширения и возможность использовать LINQ.

Листинг 2.1 — Пример использования Subject

```
using System;
using System.Reactive.Subjects;

namespace Demo
{
  class Program
  {
```

```
    static Subject<int> sub = new Subject<int>();//Declare
    static void Main()
    {
      sub.Subscribe(Console.WriteLine); //Subscribe

      sub.OnNext(234); //Publish
    }
  }
}
```

В этом листинге 2.1 создается новая последовательность, то есть `Subject` (также можно назвать последовательность объектов типа **int**), затем на нее подписывается наблюдатель (в данном случае просто выводится в консоль каждое значение последовательности), и передается значение, которое выводится в консоль с помощью наблюдателя. Каждый раз, когда подписывается новый наблюдатель, ему начинают поставляться элементы последовательности. Но есть еще несколько реализаций с другим поведением:

- ReplaySubject;
- BehaviourSubject;
- AsyncSubject;

Рассмотрим эти типы подробнее.

3.1 ReplaySubject. ReplaySubject — поставляет все элементы последовательности независимо от того, когда был подписан наблюдатель.

Листинг 2.2 — Пример использования ReplaySubject

```
using System;
using System.Reactive.Subjects;

namespace Demo
{
  class Program
  {
    static ReplaySubject<int> sub = new ReplaySubject<int>();

    static void Main()
    {
      sub.OnNext(222);

      sub.Subscribe(Console.WriteLine);

      sub.OnNext(354);
    }
  }
}
```

3.2 BehaviorSubject. BehaviorSubject — не может быть пустым, всегда содержит в себе элемент, но только последний.

Листинг 2.3 — Пример использования BehaviorSubject

```
using System;
using System.Reactive.Subjects;

namespace DemoData
{
  class Program
  {
    static BehaviorSubject<int> sub = new BehaviorSubject<int>(666);

    static void Main()
    {
      sub.OnNext(222);

      sub.Subscribe(Console.WriteLine); // 222
    }
  }
}
```

3.3 AsyncSubject. AsyncSubject — также возвращает только последнее значение, но, в отличие от остальных реализаций, данные будут публиковаться при вызове `OnCompleated`.

Листинг 2.4 — Пример использования AsyncSubject

```
using System;
using System.Reactive.Subjects;

namespace DemoData
{
  class Program
  {
    static AsyncSubject<int> sub = new AsyncSubject<int>();

    static void Main(string[] args)
    {
      sub.OnNext(222);

      sub.Subscribe(Console.WriteLine);

      sub.OnCompleted(); // Publish 222
    }
  }
}
```

Теперь сравним с стандартными Event'ами, вот как выглядел бы код:

Листинг 2.5 — Стандартная модель Event

```
using System;
namespace Demo
{
  class Program
  {
    static event Action<int> Ev; //Declare

    static void Main(string[] args)
    {
      Ev += Console.WriteLine; //Subscribe

      Ev(234); //Publish
    }
  }
}
```

Все предельно просто, выполнение будет проходить так же, но в Reactive Extensions есть ряд преимуществ перед ивентами:

Реализация `IObservable` — это классы, в которых можно делать все, что хочешь. Методы, которые объявляет `IObserver`, позволяют более корректно управлять последовательностью. Можно сообщить, что последовательность закончилась и тем самым сделать последние нужные действия и отписаться. Есть возможность управлять ошибками. В ивентах чтобы отписаться, нужно сохранить наблюдателя в какой-то переменной и как-то ими управлять. В Reactive Extensions метод `Subscribe` возвращает `IDisposable` и ему можно просто вызвать `Dispose()`, чтобы отписаться.

Листинг 2.6 — Отписка от события

```
var toDispose = sub.Subscribe(Console.WriteLine);
toDispose.Dispose();
```

Reactive Extensions содержит множество полезные инструментов для работы с событиями. Все методы и расширения схожи по синтаксису с LINQ.

Изначально LINQ позволял делать запросы к статическим источникам данных. Но так как количество данных растет, а подходы меняются, то нужно к этому приспосабливаться. Rx позволяет выполнять запросы к динамическим последовательностям.

Листинг 2.7 — Пример использования LINQ

```
using System.Reactive.Linq;

namespace Demo
```

```
{
  class Program
  {
    static void Main()
    {
      var sequence = Observable.Range(1, 10, Scheduler.Default); //
          create sequence
      var query = from s in sequence
            where s % 2 == 0
            select s; // create query
      sequence.Subscribe(Console.WriteLine); // subscribe
      query.Subscribe(Console.WriteLine); // subscribe
    }
  }
}
```

В листинге 2.7 сначала создается последовательность, которая предоставляет данные типа **int** от 1 до 10, затем к ней применяется LINQ-выражение, которое выбирает из последовательности только значения, кратные 2. Таким образом, получается две разных последовательности, на которые можно подписать разных наблюдателей. Это крайне простой пример, но набор классов Reactive Extensions предоставляет очень много методов, которые дают огромную гибкость.

Reactive Extensions позволяет создавать отдельные модули, которые будут следить за состоянием системы и реагировать на него. Каждая часть системы будет полностью независима, так как она не знает ничего об остальных модулях. Наблюдатели ожидают изменения последовательности, а ей, в свою очередь, все равно, кто наблюдает за ее изменениями. Тем самым достигается связанность модулей. Reactive Extensions имеет смысл применять для обработки UI-событий, доменных событий, изменений окружающей среды, изменений на сторонних сервисах (RSS, Twitter и т.д.). Rx также предоставляет возможность преобразовывать события в `IObservable`, что позволяет интегрироваться в систему.

ГЛАВА 3
ПОСТАНОВКА ЗАДАЧИ

Быстрая и лекго конструируемая обработка данных — одна из задач, решаемых с использованием реактивного и функционального.

Исследования и разработки, проводимые в рамках диссертации, являются основой для анализа и проектирования новых способов обработки данных. Данная работа предназначена использования в реальных проектах, в которых существует множество взаимодействий между модулями или пользователями. Инструмент, полученный в ходе работы над диссертацией, предназначен для использования в модели бизнес-логики прикладных и серверных программ для связи данных и их представления.

1 Постановка задачи

В рамках диссертации поставлена цель построить систему для обработки наборов данных. Данные представляются в виде коллекций и списков, которые обладают возможностью генерировать события о изменении этих данных. Обработка должна осуществлятся с помощью операций схожих с операциями реляционной алгебры. Должна быть построена абстрактная модель системы и реализована на одной из выбранных платформ. Все реализованные операции следует протестировать на предмет ошибок. Исследовать пути для оптимизации полученного решения. Для решения поставленной задачи необходимо:

- создать абстрактную модель исходных данных;
- создать протокол взаимодействия между частями системы;
- создать базовые модели для операций над данными;
- реализовать библиотеку согласно интерфейсам.

2 Определение границ исследования

В рамках диссертации необходимо получить библиотеку для обработки наборов данных, основанную на событийной модели. Для этого необходимо:

- построить абстрактную модель системы;
- определить интерфейс библиотеки.

Приложение будет работать с несколькими наборами исходных данных и реагировать на действия пользователя. После разработки прототипа необходимо подготовить подробное описание его структуры и сделать выводы о его эффективности. На основе проделанной работы сделать заключение о возможности использования в реальных приложениях.

ГЛАВА 4
РАЗРАБОТКА МОДЕЛИ ОБРАБОТКИ

Начать разработку следует с модели предствления динамических данных. Модель должна соответсвовать следующим критериям:

– иметь возможность предоставить данные;

– сообщить, когда они изменятся.

В .NET уже существует интейфейсы, которые предоставляют данную функциональность:

– `IReadOnlyCollection` — предоставляет доступ для чтения набора данных [17];

– `IReadOnlyList` — предоставляет доступ для чтения набора данных, наследуется от `IReadOnlyCollection` и предоставляет доступ по индексу [18];

– `INotifyCollectionChanged` — содержит событие, подписавшись на которое, можно следить за изменением набора данных [19].

В моей модели я использую только первых два интерфейса и два интерфейса, которые заменяют `INotifyCollectionChanged`. Я использую свой интерфейс, потому что `INotifyCollectionChanged` имеет ряд недостатков:

– не может сообщить о изменении коллекции без индексов;

– реализовано через стандартную модель Event — это вызвало бы нагромождение однообразного кода, при реализации операций обработки данных;

– со стандартными Event нельзя строить LINQ выражения для работы с событиями по принципам функционального программирования.

Далее я опишу получившуюся модель для динамических наборов данных.

1 Модели событий обновления наборов данных

Для замены `INotifyCollectionChanged` в моей модели существуют два интефейса: `INotifyCollectionChanged<out T>` и `INotifyListChanged<out T>`. Как можно заметить, первый служит для обычных коллекций, а второй для списков.

Рассмотрим их подробнее. Свойство генератора событий обновления коллекции `CollectionChanged` имеет тип `IObservable`. Если подписаться на это событие, то получатель будет иметь в своем распоряжении объекты типа `IUpdateCollectionQuery`, который является алгебраическим типом данных и представляет тип-сумму [20]. `IUpdateCollectionQuery` — является объектом запроса на обновление коллекции. Он является суммой данных типов:

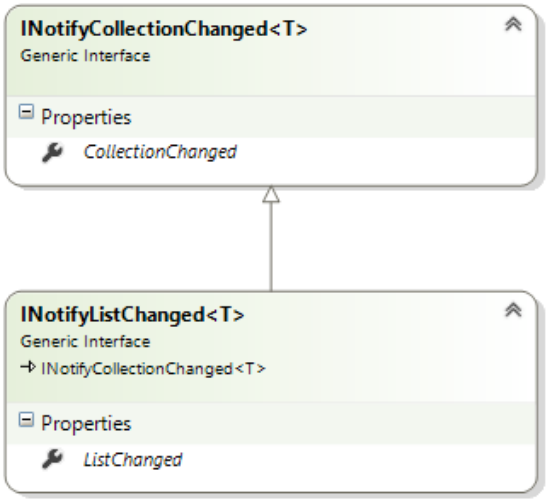

Рисунок *4.1* — Интерфейсы генераторов событий для наборов данных

– `ICollectionOnInsertArgs<T>` — запрос на вставку одного элемента типа T в коллекцию. Содержит ссылку на добавляемый объект типа T;

– `ICollectionOnRemoveArgs<T>` — запрос на удаление одного элемента типа T из коллекции. Содержит ссылку на удаляемый объект типа T;

– `ICollectionOnReplaceArgs<T>` — запрос на замену одного элемента типа T, другим элементов типа T в коллекции. Содержит ссылки на старый и новый объекты типа T;

– `ICollectionOnResetArgs<T>` — замена всего содержимого коллекции. Содержит список исходных объектов коллекции и новый список объектов коллекции;

– `ICollectionOnEmptyArgs<T>` — пустой запрос. Ничего не содержит.

`ListChanged` имеет тип `IObservable<IUpdateListQuery>`. Если подписаться на это событие, то получатель будет иметь в своем распоряжении объекты типа `IUpdateListQuery`, который является алгебраическим типом данных и представляет тип-сумму [20]. `IUpdateListQuery` — является объектом запроса на обновление списка. Он является суммой данных типов:

– `IListOnInsertArgs<T>` — запрос на вставку одного элемента типа T в список. Содержит ссылку на добавляемый объект типа T и позицию для вставки;

– `IListOnRemoveArgs<T>` — запрос на удаление одного элемента типа T из списка. Содержит ссылку на удаляемый объект типа T и позицию удаления;

– `IListOnReplaceArgs<T>` — запрос на замену одного элемента типа T, другим элементов типа T в списке. Содержит ссылки на старый и новый объекты типа T и позицию исходного элемента;

– `IListOnMoveArgs<T>` — запрос на перемещение одного элемента типа

T. Содержит ссылку объект типа T, исходную и новую позиции;

– `IListOnResetArgs<T>` — замена всего содержимого списка. Содержит список исходных объектов списка и новый список объектов списка;

– `IListOnEmptyArgs<T>` — пустой запрос. Ничего не содержит.

На рисунке 4.2 видно как соотносятся между собой запросы. Запросы для списков подходят в качестве запросов для коллекций. Данная модель учитывает то, что каждый список по сути является коллекцией и наследует её признаки.

Рисунок *4.2* — Схема наследования запросов

Интерфейсы, которые объединяют генераторы событий и хранилища данных именуются `IObservableReadOnlyCollection` для коллекций, а также `IObservableReadOnlyList` для списков. Лист наследуется коллекции. В данной модели разработаны и реалицации этих коллекций и списков, которые

можно изменять. Эти реализации и будут отправной точкой событий в данной системе. В модели они выражены в интерфейсах:

– `IObservableCollection<T>` — наследует шаблонный интерфейс коллекции с разрешением только на чтение `IObservableReadOnlyCollection<T>` и редактируемой — `ICollection<T>`. Расширен методами Replace и Reset. Первый позволяет одиним событием заменять элемент в коллекции, а второй позволяет сбрасывать содержимое коллекции за одно событие;

– `IObservableList<T>` — наследует шаблонный интерфейс списка с разрешением только на чтение `IObservableReadOnlyList<T>`, с возможностью редактирования — `IList<T>` и `IObservableCollection<T>`. Расширен методами Replace, Reset и Move. Первые два, действуют аналогично методам из интерфейса коллекции, а метод Move позволяет изменять положение объекта за одно событие.

`IObservableCollection<T>` — наследует интерфейс `ITransactional`, который позволяет выполнять несколько операций подряд, не генерируя при этом событий. События сгенерируются только по завершению транзакции.

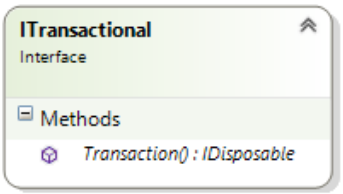

Рисунок *4.3* — Интерфейс для создания транзакций

Листинг 4.1 — Использование транзакции

```
IObservableCollection<int> collection = new ObservableCollection<
   int>();

using(collection.Transaction())
{
  collection.Add(5);
  collection.Add(2);
  collection.Add(1);
}
```

В листинге 4.1 транзакция создается и завершается в блоке using. Все операции что находятся внутри блока, сгенерируют события, когда у объекта транзакции выховется метод `Dispose`.

2 Модели операций над наборами данных

Операция — в контексте данной модели, это преобразование набора данных. Такое преобразование можно описать так: один набор данных по заданному правилу преоразуется в другой набор данных. Я выделил пять видов преобразований:

– коллекция объектов типа TIn преобразуется в коллекцию объектов типа TOut;

– коллекция объектов типа TIn преобразуется в упорядоченный список объектов типа TOut;

– список объектов типа TIn преобразуется в список объектов TOut;

– вычисление функции на основе коллекции объектов типа T;

– вычисление функции на основе списка объектов типа T.

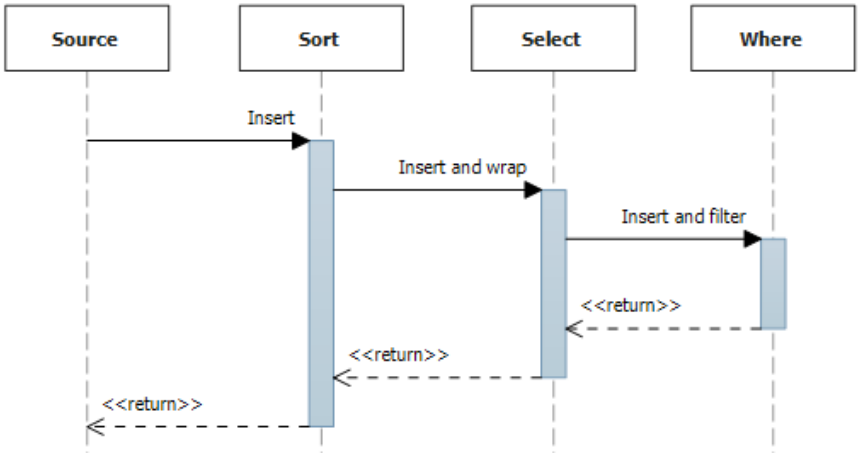

Рисунок *4.4* — Диаграмма взаимодействия наборов данных

Листинг 4.2 — Пример применения операций

```
var resultCollection = source.SortRc(...).SelectRl(...).WhereRl
   (...);
source.Add(number);
```

Операция описывается декларативно, и создает связь между исходным объектом и получившимся. Объект, который является реализацией конкретной операции, подписывыется на изменение исходного набора и, в соответсвии со своим назначением, изменяет зависимый набор. На рисунке 4.4 описан процесс взаимодействия источника(Source) и наборов, которые получились после применения операций. Код, который соответсвует рисунку 4.4, столь же простой. Его можно увидеть в листинге 4.2.

Для преобразорваний из одного типа набора в другой, было разработано три базовых класса операций:

– `CollectionToCollectionOperationBase<TIn, TOut>` — коллекция объектов типа TIn преобразуется в коллекцию объектов типа TOut;

– `CollectionToListOperationBase<TIn, TOut>` — коллекция объектов типа TIn преобразуется в упорядоченный список объектов типа TOut;

– `ListToListOperationBase<TIn, TOut>` — список объектов типа TIn преобразуется в список объектов TOut.

Для функций были разработаны следующий базовые классы:

– `CollectionFunctionBase<T>` — вычисление функции на основе коллекции объектов типа T;

– `ListFunctionBase<T>` — вычисление функции на основе списка объектов типа T.

Рассмотрим каждый из разработанных базовых реализаций, выделив ключевые детали.

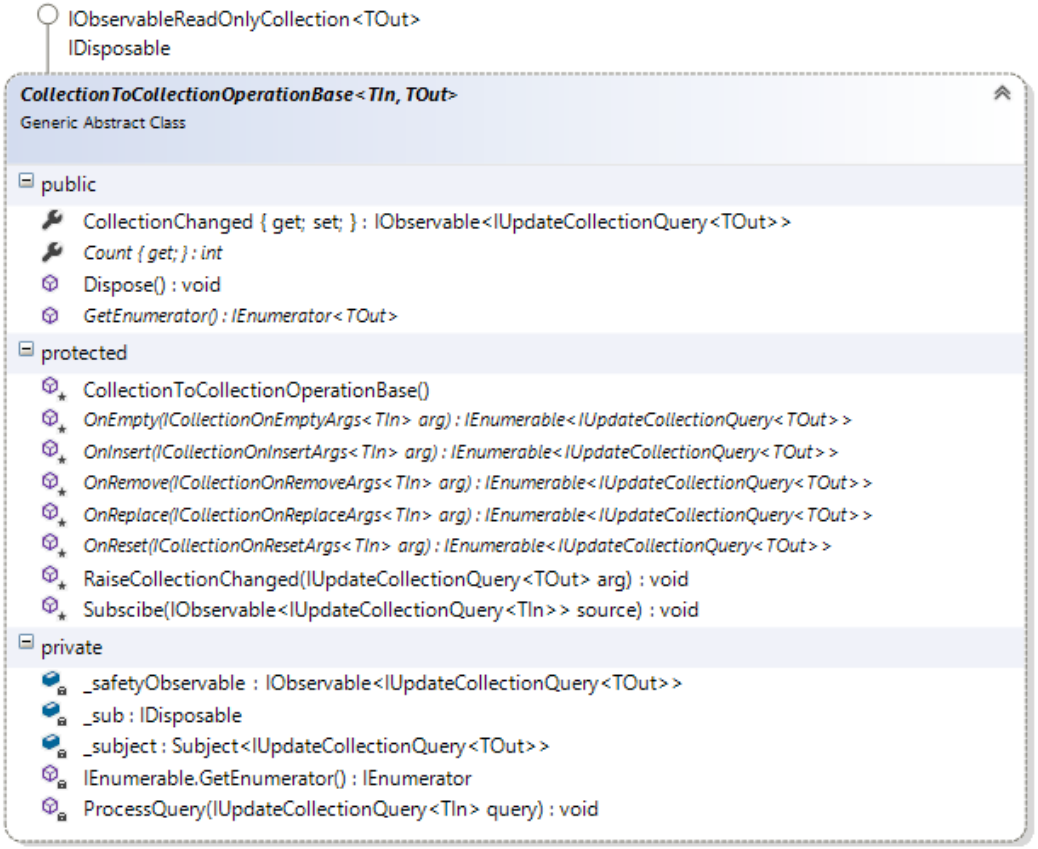

Рисунок *4.5* — Операция преобразования коллекции в коллекцию

`CollectionToCollectionOperationBase<TIn, TOut>` — наследуется от интерфейса `IObservableReadOnlyCollection<TOut>` и предназеначен для работы с генераторами событий `IObservable<IUpdateCollectionQuery<TIn>`(см. рисунок 4.5). Позволяет реализовать операции трансформации(Select), фильтрации(Where) и т. д. Например: преобразование увеличения целых чисел в 2

раза или фильтрация записей студентов по группе. Результат будет обновлятся вместе исходной коллекцией. Класс решает задачи:

– подписывается на источник событий с помощью Weak reference(слабой ссылки)[21];

– создает пустые реализации для всех существующих типов запросов; если запрос не был обработан, возникнет ошибка на этапе компиляции;

– реализует интерфейс `IObservableReadOnlyCollection<TOut>`;

– генерирует события для `IObservableReadOnlyCollection<TOut>`.

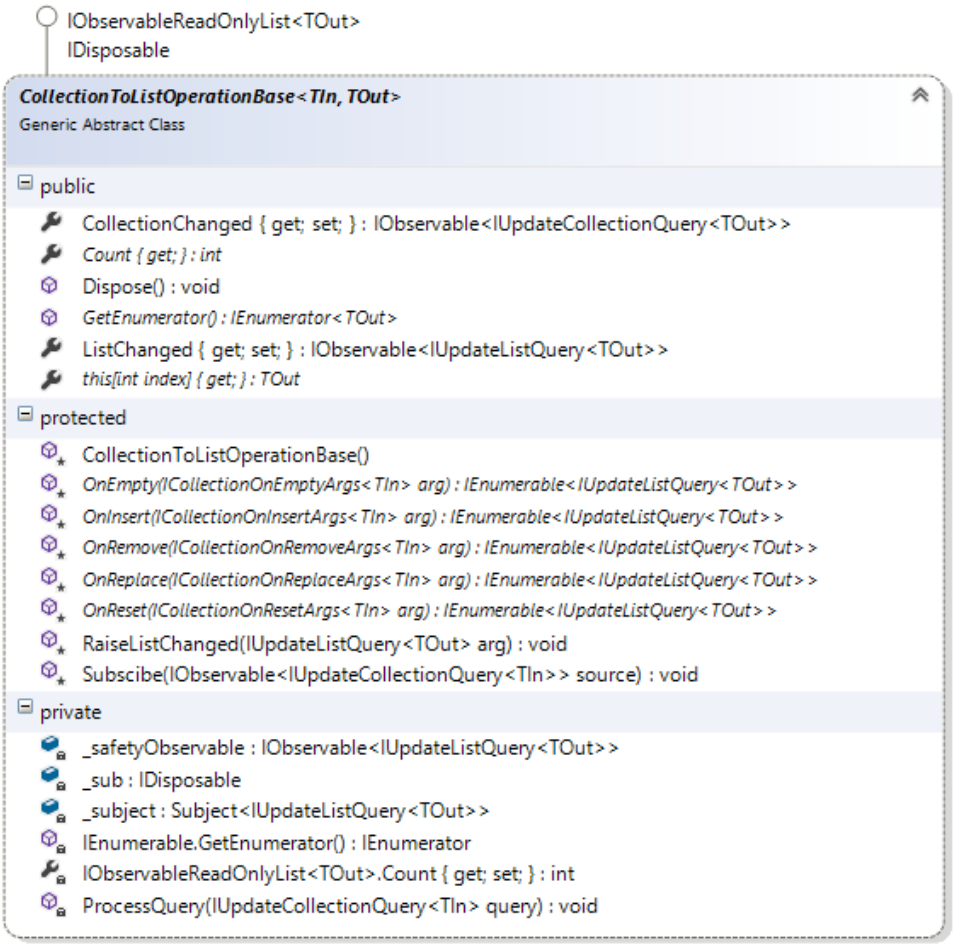

Рисунок *4.6* — Операция преобразования коллекции в список

`CollectionToListOperationBase<TIn, TOut>` — наследуется от базового интерфейса `IObservableReadOnlyList<TOut>` и предназеначен для работы с генераторами событий `IObservable<IUpdateCollectionQuery<TIn>`(см. рисунок 4.6). Позволяет реализовать операции по упорядочиванию элементов(Sort). Например: требуется сортировать заказы учитывая приоритет, который может измениться. Операция будет учитывать не только добавление и удаление элементов из коллекции, а еще и отслеживать изменения ключа сортировки. Класс решает задачи:

– подписывается на источник событий при помощи Weak reference;

– создает пустые реализации для всех существующих типов запросов; если запрос не был обработан, возникнет ошибка на этапе компиляции;

– реализует интерфейс `IObservableReadOnlyList<TOut>`;

– генерирует события для `IObservableReadOnlyList<TOut>`.

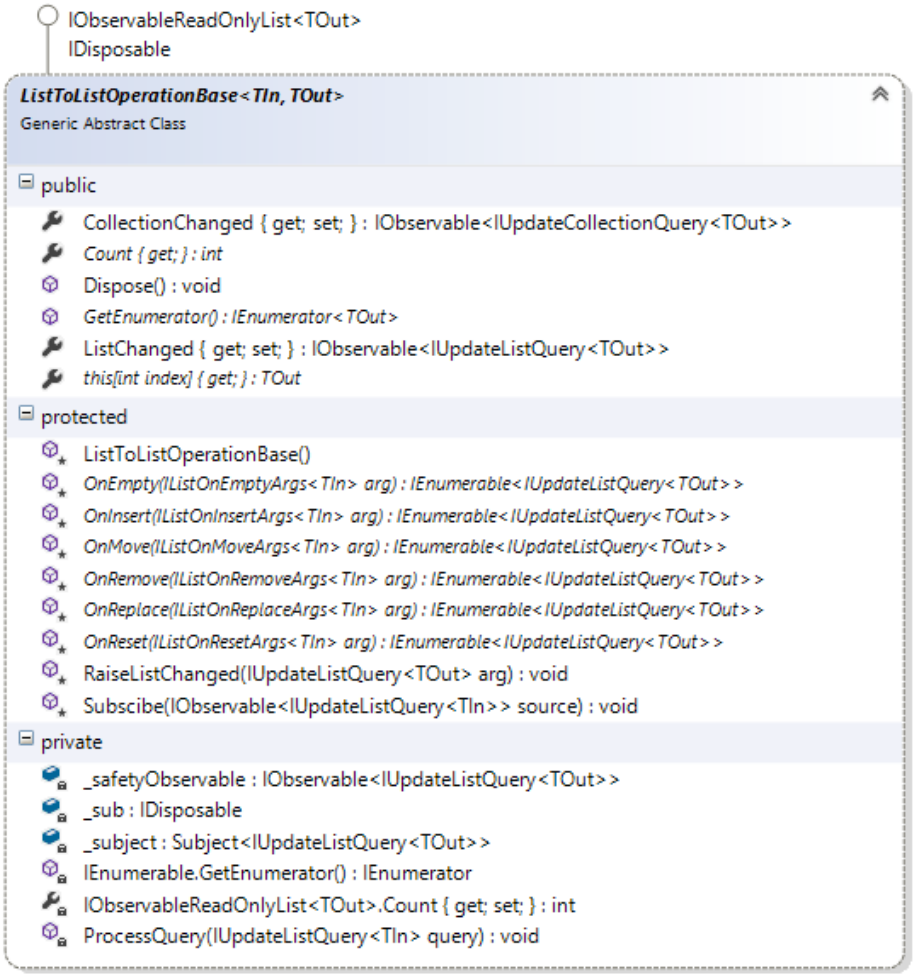

Рисунок *4.7* — Операция преобразования списка в список

`ListToListOperationBase<TIn, TOut>` — наследуется от базового интерфейса `IObservableReadOnlyList<TOut>` и предназеначен для работы с генераторами событий `IObservable<IUpdateListQuery<TIn>`(см. рисунок 4.7). Позволяет реализовать операции трансформации(Select), фильтрации(Where) и т. д. Может использоваться для переупорядочивания элементов(Sort, Take, Skip). Довольно распространённая задача реверсивного объода списка. Можно описать операцию Reverse, которая будет содержать объекты с обратной сортировкой и генерировать соответсвующие события. Класс решает задачи:

– подписывается на источник событий при помощи Weak reference;

– создает пустые реализации для всех существующих типов запросов; если запрос не был обработан, возникнет ошибка на этапе компиляции;

– реализует интерфейс `IObservableReadOnlyList<TOut>`;
– генерирует события для `IObservableReadOnlyList<TOut>`.

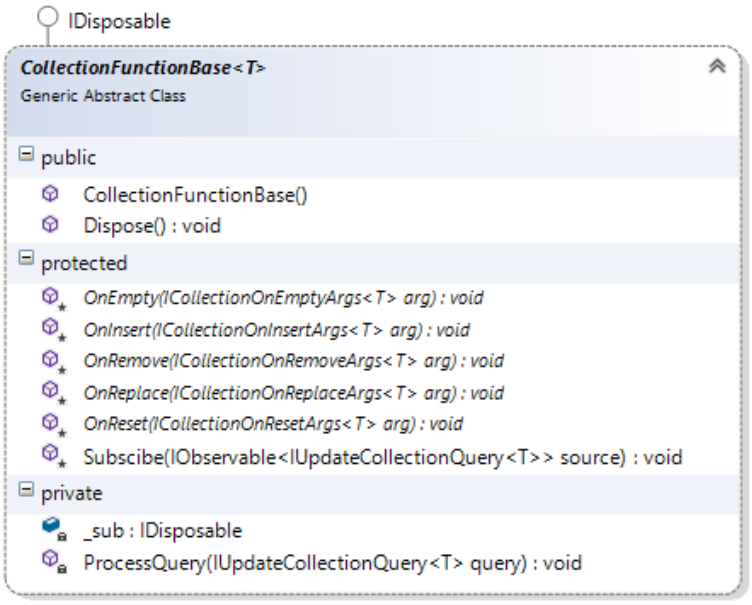

Рисунок *4.8* — Функция для коллекции

`CollectionFunctionBase<T>` — предназеначен для работы с генераторами событий `IObservable<IUpdateCollectionQuery<TIn>`(см. рисунок 4.8). Позволяет реализовать функции подсчета(Count), поиска минимума/максимума(Min/Max) и т. д. Можно реализовать функцию, которая будет возвращать объект с изменяющимся числом в соответствии с количеством контактов в адресной книге. При добавлении контактов, число автоматически будет увеличиваться, при замене контакта число останется неизменным. Класс решает задачи:

– подписывается на источник событий при помощи Weak reference;
– создает пустые реализации для всех существующих типов запросов; если запрос не был обработан, возникнет ошибка на этапе компиляции.

`ListFunctionBase<T>` — предназеначен для работы с генераторами событий `IObservable<IUpdateListQuery<TIn>`(см. рисунок 4.9). Позволяет реализовать функции подсчета(Count), поиска минимума/максимума(Min/Max) и т. д. Например: посчитать количесво чётных числел на нечётных местах или какая-либо другая функция, зависящая от индексов. Можно реализовать функцию, которая будет учитывать индексы при вычислении. Класс решает задачи:

– подписывается на источник событий при помощи Weak reference;
– создает пустые реализации для всех существующих типов запросов; если запрос не был обработан, возникнет ошибка на этапе компиляции.

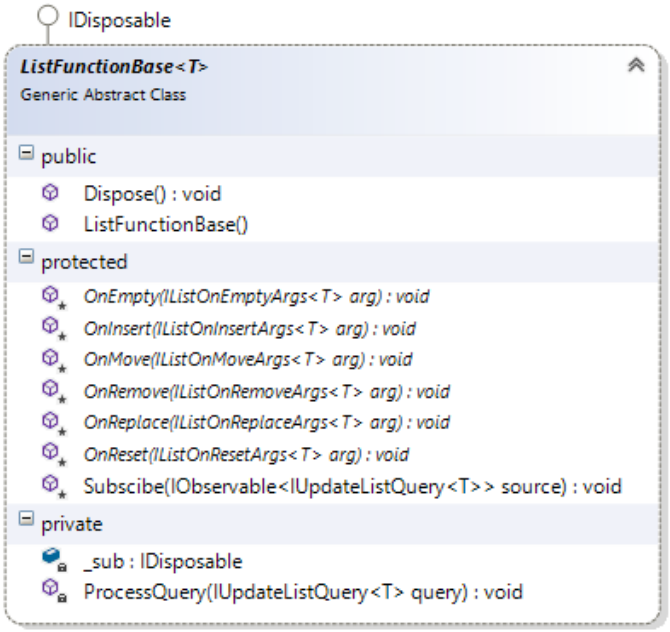

Рисунок *4.9* — Функция для списка

Создание данных абстрактных типов позволяет сокрыть детали реализации событийной модели. Это позволит оградить разработчика от реализации собственных способов обработки событий для наборов данных. Потребуется только переопределить конкретные базовых классов, что ограничивает область изменений. Изменив, что-либо в реализации одной операции — это никак не повлияет на остальные. Названия методов и свойств подобраны для повышения информативности реализуемых методов. Каждый метод выполняет конкретную задачу, поставленую запросом на обновления из события. Данное решение позволяет легко писать тесты для конкретных методов и оптимизировать для конкретных случаев. Текущие базовые классы ограничивают область видимости некоторых объектов для разработчика. Например: они сами создают объект подписки и сами его отпускают, когда ссылку на результат операции или функции никто не хранит. Но самая важная делать, что дают объекты данной модели — это возможность работы с сущностями реального мира, а не низкоуровневыми деталями реализации, потому что типы в модели сформированы исключительно в терминах функционального программирования, скрывая реализацию, основанную на реактивной парадигме. Данные классы позволяют реализовать свои операции, и если потребуется, создать дополнительные абстрактные типы.

ГЛАВА 5
ОПЕРАЦИИ И ФУНКЦИИ

Разработанное программное обеспечение предстваляет из себя библиотеку кода, написанную на языке C#. Модель проекта не ограничена только данным языком и платформой, так как в реализации используются только стандартные типы данных и библиотека Reactive Extensions. Reactive Extensions портированы на множество платформ: Java, Javascript, Python и другие.

В контексте этой главы я хочу рассказать о реализованных в проекте операций реляционной алгебры с некоторыми модификациями, которые позволяют получить обновляемые проекции. Список реализованных на текущий момент операций:

– Select — проекция;
– Where — выборка;
– SelectMany — проекция элементов в последовательности с последующим объединением;
– GroupBy — группировка элементов по ключу;
– Take — выборка указанного количества первых элементов списка;
– Skip — выборка всех элементов списка кроме указанного количества первых элементов;
– Union — объединение;
– Intersect — пересечение;
– Except — вычитание;
– Sort — сортировка;
– Join — декартово произведение;
– Reverse — отображение в список с обратным порядком;
– Distinct — отображение набора элементов в множество.

Также реализовано несколько функций:

– Count — количество элементов;
– Max — максимальный элемент;
– Min — минимальный элемент.

Далее я расскажу подробно о каждом реализованном методе из данных списков. Примеры описанные в листингах помогают понять, как их использовать. Для упрощения примеров, в каждом будет использоваться набор элементов Person(см. рисунок 5.1), который содержит свойства FirstName(Имя), LastName(Фамилия), Age(Возраст). Также класс содержит генераторы событий, которые позволяют узнать, когда и какое свойство изменилось.

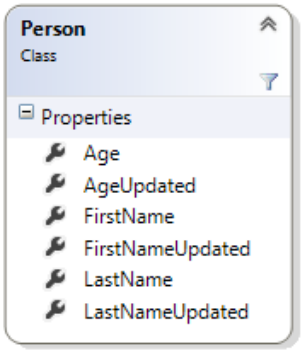

Рисунок *5.1* — Класс Person

1 Select

Select — проецирует элемент последовательности в новую форму. В данном случае метод принимает два параметра: selector — функция преобразования, применяемая к каждому элементу, updaterSelector — функция преобразования, применяемая к каждому элементу, которая получает генератор событий изменения элемента.

Листинг 5.1 — Пример использования Select

```
IObservableCollection<Person> collection = new
   ObservableCollection<Person>(); // create collection
IObservableReadOnlyCollection<string> names = collection.SelectRc
   (x => x.FirstName, x => x.FirstNameChanged.Select(_ => x)); //
    create projection
IDisposable sub = names.CollectionChanged.Subscribe(
 x => x.Match(
   onInsert: y => { Console.WriteLine(y.Item); },
   onRemove: y => { },
   onReplace: y => { Console.WriteLine($"{y.OldItem} -> {y.
      NewItem}"); },
   onReset: y => { },
   onEmpty: y => { })); \\ create subscription

collection.Add(new Person() { FirstName = "Alan", LastName = "
   Wake", Age = 32 }); // output: Alan
Person vader = new Person() { FirstName = "Darth", LastName = "
   Vader", Age = 45 };
collection.Add(vader); // output: Darth

vader.FirstName = "Anakin"; // output: Darth -> Anakin
```

В листинге 5.1 создается наблюдаемая коллекция. Затем создается проекция и обработчик событий, который обрабатывает событие добавления эле-

ментов в коллекцию и замену элементов. Проекция выбирает имена людей. При изменении имени у объекта Person изменится и проекция. Как можно заметить операция реагирует на изменение проекции, если передать ей соответствующий генератор событий.

2 Where

Where — выполняет фильтрацию последовательности значений на основе заданного предиката. В данном случае метод принимает два параметра: predicate — функция для проверки каждого элемента на соответствие условию, применяемая к каждому элементу, updaterSelector — функция преобразования, применяемая к каждому элементу, которая получает генератор событий изменения элемента.

Листинг 5.2 — Пример использования Where

```
IObservableCollection<Person> collection = new
   ObservableCollection<Person>(); // create collection
IObservableReadOnlyCollection<Person> olderThan40 = collection.
   WhereRc(x => x.Age > 40, x => x.AgeChanged.Select(_ => x)); //
    create filter
IDisposable sub = olderThan40.CollectionChanged.Subscribe(
 x => x.Match(
   onInsert: y => { Console.WriteLine($"added {y.Item.FirstName}"
      ); },
   onRemove: y => { Console.WriteLine($"removed {y.Item.FirstName
      }"); },
   onReplace: y => { },
   onReset: y => { },
   onEmpty: y => { })); \\ create subscription

Person wake = new Person() { FirstName = "Alan", LastName = "Wake
   ", Age = 32 };
collection.Add(wake);
Person vader = new Person() { FirstName = "Darth", LastName = "
   Vader", Age = 45 };
collection.Add(vader); // output: added Darth

wake.Age = 45; // output: added Alan
vader.Age = 6; // output: removed Darth
```

В листинге 5.2 создается наблюдаемая коллекция. Затем создается выборка, зависящая от возраста. Затем создается обработчик событий, который обрабатывает событие добавления и удаления элементов. Полученный фильтр обрабатывает добавленные в исходную коллекцию элементы и реагирует на изменение критерия для предиката. Как можно заметить операция реагирует

на изменение исходной коллекции и содержимого элементов, если передать ей соответствующий генератор событий для предиката.

3 SelectMany

SelectMany — проецирует каждый элемент последовательности и объединяет результирующие последовательности в одну последовательность. Полученная последовательность реагирует на изменения исходной коллекции и на изменения каждой подпоследовательности. В данном случае метод принимает один параметр: selector — функция преобразования, применяемая к каждому элементу.

Листинг 5.3 — Пример использования SelectMany

```
IObservableCollection<Person> collection = new
   ObservableCollection<Person>(); // create collection
IObservableReadOnlyCollection<Person> persons = collection.
   SelectManyRc(x => collection); // create projection
IDisposable sub = persons.CollectionChanged.Subscribe(
 x => x.Match(
   onInsert: y => { Console.WriteLine($"added {y.Item.FirstName}"
      ); },
   onRemove: y => { Console.WriteLine($"removed {y.Item.FirstName
      }"); },
   onReplace: y => { },
   onReset: y => { },
   onEmpty: y => { })); \\ create subscription

Person wake = new Person() { FirstName = "Alan", LastName = "Wake
   ", Age = 32 };
collection.Add(wake); // output: added Alan

Person vader = new Person() { FirstName = "Darth", LastName = "
   Vader", Age = 45 };
collection.Add(vader); // output: added Darth
                  // output: added Alan
                  // output: added Darth
```

В листинге 5.3 создается наблюдаемая коллекция. Затем создается проекция, которая отображает каждый элемент в исходную коллекцию. Затем создается обработчик событий, который обрабатывает событие добавления и удаления элементов. Полученная проекция обрабатывает события исходной коллекции и на изменения коллекций полученных из функции преобразования. Т. е. если удалить элемент из исходной коллекции, то удалятся все элементы, которые соответсвовали проекции этого элемента. Если удалить элемент

из проекции элемента, то из результирущей наблюдаемой коллекции удалится этот же самый элемент.

4 GroupBy

GroupBy — создает наблюдаемую коллекцию ключей, каждый из которых сопоставлен с одним или несколькими значениями в соответствии с заданной функцией выбора ключа. Метод принимает два параметр: keySelector — функция, извлекающая ключ из каждого элемента, updaterSelector — функция преобразования, применяемая к каждому элементу, которая получает генератор событий изменения ключа элемента.

Листинг 5.4 — Пример использования GroupBy

```
IObservableCollection<Person> collection = new
   ObservableCollection<Person>(); // create collection
IObservableLookup<char, Person> persons = collection.GroupBy(x =>
    x.LastName[0], x => x.LastNameChanged.Select(_ => x)); //
   create observable lookup
IDisposable sub = persons.CollectionChanged.Subscribe(
 x => x.Match(
   onInsert: y => { Console.WriteLine($"added {y.Item.Key}"); },
   onRemove: y => { Console.WriteLine($"removed {y.Item.Key}");
      },
   onReplace: y => { },
   onReset: y => { },
   onEmpty: y => { })); \\ create subscription

Person wake = new Person() { FirstName = "Alan", LastName = "Wake
   ", Age = 32 };
collection.Add(wake); // output: added W

Person vader = new Person() { FirstName = "Darth", LastName = "
   Vader", Age = 45 };
collection.Add(vader); // output: added V
```

В листинге 5.4 создается наблюдаемая коллекция. Затем создается словарь, где каждому элементу соответвует наблюдаемая коллекция эллементов. Затем создается обработчик событий, который обрабатывает событие добавления и удаления элементов. Полученный словарь реагирует на добавление новых ключей и элементов в коллекции для каждого ключа. Если для ключа отсутствуют элементы, которые ему соответсвуют — он удаляется из словаря.

5 Take и Skip

Take и Skip — создают наблюдаемый список, который соответсвует подсписку исходного. В данной реализации эти опрерации объединены в одну SkipAndTake. Метод принимает два параметр: skip — количество элементов от начала списка требуется пропустить, take — максимальное количество элементов после пропущенных, которое можно взять.

Листинг 5.5 — Пример использования SkipAndTake

```
IObservableList<Person> list = new ObservableList<Person>(); //
   create list
ObservableValue<int> skip = new ObservableValue<int>(0);
ObservableValue<int> take = new ObservableValue<int>(1);
IObservableReadOnlyList<Person> persons = list.SkipAndTakeRl(skip
   , take); // create sublist
IDisposable sub = persons.ListChanged.Subscribe(
 x => x.Match(
   onInsert: y => { Console.WriteLine($"added {y.Item.FirstName}"
      ); },
   onRemove: y => { Console.WriteLine($"removed {y.Item.FirstName
      }"); },
   onReplace: y => { },
   onMove: y => { }
   onReset: y => { },
   onEmpty: y => { })); \\ create subscription

Person wake = new Person() { FirstName = "Alan", LastName = "Wake
   ", Age = 32 };
collection.Add(wake); // output: added Alan

Person vader = new Person() { FirstName = "Darth", LastName = "
   Vader", Age = 45 };
collection.Add(vader);

skip.Value = 1; // output: removed Alan
            // output: added Darth
```

В листинге 5.5 создается наблюдаемая коллекция. Затем создаются объекты, у которых можно менять значение и подписаться на его изменение. Затем создается подсписок, у которого пропускается 0 элементов и берется максимум 1. При добавлении первой записи, она также попадает в подсписок. При добавлении второй, места в подсписке нет, так как он ограничен одним элементом. Далее мы изменяем количество элементов, которое мы хотим пропускать с 0 до 1. Это значит что запись Alan удалится из подсписка, а Darth добавится. Обработка событий обновления исходного списка идет с сохране-

нием позиций элементов относительно друг друга.

6 Union

Union — находит объединения наборов двух последовательностей, используя компаратор проверки на равенство. Метод принимает три параметра: second — вторую последовательность элементов такого же типа, что и исходная, comparer — функция для сравнения элементов, updaterSelector — функция преобразования, применяемая к каждому элементу, которая получает генератор событий изменения элемента.

Листинг 5.6 — Пример использования Union

```
IObservableCollection<Person> collection1 = new
   ObservableCollection<Person>(); // create collection1
IObservableCollection<Person> collection2 = new
   ObservableCollection<Person>(); // create collection2
IObservableReadOnlyCollection<Person> persons = collection1.
   UnionRc(collection2, EqualityComparer<Person>.Default, x =>
   Observable.Never<Person>()); // create union
IDisposable sub = persons.CollecionChanged.Subscribe(
 x => x.Match(
   onInsert: y => { Console.WriteLine($"added {y.Item.FirstName}"
      ); },
   onRemove: y => { Console.WriteLine($"removed {y.Item.FirstName
      }"); },
   onReplace: y => { },
   onMove: y => { }
   onReset: y => { },
   onEmpty: y => { })); \\ create subscription

Person wake = new Person() { FirstName = "Alan", LastName = "Wake
   ", Age = 32 };
collection1.Add(wake); // output: added Alan

Person vader = new Person() { FirstName = "Darth", LastName = "
   Vader", Age = 45 };
collection2.Add(vader); // output: added Darth
```

В листинге 5.6 создается две наблюдаемые коллекции для объектов Person. Затем создается наблюдаемая коллекция, которая является объединением исходных двух. При добавлении элементов в какую-либо из исходных коллекций, элементы добавляются в результирующую. Таким образом можно объединять несколько коллекций и создавать глобальные коллекции, которые следят, например, за списком контактов из разных учетных записей пользователя.

7 Intersect

Intersect — находит пересечение наборов двух последовательностей, используя компаратор проверки на равенство. Метод принимает три параметра: second — вторую последовательность элементов такого же типа, что и исходная, comparer — функция для сравнения элементов, updaterSelector — функция преобразования, применяемая к каждому элементу, которая получает генератор событий изменения элемента.

Листинг 5.7 — Пример использования Intersect

```
IObservableCollection<Person> collection1 = new
   ObservableCollection<Person>(); // create collection1
IObservableCollection<Person> collection2 = new
   ObservableCollection<Person>(); // create collection2
IObservableReadOnlyCollection<Person> persons = collection1.
   IntersectRc(collection2, EqualityComparer<Person>.Default, x
   => Observable.Never<Person>()); // create intersect
IDisposable sub = persons.CollecionChanged.Subscribe(
 x => x.Match(
   onInsert: y => { Console.WriteLine($"added {y.Item.FirstName}"
      ); },
   onRemove: y => { Console.WriteLine($"removed {y.Item.FirstName
      }"); },
   onReplace: y => { },
   onMove: y => { }
   onReset: y => { },
   onEmpty: y => { })); \\ create subscription

Person wake = new Person() { FirstName = "Alan", LastName = "Wake
   ", Age = 32 };
Person vader = new Person() { FirstName = "Darth", LastName = "
   Vader", Age = 45 };
Person geralt = new Person() { FirstName = "Geralt", LastName = "
   From Rivia", Age = 60 };

collection1.Add(wake);
collection1.Add(vader);
collection2.Add(vader); // output: added FirstName
collection2.Add(geralt)
```

В листинге 5.7 создается две наблюдаемые коллекции для объектов Person. Затем создается наблюдаемая коллекция, которая является пересечением исходных двух. При добавлении элементов в какую-либо из исходных коллекций, алгоритм операции использует компаратор определяет вхождение элемента и решает добавлять или не добавлять в результирующую коллекцию. Результат операции сходен с результатом выполнения Intersect из LINQ.

Таким образом можно находить пересечения нескольких коллекций.

8 Except

Except — находит разницу наборов двух последовательностей, используя компаратор проверки на равенство. Метод принимает три параметра: second — вторую последовательность элементов такого же типа, что и исходная, comparer — функция для сравнения элементов, updaterSelector — функция преобразования, применяемая к каждому элементу, которая получает генератор событий изменения элемента.

Листинг 5.8 — Пример использования Except

```
IObservableCollection<Person> collection1 = new
   ObservableCollection<Person>(); // create collection1
IObservableCollection<Person> collection2 = new
   ObservableCollection<Person>(); // create collection2
IObservableReadOnlyCollection<Person> persons = collection1.
   ExceptRc(collection2, EqualityComparer<Person>.Default, x =>
   Observable.Never<Person>()); // create Except
IDisposable sub = persons.CollecionChanged.Subscribe(
 x => x.Match(
   onInsert: y => { Console.WriteLine($"added {y.Item.FirstName}"
      ); },
   onRemove: y => { Console.WriteLine($"removed {y.Item.FirstName
      }"); },
   onReplace: y => { },
   onMove: y => { }
   onReset: y => { },
   onEmpty: y => { })); \\ create subscription

Person wake = new Person() { FirstName = "Alan", LastName = "Wake
   ", Age = 32 };
Person vader = new Person() { FirstName = "Darth", LastName = "
   Vader", Age = 45 };
Person geralt = new Person() { FirstName = "Geralt", LastName = "
   From Rivia", Age = 60 };

collection1.Add(wake); // output: added Alan
collection1.Add(vader); // output: added Darth
collection2.Add(vader); // output: removed Darth
collection2.Add(geralt)
```

В листинге 5.8 создается две наблюдаемые коллекции для объектов класса Person. Затем создается наблюдаемая коллекция, которая является разницей множеств исходных двух. При добавлении элементов в какую-либо из исходных коллекций, алгоритм операции использует компаратор определяет

вхождение элемента и решает добавлять или не добавлять в результирующую коллекцию. Сначала добавляется Alan в первую коллекцию — разница первой и второй теперь Alan. Затем добавляется Darth — разница становится Alan и Darth. При добавлении Darth во вторую коллекцию, он удалится из результирующей, так как он был в первой и согласно правилам вычитания множеств: должен отсутствовать в результате. Результат операции сходен с результатом выполнения Except из LINQ.

9 Sort

Sort — создает сортированный список согласно критерию и компаратору. Метод принимает три параметра: selector — функция, которая строит отображения критерия для сравнения, comparer — компаратор для критериев, updaterSelector — функция преобразования, применяемая к каждому элементу, которая получает генератор событий изменения элемента.

Листинг 5.9 — Пример использования Sort

```
IObservableCollection<Person> collection = new
   ObservableCollection<Person>(); // create collection
IObservableReadOnlyList<Person> persons = collection.SortRc(x =>
   x.Age, Comparer<uint>.Default, x => x.AgeUpdated.Select(_ => x
   )); // create sort
IDisposable sub = persons.ListChanged.Subscribe(
 x => x.Match(
   onInsert: y => { Console.WriteLine($"added {y.Item.FirstName}
      to {y.Index}"); },
   onRemove: y => { Console.WriteLine($"removed {y.Item.FirstName
      } from {y.Index}"); },
   onReplace: y => { },
   onMove: y => { }
   onReset: y => { },
   onEmpty: y => { })); \\ create subscription

Person vader = new Person() { FirstName = "Darth", LastName = "
   Vader", Age = 45 };
Person wake = new Person() { FirstName = "Alan", LastName = "Wake
   ", Age = 32 };
Person geralt = new Person() { FirstName = "Geralt", LastName = "
   From Rivia", Age = 60 };

collection.Add(vader); // output: added Darth to 0
collection.Add(wake); // output: added Alan to 0
collection.Add(geralt); // output: added Geralt to 2
// result [Alan(32), Darth(45), Geralt(60)]
```

В листинге 5.9 создается коллекция для объектов Person. Затем создается список, в котором записи сортируются согласно возрасту в порядке возрастания. При добавлении элементов в исходный список, операция определяет позицию для нового элемента согласно компаратору. Сначала добавляется Darth в первую коллекцию — его позиция будет 0. Затем добавляется Alan — его возраст мешьне чем Darth и поэтому его позиция будет 0, а Darth сдвинется на позицию 1. И наконец Geralt вставится в последнюю позицию. В результате возрасты персонажей будут 32, 45 и 60 соответсвенно.

10 Join

Join — Устанавливает корреляцию между элементами двух последовательностей на основе сопоставления ключей. Метод принимает семь параметров:

– inner — последовательность, соединяемая с исходной последовательностью;

– outerKeySelector — функция, извлекающая ключ соединения из каждого элемента первой последовательности;

– outerKeyUpdated — функция, извлекающая генератой событий изменения ключа для первой коллекции;

– innerKeySelector — функция, извлекающая ключ соединения из каждого элемента второй последовательности;

– innerKeyUpdated — функция, извлекающая генератой событий изменения ключа для второй коллекции;

– resultSelector — функция для создания результирующего элемента для пары соответствующих элементов;

– comparer — функция для хэширования и сравнения ключей.

Листинг 5.10 — Пример использования Join

```
IObservableCollection<Person> collection1 = new
   ObservableCollection<Person>(); // create collection1
IObservableCollection<Person> collection2 = new
   ObservableCollection<Person>(); // create collection2
IObservableReadOnlyCollection<string> persons = collection1.
   JoinRc(
  collection2,
  x => x.Age,
  x => Observable.Never<Person>(),
  x => x.Age,
  x => Observable.Never<Person>(),
  (x, y) => $"{x.FirstName} + {y.FirstName}",
  EqualityComparer<uint>.Default); // create join
```

```
IDisposable sub = persons.CollecionChanged.Subscribe(
 x => x.Match(
   onInsert: y => { Console.WriteLine($"added {y.Item}"); },
   onRemove: y => { Console.WriteLine($"removed {y.Item}"); },
   onReplace: y => { },
   onMove: y => { }
   onReset: y => { },
   onEmpty: y => { })); \\ create subscription

Person wake = new Person() { FirstName = "Alan", LastName = "Wake
   ", Age = 32 };
Person vader = new Person() { FirstName = "Darth", LastName = "
   Vader", Age = 45 };
Person geralt = new Person() { FirstName = "Geralt", LastName = "
   From Rivia", Age = 45 };

collection1.Add(wake);
collection1.Add(vader);
collection2.Add(vader); // output: added Darth + Darth
collection2.Add(geralt) // output: added Darth + Geralt
```

В листинге 5.10 создается две наблюдаемые коллекции для объектов Person. Затем создается наблюдаемая коллекция, которая является декартовывым произведением с фильтром(inner join) исходных двух. В данном случае ключом является возраст персонажей. При добавлении элементов в какую-либо из исходных коллекций, алгоритм извлекает ключ и сравнивает его с уже существующмими ключами. Если нужно создается новый элемент и строится проекция Результат операции сходен с результатом выполнения Join из LINQ.

11 Reverse

Reverse — создает список согласно с порядком элементов обратным исходному. Метод не принимает параметров.

Листинг 5.11 — Пример использования Reverse

```
IObservableList<Person> list = new ObservableLit<Person>(); //
   create list
IObservableReadOnlyList<Person> persons = list.ReverseRl(); //
   create reversed list
IDisposable sub = persons.ListChanged.Subscribe(
 x => x.Match(
   onInsert: y => { Console.WriteLine($"added {y.Item.FirstName}
      to {y.Index}"); },
   onRemove: y => { Console.WriteLine($"removed {y.Item.FirstName
      } from {y.Index}"); },
```

```
    onReplace: y => { },
    onMove: y => { }
    onReset: y => { },
    onEmpty: y => { })); \\ create subscription

Person vader = new Person() { FirstName = "Darth", LastName = "
    Vader", Age = 45 };
Person wake = new Person() { FirstName = "Alan", LastName = "Wake
    ", Age = 32 };
Person geralt = new Person() { FirstName = "Geralt", LastName = "
    From Rivia", Age = 60 };

list.Add(vader); // output: added Darth to 0
list.Add(wake); // output: added Alan to 0
list.Add(geralt); // output: added Geralt to 0
// result [Geralt, Alan, Darth]
```

В листинге 5.11 создается список для объектов Person. Затем создается список, в котором записи сортируются в обратном порядке. При добавлении элементов в исходный список, элементы в результирующем получают отраженный индекс. Например, добавление в конец списка вызывает добавление в начало результирующего списка. Результать операции сходен с результатом выполнения Reverse из LINQ, с тем лишь ограничением, что применяется только для списков.

12 Distinct

Distinct — возвращает различающиеся элементы последовательности, используя указанный компаратор для сравнения значений. Метод принимает два параметра: comparer — функция для сравнения элементов, updaterSelector — функция преобразования, применяемая к каждому элементу, которая получает генератор событий изменения элемента.

Листинг 5.12 — Пример использования Distinct

```
IObservableCollection<Person> collection = new
    ObservableCollection<Person>(); // create collection
IObservableReadOnlyCollection<Person> persons = collection.
    DistinctRc(collection, EqualityComparer<Person>.Default, x =>
    Observable.Never<Person>()); // create distinct
IDisposable sub = persons.CollecionChanged.Subscribe(
  x => x.Match(
    onInsert: y => { Console.WriteLine($"added {y.Item.FirstName}"
      ); },
    onRemove: y => { Console.WriteLine($"removed {y.Item.FirstName
      }"); },
```

```
    onReplace: y => { },
    onMove: y => { }
    onReset: y => { },
    onEmpty: y => { })); \\ create subscription

Person wake = new Person() { FirstName = "Alan", LastName = "Wake
    ", Age = 32 };
Person vader = new Person() { FirstName = "Darth", LastName = "
    Vader", Age = 45 };

collection.Add(wake); // output: added Alan
collection.Add(vader); // output: added Darth
collection.Add(vader); // output: added Darth
collection.Remove(wake) // output: removed Alan
```

В листинге 5.12 создается наблюдаемая коллекция для объектов Person. Затем создается наблюдаемая коллекция, которая является набором уникальных элементов из исходной. При добавлении элементов равных относительно компаратора, в результирующую коллекцию добавится лишь один и будет существовать, пока в исходной коллекции будет находится хотя бы одна его копия.

13 Count

Count — функция, которая возвращает число равное количеству элементов коллекции, обновляемое с изменением коллекции. Метод не принимает параметров.

Листинг 5.13 — Пример использования Count

```
IObservableCollection<Person> collection = new
    ObservableCollection<Person>(); // create collection
ObservableValue<int> count = collection.CountRc(); // create
    count
IDisposable sub = count.ValueChanged.Subscribe(
  x => { Console.WriteLine($"count = {x}"); }); // create
     subscription

Person vader = new Person() { FirstName = "Darth", LastName = "
    Vader", Age = 45 };
Person wake = new Person() { FirstName = "Alan", LastName = "Wake
    ", Age = 32 };
Person geralt = new Person() { FirstName = "Geralt", LastName = "
    From Rivia", Age = 60 };

collection.Add(vader); // output: count = 1
collection.Add(wake); // output: count = 2
```

```
collection.Add(geralt); // output: count = 3
```

В листинге 5.9 создается коллекция для объектов Person. Затем вычисляется функция Count, значение которой изменяется вместе с изменением исходной коллекции.

14 Min и Max

Min и Max — функции вычисляющие минимум и максимум соответсвенно. Метод принимает три параметра: selector — функция, которая строит отображения критерия для сравнения, comparer — компаратор для критериев, updaterSelector — функция преобразования, применяемая к каждому элементу, которая получает генератор событий изменения элемента.

Листинг 5.14 — Пример использования Min и Max

```
IObservableCollection<Person> collection = new
   ObservableCollection<Person>(); // create collection
ObservableValue<Person?> max = collection.MaxRc(x => x.Age,
   Comparer<uint>.Default, x => x.AgeUpdated.Select(_ => x)); //
   create max
ObservableValue<Person?> min = collection.MinRc(x => x.Age,
   Comparer<uint>.Default, x => x.AgeUpdated.Select(_ => x)); //
   create min

IDisposable sub1 = max.ValueChanged.Subscribe(
  x => { Console.WriteLine($"max age = {x?.Age}"); }); // create
     subscription
IDisposable sub2 = min.ValueChanged.Subscribe(
  x => { Console.WriteLine($"min age = {x?.Age}"); }); // create
     subscription

Person vader = new Person() { FirstName = "Darth", LastName = "
   Vader", Age = 45 };
Person wake = new Person() { FirstName = "Alan", LastName = "Wake
   ", Age = 32 };
Person geralt = new Person() { FirstName = "Geralt", LastName = "
   From Rivia", Age = 60 };
collection.Add(vader); // output: max age = 45
                              // output: min age = 45
collection.Add(wake); // output: min age = 32
collection.Add(geralt); // output: max age = 60
```

В листинге 5.14 создается коллекция для объектов Person. Затем вычисляются функции Max и Min. Результат функции пересчитывается при изменении исходной коллекции или её элементов согласно компаратору. Если коллекция пуста, то значение функций примет null.

ГЛАВА 6
ТЕСТИРОВАНИЕ И АНАЛИЗ ПОЛУЧЕННОЙ МОДЕЛИ

В данной главе приводится описание метода тестирования полученного инструмента для работы с данными.

Для тестирования использовался подход модульного тестирования. Причины данного выбора:

– проверка корректности работы операций и функций библиотеки;

– подготовка к последующим оптимизациям алгоритмов;

– подготовка примеров использования разработанных методов в реальной ситуации.

Идея состоит в том, чтобы писать тесты для каждой реализованной операции. Это позволяет достаточно быстро проверить, не привело ли очередное изменение кода к регрессии, то есть к появлению ошибок в уже оттестированных местах программы, а также облегчает обнаружение и устранение таких ошибок. Конечный пользователь данного продукта, может использовать эти тесты как примеры. В качестве эталона в тестах, использовались операции и функции LINQ, где это было возможно.

В ходе тестирования важно было проверить два свойства реализованных структур данных:

– корректность применение операции;

– корректно сгенерированные события изменения данных.

Корректность применения операции проверялся сравнением результата выполнения операции с эталоном, события — с помощью специальных обработчиков, которые определяли тип сгенерированного изменения.

Тесты сгруппированы по операциям и протестированы все возможные методы и события для реализованных структур данных в данном проекте. Стандартный тест для операции включает в себя:

– инициализация исходных данных;

– применение операции к данным;

– применение эталонной операции к данным;

– функция тестирования свойства(добавление, удаление и т. д.);

– пуск теста с сгенерированными данными.

Листинг 6.1 — Пример теста

```
[TestMethod]
public void UpdateItem()
{
  \\ initialization
```

```
  IObservableCollection<BehaviorSubject<int>> collection = new
     ObservableCollection<BehaviorSubject<int>>();

  \\ create actual data
  IObservableReadOnlyCollection<int> actualOperation = collection
    .WhereRc(_filter, _getUpdater)
    .SelectRc(_selector, _getUpdater);

  \\ create expected data
  IEnumerable<int> expectedOperation = collection.Where(_filter).
     Select(_selector);

  \\test function body
  Action<BehaviorSubject<int>> assertAddAndUpdate = item =>
  {
    collection.Add(item);
    Assert.IsTrue(Enumerable.SequenceEqual(expectedOperation,
       actualOperation));
    item.OnNext(item.Value + 1);
    Assert.IsTrue(Enumerable.SequenceEqual(expectedOperation,
       actualOperation));
    item.OnNext(item.Value + 1);
    Assert.IsTrue(Enumerable.SequenceEqual(expectedOperation,
       actualOperation));
  };

  \\start test
  Prop.ForAll(Arb.From(_intGen), assertAddAndUpdate).
     QuickCheckThrowOnFailure();
}
```

В листинге 6.1 приведен пример одного теста на изменение обекта в коллекции. Данный тест проверят как применяется операция и правильные ли генерирует события. В качестве платформы тестирования использовался Unit Test Framework, который поставляется вместе с Visual Studio. Для генерации тестовых данных используется библиотека FsCheck, написанная на языке F#. Она предназначена для реализации модульного и функционального тестирования программных систем. Уже на примере теста видно, как мало требуется написать кода, чтобы создать коллекцию реагирующую на изменения исходных данных.

Рассмотрим пример использования данной библиотеки вместе с патерном проектирования MVVM(Model-View-ViewModel) [22]. На рисунке 6.1 представлено:

– External service layer — внешний сервис, который предоставил новую модель;

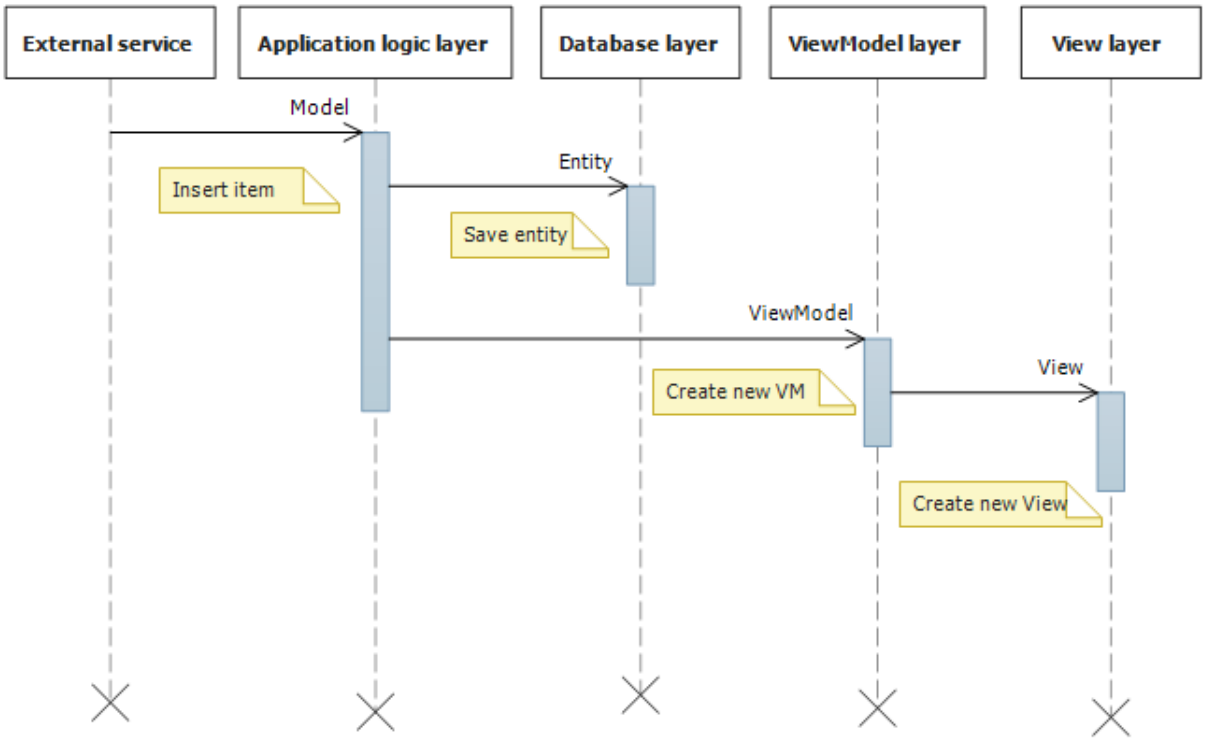

Рисунок *6.1* — Использование с паттерном MVVM

– Application logic layer — бизнес логика приложения;
– Database layer — слой доступа к базе данных;
– ViewModel layer — слой абстракций для моделей и представлений;
– View layer — слой пользовательского интерфейса.

По сценарию в систему приходит новая модель. Нам требуется сохранить её в базу данных, создать из нее ViewModel, а из ViewModel — View. А данном случае, Application logic layer будет содержать объект наблюдаемой коллекции, на который будут подписаны Database layer и ViewModel layer. При добавлении элемента в коллекцию, изменения распространятся автоматически на всю систему. Для реализации данного примера достаточно операции Select(проекции) и операции Dispatch(перенаправления в другой поток исполнения).

Так добавив по несколько строк кода в каждый уровень приложения, можно создать связь между пользовательским интерфейсом и бизнес-логикой, используя только операции данной библиотеки. Используя понятия реляционной алгебры, код получается легко читаемым для широкого круга разработчиков.

ЗАКЛЮЧЕНИЕ

В данной работе были рассмотрены вопросы использования функционального и реактивного программирования для обработки данных. Также было разработано подмножество операций реляционной алгебры для динамических данных. Была разработана библиотека типов и функций в качестве реализации данной алгебры на платформе .NET. Разработанная библиотека функций может быть использована при разработке коммерческих продуктов на платформе Microsoft .NET для реализации взаимодействий между моделями, сервисами, для построенения отзывчивых пользовательских интерфейсов, созданных для получения актуальной информации. Единственными наиболее близким по функциональности продуктом со схожей областью применения для платформы Microsoft .NET является библиотека Dynamic Data, разработанная Roland Pheasant. Отличие данной работы, что операции и функции могут использовать не только события изменения коллекций, но и изменения самих элементов.

В целом разработанная библиотека включает требуемую функциональность, необходимую для практического использования. В библиотеке присутствуют функции для решения ряда задач связанных с применением реляционной алгебры: объединение, пересечение, вычитание, декартово произведение, выборка, проекция, соединение. Помимо этого реализованы функции агрегаторы и операции сортировки.

В результате цель работы была достигнута. Было создано программное обеспечение решающие задачи, связанных с применением релационной алгебры и событийной модели на практике. Было разработан алгоритм и проверена его работоспособность на реальных данных. За рамками проделанной работы остались некоторые специфические вопросы, например, оптимизация выполнение данных операций, реализация более специфичных функций для работы с данными. Эти вопросы возникают не во всех практических задачах, но при необходимости разработанная библиотека может быть доработана. Эти задачи также являются нетривиальными и требуют детального изучения и проработки, они не рассматривались в данной работе из-за временных ограничений на их исследвание.

В дальнейшем планируется развивать и довести существующее ПО до полноценной библиотеки, способной решать более широкий класс задач, возникающих в области построения отзывчивых пользовательских интерфейсов.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] Harrison, John. Введение в функциональное программирование / John Harrison. — 1997.

[2] W., Dijkstra E. A Discipline of Programming / Dijkstra E. W. — Prentice-Hall, 1976.

[3] Tatum, Malcolm. What Is Location Transparency? [Эл[Электронн ресурс. — Электронные данные. — 2016. — April. — Режим доступа: http://www.wisegeek.com/what-is-location-transparency.htm.

[4] Д. Грин, Д. Кнут. Математические методы анализа алгоритмов / Д. Кнут Д. Грин. — Мир, 1987.

[5] Leon-Garcia, Alberto. Probability, statistics, and random processes for electrical engineering (3rd ed.). / Alberto Leon-Garcia. — Prentice Hall, 2008.

[6] Circuit breaker design pattern [Электронный ресурс]. — Электронные данные. — Режим доступа: https://msdn.microsoft.com/en-us/library/dn589784.aspx.

[7] The Reactive Manifesto v2.0 [Электронный ресурс]. — Электронные данные. — 2014. — September. — Режим доступа: http://www.reactivemanifesto.org/.

[8] Макконнелл, С. Совершенный код. Мастер-класс / Пер. с англ. / С. Макконнелл. — СПб. : Издательско-торговый дом «Русская редакция», 2005. — 896 с.

[9] Common Language Infrastructure (CLI). Partitions I to VI. — 2012. — June. `http://www.ecma-international.org/publications/standards/Ecma-335.htm`.

[10] Рихтер, Джеффри. CLR via C#. Программирование на платформе Microsoft .NET Framework 2.0 на языке C# / Джеффри Рихтер. — 2-е изд. — СПб. : Питер, Русская Редакция, 2007. — 656 с.

[11] Марченко, А. Л. Основы программирования на C# 2.0 / А. Л. Марченко. — БИНОМ. Лаборатория знаний, Интернет-университет информационных технологий — ИНТУИТ.ру, 2007. — 552 с.

[12] Richter, Jeffrey. CLR via C# / Jeffrey Richter. Microsoft, Developer Reference. — 4-th edition. — One Microsoft Way, Redmond, Washington 98052-6399 : Microsoft Press, 2012. — 896 P.

[13] Абельсон, Харольд. Структура и интерпретация компьютерных программ / Харольд Абельсон, Джеральд Джей Сассман, Джули Сассман. — Добросвет, 2006. — 608 с.

[14] Albahari, Joseph. C# 5.0 in a Nutshell / Joseph Albahari, Ben Albahari. — 5-th edition. — O'Reilly Media, Inc, 2012. — June. — 1062 P.

[15] C Sharp [Электронный ресурс]. — Электронные данные. — Режим доступа: `http://ru.wikipedia.org/wiki/C_Sharp`. — Дата доступа: 22.03.2013.

[16] Michaelis, Mark. The New and Improved C 6.0 / Mark Michaelis // MSDN Magazine. — 2014.

[17] MSDN. IReadOnlyCollection [Электронный ресурс]. — Электронные данные. — Режим доступа: https://msdn.microsoft.com/ru-ru/library/hh881542(v=vs.110).aspx.

[18] MSDN. IReadOnlyList [Электронный ресурс]. — Электронные данные. — Режим доступа: https://msdn.microsoft.com/ru-ru/library/hh192385(v=vs.110).aspx.

[19] MSDN. INotifyCollectionChanged [Электронный ресурс]. — Электронные данные. — Режим доступа: https://msdn.microsoft.com/ru-ru/library/system.collections.specialized.inotifycollectionchanged(v=vs.110).aspx.

[20] line Dictionary of Computing, Free On. Algebraic data type [Электронный ресурс]. — Электронные данные. — 1994. — November. — Режим доступа: http://foldoc.org/algebraic

[21] MSDN. Weak References [Электронный ресурс]. — Электронные данные. — Режим доступа: https://msdn.microsoft.com/en-us/library/ms404247(v=vs.110).aspx.

[22] MSDN. The MVVM Pattern [Электронный ресурс]. — Электронные данные. — 2012. — Февраль. — Режим доступа: https://msdn.microsoft.com/en-us/library/hh848246.aspx.