

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«ПЕНЗЕНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»
ПОЛИТЕХНИЧЕСКИЙ ИНСТИТУТ

Факультет
вычислительной техники

Кафедра
МОиПЭВМ

Направление подготовки 09.03.04 «Программная инженерия»

БАКАЛАВРСКАЯ РАБОТА
на тему
Библиотека для построения
реактивного пользовательского интерфейса

Студент	_____	Ванифатов Владислав Александрович
	(подпись, дата)	(ФИО полностью)
Руководитель	_____	Попова Н.А.
	(подпись, дата)	(фамилия, инициалы)
Нормоконтролёр	_____	Такташкин Д.В.
	(подпись, дата)	(фамилия, инициалы)

Работа допущена к защите (протокол заседания кафедры от _____ № _____)

Заведующий кафедрой	_____	Макарычев П.П.
	(подпись)	(фамилия, инициалы)

Работа защищена с отметкой _____ (протокол заседания ГЭК от _____ № _____)

Секретарь ГЭК	_____	Попова Н.А.
	(подпись)	(фамилия, инициалы)

Пенза, 2020

Федеральное государственное бюджетное образовательное учреждение
высшего образования
ПЕНЗЕНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ПОЛИТЕХНИЧЕСКИЙ ИНСТИТУТ

Факультет

Вычислительной техники

Кафедра

МО и ПЭВМ

«Утверждаю»
Заведующий кафедрой
П.П. Макарычев
«__» _____ 2019 г

**ЗАДАНИЕ
НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ
БАКАЛАВРА**

1. Студент Ванифатов Владислав Александрович гр. 16ВП1 факультета ВТ
<Фамилия, имя, отчество полностью>

2. Тема работы Библиотека для построения реактивного пользовательского интерфейса

Тема утверждена приказом ПГУ № _____ от "_____" _____ 20____

2. Руководитель работы к.т.н, доцент кафедры «МОиПЭВМ» Попова Н.А.

3. Задание на работу (назначение разработки, исходные данные и т.п.) _____

Назначение: библиотека предоставляет набор классов для построения реактивного
Пользовательского интерфейса

Основные функции:

а) Создание ячеек для хранения данных программы

б) Автоматическая актуализация значения ячеек

в) Создание автоматических реакций на изменение данных

г) Построение интерфейса по заданному описанию

д) Автоматическое обновление интерфейса при изменении данных

Технология разработки: ООП, RUP

Языковые средства разработки: C#

Инструментальные средства разработки: Unity, JetBrains Rider

Среда исполнения: Unity

4. Перечень подлежащих разработке вопросов _____

1. Анализ предметной области и постановка задач

2. Проектирование библиотеки

3. Реализация библиотеки

4. Тестирование и анализ качества

5. Планирование разработки и анализ бюджета

6. Оформление пояснительной записки

5. Календарный график выполнения работы

№ п/п	Наименование этапов работы	Объем работы	Срок выполнения	Подпись руководителя
1	<i>Анализ предметной области и постановка задач</i>	10	13.04.2020 – 28.04.2020 г.	
2	<i>Проектирование библиотеки</i>	20	29.04.2020 – 30.04.2020 г.	
3	<i>Реализация библиотеки</i>	35	01.05.2020 – 20.05.2020 г.	
4	<i>Тестирование и анализ качества</i>	20	21.05.2020 – 29.05.2020 г.	
5	<i>Планирование разработки и анализ бюджета</i>	5	01.06.2020 – 01.06.2020 г.	
6	<i>Оформление пояснительной записки</i>	10	02.06.2020 – 13.06.2020 г.	

Дата выдачи задания "30" декабря 2019

Руководитель бакалаврской работы _____
(подпись, дата)

Задание к исполнению принял студент _____
(подпись, дата)

Работу к защите допустить
Декан факультета _____
(подпись, дата)

Реферат

Пояснительная записка содержит 89 листов, 19 рисунков, 19 использованных источников и 15 таблиц.

ООП, RUP, UNITY, C#, UML, РЕАКТИВНОЕ ПРОГРАММИРОВАНИЕ, ПОЛЬЗОВАТЕЛЬСКИЙ ИНТЕРФЕЙС.

Предмет работы: системы построения реактивного пользовательского интерфейса.

Объект работы: пользовательский интерфейс программного обеспечения.

Цель работы: разработать библиотеку для построения реактивного пользовательского интерфейса в Unity.

Технологии разработки: ООП, RUP, C#, Unity.

Результаты работы: библиотека для построения реактивного пользовательского интерфейса в Unity.

Область применения: разработанная библиотека может быть использована для построения пользовательского интерфейса в приложениях и компьютерных играх, разрабатываемых на платформе Unity.

					ПГУ 09.03.04–8BP161.05 ПЗ			
Изм.	Лист	№ докум.	Подпись	Дата	«Библиотека для построения реактивного пользовательского интерфейса» Пояснительная записка	Лит.	Лист	Листов
Разраб.		Ванифатов В.А.					2	85
Провер.		Попова Н.А.						
Н. Контр.		Такташкин Д.В.				Группа 16ВП1		
Утверд.		Макарычев П.П.						

Содержание

Введение.....	5
1 Анализ предметной области	7
2 Анализ продуктов-аналогов.....	10
2.1 Библиотека Mol	10
2.2 Библиотека Jetpack Compose.....	11
2.3 Библиотека Swift UI	12
2.4 Платформа Flutter.....	12
2.5 Сравнение готовых решений	13
3 Постановка задачи	14
3.1 Требования к подсистеме реактивного программирования	14
3.2 Требования к подсистеме построения интерфейса	15
4 Анализ требований.....	17
4.1 Функциональные требования	17
4.2 Нефункциональные требования	21
4.3 Технологии и средства разработки	21
5 Проектирование библиотеки	23
5.1 Проектирование подсистемы реактивного программирования.....	23
5.2 Проектирование подсистемы построения интерфейса	27
6 Реализация библиотеки	32
6.1 Реализация подсистемы реактивного программирования.....	33
6.2 Реализация подсистемы построения интерфейса	35
7 Тестирование и анализ качества.....	37
7.1 Тестирование программного обеспечения	37
7.1.1 Метод причинно-следственных связей.....	38

7.1.2 Автоматизированное тестирование	40
7.1.3 Структурное тестирование.....	42
7.1.4 Тестирование интерфейса	47
7.2 Метрики качества.....	48
8 Планирование разработки и анализ бюджета	51
Заключение	57
Список использованных источников	58
Приложение А. Глоссарий	60
Приложение Б. Листинг программы	62
Приложение В. Результаты тестирования	80
Приложение Г. Исходный код тестов	85

Введение

Современные программные продукты, такие как приложения, web-сайты и компьютерные игры, представляют собой сложные механизмы, состоящие из множества независимых частей. Такие системы сложно представить без графического пользовательского интерфейса, который по сложности разработки зачастую не уступает системам, реализующим основной функционал программы.

Графический интерфейс является сложной системой, задачей которой является предоставление механизма взаимодействия пользователя с приложением и отображение актуального состояния программы в удобном для пользователя виде. Наиболее распространенным способом отображения информация является их вывод в виде множества графических элементов, таких как окна, списки, кнопки [1].

В большинстве случаев программное обеспечение оперирует моделями данных, удобными для вычислений внутри программы. Но такие модели не всегда понятны для пользователей, поэтому перед отображением их необходимо конвертировать в понятный человеку вид, после чего они преобразуются в множество элементов интерфейса.

Данные операции могут быть выполнены вручную, однако в сложных системах это требует больших трудозатрат и внимательности от разработчиков программного обеспечения, что значительно усложняет разработку.

Для автоматизации данного класса задач используют специализированные библиотеки для реактивного программирования и библиотеки для построения графического интерфейса.

Библиотеки для реактивного программирования выполняют автоматический перерасчет данных, представленных математическими формулами или алгоритмами на основе других данных [2]. Это гарантирует,

что в любой момент систем находится в корректном состоянии вне зависимости от частоты изменения данных.

Библиотеки для построения интерфейса предоставляют механизм, позволяющий по исходным данным выстраивать графический интерфейс, что в сочетании с реактивным программированием может быть использовано для автоматического обновления интерфейса.

В совокупности данные подходы позволяют автоматизировать часть задач, что упрощает работу программистов, сокращает время и стоимость разработки, а также повышает качество программного продукта.

Целью выпускной квалификационной работы является создание библиотеки для построения реактивного пользовательского интерфейса для платформы Unity.

Основные задачи:

- изучение подходов реактивного программирования;
- изучение способов построения интерфейса;
- классификация систем построения интерфейса;
- проведение сравнительного анализа продуктов аналогов;
- выделение особенностей реализуемой программы;
- проектирование библиотеки;
- реализация библиотеки;
- тестирование библиотеки;
- планирование разработки и анализ бюджета.

Предметом исследования стали системы построения реактивного пользовательского интерфейса.

Объектом исследования является пользовательский интерфейс программного обеспечения.

1 Анализ предметной области

Библиотеки для построения пользовательского интерфейса служат для упрощения разработки сложных интерфейсов. Они предоставляют набор готовых элементов, таких как текстовые поля, таблицы, списки, а также определяют подходы, используемые для реализации графической части программного обеспечения. Использование таких библиотек позволяет стандартизировать разработку и создавать модульный пользовательский интерфейс, части которого могут быть перенесены и использованы в других приложениях.

Наиболее распространенными являются два подхода для описания интерфейса: императивный и декларативный.

При императивном подходе программисту необходимо создавать отдельные элементы интерфейса, указывать их расположение на экране. В этом случае программа представляет собой алгоритм построения и обновления интерфейса, но не отображает итоговый результат, что является недостатком подхода.

Декларативный подход противоположен императивному и предполагает описание итогового представления интерфейса. Механизм создания отдельных элементов и перехода от одного состояния к другому в данном подходе скрыт от программиста. Одним из распространенных декларативных языков является HTML.

Важной задачей библиотек для построения интерфейсов является связывание объектов доменной области, описывающих внутреннее состояние программы, и их графического отображения.

Одним из способов решения этой задачи является ручное управление перерисовкой интерфейса. В этом случае после любых изменениях, влияющих на отображение интерфейса, программисту необходимо запускать специальный механизм, который будет приводить интерфейс в актуальное состояние.

Другим способом является использование библиотек для реактивного управления данными, основанных на принципах реактивного программирования.

Реактивное программирование – парадигма, в которой состояние программы представляется формулами, вычисляемыми на основе исходных данных. При реактивном подходе при изменении исходных данных происходит автоматическое перевычисление зависимого состояния.

Многие языки программирования не реализуют такой функционал на уровне языка. В этом случае требуются специальные библиотеки для реактивного управления состоянием, которые реализуют парадигму реактивного программирования.

Существует два вида реактивного программирования:

- pull-based;
- push-based.

Подход на основе проталкивания данных (push-based) реализуется с помощью реактивных потоков, которые императивно описывают трансформации данных. В этом случае, при получении нового значения поток произведет трансформирование данных и вычисленный результат будет отправлен на дальнейшую обработку.

Недостатками такого подхода для использования в качестве части системы построения интерфейса являются:

- вычисление в потоках происходит независимо от того, используются ли результаты вычисления или нет, что приводит к лишним вычислениям;
- отсутствие явного состояния системы, что может привести к пребыванию интерфейса в неопределенном состоянии;
- сложность создания динамических потоков, основанных на данных из нескольких источников, что может привести к увеличению количества ошибок и усложнению исходного кода.

В связи с этим данная модель не подходит для использования в качестве основы для создания библиотеки построения интерфейса.

Подход на основе затягивания данных (pull-based) реализуется с использованием реактивных функций, которые декларативно описывают механизм вычисления состояния, самостоятельно получают необходимые данные из источников, а результат вычисления сохраняется. Здесь вычисления происходят только при использовании данных, существует явное состояние системы и построение реактивных связей происходит автоматически, что устраняет проблемы push-based в контексте использования для библиотеки построения интерфейса.

Также, важным является то, что pull-based подход реактивного программирования может быть скомбинирован с декларативным подходом построения пользовательского интерфейса.

В проведенной классификации были рассмотрены подходы к построению интерфейса и реализации реактивного программирования. Выявлено, что для построения реактивного пользовательского интерфейса оптимальным является использование декларативного подхода к описанию требуемого результата и использование pull-based модели реактивного программирования.

2 Анализ продуктов-аналогов

Был произведен анализ библиотек для построения реактивного пользовательского интерфейса в таких областях как unity, web, android и iOS разработка. Рассмотренные в данном разделе программы являются либо полными аналогами разрабатываемого продукта, либо представляют собой комплексные решения, содержащие в том числе подсистему построения интерфейса и подсистему реактивного обновления интерфейса. В рассмотрении участвовали только библиотеки, распространяемыми под лицензиями, позволяющие использовать их как для персонального, так и для коммерческого использования.

Было выявлено несколько аналогов в смежных областях разработки. Готовых решений для реализации реактивного пользовательского интерфейса для платформы Unity выявлено не было.

2.1 Библиотека Mol

Библиотека Mol является полноценным фреймворком для построения реактивных web приложений, который состоит из множества небольших библиотек, реализующих задачи декларативного построение интерфейса, реактивного (pull-based) обновления системы, взаимодействия с сервером и множества других. Mol также предоставляет большой набор готовых компонентов для реализации интерфейса, функционал которых при необходимости может быть расширен [3].

Библиотека обладает следующими достоинствами:

- является полностью реактивной;
- полностью декларативная;
- простая конфигурация;
- высокая расширяемость путем наследования компонентов;
- обладает оптимизированным механизмом обновления;
- исходный код является модульным;
- распространяется под свободной лицензией.

Недостатки:

- поддерживается только платформа web;
- сложный синтаксис описания интерфейса;
- низкая распространенность;
- отсутствие документации.

Данная библиотеки рассчитана на web разработку и не применима для платформы Unity.

2.2 Библиотека Jetpack Compose

Jetpack Compose является набором инструментов для создания пользовательского интерфейса. Библиотека разрабатывается компанией Google Inc и является частью платформы Android. Отличительной особенностью является то, что для использования библиотеки требуется установка компилятора Compose Compiler, который расширяет функционал языка программирования Kotlin и добавляет поддержку реактивного программирования. Jetpack Compose в настоящее время находится в стадии исследований, предоставляет только «Developer Preview» версию и в текущем состоянии не рекомендуется для использования в крупных проектах [4].

Достоинства:

- поддержка реактивности на уровне языка программирования;
- является частью платформы;
- наличие встроенных стилей оформления;
- полная поддержка средой разработки Android Studio.

Недостатки:

- поддерживается только язык Kotlin;
- использует медленные Reflection методы;
- отсутствует стабильная версия;
- сложная установка и интеграция;
- для компилятора требуются обязательные аннотации в коде.

Данная библиотека использует возможности расширения языка Kotlin и не может быть использована для Unity, использующей язык C#.

2.3 Библиотека Swift UI

Swift UI разрабатывается Apple Inc как часть системы Xcode и предоставляет набор утилит для построения пользовательских интерфейсов для всех Apple платформ, таких как iOS и macOS. Библиотека предоставляет различные элементы интерфейса и элементы управления для построения интерфейса [5].

Достоинства:

- поддержка реактивности с помощью аннотаций в коде;
- является частью платформы;
- декларативный синтаксис;
- встроенная поддержка локализации;
- отображение разрабатываемого интерфейса в среде разработки;
- поддержка средой Xcode.

Недостатки:

- поддерживаются только платформы Apple;
- недоработанный программный интерфейс;
- сложная установка.

Данная библиотека может быть использована только в экосистеме Apple и не может быть использована для Unity, допускающей разработку проектов под множество платформ.

2.4 Платформа Flutter

Flutter разрабатывается Google Inc и является полноценной платформой для разработки приложений под мобильные, web и desktop платформы. Написана на языке Dart и использует для работы библиотеку Flutter Foundation [6].

Достоинства:

- декларативный синтаксис;
- простой синтаксис описания интерфейса.

Недостатки:

- не допускает встраивание во внешние системы;
- нет встроенных механизмов реактивности;
- выполняется в виртуальной машине;
- не подходит для разработки игр.

Flutter является платформой разработки и может быть использован как замена Unity, но не позволяет использование в качестве библиотеки для построения интерфейса в рамках Unity.

2.5 Сравнение готовых решений

В результате исследования были проанализированы несколько продуктов-аналогов. Все они выполняют примерно одинаковый набор задач и отличаются деталями реализации и поддерживаемыми платформами.

Готовых продуктов, которые могут быть использованы для построения интерфейса в Unity выявлено не было. На основе этого было принято решение о необходимости разработки собственной библиотеки с учетом существующих разработок в смежных областях.

Разрабатываемая в рамках выпускной квалификационной работы библиотека должна обладать следующими характеристиками:

- поддерживать все платформы (с помощью Unity);
- быть полностью реактивной;
- поддерживать реактивность с помощью аннотаций в коде;
- обладать простой конфигурацией;
- обладать расширяемостью путем наследования;
- обладать простым синтаксисом описания интерфейса;
- не использовать Reflection методы.

3 Постановка задачи

На основе данных, полученных в ходе изучения предметной области и анализа продуктов аналогов, было принято решение о необходимости разработки библиотеки для построения реактивного пользовательского интерфейса, которая будет состоять из двух подсистем:

- подсистема реактивного программирования;
- подсистема построения интерфейса.

Индивидуальные требования к подсистемам приведены в подпунктах 3.1 и 3.2. Подсистемы должны быть реализованы на языке C# для среды Unity без применения зависящих от конкретной платформы плагинов и механизма Reflection, что позволит использовать библиотеку на всех платформах поддерживаемых Unity. Подсистемы должны быть оформлены в виде модулей, которые могут быть интегрированы в любое программное обеспечение.

3.1 Требования к подсистеме реактивного программирования

Подсистему реактивного программирования необходимо реализовать с помощью pull-based подхода на основе аннотаций в коде.

Библиотека должна оперировать следующими типами данных:

- исходная ячейка;
- вычисляемая ячейка;
- реакция.

Исходная ячейка сравнима с переменными в языках программирования. Из такой ячейки можно получить текущее значение и записать новое.

Вычисляемая ячейка может быть представлена в виде функции, которая рассчитывает свое значение на основе других ячеек.

Реакции необходимы для взаимодействия программы с внешней средой. Печать содержимого ячейки в консоль или отображение значения в пользовательском интерфейсе является реакцией.

Пример использования подсистемы приведен в листинге 1.

Листинг 1 – Пример использования подсистемы реактивного программирования

```
// Создание исходной ячейки
[Atom] int Counter = 0;

// Создание вычисляемой ячейки
[Atom] int CounterSqr => Counter * Counter;

// Создание реакции
var printCounter = Atom.Reaction(delegate {
    Console.WriteLine(CounterSqr);
});

// Запуск реакции
printCounter.Run();

// Отключение реакции
printCounter.Deactivate();
```

В данном примере на основе исходной ячейки «Counter» рассчитывается значение вычисляемой ячейки «CounterSqr». Ячейки можно отличить от обычных переменных и функций по аннотации «Atom». Далее создается реакция «printCounter», выводящая сообщение в консоль при каждом изменении значения «CounterSqr».

3.2 Требования к подсистеме построения интерфейса

Подсистема построения интерфейса должна быть построена на основе декларативного подхода. Интерфейс должен быть представлен множеством элементов, таких как текстовые поля, таблицы, строки, называемых виджетами [7]. Каждый виджет представлен в виде отдельного класса. В составе библиотеки должны быть реализованы следующие базовые виджеты:

- Text (текстовое поле);
- Button (кнопка);
- Container (контейнер);
- Row (строка из нескольких виджетов);

- Column (столбец из нескольких виджетов);
- Grid (таблица из нескольких виджетов);
- Scroll List (прокручиваемый список).

Необходимо предусмотреть возможность создания пользователем собственных виджетов для расширения функционала библиотеки.

Построение виджетов должно выполняться на основе ячеек с данными, что позволит автоматически обновлять интерфейс при изменении значения в ячейках.

Пример построения простейшего интерфейса с использованием библиотеки представлен в листинге 2.

Листинг 2 – Пример использования подсистемы построения интерфейса

```
// Создание исходной ячейки
[Atom] int Counter = 0;

// Создание вычисляемой ячейки
[Atom] int CounterSqr => Counter * Counter;

// Описание интерфейса
[Atom] Widget BuildInterface () {
    return new Text (
        value: "Значение счетчика:" + CounterSqr
    );
}

// Запуск отрисовки интерфейса
var app = App.Run(BuildInterface);

// Приостановка отрисовки интерфейса
app.Deactivate();
```

В данном примере создаются ячейки «Counter» и «CounterSqr». Далее в функции BuildInterface создается текстовое поле, после чего запускается его отрисовка на экран. При изменении значения в любой из использованных для описания интерфейса ячейках пользовательский интерфейс должен быть автоматически обновлен библиотекой.

4 Анализ требований

4.1 Функциональные требования

В результате анализа поставленной задачи для каждой из подсистем разрабатываемой библиотеки были выявлены и систематизированы требования, что является одним из важнейших этапов разработки [8].

Подсистема реактивного программирования необходима для вычисления зависимостей между ячейками программы и автоматического их обновления при обновлении исходных данных. Основными функциями подсистемы, с которыми взаимодействует пользователь, являются:

- создание исходной ячейки данных;
- обновления значения исходной ячейки данных;
- получение значения исходной ячейки данных;
- создание вычисляемой ячейки данных;
- получение значения вычисляемой ячейки данных;
- создание реакции;
- запуск реакции;
- отключение реакции.

Таким образом, пользователю библиотеки требуется при запуске программы создать ячейки с данными и реакции, после чего можно изменять значения созданных ячеек.

Остальные функции подсистемы, такие как отслеживание зависимостей между ячейками, перевычисление значений и кеширование данных выполняются автоматически без управления со стороны пользователя.

Диаграмма вариантов использования для подсистемы реактивного программирования приведена на рисунке 1.

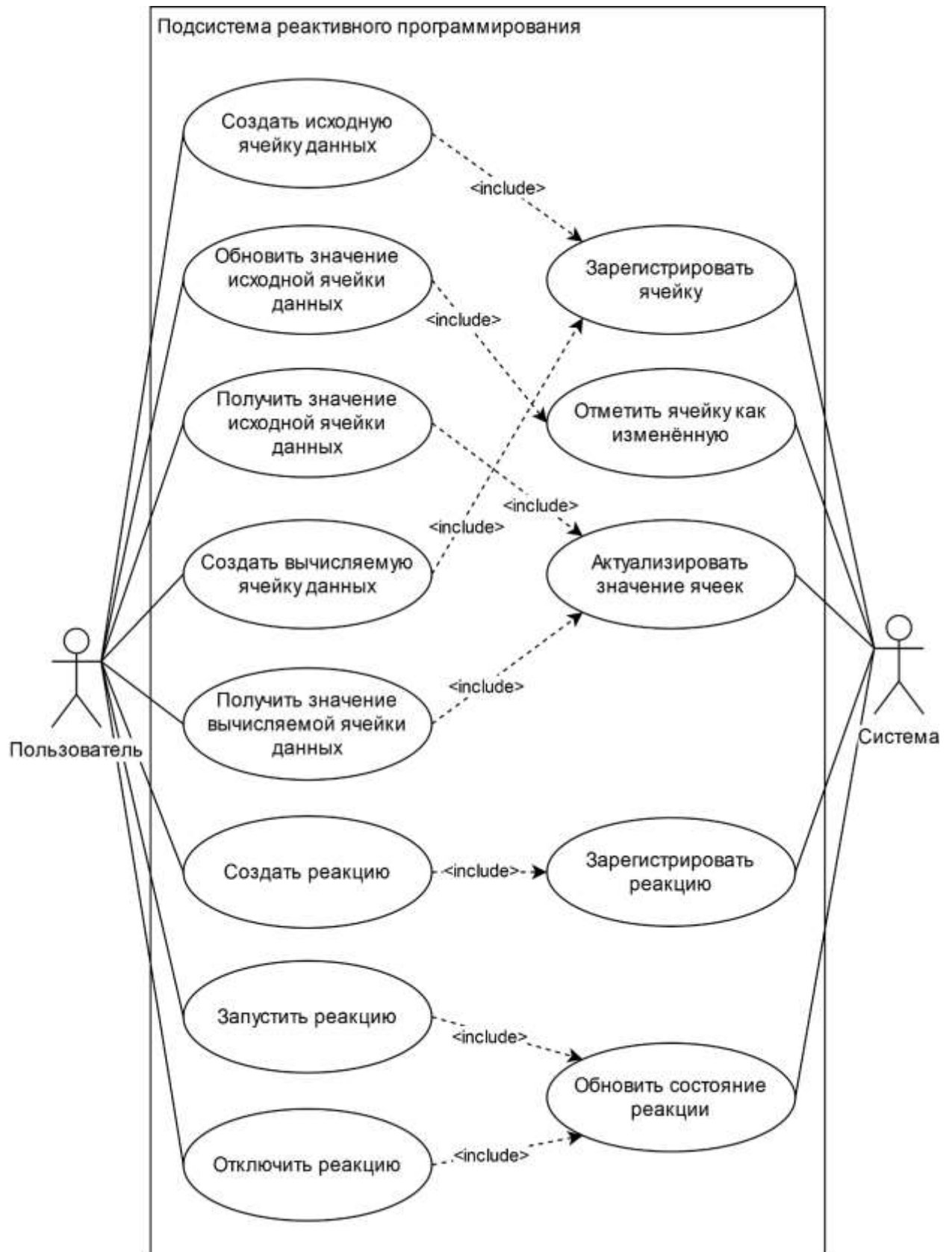


Рисунок 1 – Диаграмма вариантов использования подсистемы реактивного программирования

Диаграмма вариантов использования описывает работу системы на высоком уровне абстракции. Для уточнения алгоритмов работы системы была составлена диаграмма деятельности для варианта использования «Обновить значение исходной ячейки данных» (рисунок 2), демонстрирующая детализированное описание алгоритма.



Рисунок 2 – Диаграмма деятельности «Обновить значение исходной ячейки данных»

Подсистема построения интерфейса отвечает за преобразование виджетов в элементы графического интерфейса и их отрисовку. Она обладает следующим функционалом:

- описание интерфейса на основе ячеек с данными;
- запуск отрисовки интерфейса;
- отключение отрисовки интерфейса.

Диаграмма вариантов использования для подсистемы построения интерфейса приведена на рисунке 3.



Рисунок 3 – Диаграмма вариантов использования подсистемы построения интерфейса

При использовании подсистемы от пользователя требуется описать интерфейс программы с помощью виджетов, после чего можно запустить или приостановить его отрисовку. Остальную часть работы подсистема выполняет в автоматическом режиме.

Выделенные функции будут использованы при реализации библиотеки для построения реактивного пользовательского интерфейса как внешнее API системы.

4.2 Нефункциональные требования

Нефункциональные требования задают ограничения системы, не относящиеся к ее функциям [9].

Библиотека для построения интерфейса должна быть разработана для платформы Unity 2019.1 с использованием языка программирования C#. Разработка должна проводиться с использованием ООП.

Библиотека должна корректно отображать пользовательский интерфейс на наиболее распространенных моделях ЭВМ, использующих различные операционные системы и разрешения экрана. В качестве основы для реализации интерфейса необходимо использовать Unity UI, как наиболее распространенную систему для отрисовки графики в Unity.

Библиотека должна обладать возможностью расширения функционала, для чего необходимо предусмотреть модульную структуру исходного кода. Качество кода должно быть подтверждено метриками.

Каждая разрабатываемая подсистема должна быть скомпонована в виде пакета для Unity Package Manager, что позволит устанавливать и использовать библиотеку в любом проекте, использующем Unity версии 2019.1 или выше.

4.3 Технологии и средства разработки

Разработка библиотеки будет проводиться на ЭВМ с установленной операционной системой Windows 10, которая является наиболее распространенной при разработке компьютерных игр, процессором Intel Core i7 с поддержкой SSE2 набора команд процессора, графический процессор GTX 1050 с поддержкой DirectX10. Данные требования являются необходимым для работы Unity.

Кодирование будет проводиться в JetBrains Rider. Особенностью этой среды разработки являются высокая степень интеграции с Unity, что значительно упрощает разработку.

Тестирование библиотеки будет проводиться на используемом для разработки ПК, а также на смартфоне с установленной ОС Android. Для

компиляции бинарных файлов под эту платформу требуются установленные и настроенные Java SDK и NDK.

Минимальные требования к аппаратному окружению в дальнейшем будут использованы для выбора средств разработки программного продукта.

Для разработки будет использована методология RUP. Особенностью этой методологии является наличие итераций. В конце каждой итерации получается новая версия готового продукта. Каждая итерация занимает несколько недель, после чего идет этап тестирования и интеграции. Это позволяет контролировать качество разрабатываемой библиотеки, а также подстраивать процесс разработки под требования, которые могут часто меняться. Для моделирования будет использован язык UML, который описывает работу системы в понятном графическом виде. Это позволяет организовать совместную работу над реализацией библиотеки несколькими разработчиками. UML диаграммы будут создаваться в онлайн-редакторе diagrams.net.

Для планирования разработки библиотеки, анализа бюджетов и контроля загруженности ресурсов будет использоваться Microsoft Project.

5 Проектирование библиотеки

5.1 Проектирование подсистемы реактивного программирования

Основой подсистемы является класс `AtomBase`, который реализует функционал реактивной ячейки. Главными элементами данного класса являются списки `subscribers` и `children`.

`Subscribers` – список ячеек, использующих значение текущей ячейки. При изменении значения в ячейке, все элементы этой коллекции будут уведомлены о изменениях.

`Children` – список ячеек, которые были использованы при вычислении значения текущей ячейки. Данная коллекция нужна для возможности отменить отслеживание изменений используемых ячеек при приостановке текущей ячейки. Этот список является обратным к списку `subscribers`. Это означает, что если в ячейке «А» в списке `subscribers` есть ячейка «Б», то в списке `children` ячейки «Б» будет находится ячейка «А».

Данные два списка выстраивают неявный граф зависимостей. Благодаря этому при изменении значения любой ячейки данных в системе можно найти все ячейки, которые вычисляются на основе этой ячейки, и выполнить перевычисление их значения, что приведет к актуализации состояния системы.

Пример графа зависимостей для формулы изображен на рисунке 4.

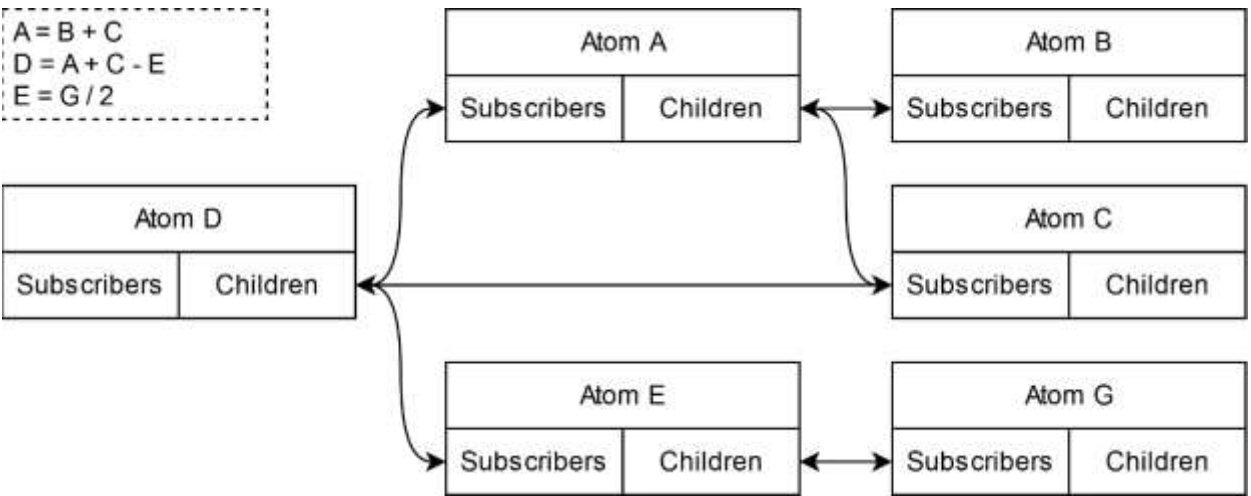


Рисунок 4 – Схема зависимостей ячеек для формулы

При каждом обновлении структура графа перестраивается, что позволяет использовать в ячейках не только простые формулы, но и любые сложные конструкции доступные в языке программирования, такие как условия и циклы.

Пример условного оператора и построенного для него графа приведен на рисунке 5. Как можно заметить, в случае если переменная В равна 0, то значение D не используется. В противоположном случае, если В не равно 0, то не используется значение С. Связи между неиспользуемыми ячейками полностью исключаются из графа, что позволяет сократить значительную часть лишних вычислений.

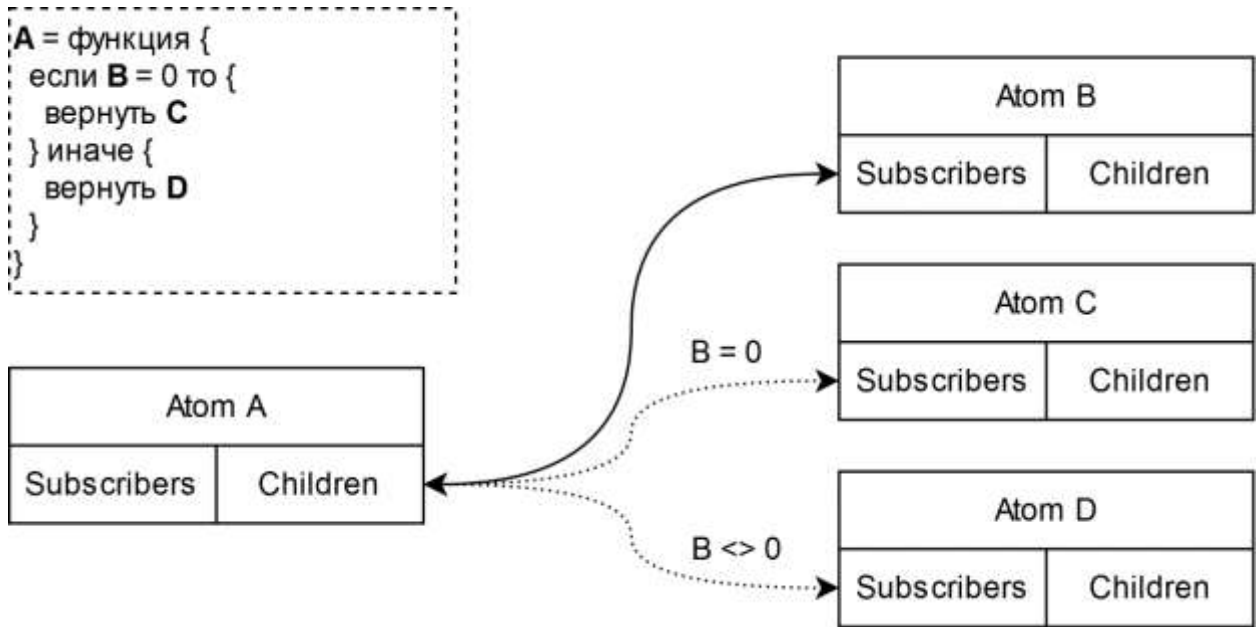


Рисунок 5 – Схема зависимостей для условного оператора

При вычислении могут появиться циклические зависимости между ячейками. Такая ситуация не может быть корректно обработана. Этот случай отдельно отслеживается системой, и в случае наличия циклов будет выброшено исключение «Cyclic Atom Dependency Exception».

На основе списков subscribers и children происходит выполнение всех механизмов по отслеживанию изменений, а также активация и деактивация ячеек.

Для обновления значений требуется знать состояние ячеек. Каждая ячейка может находиться в одном из следующих состояний: actual, obsolete, checking, pulling.

Состояние actual означает что значение ячейки находится в актуальном состоянии и его обновление не требуется.

Состояние obsolete означает что значение ячейки устарело и при следующей попытке получить ее значение оно будет вычислено повторно. Ячейка может устареть в случае, если ее значение будет указано из вне.

Состояние checking является подвидом состояния obsolete и означает, что хранящееся в ячейке значение могло устареть. Ячейка переходит в это состояние при устаревании одной или более из ее зависимостей. При попытке получить значение такой ячейки она перейдет либо в состояние actual, если изменение значений зависимостей не привело к изменению значения самой ячейки, либо в состояние obsolete.

Состояние pulling присваивается ячейке во время вычисления значения и необходимо для отслеживания циклических зависимостей.

Для формулы, изображенной на рисунке 4, была составлена схема распространения изменений при обновлении значения ячейки «С» (рисунок 6). В этом случае состояние ячейки «С» будет установлено в obsolete, а состояния ячеек «А» и «D» в checking. При этом изменение «С» не повлияло на ячейки «В», «Е» и «G», и они не будут вычисляться заново.

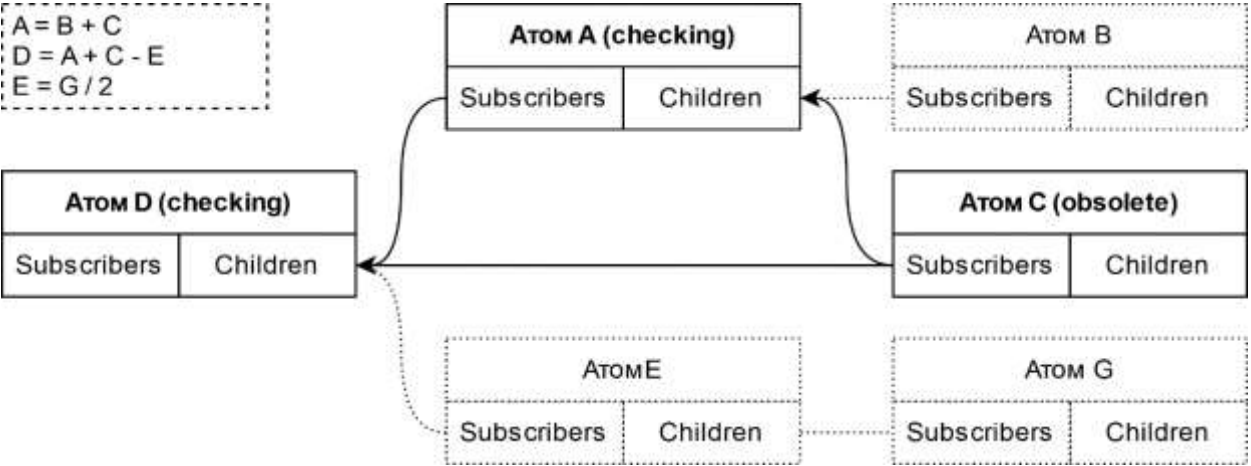


Рисунок 6 – Схема распространения изменений

На основе базового абстрактного класса AtomBase были созданы три реализации.

Диаграмма классов представлена на рисунке 7.

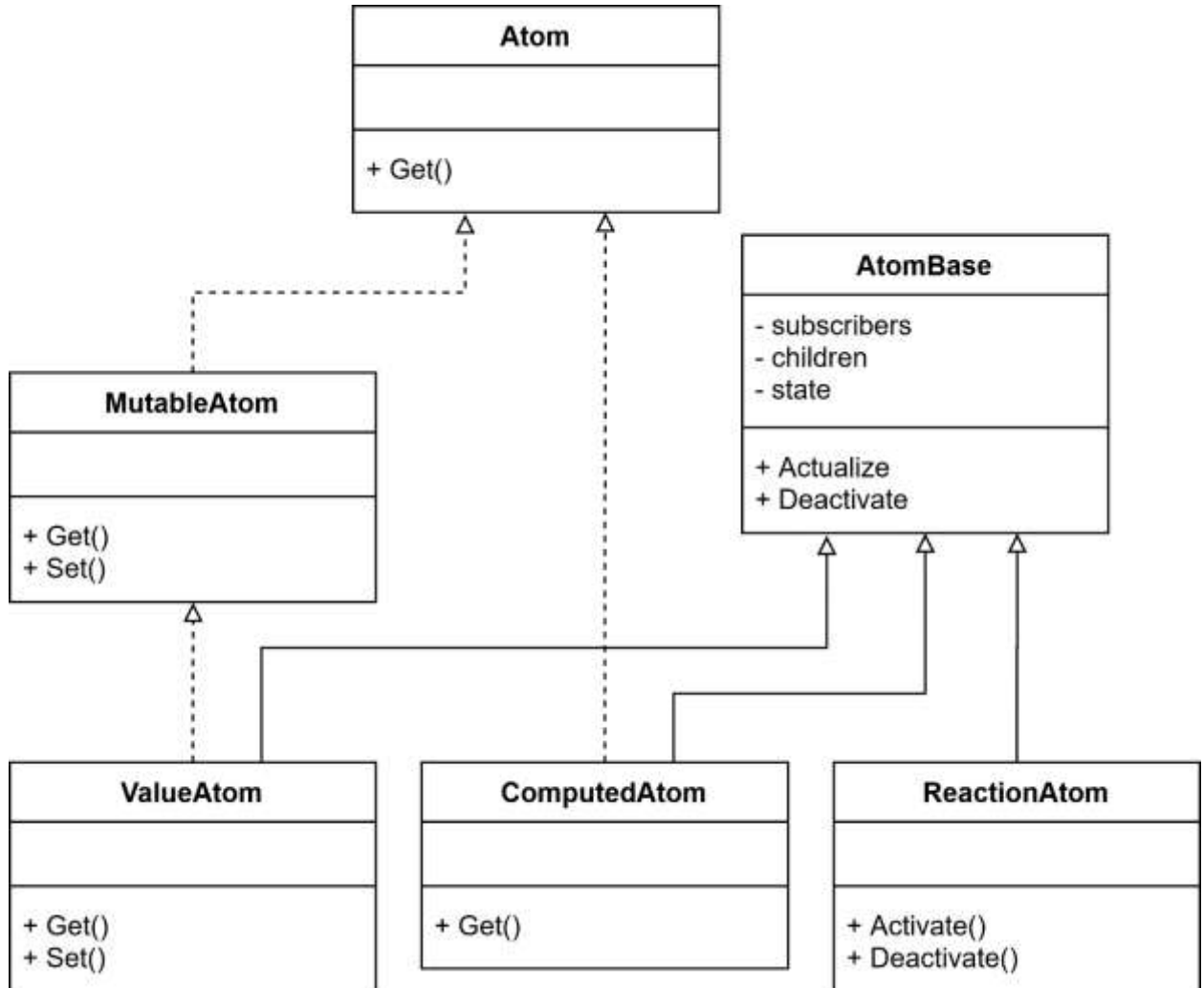


Рисунок 7 – Диаграмма классов ячеек

ValueAtom представляет исходную ячейку данных. Значение такой ячейки может быть установлено из вне. Отличительной особенностью таких ячеек является то, что их список children всегда пустой.

ComputedAtom представляет вычисляемую ячейку данных. Для создания такой ячейки требуется указать функцию, по которой будет вычисляться значение. Изменить значение такой ячейки можно только путем изменения ее зависимостей.

ReactionAtom представляют реакции. В отличие от исходных и вычисляемых ячеек, реакции необходимы для выполнения каких-либо действий. Реакции не возвращают значение, и не могут быть использованы из других ячеек, что означает что их список subscribers будет всегда пуст. Запуск и отключение реакций требуется производить вручную.

Для упрощения внешнего API библиотеки были выделены два интерфейса: Atom и MutableAtom.

Atom представляет ячейку данных, доступную только для чтения и содержит метод Get для получения значения.

MutableAtom реализует функционал интерфейса Atom, и в дополнение предоставляет метод Set для изменения значения ячейки.

Введение данных интерфейсов позволило ограничить область видимости классов ValueAtom, ComputedAtom и ReactionAtom до уровня библиотеки и скрыть их программистов, использующих библиотеку в своих проектах.

5.2 Проектирование подсистемы построения интерфейса

Подсистема построения интерфейса состоит из множества классов, представляющих отдельные элементы пользовательского интерфейса. Каждый элемент состоит из трех частей: View, State и Widget.

View отвечает за отрисовку элемента пользовательского интерфейса. Наследует MonoBehaviour, который является базовым классом для скриптов, используемых в Unity. Это позволяет настраивать переменные класса в редакторе Unity. Для отрисовки View использует данные, предоставляемые объектом State.

State отвечает за преобразование данных доменной области в данные, пригодные для отображения в интерфейсе. Также в этом классе могут храниться данные, описывающие текущее состояние интерфейса, такие как отступ в прокручиваемом списке или номер выделенной строки в таблице.

Widget необходим для описание интерфейса пользователем библиотеки путем построения дерева виджетов, которое в дальнейшем будет преобразовано системой в наборы View для отображения на экране.

Общая схема взаимодействия этих частей выглядит следующим образом: пользователь передает в библиотеку Widget, который необходимо отрисовать. Далее, система по Widget создает объект State для элемента. По объекту State создается View, который отображается на экране.

Детализированная диаграмма последовательности представлена на рисунке 8.

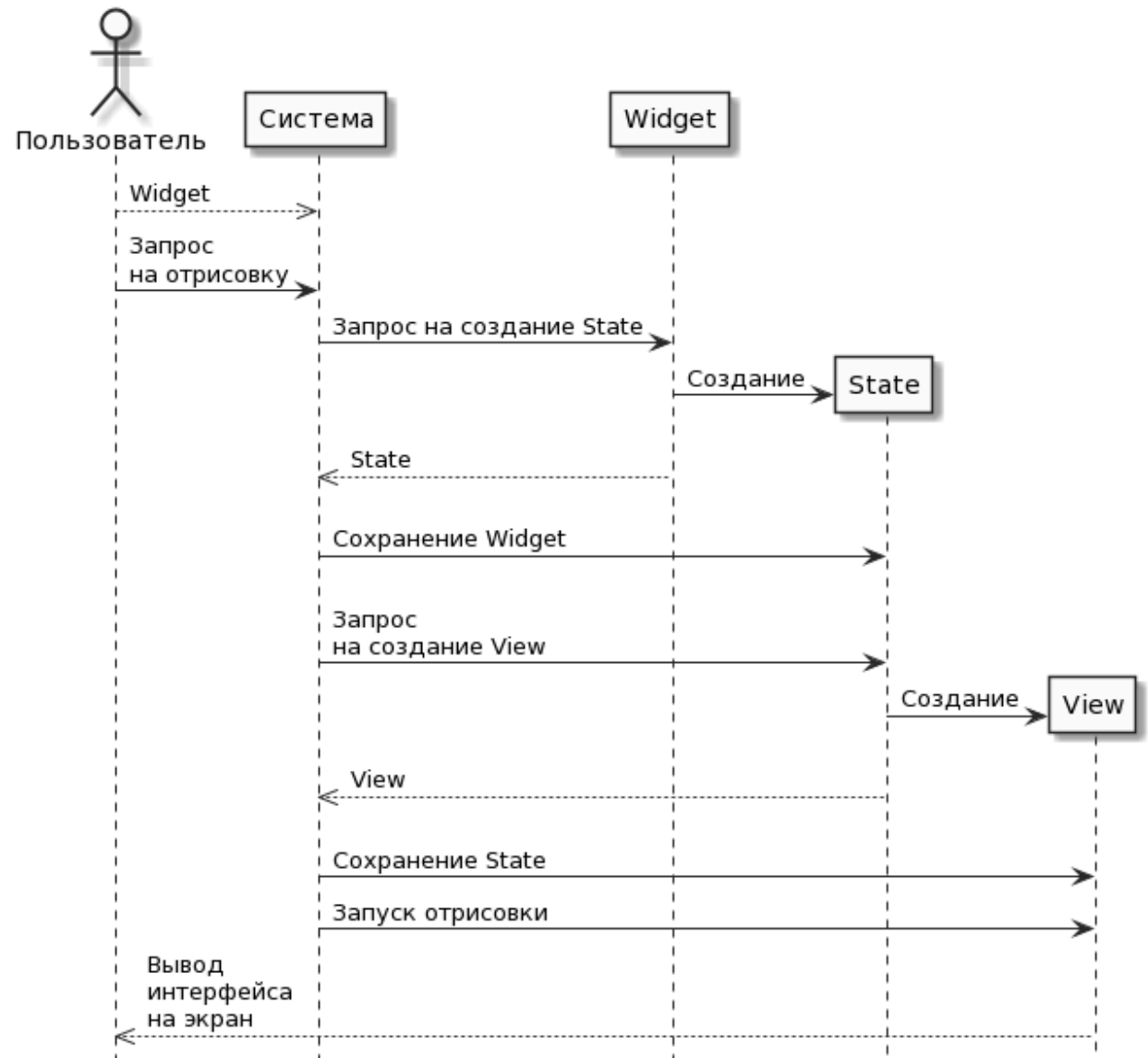


Рисунок 8 – Диаграмма последовательности

Разделение на Widget, View и State необходимо по следующим причинам.

Во-первых, дерево виджетов перестраивается при любом изменении ячеек с данными. Так как это может происходить очень часто, необходимо чтобы Widget занимал как можно меньше оперативной памяти и хранил в себе только минимальную информацию, которой будет достаточно для отрисовки элемента. Из-за этого данный класс является DTO, то есть хранит в себе только данные. Это позволяет часто пересоздавать объекты без снижения производительности системы. При этом, система сопоставляет пересоздаваемые виджеты с уже существующими, что гарантирует что для одного элемента интерфейса будет создан только один экземпляр State.

Во-вторых, для отрисовки элементов используется Unity UI. В этой библиотеке создание и отображение элементов является ресурсоемкой задачей и при неправильном использовании может привести к низкой производительности системы. Из-за этого разрабатываемая библиотека автоматически скрывает невидимые элементы интерфейса для уменьшения количества отображаемых элементов и при возможности повторно их использует в других частях интерфейса для минимизации количества операций по созданию и удалению объектов.

Таким образом, для одного элемента пользовательского интерфейса создается один экземпляр State, может быть создан один или больше экземпляров Widget в зависимости от частоты обновления данных, и один или ни одного объекта View в зависимости от того, отображается ли в данный момент элемент на экране.

Для каждого элемента пользовательского интерфейса реализуются три класса, являющиеся наследниками View, State и Widget соответственно. В данных классах реализуется специфический для конкретного элемента функционал.

Диаграмма классов, унаследованных от базового класса Widget представлена на рисунке 9.

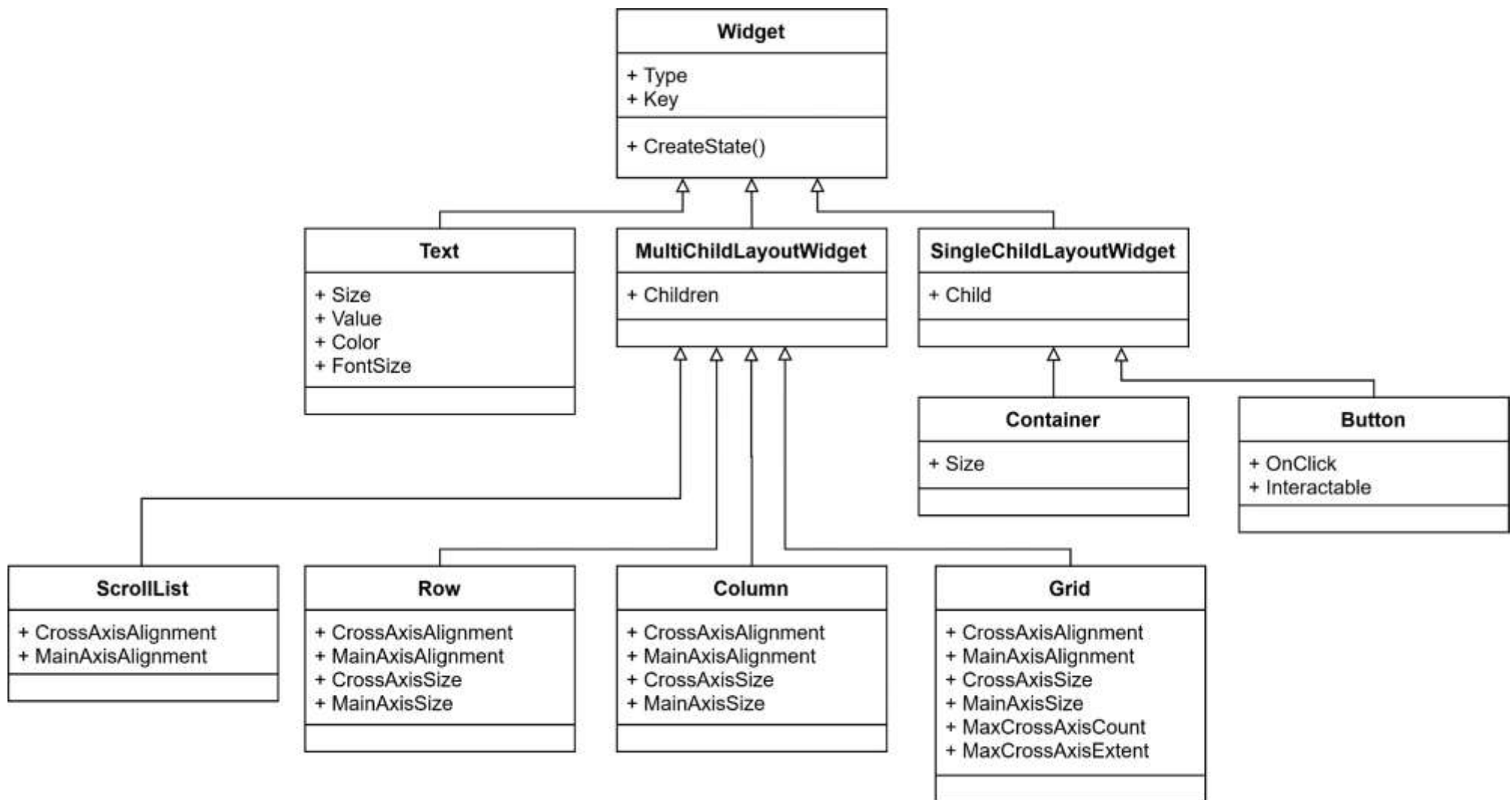


Рисунок 9 – Диаграмма классов виджетов

Согласно задачам, поставленным на выпускную квалификационную работу, были спроектированы шесть реализаций виджетов: Text, Button, Container, Row, Column, Scroll List.

Для упрощения реализации виджетов-контейнеров были выделены абстрактные классы SingleChildLayoutWidget и MultiChildLayoutWidget.

SingleChildLayoutWidget является базовым для виджетов, отображающих один вложенный элемент. Данный класс отвечает за хранение состояния и отображение вложенного элемента, что позволяет устранить дублирование кода при реализации данного функционала в классах-наследниках. Примерами виджетов, реализующих данный класс, являются контейнер и кнопка.

MultiChildLayoutWidget аналогичен SingleChildLayoutWidget, за исключением того, что данный класс необходим для реализации виджетов, отображающих несколько элементов, а не один. Данный класс наследуют такие виджеты как список, колонка, таблица, прокручиваемый список.

В случае если виджет является примитивным, то есть не содержит вложенных виджетов, выполняется наследование от базового класса Widget.

При необходимости, пользователь библиотеки может реализовать собственные виджеты. Для этого необходимо создать класс, являющийся наследником одного из базовых классов, предоставляемых библиотекой. За счет этого достигается высокая степень расширяемости библиотеки.

Аналогично иерархии наследования классов, реализующих Widget, реализованы иерархии наследования для классов State и View.

Спроектированная архитектура подсистем будет использована при реализации библиотеки для построения реактивного пользовательского интерфейса.

6 Реализация библиотеки

На основе спроектированной архитектуры была реализована библиотека для построения реактивного пользовательского интерфейса.

Библиотека разделена на следующие программные модули:

- Core (реактивное программирование);
- UI (построение интерфейса).

Core модуль полностью независим от UI и может быть использован отдельно. Это позволяет применять библиотеку в приложения, построенных на основе многоуровневой архитектуры. В таких приложениях реактивное программирование может быть использовано во всех слоях программы, а использование подсистемы построения интерфейса будет ограничено в пределах слоя представления.

Исходный код подсистемы реактивного программирования был разделен на Runtime и Editor сборки.

Runtime часть выполняется во время исполнения скомпилированной программы. В данной сборке содержится основной код подсистемы. Editor часть выполняется только в режиме редактора и содержит код, отвечающий за модификацию кода.

Диаграмма пакетов представлена на рисунке 10 [13].

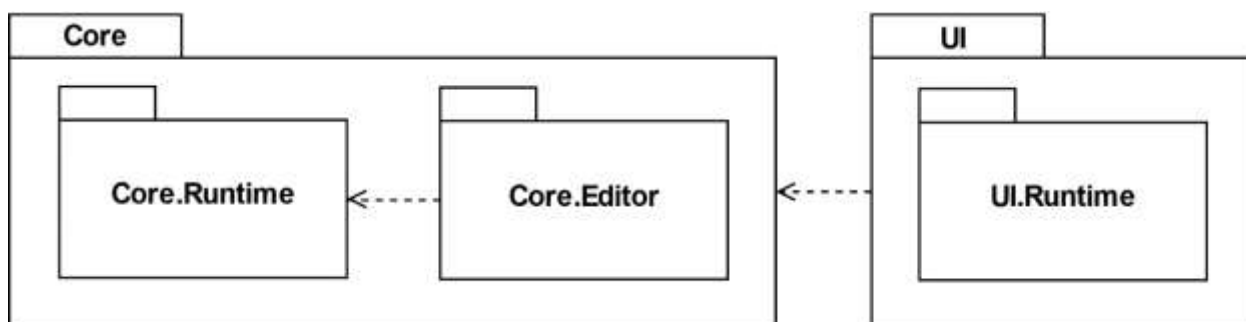


Рисунок 10 – Диаграмма пакетов

Для каждого пакета был создан файл `package.json`, позволяющий распространять библиотеку через Unity Package Manager. Данная утилита в настоящее время широко распространена для установки пакетов в Unity.

6.1 Реализация подсистемы реактивного программирования

В ходе реализации подсистемы реактивного программирования был создан ряд классов. Описание основных классов приведено в таблице 1.

Таблица 1 – Описание основных классов

Название	Назначение
Atom	Интерфейс ячейки, доступной только для чтения
MutableAtom	Интерфейс ячейки, доступной для чтения и модификации
AtomBase	Базовый класс для реализации реактивных ячеек
ValueAtom	Класс, представляющий исходную ячейку данных
ComputedAtom	Класс, представляющий вычисляемую ячейку данных
ReactionAtom	Класс, представляющий реакцию
AtomFactory	Класс, предоставляющий методы создания ячеек
AtomWeaver	Класс, выполняющий модификацию исходного кода
AtomAttribute	Класс для аннотации реактивных ячеек в исходном коде

Классы AtomBase, ValueAtom, ComputedAtom и ReactionAtom реализованы согласно описанию, представленному в предыдущей главе.

Область видимости классов ValueAtom, ComputedAtom и ReactionAtom ограничена библиотекой, поэтому для их использования вне библиотеки был создан класс-фабрика AtomFactory, отвечающий за создание экземпляров ячеек. Данный класс скрывает логику по созданию ячеек от пользователя библиотеки, и передает пользователям ячейки по их интерфейсам Atom и MutableAtom.

Отдельно следует рассмотреть классы AtomAttribute и AtomWeaver.

Согласно требованиям, предъявляемым к подсистеме реактивного программирования, реактивное программирование должно быть реализовано с помощью аннотаций в коде.

Для реализации этого требования был создан атрибут `AtomAttribute`, который необходим для добавления в код программы метаданных о реактивных ячейках.

Для преобразования аннотаций, содержащих метаданные, в программный код был реализован класс `AtomWeaver`, использующий библиотеку `Mono.Cecil` [10]. Данная библиотека позволяет выполнять модификацию промежуточного кода, компилируемого средой .NET [11].

Пример исходного кода программы представлен в листинге 3. Во время компиляции данный код будет преобразован в код, представленный в листинге 4. Как видно из листингов, при использовании аннотаций код получается значительно компактнее и проще для понимания.

Все модификации происходят на уровне промежуточного кода, не видны пользователю библиотеки и не влияют на исходные C# файлы.

Для просмотра финального кода, полученного после модификации программы, была использована программа `ILSpy`, позволяющая конвертировать промежуточный код в код на языке C# [12].

Листинг 3 – Код программы до модификации

```
[Atom] public int A {
    get => B + C;
}
```

Листинг 4 – Код программы после модификации

```
private MutableAtom<int> reactive_A = Atom.Computed<int>(Get_A);

public int A {
    get => reactive_A.Value;
}

private int Get_A () {
    return B + C;
}
```

Таким образом, модификация промежуточного кода позволяет преобразовать переменные C# в реактивные ячейки, что уменьшает количество исходного кода, которое необходимо писать пользователям библиотеки.

6.2 Реализация подсистемы построения интерфейса

При реализации подсистемы построения интерфейса был создан ряд классов. Описание основных классов приведено в таблице 2.

Таблица 2 – Описание основных классов

Название	Назначение
Widget	Базовый класс для описания виджета
State	Базовый класс для хранения состояния виджета
View	Базовый класс для отображения виджета на экране
Column	Класс для описания виджета «Колонка»
ColumnState	Класс для хранения состояния виджета «Колонка»
ColumnView	Класс для отображения виджета «Колонка»
Container	Класс для описания виджета «Контейнер»
ContainerState	Класс для хранения состояния виджета «Контейнер»
ContainerView	Класс для отображения виджета «Контейнер»
Grid	Класс для описания виджета «Таблица»
GridState	Класс для хранения состояния виджета «Таблица»
GridView	Класс для отображения виджета «Таблица»
Row	Класс для описания виджета «Строка»
RowState	Класс для хранения состояния виджета «Строка»
RowView	Класс для отображения виджета «Строка»
Text	Класс для описания виджета «Текст»
TextState	Класс для хранения состояния виджета «Текст»
TextView	Класс для отображения виджета «Текст»
Button	Класс для описания виджета «Кнопка»

Продолжение таблицы 2

Название	Назначение
ButtonState	Класс для хранения состояния виджета «Кнопка»
ButtonView	Класс для отображения виджета «Кнопка»
ScrollList	Класс для описания виджета «Прокручиваемый список»
ScrollListState	Класс для хранения состояния виджета «Прокручиваемый список»
ScrollListView	Класс для отображения виджета «Прокручиваемый список»
SingleChildLayoutWidget	Базовый класс виджета, содержащего один элемент
SingleChildLayoutState	Базовый класс для хранения состояния виджета, содержащего один элемент
MultiChildLayoutWidget	Базовый класс виджета, содержащего несколько элементов
MultiChildLayoutState	Базовый класс для хранения состояния виджета, содержащего несколько элементов
PooledViewMapper	Класс, отвечающий за обновление и отображение коллекций из нескольких элементов

Согласно спроектированной архитектуре, для каждого виджета было создано три класса, наследующие Widget, State и View.

Для отображения сложных виджетов, таких как колонка, строка, таблица и прокручиваемый список был реализован специальный класс PooledViewMapper. Данный класс отвечает за создание элементов графического интерфейса. Отличительной особенностью класса является то, что все элементы интерфейса после их отключения не удаляются, а переносятся в специальный список, из которого могут быть в дальнейшем использованы повторно. Благодаря такой оптимизации, из-за устранения лишних операций по созданию и удалению элементов, снижается нагрузка на систему что приводит к увеличению производительности библиотеки.

Исходный код классов приведен в приложении Б.

7 Тестирование и анализ качества

7.1 Тестирование программного обеспечения

Тестирование – процесс выполнения программы на заданном множестве данных, необходимый для проверки соответствия полученных результатов с ожидаемыми результатами [14].

Процесс тестирования направлен на выявление возможных ошибок в программе и позволяет проверить корректность реализации программного обеспечения. Тестирование направлено на выявление ошибок во время разработки, что уменьшает вероятность возникновения ошибок у конечных пользователей.

В дополнение к корректности реализации, важным показателем программы является сложность ее исходного кода. Увеличение сложности кода ведет к усложнению дальнейшей модификации. Для вычисления сложности программы служат метрики кода.

Метрики необходимы для количественной оценки качества программного обеспечения, повышения объективности и оценки динамики изменений [15, 16].

В рамках выпускной квалификационной работы было проведено тестирование основных задач разработанной библиотеки, таких как:

- перевычисление значений ячеек с данными;
- отображение элементов интерфейса;
- обновление элементов интерфейса.

Для тестирования были использованы метод причинно-следственных связей, автоматизированное тестирование, структурное тестирование и тестирование интерфейса.

Согласно заданию на разработку библиотеки для построения реактивного пользовательского интерфейса были вычислены метрики качества кода.

7.1.1 Метод причинно-следственных связей

Техническое задание: реализовать виджет-контейнер для управления размером относительно одного вложенного элемента.

Описание: тестируемая часть исходного кода является реализацией виджета Container, который служит для хранения вложенного элемента и позволяет менять свой размер. Он принимает два необязательных аргумента:

- размер (size);
- экземпляр вложенного виджета (child).

Если ни один из аргументов не указан, то виджет должен быть нулевого размера и не должен отображаться на экране. Если указан аргумент «size», то виджет должен занимать на экране указанное количество пространства вне зависимости от того, указан аргумент «child» или нет. Если указан только аргумент «child», размер виджета-контейнера должен совпадать с размером вложенного виджета.

По описанию была составлена функциональная диаграмма (рисунок 11).

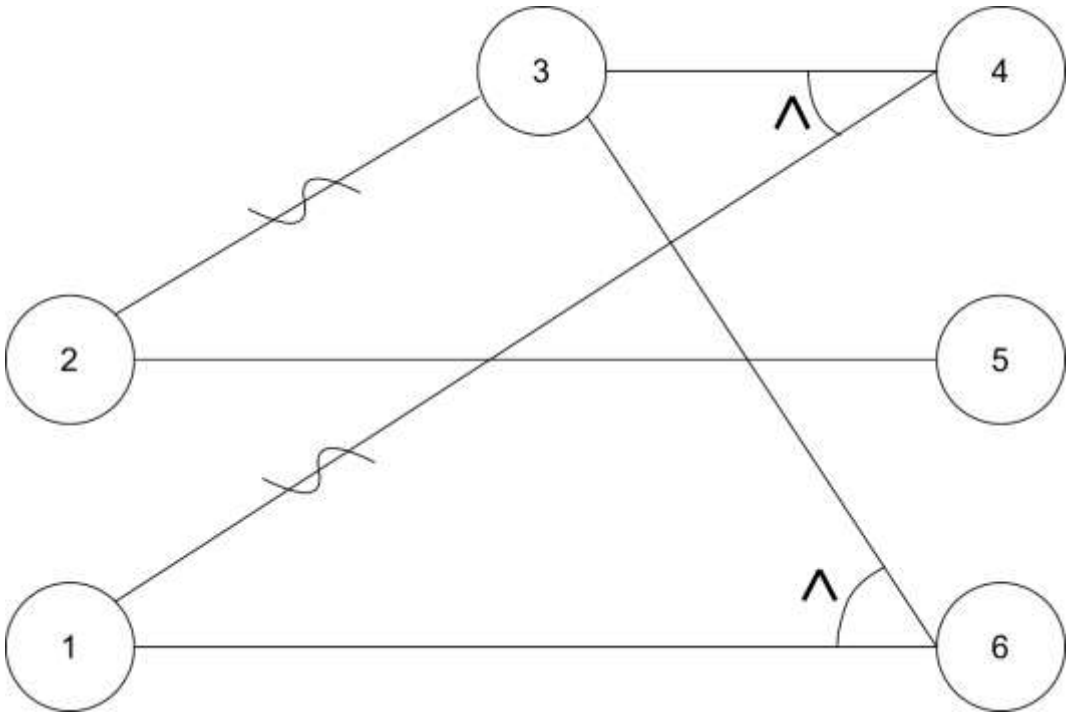


Рисунок 11 – Функциональная диграмма

Цифровые обозначения на диаграмме:

1 – указан вложенный элемент интерфейса;

2 – указан размер элемента интерфейса;

3 – не указан размер элемента интерфейса;

4 – элемент нулевого размера;

5 – элемент фиксированного размера;

6 – размер элемента совпадает с размером вложенного элемента.

По диаграмме построена таблица решений (таблица 3), которая отображает связь между входными данными и предполагаемым результатом.

Таблица 3 – Таблица решений

Входные данные	1	0	0	1	1
	2	0	1	0	1
Предполагаемый результат	4	1	0	0	0
	5	0	1	0	1
	6	0	0	1	0

По таблице решений был составлен набор тестов (таблица 4).

Таблица 4 – Тесты на основе таблицы решений

Входные данные	Предполагаемый результат	Полученный результат
Container()	Элемент нулевого размера	Элемент нулевого размера
Container(size: Size(w: 200, h: 10))	Элемент фиксированного размера	Элемент фиксированного размера
Container(child: Image(...))	Размер элемента совпадает с размером вложенного элемента	Размер элемента совпадает с размером вложенного элемента
Container(size: Size(w: 200, h: 10), child: Image(...))	Элемент фиксированного размера	Элемент фиксированного размера

С помощью составленного набора тестов было проведено тестирование виджета для управления размером относительно вложенного элемента. Результаты тестирования приведены в приложении В.

7.1.2 Автоматизированное тестирование

Для подсистемы реактивного программирования, являющейся частью разработанной библиотеки для построения реактивного пользовательского интерфейса, был составлен ряд unit-тестов. Тестирование осуществлялось с помощью утилиты Test Runner, поставляемой вместе с Unity3d [17].

Код теста для проверки автоматического изменения вычисляемых значений при изменении зависимостей приведен в листинге 5.

Листинг 5 – Код теста автоматического обновления значения

```
[Test]
public void AutoRun() => TestZone.Run(tick => {
    // установка начального значения
    var src = Atom.Value(0);

    // счетчик обновлений
    int runs = 0;
    var runner = Atom.AutoRun(() => {
        // подписка на изменение значения src
        src.Get();
        // увеличение счетчика обновлений
        ++runs;
    });

    // вызов метода AutoRun приводит к одному обновлению
    Assert.AreEqual(1, runs);

    // изменение src приводит к обновлению runner
    src.Value++;
    tick(time: 0);
    Assert.AreEqual(2, runs);

    // остановка выполнения
    runner.Dispose();

    // изменение src не влияет на runner после остановки runner
    src.Value++;
    tick(time: 0);
    Assert.AreEqual(2, runs);
});
```

Описание основных unit-тестов приведено в таблице 5. Исходный код основных unit-тестов приведен в приложении Г.

Таблица 5 – Описание unit-тестов

Название	Описание	Статус
NoSubscribed	При получении значения ячейки вне реактивного контекста запуск механизма отслеживания изменений не осуществляется	Пройдено
AutoUnsubscribed_Reaction	При деактивации ячейки происходит очищение списка зависимостей	Пройдено
AutoActivation	При появлении зависимых ячеек происходит автоматическая активация ячейки	Пройдено
Invalidate	При изменении зависимостей происходит перевычисление значения зависимой ячейки	Пройдено
Caching	При получении значения активной ячейки результат вычисления берется из кэша	Пройдено
Lazy	Вычисление значения ячейки производится только при появлении зависимых ячеек	Пройдено
ActualizedInInitialOrder	Перевычисление значения ячейки происходит в детерминированном порядке	Пройдено
EqualStateIgnored	Обновление значения ячейки на идентичное не приводит к перевычислению зависимых ячеек	Пройдено
InstantActualization	Ячейка производит мгновенную актуализацию значения при обращении вне зависимости от состояния зависимых ячеек	Пройдено

Результаты выполнения тестов отображены на рисунке 12.

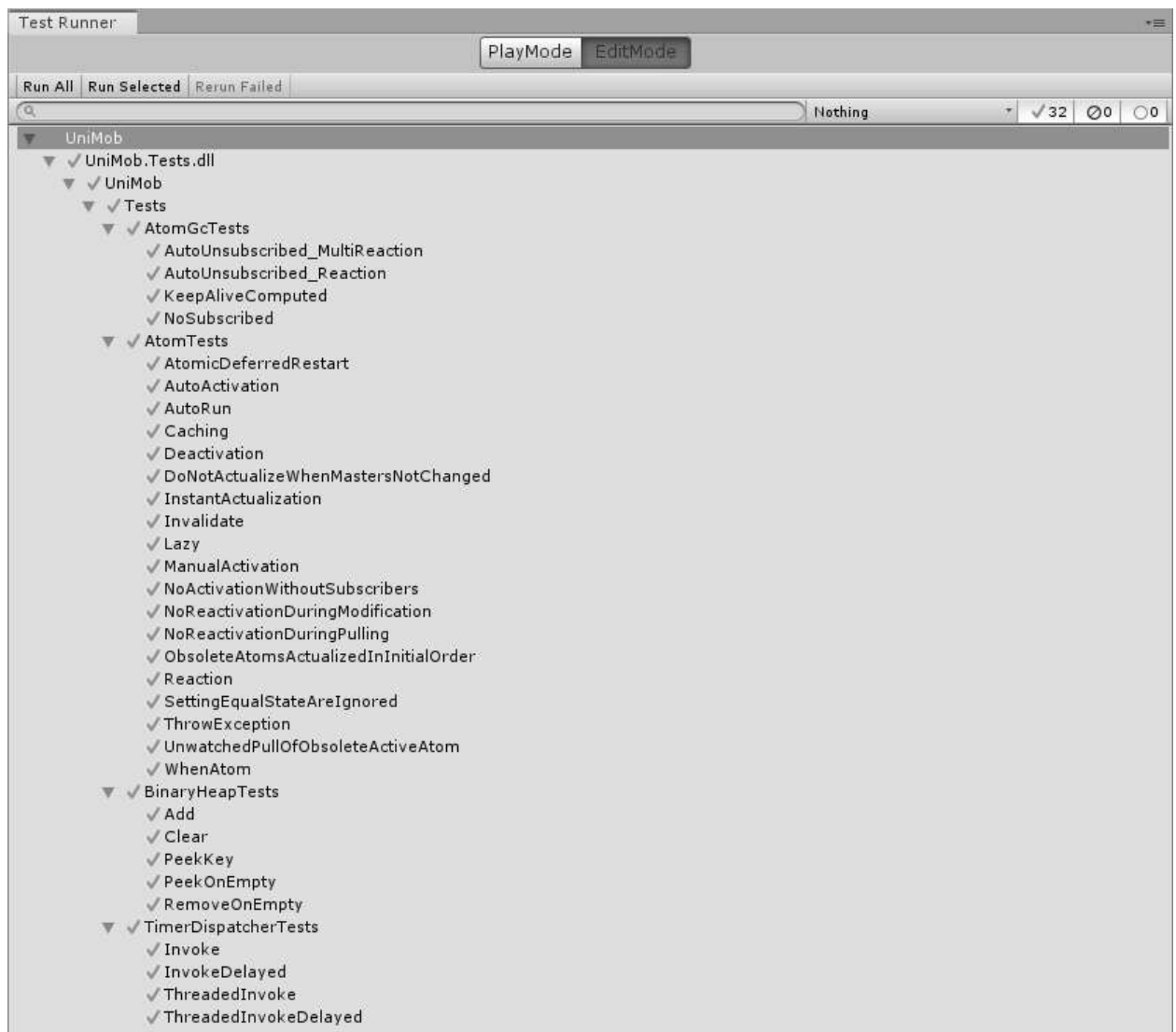


Рисунок 12 – Результаты выполнения unit-тестов

Итогом выполнения данной части работы стало покрытие кода программы unit-тестами.

7.1.3 Структурное тестирование

С помощью структурного тестирования был проверен участок кода, отвечающий за обновление элемента интерфейса (листинг 6). Данный метод принимает два аргумента:

- объект «State», содержащий данные о текущем состоянии;
- объект «Widget», описывающий требуемое состояние.

Если при вызове функции аргумент `child` не указан, то по описанию создается новое состояние с помощью метода `CreateState`. Если старый виджет равен новому, что означает что изменений не было, тогда метод не производит никаких вычислений. Если старый виджет не совпадает с новым, но может быть обновлен, то выполняется обновление с помощью метода `Update`. Иначе, если старый виджет не совпадает с новым и не может быть обновлен, происходит уничтожение старого состояния вызовом метода `Dispose` и создание нового.

Листинг 6 – Обновление элемента интерфейса

```
State UpdateChildElement([CanBeNull] State child,
    [NotNull] Widget nextWidget) {

    if (child != null) { // 1
        if (child.RawWidget == nextWidget) { // 2
            return child; // 3
        }

        if (child.RawWidget.Key == nextWidget.Key &&
            child.RawWidget.Type == nextWidget.Type) { // 4
            child.Update(nextWidget); // 5

            return child; // 6
        }

        child.Dispose(); // 7
    }

    var nextChild = nextWidget.CreateState(); // 8

    nextChild.Update(nextWidget); // 9
    nextChild.InitState(); // 10

    return nextChild; // 11
} // 12 (end)
```

Для функции `UpdateChildElement` был составлен граф потока исполнения (рисунок 13). Значение цифровых обозначений отмечено комментариями к коду (смотри листинг 6).

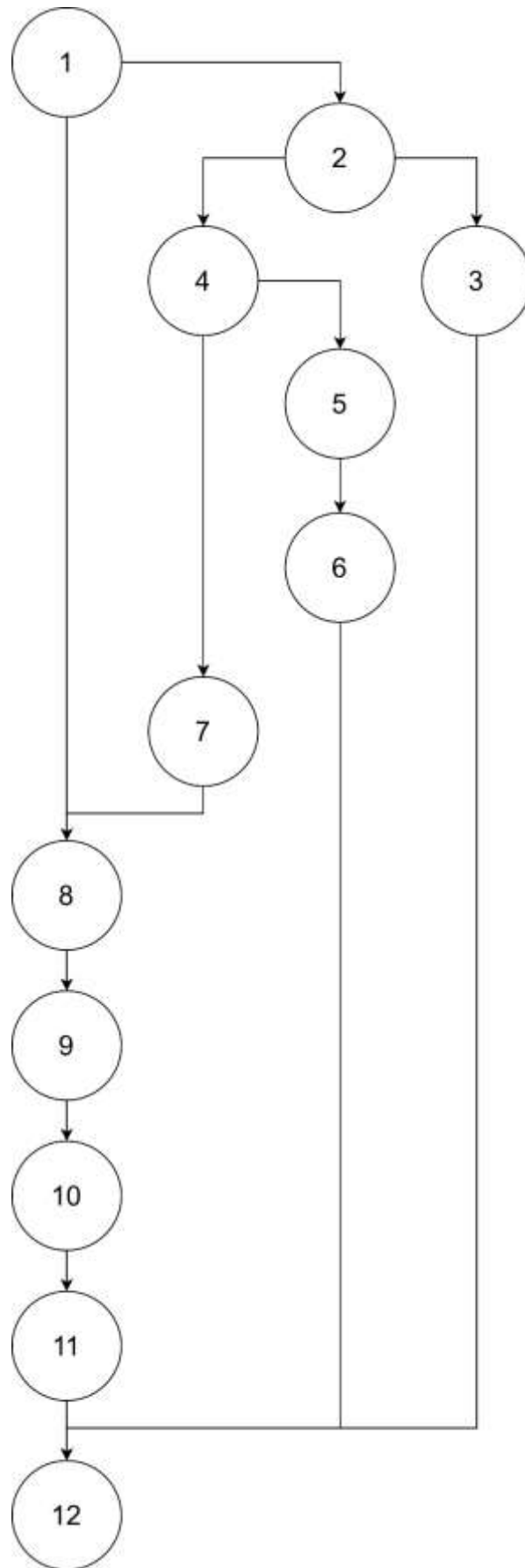


Рисунок 13 – Граф потока управления

Для проведения структурного тестирования был составлен ряд тестов (таблица 6).

Таблица 6 – Тесты для структурного тестирования

Номер теста	Входные данные	Предполагаемый результат	Полученный результат
1	Состояние указано, новый виджет не совпадает с предыдущим и не может быть обновлен	Старое состояние уничтожено, создано новое состояние	Старое состояние уничтожено, создано новое состояние
2	Состояние указано, новый виджет не совпадает с предыдущим, но может быть обновлен	Старое состояние обновлено по описанию требуемого состояния	Старое состояние обновлено по описанию требуемого состояния
3	Состояние указано, новый виджет совпадает с предыдущим	Старое состояние не изменено	Старое состояние не изменено
4	Состояние не указано	Создано новое состояние	Создано новое состояние

Соотношение условий и покрывающих их тестов представлено в таблице 7.

Таблица 7 – Покрытие условий

Условие	Номер теста
1 – да	1, 2, 3
1 – нет	4
2 – да	3
2 – нет	2, 3
4 – да	2
4 – нет	1

Соотношение операторов и тестов, которые их покрывают, представлено в таблице 8. Для покрытия всех операторов достаточно трех тестов, тест номер 4 избыточен.

Таблица 8 – Покрытие операторов

Оператор	Номер теста
1	1, 2, 3, 4
2	1, 2, 3
3	3
4	1, 2
5	2
6	2
7	1
8	1, 4
9	1, 4
10	1, 4
11	1, 4
12	1, 2, 3, 4

Соотношение маршрутов и покрывающих их тестов представлено в таблице 9.

Таблица 9 – Покрытие маршрутов

Маршрут	Номер теста
1-2-3-12	3
1-2-4-5-6-10	3
1-2-4-7-8-9-10-11-12	1
1-8-9-10-11-12	4

В результате тестирования для каждого условия, оператора и маршрута был составлен как минимум один тест. Тестирование подтвердило корректность реализации метода для обновления состояния виджета.

7.1.4 Тестирование интерфейса

Среда Unity3d позволяет разрабатывать приложения под множество устройств, работающих на разных операционных системах и обладающих различными разрешениями экрана. Разрабатываемая библиотека должна корректно отображать пользовательский интерфейс на наиболее распространенных устройствах.

Для тестирования было реализовано демонстрационное приложение, отображающее список задач с использованием основных возможностей библиотеки, таких как текст, кнопка, прокручиваемый список, колонка, безопасная зона. Тестирование осуществлялось с помощью утилиты Device Simulator, поставляемой в комплекте со средой Unity3d [18].

Протокол тестирования представлен в таблице 10.

Таблица 10 – Протокол тестирования интерфейса

Устройство	Операционная система	Разрешение экрана	Результат тестирования
Apple iPad Air	iOS	1536x2048	Тест пройден (рисунок В.4)
Apple iPhone X	iOS	2436x1125	Тест пройден (рисунок В.5)
Huawei Mate 20 Pro	Android	1440x3120	Тест пройден (рисунок В.6)
Sony Xperia Z	Android	1080x1920	Тест пройден (рисунок В.7)

В результате тестов было проверено что разработанная библиотека корректно выполняет построение интерфейса на различных моделях устройств.

7.2 Метрики качества

Метрики позволяют оценивать качество кода. Наиболее распространенная – комплексный показатель качества кода (Maintainability Index), которая рассчитывается по формуле:

$$MI = \max(0, (171 - 5.2 \times \ln(HV) - 0.23 \times M - 16.2 \times \ln(LoC)) \times 100 / 171),$$

где HV – вычислительная сложность, зависящая от количества операторов;
M – структурная сложность, которая зависит от количества ветвлений;
LoC – количество строк в коде.

Структурная сложность рассчитывается по функциональному графу по формуле:

$$M = e - n + 2p,$$

где e – число ребер;
n – число узлов;
p – число компонент связности.

Вычислительную сложность можно рассчитать по формуле:

$$HV = N \times \log_2(n),$$

где N – длина программы;
n – словарь программы.

Длина программы рассчитывается как общее количество операторов и операндов в программе. Словарь программы – общее число уникальных операторов и операндов.

Далее представлен расчет сложности для кода из листинга 6.

Метод UpdateChildElement содержит 23 строки, из которых 11 являются пустыми или содержат только скобки. Следовательно:

$$LoC = 23 - 11 = 12.$$

Для функционального графа метода UpdateChildElement (смотри рисунок 13), структурная сложность равна:

$$M = 14 - 12 + 2 \times 1 = 4.$$

Таблица 11 содержит информацию о количестве операндов. В таблице 12 представлены данные о количестве операторов.

Таблица 11 – Количество операндов

Операнд	Количество
Child	8
RawWidget	3
nextWidget	6
Key	2
Type	2
Update	2
Dispose	1
CreateState	1
nextChild	4
InitState	1
null	1

Таблица 12 – Количество операторов

Оператор	Количество
If	3
return	3
.	12
&&	1
!=	1
==	3

В результате анализа исходного кода программы получено:

- число уникальных операндов равно 11;
- общее число операндов равно 31;
- число уникальных операторов равно 6;
- общее число операторов равно 23.

Тогда вычислительную сложность можно рассчитать по формуле:

$$N = 31 + 23 = 54;$$

$$n = 11 + 6 = 17;$$

$$NV = 54 \times \log_2(17) = 54 \times 4.08 = 220.$$

Показатель качества:

$$MI = \max(0, (171 - 5.2 \times \ln(220) - 0.23 \times 4 - 16.2 \times \ln(12)) \times 100 / 171) = 59.51.$$

Показатель качества принимает значение от 0 до 100. Чем ближе значение показателя к 100, тем проще поддерживать код. Показатель 59.51 свидетельствует о хорошем качестве кода.

8 Планирование разработки и анализ бюджета

Было осуществлено планирование разработки библиотеки для построения реактивного пользовательского интерфейса.

В ходе планирования было выявлено 19 задач. Задачи представлены в таблице 13.

Таблица 13 – Задачи проекта

Номер	Название задачи	Длительность (дни)
1	Изучение предметной области	3
2	Изучение готовых решений	2
3	Сравнение готовых решений	1
4	Составление требований к подсистеме реактивного программирования	2
5	Составление требований к подсистеме построения интерфейса	2
6	Анализ функциональных требований	2
7	Анализ нефункциональных требований	1
8	Выбор технологий и средств разработки	1
9	Проектирование подсистемы реактивного программирования	2
10	Проектирование подсистемы построения интерфейса	2
11	Реализация подсистемы реактивного программирования	10
12	Реализация подсистемы построения интерфейса	14
13	Тестирование методом причинно-следственных связей	2
14	Автоматизированное тестирование	1
15	Структурное тестирование	1
16	Тестирование интерфейса	1
17	Вычисление метрик качества кода	2
18	Составление глоссария	1
19	Составление отчета	4

Список ресурсов, используемых при реализации проекта, представлен в таблице 14.

Таблица 14 – Ресурсы проекта

Номер	Название ресурса	Ставка
1	Программист 1	30 тысяч руб/мес
2	Программист 2	30 тысяч руб/мес
3	Тестировщик	25 тысяч руб/мес
4	Документовед	20 тысяч руб/мес

Для каждой задачи были назначены ресурсы (таблица 15).

Таблица 15 – Назначения ресурсов задачам

Номер ресурса	Номера задач
1	1, 2,3, 4, 6, 7, 8, 9, 11, 13, 14, 17
2	1, 2, 3, 5, 6, 7, 8, 10, 12, 15, 16, 17
3	13, 14, 15, 16
4	18, 19

Составленные список задач и список ресурсов были перенесены в Microsoft Project. Данное программное обеспечение позволяет автоматизировать построения плана разработки и анализ бюджета [19].

Ресурсы, а также их типы и ставки, представлены на рисунке 14.

		Название ресурса	Тип	Крат. назв.	Макс. единиц	Стандартная ставка	Ставка сверхурочн.	Затраты на использ.	Начисление	Базовый календарь
1		Программист 1	Трудовой	П1	100%	30 000,00 ₽/месяц	0,00 ₽/час	0,00 ₽	Пропорциональ	Стандартный
2		Программист 2	Трудовой	П2	100%	30 000,00 ₽/месяц	0,00 ₽/час	0,00 ₽	Пропорциональ	Стандартный
3		Тестировщик	Трудовой	Т	100%	25 000,00 ₽/месяц	0,00 ₽/час	0,00 ₽	Пропорциональ	Стандартный
4		Документовед	Трудовой	Д	100%	20 000,00 ₽/месяц	0,00 ₽/час	0,00 ₽	Пропорциональ	Стандартный

Рисунок 14 – Список ресурсов проекта

Задачи проекта с учетом вех и этапов представлены на рисунке 15.

		Название задачи	Длительн	Начало	Окончание	Названия ресурсов
0		Библиотека для построения реактивного пользовательского интерфейса	40 дней	Пн 13.04.20	Пт 05.06.20	
1		Начало проекта	0 дней	Пн 13.04.20	Пн 13.04.20	
2		Анализ предметной области	12 дней	Пн 13.04.20	Вт 28.04.20	1
3		Изучение предметной области	3 дней	Пн 13.04.20	Ср 15.04.20	Программист 1 ;Программист 2
4		Анализ продуктов-аналогов	3 дней	Чт 16.04.20	Пн 20.04.20	3
5		Изучение готовых решений	2 дней	Чт 16.04.20	Пт 17.04.20	Программист 1 ;Программист 2
6		Сравнение готовых решений	1 день	Пн 20.04.20	Пн 20.04.20	5 Программист 1 ;Программист 2
7		Постановка задачи	2 дней	Вт 21.04.20	Ср 22.04.20	4
8		Составление требований к подсистеме реактивного программирования	2 дней	Вт 21.04.20	Ср 22.04.20	Программист 1
9		Составление требований к подсистеме построения интерфейса	2 дней	Вт 21.04.20	Ср 22.04.20	Программист 2
10		Анализ требований	3 дней	Чт 23.04.20	Пн 27.04.20	7
11		Анализ функциональных требований	2 дней	Чт 23.04.20	Пт 24.04.20	Программист 1 ;Программист 2
12		Анализ нефункциональных требований	1 день	Пн 27.04.20	Пн 27.04.20	11 Программист 1 ;Программист 2
13		Выбор технологий и средств разработки	1 день	Вт 28.04.20	Вт 28.04.20	10 Программист 1 ;Программист 2
14		Проектирование	2 дней	Ср 29.04.20	Чт 30.04.20	2
15		Проектирование подсистемы реактивного программирования	2 дней	Ср 29.04.20	Чт 30.04.20	Программист 1
16		Проектирование подсистемы построения интерфейса	2 дней	Ср 29.04.20	Чт 30.04.20	Программист 2
17		Реализация	14 дней	Пт 01.05.20	Ср 20.05.20	14
18		Реализация подсистемы реактивного программирования	10 дней	Пт 01.05.20	Чт 14.05.20	Программист 1
19		Реализация подсистемы построения интерфейса	14 дней	Пт 01.05.20	Ср 20.05.20	Программист 2
20		Тестирование	7 дней	Чт 21.05.20	Пт 29.05.20	17
21		Тестирование методом причинно-следственных связей	2 дней	Чт 21.05.20	Пт 22.05.20	Тестирующий ;Программист 1
22		Автоматизированное тестирование	1 день	Пн 25.05.20	Пн 25.05.20	21 Тестирующий ;Программист 1
23		Структурное тестирование	1 день	Вт 26.05.20	Вт 26.05.20	22 Тестирующий ;Программист 2
24		Тестирование интерфейса	1 день	Ср 27.05.20	Ср 27.05.20	23 Тестирующий ;Программист 2
25		Вычисление метрик качества кода	2 дней	Чт 28.05.20	Пт 29.05.20	24 Программист 1 ;Программист 2
26		Документирование	5 дней	Пн 01.06.20	Пт 05.06.20	20
27		Составление глоссария	1 день	Пн 01.06.20	Пн 01.06.20	Документовед
28		Составление отчета	4 дней	Вт 02.06.20	Пт 05.06.20	27 Документовед
29		Конец проекта	0 дней	Пт 05.06.20	Пт 05.06.20	26

Рисунок 15 – Список задач проекта

Диаграмма Ганта, иллюстрирующая разработанный план разработки, представлена на рисунке 16.



Рисунок 16 – Диаграмма Ганта

По составленному плану разработки была проанализирована загруженность ресурсов и составлен график трудозатрат.

Загруженность ресурсов представлена на рисунке 17.

Название	Начало	Окончание	Оставшиеся трудозатраты
Программист 1	Пн 13.04.20	Пт 29.05.20	232 часов
Программист 2	Пн 13.04.20	Пт 29.05.20	256 часов
Тестировщик	Чт 21.05.20	Ср 27.05.20	40 часов
Документовед	Пн 01.06.20	Пт 05.06.20	40 часов

Рисунок 17 – Загруженность ресурсов

График трудозатрат представлен на рисунке 18.

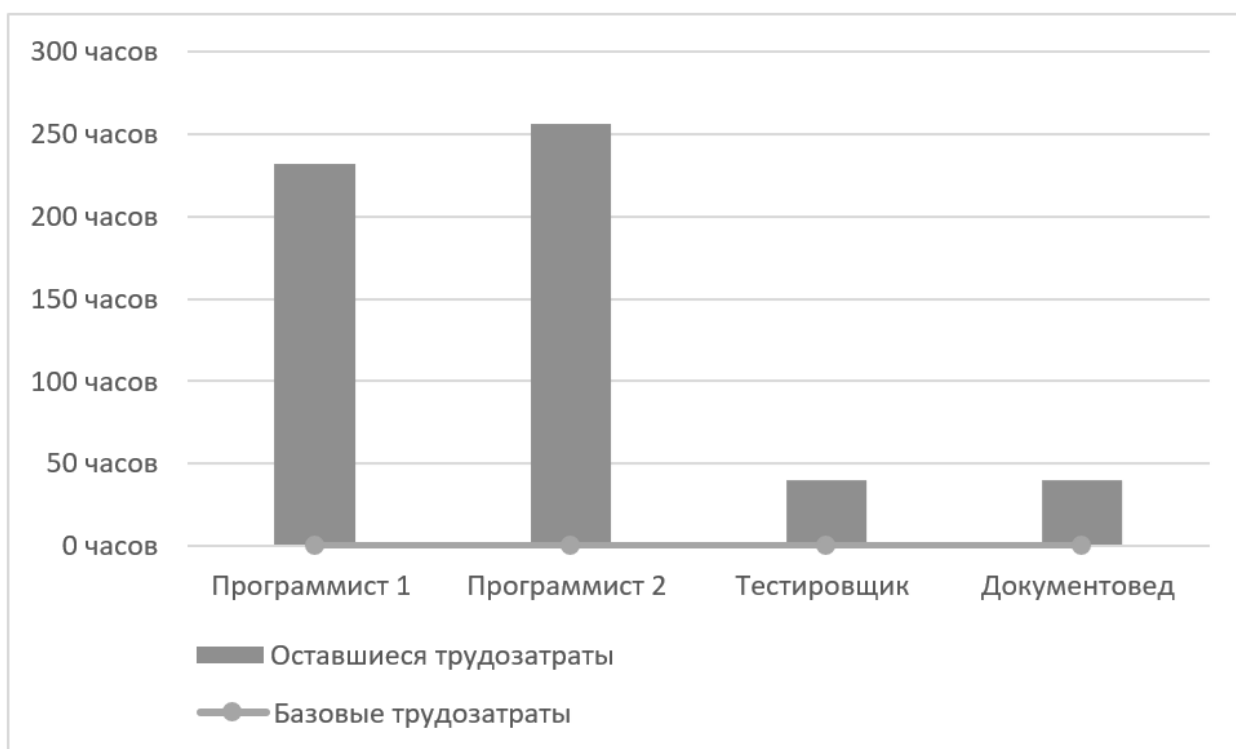


Рисунок 18 – График трудозатрат

Был составлен график затрат на выполнение отдельных задач первого уровня, благодаря чему можно оценить стоимость реализации отдельных этапов разработки библиотеки.

Затраты на выполнение отдельных задач представлены на рисунке 19.

Название	Фактически затраты	Оставшиеся затраты	Базовые затраты	Затраты	Отклонение по стоимости
Начало проекта	0,00 Р	0,00 Р	0,00 Р	0,00 Р	0,00 Р
Анализ предметной области	0,00 Р	36 000,00 Р	0,00 Р	36 000,00 Р	36 000,00 Р
Проектирование	0,00 Р	6 000,00 Р	0,00 Р	6 000,00 Р	6 000,00 Р
Реализация	0,00 Р	36 000,00 Р	0,00 Р	36 000,00 Р	36 000,00 Р
Тестирование	0,00 Р	19 750,00 Р	0,00 Р	19 750,00 Р	19 750,00 Р
Документирование	0,00 Р	5 000,00 Р	0,00 Р	5 000,00 Р	5 000,00 Р
Конец проекта	0,00 Р	0,00 Р	0,00 Р	0,00 Р	0,00 Р

Рисунок 19 – Затраты на задачи

Стоимость реализации проекта составляет сто две тысячи семьсот пятьдесят рублей. Разработка будет идти с 13 апреля 2020 года по 6 июня 2020 года. Длительность разработки составит 40 рабочих дней. Трудозатраты на реализацию составят 568 часов.

Заключение

В выпускной квалификационной работе были изучены методы реактивного программирования и способы построения интерфейсов, была произведена классификация систем построения интерфейсов, проведен сравнительный анализ продуктов-аналогов.

В результате исследования была выявлена высокая востребованность библиотек для построения реактивных интерфейсов в смежных областях (android, iOS, web) и отсутствие готовых решений для Unity.

На основе этих данных было принято решение в рамках выпускной квалификационной работы осуществить реализацию библиотеки для построения реактивного пользовательского интерфейса на основе декларативного подхода и pull-based реактивного программирования.

Была реализована библиотека для построения реактивного пользовательского интерфейса, состоящая из двух подсистем:

- подсистема реактивного программирования;
- подсистема построения интерфейса.

Были поставлены задачи на разработку, проанализированы требования, выбраны технологии и средства разработки, осуществлено проектирование, реализация и тестирование подсистем, проведено планирование разработки и анализ бюджета.

Разработка осуществлялась на операционной системе Windows в среде разработки JetBrains Rider.

Планирование разработки проводилось в Microsoft Project. Стоимость разработки составляет 102750 рублей.

В результате выполнения выпускной квалификационной работы была разработана библиотека для построения реактивного пользовательского интерфейса.

Список использованных источников

1. Графический интерфейс пользователя, графический пользовательский интерфейс // Википедия [Электронный ресурс]. URL: https://ru.wikipedia.org/wiki/Графический_интерфейс_пользователя (дата обращения: 20.04.2020).
2. Реактивное программирование // Википедия [Электронный ресурс]. URL: https://ru.wikipedia.org/wiki/Реактивное_программирование (дата обращения: 20.04.2020).
3. Mol libs for web ui // The JavaScript organization [Электронный ресурс]. URL: <https://mol.js.org/> (дата обращения: 20.04.2020).
4. Jetpack Compose // Android platform [Электронный ресурс]. URL: <https://developer.android.com/jetpack/compose/> (дата обращения: 20.04.2020).
5. Xcode Swift UI Apple Developer // Apple [Электронный ресурс]. URL: <https://developer.apple.com/xcode/swiftui/> (дата обращения: 20.04.2020).
6. Flutter – Beautiful native apps in record time // Flutter [Электронный ресурс]. URL: <https://flutter.dev/> (дата обращения: 20.04.2020).
7. Что такое виджеты? // Технологии Яндекса [Электронный ресурс]. URL: <https://yandex.ru/dev/wdgt/doc/dg/concepts/widget-definition-docpage/> (дата обращения: 20.04.2020).
8. Химонин Юрий. Сбор и анализ требований к программному продукту // Файловый архив [Электронный ресурс]. URL: <https://studfile.net/preview/1869165/> (дата обращения: 07.05.2020).
9. Вигерс Карл. Разработка требований к программному обеспечению / К. Вингерс – М.: Издательско-торговый дом «Русская Редакция», 2004. – 576с
10. Mono.Cecil // Mono [Электронный ресурс]. URL: <https://www.mono-project.com/docs/tools+libraries/libraries/Mono.Cecil/> (дата обращения: 07.05.2020).

11. Что такое управляемый код // Microsoft [Электронный ресурс]. URL: <https://docs.microsoft.com/ru-ru/dotnet/standard/managed-code> (дата обращения: 07.05.2020).
12. ISharpCode .NET Decompiler with support for PDB generation, ReadyToRun, Metadata (&more) // Github [Электронный ресурс]. URL: <https://github.com/icsharpcode/ILSpy> (дата обращения: 07.05.2020).
13. Диаграммы пакетов // НОУ ИНТУИТ [Электронный ресурс]. URL: <https://www.intuit.ru/studies/courses/1041/218/lecture/27263?page=3> (дата обращения: 07.05.2020).
14. Тестирование программных средств // RSDN [Электронный ресурс]. URL: <http://rsdn.org/article/testing/SoftwareTesting.xml> (дата обращения: 07.05.2020).
15. Модели качества и надежности в программной инженерии. Метрики качества программного обеспечения // НОУ ИНТУИТ [Электронный ресурс]. URL: <https://www.intuit.ru/studies/courses/2190/237/lecture/6136> (дата обращения: 07.05.2020).
16. Метрики на службе у QA // Тестирование и качество ПО [Электронный ресурс]. URL: <https://www.software-testing.ru/library/around-testing/processes/3002-metrics> (дата обращения: 07.05.2020).
17. About Unity Test Framework // Unity [Электронный ресурс]. URL: <https://docs.unity3d.com/Packages/com.unity.test-framework@1.1/manual/index.html> (дата обращения: 07.05.2020).
18. Device Simulator // Unity [Электронный ресурс]. URL: <https://docs.unity3d.com/Packages/com.unity.device-simulator@2.2/manual/index.html> (дата обращения: 07.05.2020).
19. Решение для управления проектами // Microsoft [Электронный ресурс] URL: <https://www.microsoft.com/ru-ru/microsoft-365/project/project-management-software> (дата обращения: 07.05.2020).

Приложение А
ГЛОССАРИЙ
(справочное)

Виджет – часть исходного кода библиотеки, оформленная в виде одного или нескольких классов и реализующая примитивный элемент интерфейса. Примеры виджетов: текстовое поле, кнопка, таблица, список.

Дерево виджетов – несколько виджетов составляющих древовидную структуру, полученную методом композиции виджетов.

Примитивный виджет – виджет, не содержащий вложенных элементов. В дереве виджетов такой виджет является терминальным узлом.

Виджет-контейнер – виджет, служащий для отображения и позиционирования одного или нескольких вложенных элементов. Примеры виджетов-контейнеров: таблица, список. В дереве виджетов такой виджет является внутренним узлом.

Unity3d – набор инструментов для разработки компьютерных игр.

Ячейка – класс-обертка над переменной, разрабатываемый в рамках данной работы, служащий для автоматического перевычисления значений вычисляемых переменных.

DTO (Data Transfer Object) – класс, содержащий только данные. Используется для передачи информации между системами.

Java SDK (Java Software Development Kit) – набор утилит для разработки приложений на языке Java.

Android NDK (Android Native Development Kit) – набор утилит для реализации части Android приложений с помощью языков C/C++.

RUP (Rational Unified Process) – методология разработки программного обеспечения.

Diagrams.net – онлайн редактор диаграмм, поддерживающий с том числе язык UML.

UML (Unified Modelling Language) – унифицированный язык моделирования.

API (Application Programming Interface) – описание способов взаимодействия программ, либо частей программы между собой.

Приложение Б
ЛИСТИНГ ПРОГРАММЫ
(рекомендуемое)

Листинг Б.1 – Файл «Atom.cs»

```
namespace UniMob
{
    public interface Atom<out T>
    {
        T Value { get; }
    }
}
```

Листинг Б.2 – Файл «MutableAtom.cs»

```
namespace UniMob
{
    public interface MutableAtom<T> : Atom<T>
    {
        new T Value { get; set; }
    }
}
```

Листинг Б.3 – Файл «ValueAtom.cs»

```
using System;
using System.Collections.Generic;

namespace UniMob
{
    public class ValueAtom<T> : AtomBase, MutableAtom<T>
    {
        private readonly IEqualityComparer<T> _comparer;
        private T _value;

        internal ValueAtom(
            T value,
            Action onActive = null,
            Action onInactive = null,
            IEqualityComparer<T> comparer = null)
            : base(false, onActive, onInactive)
        {
            _value = value;
            _comparer = comparer ?? EqualityComparer<T>.Default;
        }

        public T Value
        {
            get
            {
                if (!IsActive && Stack == null && !KeepAlive)
                {
                    return _value;
                }

                Actualize();
                SubscribeToParent();
                return _value;
            }
            set
            {
                try
```

```

        {
            using (Atom.NoWatch)
            {
                if (_comparer.Equals(value, _value))
                    return;

                State = AtomState.Actual;
                _value = value;
            }
        }
        finally
        {
            ObsoleteSubscribers();
        }
    }

    protected override void Evaluate()
    {
        State = AtomState.Actual;

        ObsoleteSubscribers();
    }

    public void Invalidate()
    {
        ObsoleteSubscribers();
    }

    public override string ToString()
    {
        return Convert.ToString(_value);
    }
}
}

```

Листинг Б.4 – Файл «ComputedAtom.cs»

```

using System;
using System.Collections.Generic;
using System.Runtime.ExceptionServices;
using JetBrains.Annotations;

namespace UniMob
{
    public class ComputedAtom<T> : AtomBase, MutableAtom<T>
    {
        private readonly IEqualityComparer<T> _comparer;
        private readonly AtomPull<T> _pull;
        private readonly AtomPush<T> _push;
        private readonly bool _requiresReaction;

        private bool _hasCache;
        private T _cache;
        private ExceptionDispatchInfo _exception;
        private bool _isRunningSetter;

        internal ComputedAtom(
            [NotNull] AtomPull<T> pull,
            AtomPush<T> push = null,
            bool keepAlive = false,
            bool requiresReaction = false,

```

```

    Action onActive = null,
    Action onInactive = null,
    IEqualityComparer<T> comparer = null)
    : base(keepAlive, onActive, onInactive)
{
    _pull = pull
        ?? throw new ArgumentNullException(nameof(pull));
    _push = push;
    _comparer = comparer ?? EqualityComparer<T>.Default;
    _requiresReaction = requiresReaction;
}

public T Value
{
    get
    {
        if (State == AtomState.Pulling)
        {
            throw new CyclicAtomDependencyException(this);
        }

        if (!IsActive && Stack == null && !KeepAlive)
        {
            WarnAboutUnTrackedRead();

            var oldState = State;
            try
            {
                State = AtomState.Pulling;
                return _pull();
            }
            finally
            {
                State = oldState;
            }
        }

        Actualize();
        SubscribeToParent();

        if (_exception != null)
        {
            _exception.Throw();
        }

        return _cache;
    }
    set
    {
        if (_push == null)
            throw new InvalidOperationException();

        if (_isRunningSetter)
        {
            throw new InvalidOperationException();
        }

        try
        {
            using (Atom.NoWatch)
            {
                if (_hasCache &&
                    comparer.Equals(value, cache))

```

```

        return;

        State = AtomState.Obsolete;
        _cache = default;
        _exception = null;
        _hasCache = false;

        _isRunningSetter = true;
        _push(value);
    }
}
finally
{
    _isRunningSetter = false;
    ObsoleteSubscribers();
}
}

public override void Deactivate()
{
    base.Deactivate();

    _hasCache = false;
    _cache = default;
    _exception = null;
}

protected override void Evaluate()
{
    try
    {
        var value = _pull();

        using (Atom.NoWatch)
        {
            if (_hasCache
                && _comparer.Equals(value, _cache))
                return;
        }

        _hasCache = true;
        _cache = value;
        _exception = null;
    }
    catch (Exception exception)
    {
        _hasCache = false;
        _cache = default;
        _exception = ExceptionDispatchInfo.Capture(exception);
    }
    finally
    {
        State = AtomState.Actual;
    }

    ObsoleteSubscribers();
}

public void Invalidate()
{
    State = AtomState.Obsolete;
}

```

```

        _hasCache = false;
        _cache = default;
        _exception = null;

        ObsoleteSubscribers();
    }

    public override string ToString()
    {
        if (_exception != null)
        {
            return _exception.SourceException.ToString();
        }

        return _hasCache ? Convert.ToString(_cache) : "[undefined]";
    }
}

```

Листинг Б.5 – Файл «ReactionAtom.cs»

```

using System;
using System.Collections.Generic;
using System.Runtime.ExceptionServices;
using JetBrains.Annotations;

namespace UniMob
{
    public class ComputedAtom<T> : AtomBase, MutableAtom<T>
    {
        private readonly IEqualityComparer<T> _comparer;
        private readonly AtomPull<T> _pull;
        private readonly AtomPush<T> _push;
        private readonly bool _requiresReaction;

        private bool _hasCache;
        private T _cache;
        private ExceptionDispatchInfo _exception;
        private bool _isRunningSetter;

        internal ComputedAtom(
            [NotNull] AtomPull<T> pull,
            AtomPush<T> push = null,
            bool keepAlive = false,
            bool requiresReaction = false,
            Action onActive = null,
            Action onInactive = null,
            IEqualityComparer<T> comparer = null)
            : base(keepAlive, onActive, onInactive)
        {
            _pull = pull
                ?? throw new ArgumentNullException(nameof(pull));
            _push = push;
            _comparer = comparer ?? EqualityComparer<T>.Default;
            _requiresReaction = requiresReaction;
        }

        public T Value
        {
            get
            {
                if (State == AtomState.Pulling)

```

```

    {
        throw new CyclicAtomDependencyException(this);
    }

    if (!IsActive && Stack == null && !KeepAlive)
    {
        WarnAboutUnTrackedRead();

        var oldState = State;
        try
        {
            State = AtomState.Pulling;
            return _pull();
        }
        finally
        {
            State = oldState;
        }
    }

    Actualize();
    SubscribeToParent();

    if (_exception != null)
    {
        _exception.Throw();
    }

    return _cache;
}
set
{
    if (_push == null)
        throw new InvalidOperationException();

    if (_isRunningSetter)
    {
        throw new InvalidOperationException();
    }

    try
    {
        using (Atom.NoWatch)
        {
            if (_hasCache &&
                _comparer.Equals(value, _cache))
                return;

            State = AtomState.Obsolete;
            _cache = default;
            _exception = null;
            _hasCache = false;

            _isRunningSetter = true;
            _push(value);
        }
    }
    finally
    {
        _isRunningSetter = false;
        ObsoleteSubscribers();
    }
}

```

```

    }

    public override void Deactivate()
    {
        base.Deactivate();

        _hasCache = false;
        _cache = default;
        _exception = null;
    }

    protected override void Evaluate()
    {
        try
        {
            var value = _pull();
            using (Atom.NoWatch)
            {
                if (_hasCache &&
                    _comparer.Equals(value, _cache))
                    return;
            }

            _hasCache = true;
            _cache = value;
            _exception = null;
        }
        catch (Exception exception)
        {
            _hasCache = false;
            _cache = default;
            _exception = ExceptionDispatchInfo.Capture(exception);
        }
        finally
        {
            State = AtomState.Actual;
        }

        ObsoleteSubscribers();
    }

    public void Invalidate()
    {
        State = AtomState.Obsolete;

        _hasCache = false;
        _cache = default;
        _exception = null;

        ObsoleteSubscribers();
    }

    public override string ToString()
    {
        if (_exception != null)
        {
            return _exception.SourceException.ToString();
        }

        return _hasCache ? Convert.ToString(_cache) : "[undefined]";
    }
}

```

Листинг Б.6 – Файл «AtomFactory.cs»

```

namespace UniMob
{
    public static class Atom
    {
        public static MutableAtom<T> Value<T>(
            T value,
            Action onActive = null,
            Action onInactive = null,
            IEqualityComparer<T> comparer = null)
        {
            return new ValueAtom<T>(value, onActive, onInactive, comparer);
        }

        public static Atom<T> Computed<T>(
            AtomPull<T> pull,
            bool keepAlive = false,
            bool requiresReaction = false,
            Action onActive = null,
            Action onInactive = null,
            IEqualityComparer<T> comparer = null)
        {
            return new ComputedAtom<T>(pull, null, keepAlive,
                requiresReaction, onActive, onInactive, comparer);
        }

        public static IDisposable Reaction<T>(
            AtomPull<T> pull,
            Action<T, IDisposable> reaction,
            IEqualityComparer<T> comparer = null,
            bool fireImmediately = false,
            Action<Exception> exceptionHandler = null)
        {
            var valueAtom = Computed(pull, comparer: comparer);
            bool firstRun = true;

            ReactionAtom atom = null;
            atom = new ReactionAtom(() =>
            {
                var value = valueAtom.Value;

                using (NoWatch)
                {
                    if (firstRun)
                    {
                        firstRun = false;

                        if (!fireImmediately)
                            return;
                    }

                    reaction(value, atom);
                }
            }, exceptionHandler);

            atom.Get();
            return atom;
        }
    }
}

```

Листинг Б.7 – Файл «AtomWeaver.cs»

```

using System;
using System.Linq;
using Mono.CecilX;
using Mono.CecilX.Cil;

namespace UniMob.Editor.Weaver
{
    public class AtomWeaver
    {
        private const string MemFieldNameFormat = "__atom_{0}";
        private const string PullMethodNameFormat = "__atom_get_{0}";
        private const string PushMethodNameFormat = "__atom_set_{0}";

        private const string ValueName = "Value";
        private const string ConstructorName = ".ctor";
        private const string ComputedName = "Computed";

        private ModuleDefinition _module;

        private TypeDefinition _atomFactoryDef;
        private TypeReference _atomTypeGeneric;
        private TypeDefinition _atomDefGeneric;
        private TypeDefinition _atomPullDef;
        private TypeDefinition _atomPushDef;

        private MethodReference _atomGetValue;
        private MethodReference _atomSetValue;
        private MethodReference _atomPullCtor;
        private MethodReference _atomPushCtor;
        private MethodReference _atomFactoryComputed;

        public bool Weave(AssemblyDefinition asmDef)
        {
            _module = asmDef.MainModule;

            _atomTypeGeneric =
                module.ImportReference(typeof(MutableAtom<>));
            _atomDefGeneric = _atomTypeGeneric.Resolve();
            _atomPullDef =
                _module.ImportReference(typeof(AtomPull<>)).Resolve();
            _atomPushDef =
                _module.ImportReference(typeof(AtomPush<>)).Resolve();
            _atomFactoryDef =
                _module.ImportReference(typeof(Atom)).Resolve();

            bool ValuePropertyFilter(PropertyDefinition p) =>
                p.Name == ValueName;
            _atomGetValue =
                _module.ImportReference(_atomDefGeneric.Properties
                    .Single(ValuePropertyFilter).GetMethod);
            _atomSetValue =
                module.ImportReference(_atomDefGeneric.Properties
                    .Single(ValuePropertyFilter).SetMethod);

            bool ConstructorFilter(MethodDefinition m) =>
                m.Name == ConstructorName;
            _atomPullCtor =
                _module.ImportReference(_atomPullDef.Methods
                    .Single(ConstructorFilter));
            _atomPushCtor =

```

```

        _module.ImportReference(_atomPushDef.Methods
            .Single(ConstructorFilter));

bool ComputedMethodFilter(MethodDefinition m) =>
    m.Name == ComputedName && m.Parameters.Count == 7;
_atomFactoryComputed =
    _module.ImportReference(_atomFactoryDef.Methods
        .Single(ComputedMethodFilter));

bool dirty = false;

var types = _module.Types;
for (var typeIndex = 0; typeIndex < types.Count;
    typeIndex++)
{
    var type = types[typeIndex];
    var properties = type.Properties;
    for (var propIndex = 0; propIndex < properties.Count;
        propIndex++)
    {
        var property = properties[propIndex];
        dirty |= Weave(property);
    }
}

return dirty;
}

public bool Weave(PropertyDefinition propDef)
{
    var customAttr =
        Helpers.GetCustomAttribute<AtomAttribute>(propDef);
    if (customAttr == null)
        return false;

    //propDef.CustomAttributes.Remove(customAttr);

    var pull = (propDef.GetMethod != null) ?
        CreatePullMethod(propDef) : null;
    var push = (propDef.SetMethod != null) ?
        CreatePushMethod(propDef) : null;
    var atom = CreateAtomField(propDef);

    if (pull != null) propDef.DeclaringType.Methods.Add(pull);
    if (push != null) propDef.DeclaringType.Methods.Add(push);
    propDef.DeclaringType.Fields.Add(atom);

    ReplacePropertyCall(propDef, atom, pull, push);

    return true;
}

private static MethodDefinition
CreatePullMethod(PropertyDefinition propDef)
{
    var ops = propDef.GetMethod.Body.Instructions;
    if (ops.Count == 3 &&
        ops[0].OpCode == OpCodes.Ldarg_0 &&
        ops[2].OpCode == OpCodes.Ret &&
        ops[1].OpCode == OpCodes.Ldfld &&
        ops[1].Operand is FieldDefinition backingField &&
        backingField.Name.Equals(
            $"<{propDef.Name}>k__BackingField"))

```

```

    {
        backingField.Name =
            $"__atom__{propDef.Name}__BackingField";
    }

    var name = string.Format(PullMethodNameFormat,
        propDef.Name);
    var method = new MethodDefinition(name,
        MethodAttributes.Private, propDef.PropertyType)
    {
        Body = propDef.GetMethod.Body,
    };
    return method;
}

private static MethodDefinition
CreatePushMethod(PropertyDefinition propDef)
{
    var name = string.Format(PushMethodNameFormat,
        propDef.Name);
    var method = new MethodDefinition(name,
        MethodAttributes.Private,
        propDef.SetMethod.ReturnType)
    {
        Body = propDef.SetMethod.Body,
    };

    var parameter = new ParameterDefinition("value",
        ParameterAttributes.None, propDef.PropertyType);
    method.Parameters.Add(parameter);

    return method;
}

private FieldDefinition CreateAtomField(
    PropertyDefinition propDef)
{
    var name = string.Format(MemFieldNameFormat, propDef.Name);
    var type = Helpers.MakeGenericType(_atomTypeGeneric,
        propDef.PropertyType);
    return new FieldDefinition(name,
        FieldAttributes.Private, type);
}

private void ReplacePropertyCall(PropertyDefinition prop,
    FieldReference field,
    MethodReference pullMethod, MethodReference pushMethod)
{
    var type = prop.PropertyType;

    if (prop.GetMethod != null)
    {
        prop.GetMethod.Body = new MethodBody(prop.GetMethod);
        prop.GetMethod.Body.Variables.Add(
            new VariableDefinition(_atomTypeGeneric));

        var ilProcessor =
            prop.GetMethod.Body.GetILProcessor();
        var typedAtomGet =
            Helpers.MakeHostInstanceGeneric(_atomGetValue,
                type);

        GenerateAtom(ilProcessor, type, field, pullMethod,

```

```

        pushMethod,
        proc =>
        {
            Instruction first;
            proc.Append(first =
                proc.Create(OpCodes.Callvirt,
                    typedAtomGet));
            return first;
        });
    }

    if (prop.SetMethod != null)
    {
        prop.SetMethod.Body = new MethodBody(prop.SetMethod);
        prop.SetMethod.Body.Variables.Add(new
            VariableDefinition(_atomTypeGeneric));

        var ilProcessor =
            prop.SetMethod.Body.GetILProcessor();
        var typedAtomSet =
            Helpers.MakeHostInstanceGeneric(_atomSetValue,
                type);

        GenerateAtom(ilProcessor, type, field, pullMethod,
            pushMethod, proc =>
        {
            Instruction first;
            proc.Append(first =
                proc.Create(OpCodes.Ldarg_1));
            proc.Append(proc.Create(OpCodes.Callvirt,
                typedAtomSet));
            return first;
        });
    }
}

private void GenerateAtom(ILProcessor proc, TypeReference type,
    FieldReference field,
    MethodReference pullMethod, MethodReference pushMethod,
    Func<ILProcessor, Instruction> generate)
{
    Instruction br;

    //load class instance onto stack
    proc.Append(proc.Create(OpCodes.Ldarg_0));

    // pop class instance and push field value
    proc.Append(proc.Create(OpCodes.Ldfld, field));

    // duplicate field value
    proc.Append(br = proc.Create(OpCodes.Dup));

    {
        // pop field value and jump if atom not null
        //worker.Append(worker.Create(OpCodes.Brtrue_S,
            label)); //will be inserted later

        // pop field value (now stack is empty)
        proc.Append(proc.Create(OpCodes.Pop));

        //load class instance
        proc.Append(proc.Create(OpCodes.Ldarg_0));
    }
}

```

```

//create pull delegate or null
if (pullMethod != null)
{
    proc.Append(proc.Create(OpCodes.Ldarg_0));
    proc.Append(proc.Create(OpCodes.Ldftn,
        pullMethod));
    proc.Append(proc.Create(OpCodes.Newobj,
        Helpers.MakeHostInstanceGeneric(
            _atomPullCtor, type)));
}
else
{
    proc.Append(proc.Create(OpCodes.Ldnull));
}

//create push delegate or null
if (pushMethod != null)
{
    proc.Append(proc.Create(OpCodes.Ldarg_0));
    proc.Append(proc.Create(OpCodes.Ldftn,
        pushMethod));
    proc.Append(proc.Create(OpCodes.Newobj,
        Helpers.MakeHostInstanceGeneric(
            _atomPushCtor, type)));
}
else
{
    proc.Append(proc.Create(OpCodes.Ldnull));
}

// keepAlive = false
proc.Append(proc.Create(OpCodes.Ldc_I4_0));

// requireReaction = false
proc.Append(proc.Create(OpCodes.Ldc_I4_0));

// onActive = null
proc.Append(proc.Create(OpCodes.Ldnull));

// onInactive = null
proc.Append(proc.Create(OpCodes.Ldnull));

//comparer = null
proc.Append(proc.Create(OpCodes.Ldnull));

//invoke Atom.Computed(pull, push, keepAlive,
    requiredReaction, onActive, onInactive,
    comparer)
proc.Append(proc.Create(OpCodes.Call,
    Helpers.MakeGenericMethod(
        _atomFactoryComputed, type)));

// duplicate created Atom
proc.Append(proc.Create(OpCodes.Dup));

//pop created atom and save it to local variable
proc.Append(proc.Create(OpCodes.Stloc_0));

//pop atom and save it to class field
proc.Append(proc.Create(OpCodes.Stfld, field));

//load atom onto stack
proc.Append(proc.Create(OpCodes.Ldloc_0));

```

```

    }

    //use atom
    var brTarget = generate(proc);

    proc.InsertAfter(br, proc.Create(OpCodes.Brtrue_S,
        brTarget));

    // return
    proc.Append(proc.Create(OpCodes.Ret));
}
}
}

```

Листинг Б.8 – Файл «State.cs»

```

using System;
using System.Collections.Generic;
using System.Linq;
using UniMob.UI.Internal;
using UnityEngine.Assertions;

namespace UniMob.UI
{
    public abstract class State : IState, IDisposable
    {
        private readonly MutableBuildContext _context;

        public BuildContext Context => _context;

        internal Widget RawWidget { get; private set; }

        public abstract IViewState InnerViewState { get; }

        public abstract WidgetSize Size { get; }

        protected State()
        {
            Assert.IsNull(Atom.CurrentScope);
            _context = new MutableBuildContext(this, null);
        }

        internal virtual void Update(Widget widget)
        {
            RawWidget = widget;
        }

        internal void Mount(BuildContext context)
        {
            Assert.IsNull(Atom.CurrentScope);

            if (Context.Parent != null)
                throw new InvalidOperationException();

            _context.SetParent(context);
        }

        public virtual void InitState()
        {
            Assert.IsNull(Atom.CurrentScope);
        }
    }
}

```

```

public virtual void Dispose()
{
    Assert.IsNull (Atom.CurrentScope);
}

internal static Atom<IState> Create(
    BuildContext context, WidgetBuilder builder)
{
    Assert.IsNull (Atom.CurrentScope);

    State state = null;
    return Atom.Computed<IState>(() =>
    {
        var newWidget = builder(context);
        using (Atom.NoWatch)
        {
            state = StateUtilities
                .UpdateChildElement(
                    context, state, newWidget);
        }

        return state;
    }, onInactive: () =>
        StateUtilities.DeactivateChild(state),
        requiresReaction: true);
}

internal static Atom<IState[]> CreateList(
    BuildContext context, Func<BuildContext,
    List<Widget>> builder)
{
    Assert.IsNull (Atom.CurrentScope);

    var states = new State[0];
    return Atom.Computed<IState[]>(() =>
    {
        var newWidgets = builder.Invoke(context);
        using (Atom.NoWatch)
        {
            states = StateUtilities.UpdateChildren(
                context, states, newWidgets);
        }

        return states.ToArray();
    }, onInactive: () =>
    {
        foreach (var state in states)
        {
            StateUtilities.DeactivateChild(state);
        }
    }, requiresReaction: true);
}
}
}

```

Листинг Б.9 – Файл «ColumnView.cs»

```

using UniMob.UI.Internal;
using UniMob.UI.Widgets;
using UnityEngine;

```

```
[assembly: RegisterComponentViewFactory($"$_Column",
    typeof(RectTransform), typeof(ColumnView))]

namespace UniMob.UI.Widgets
{
    internal sealed class ColumnView : View<IColumnState>
    {
        private IMapperBase _mapper;

        protected override void Activate()
        {
            base.Activate();

            if (_mapper == null)
                _mapper = new PooledViewMapper(transform);
        }

        protected override void Render()
        {
            var children = State.Children;
            var crossAxis = State.CrossAxisAlignment;
            var mainAxis = State.MainAxisAlignment;
            var columnSize = State.InnerSize;

            var alignX = crossAxis == CrossAxisAlignment.Start
                ? Alignment.TopLeft.X
                : crossAxis == CrossAxisAlignment.End
                ? Alignment.TopRight.X
                : Alignment.Center.X;

            var alignY = mainAxis == MainAxisAlignment.Start
                ? Alignment.TopCenter.Y
                : mainAxis == MainAxisAlignment.End
                ? Alignment.BottomCenter.Y
                : Alignment.Center.Y;

            var offsetMultiplierY =
                mainAxis == MainAxisAlignment.Start ? 0
                : mainAxis == MainAxisAlignment.End ? 1f
                : 0.5f;

            var childAlignment = new Alignment(
                alignX, alignY);
            var corner = childAlignment.WithTop();
            var cornerPosition = new Vector2(
                0, -columnSize.Height * offsetMultiplierY);

            using (var render = _mapper.CreateRender())
            {
                foreach (var child in children)
                {
                    var childSize = child.Size;
                    var childView =
                        render.RenderItem(child);

                    LayoutData layout;
                    layout.Size = childSize;
                    layout.Alignment = childAlignment;
                    layout.Corner = corner;
                    layout.CornerPosition = cornerPosition;

                    ViewLayoutUtility.SetLayout(
                        childView.rectTransform, layout);
                }
            }
        }
    }
}
```

```

        cornerPosition += new Vector2(
            0, childSize.Height);
    }
}

internal interface IColumnState : IViewState
{
    WidgetSize InnerSize { get; }
    IState[] Children { get; }
    CrossAxisAlignment CrossAxisAlignment { get; }
    MainAxisAlignment MainAxisAlignment { get; }
}

```

Приложение В
РЕЗУЛЬТАТЫ ТЕСТИРОВАНИЯ
(рекомендуемое)



Рисунок В.1 – Тест «Элемент нулевого размера» (Метод причинно-следственных связей)

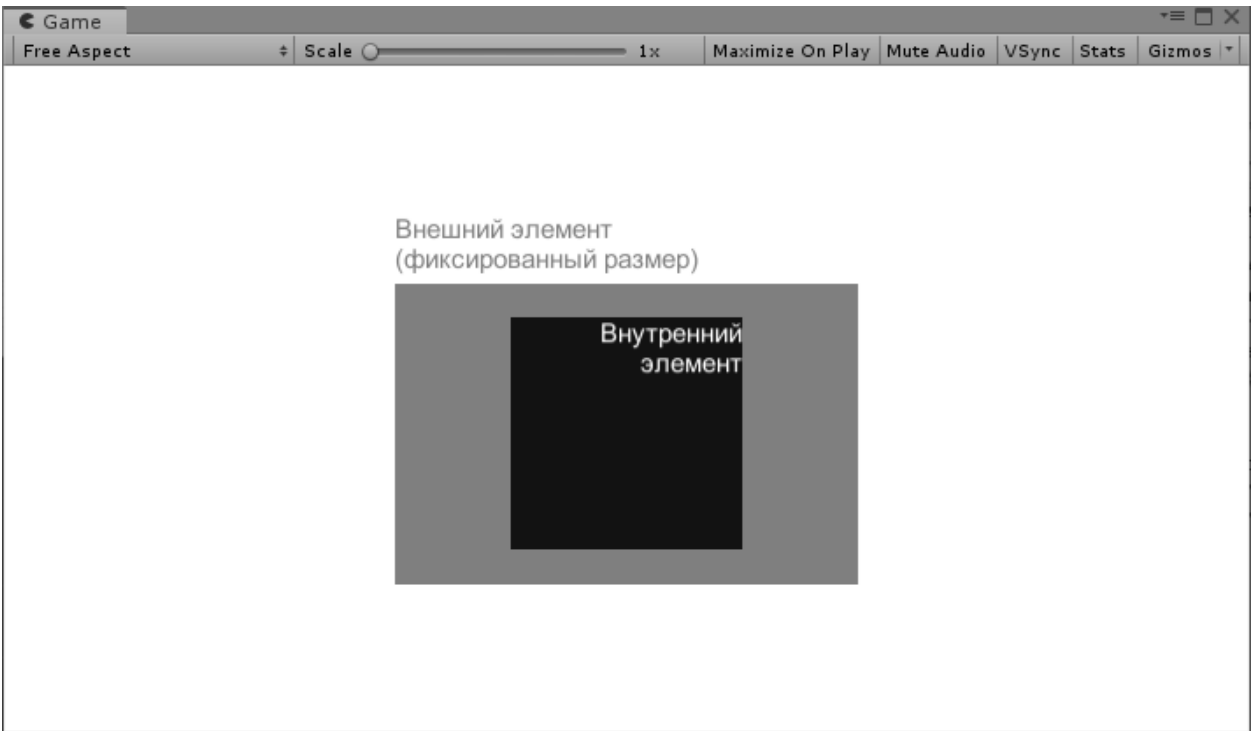


Рисунок В.2 – Тест «Элемент фиксированного размера» (Метод причинно-следственных связей)



Рисунок В.3 – Тест «Размер элемента совпадает с размером вложенного элемента» (Метод причинно-следственных связей)



Рисунок В.4 – Тест на iPad Air (тестирование интерфейса)

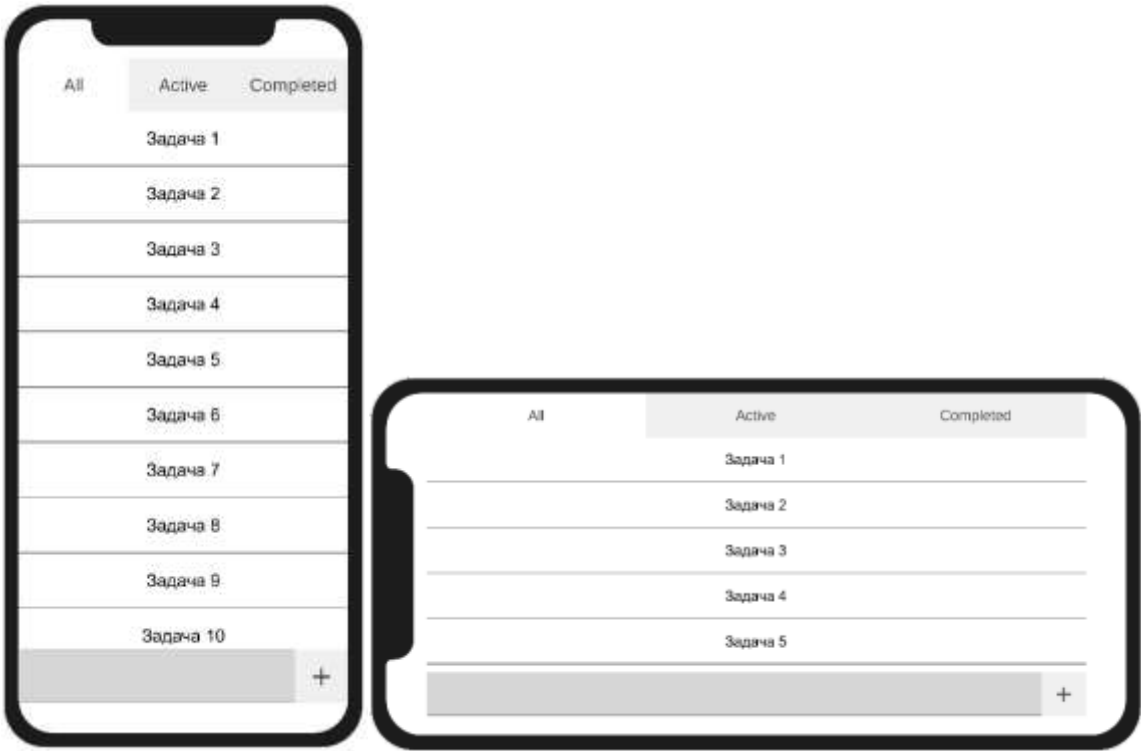


Рисунок В.5 – Тест на iPhone X (тестирование интерфейса)



Рисунок В.6 – Тест на Huawei Mate 20 Pro (тестирование интерфейса)



Рисунок В.7 – Тест на Sony Xperia Z (тестирование интерфейса)

Приложение Г
ИСХОДНЫЙ КОД ТЕСТОВ
(рекомендуемое)

Листинг Г.1 – Код основных unit-тестов

```

[Test]
public void NoSubscribed() => TestZone.Run(tick => {
    var source = Atom.Value(1);
    var middle = Atom.Computed(() => source.Value + 1);
    var target = Atom.Computed(() => middle.Value +
        source.Value);

    Assert.AreEqual(3, target.Value);
    Assert.AreEqual(0, source.SubscribersCount());
    Assert.AreEqual(0, middle.SubscribersCount());
    Assert.AreEqual(0, target.SubscribersCount());
});

[Test]
public void AutoUnsubscribedReaction() => TestZone.Run(tick => {
    var source = Atom.Value(1);
    var middle = Atom.Computed(() => source.Value + 1);
    var target = Atom.Computed(() => middle.Value +
        source.Value);
    var run = Atom.AutoRun(() => target.Get());

    Assert.AreEqual(2, source.SubscribersCount());
    Assert.AreEqual(1, middle.SubscribersCount());
    Assert.AreEqual(1, target.SubscribersCount());

    run.Dispose();
    tick(0);

    Assert.AreEqual(0, source.SubscribersCount());
    Assert.AreEqual(0, middle.SubscribersCount());
    Assert.AreEqual(0, target.SubscribersCount());
});

[Test]
public void EqualStateIgnored() => TestZone.Run(tick => {
    var atom = Atom.Value(new[] {1, 2, 3},
        new TestComparer<int[]>((a, b) => a.SequenceEqual(b)));

    var v1 = atom.Value;
    var v2 = new[] {1, 2, 3};
    atom.Value = v2;
    var v3 = atom.Value;

    Assert.IsTrue(ReferenceEquals(v1, v3));
    Assert.IsFalse(ReferenceEquals(v2, v3));
});

```

```
[Test]
public void AutoActivation() => TestZone.Run(tick => {
    var activation = "";

    var atom = Atom.Computed(
        pull: () => 1,
        onActive: () => activation += "A",
        onInactive: () => activation += "D");

    var listener = Atom.Computed(() => atom.Value + 1, true);

    Assert.AreEqual("", activation);

    listener.Get();
    Assert.AreEqual("A", activation);

    listener.Deactivate();
    Assert.AreEqual("A", activation);

    tick(0);
    Assert.AreEqual("AD", activation);
});
```

```
[Test]
public void Invalidate() => TestZone.Run(tick => {
    var source = Atom.Value(0);

    string actualization = "";

    var dispose = Atom.AutoRun(() => {
        source.Get();
        actualization += "T";
    });

    tick(0);
    Assert.AreEqual("T", actualization);

    source.Invalidate();

    tick(0);
    Assert.AreEqual("TT", actualization);

    dispose.Dispose();
});
```

```

[Test]
public void Caching() => TestZone.Run(tick => {
    var random = new Random();
    var atom = Atom.Computed(() => random.Next(),
        keepAlive: true);

    Assert.AreEqual(atom.Value, atom.Value);
});

[Test]
public void Lazy() => TestZone.Run(tick => {
    var value = 0;
    var atom = Atom.Computed(() => value = 1);

    tick(0f);
    Assert.AreEqual(0, value);

    atom.Get();
    Assert.AreEqual(1, value);
});

[Test]
public void ActualizedInInitialOrder() => TestZone.Run(tick => {
    var actualization = "";

    var source = Atom.Value(1);
    var middle = Atom.Computed(() => {
        actualization += "M";
        return source.Value;
    });
    var target = Atom.Computed(() => {
        actualization += "T";
        source.Get();
        return middle.Value;
    });

    var autoRun = Atom.AutoRun(() => target.Get());
    Assert.AreEqual("TM", actualization);

    source.Value = 2;

    tick(0);
    Assert.AreEqual("TMTM", actualization);

    autoRun.Dispose();
});

```

```
[Test]
public void InstantActualization() => TestZone.Run(tick => {
    var source = Atom.Value(1);
    var middle = Atom.Computed(() => source.Value + 1);
    var target = Atom.Computed(() => middle.Value + 1);

    Assert.AreEqual(3, target.Value);

    source.Value = 2;

    Assert.AreEqual(4, target.Value);
});
```