**СОДЕРЖАНИЕ**

[ОПРЕДЕЛЕНИЯ И СОКРАЩЕНИЯ 5](#_Toc99481053)

[ВВЕДЕНИЕ 6](#_Toc99481054)

[1 ИСПОЛЬЗУЕМЫЕ ТЕХНОЛОГИИ 7](#_Toc99481055)

[1.1 Программная платформа Microsoft .NET 8](#_Toc99481056)

[1.2 Язык программирования C# 11](#_Toc99481057)

[1.3 Язык программирования F# 15](#_Toc99481058)

[1.4 Akka.Net 21](#_Toc99481059)

[2 АРХИТЕКТУРА И МОДУЛИ СИСТЕМЫ 27](#_Toc99481060)

[2.1 Однопоточный модуль 27](#_Toc99481061)

[2.2 Многопоточный модуль 28](#_Toc99481062)

[2.3 Асинхронный модуль 29](#_Toc99481063)

[2.4 Акторный модуль 33](#_Toc99481064)

[3 АНАЛИЗ ДАННЫХ 34](#_Toc99481065)

[3.1 Сравнительные таблицы 34](#_Toc99481066)

[ЗАКЛЮЧЕНИЕ 35](#_Toc99481067)

[СПИСОК ИСПОЛЬЗУЕМЫХ ИСТОЧНИКОВ 36](#_Toc99481068)

[ПРИЛОЖЕНИЕ А 37](#_Toc99481069)

[ПРИЛОЖЕНИЕ Б 45](#_Toc99481070)

# ОПРЕДЕЛЕНИЯ И СОКРАЩЕНИЯ

В настоящей пояснительной записке применяются следующие определения и сокращения.

Спецификация – документ, который желательно полно, точно и верифицируемо определяет требования, дизайн, поведение или другие характеристики компонента или системы, и, часто, инструкции для контроля выполнения этих требований.

Кроссплатформенность – способность программного обеспечения работать более чем на одной аппаратной платформе и (или) операционной системе.

Нативное программное средство – программное средство, специфичное для какой-либо платформы.

Проприетарное программное обеспечение – программное обеспечение, являющееся частной собственностью авторов или правообладателей и не удовлетворяющее критериям свободного ПО: свобода запуска программы в любых целях, свобода адаптации программы для любых нужд, свобода распространения, свобода улучшений исходных кодов и публикации улучшений.

ВУЗ – высшее учебное заведение.

ПС – программное средство.

ПО – программное обеспечение.

БД – база данных.

СУБД – система управления базами данных.

ЯП – язык программирования.

API – application programming interface (сетевой программный интерфейс).

UI – user interface (пользовательский интерфейс).

ТЭО – технико-экономическое обоснование.

# ВВЕДЕНИЕ

Прогресс производства аппаратуры намного опередил развитие технологий программирования. Параллельные архитектуры теперь позволяют принципиально повышать производительность программ решения многих задач. Методы автоматического распараллеливания программ способны обеспечить значительное ускорение вычислений, явно сводимых к комплексу независимых процессов обработки элементов векторов, но отступают перед программами более широкого класса. Перспектива выражения языковыми средствами параллелизма на уровне постановки новой задачи приторможена проблемами оптимизирующей верифицирующей компиляции, обеспечивающей аккуратный выбор эффективной и надежной схемы параллелизма. Кроме того, трудоёмкость повторного программирования и отладки параллельных алгоритмов обуславливает рассмотрение готовых последовательных программ в качестве эталонов при оценке результатов параллельных вычислений, что сдерживает развитие языков параллельного программирования и методов их реализации.

В настоящее время осознана актуальность формирования новых парадигм параллельного программирования и использование уже существующих, вызванная расширением и развитием системы базовых понятий, необходимых для рациональной разработки систем управления процессами на современной аппаратуре. Парадигмы в программировании характеризуются стилем мышления при решении задачи, системой используемых понятий и особенностями их реализации. Можно констатировать, что стиль мышления и система понятий для параллельного программирования уже сложились в процессе эволюции языков программирования, но новые подходы всё ещё появляются.

Одним из таких методов и является акторная модель вычислений, которая будет показана в сравнении с более традиционными подходами в параллельном и асинхронном программировании в рамках выбранного языка программирования.

# 1 ИСПОЛЬЗУЕМЫЕ ТЕХНОЛОГИИ

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

Ниже перечислены некоторые факторы, повлиявшие на выбор технологий:

– разрабатываемое ПО должно работать на операционной системе Windows 7 и более новых версиях системы;

– среди различных платформ разработки имеющийся программист лучше всего знаком с разработкой на платформе Microsoft .NET;

– дальнейшей поддержкой проекта, возможно, будут заниматься разработчики, не принимавшие участие в выпуске первой версии;

– имеющийся разработчик имеет опыт работы с объекто-ориентированными и с функциональными языками программирования.

Основываясь на опыте работы имеющихся программистов разрабатывать ПО целесообразно на платформе Microsoft .NET. Приняв во внимание необходимость обеспечения доступности дальнейшей поддержки ПО, возможно, другой командой программистов, целесообразно не использовать малоизвестные и сложные языки программирования. С учетом этого фактора выбор языков программирования сужается до четырех официально поддерживаемых Microsoft и имеющих изначальную поддержку в Visual Studio 2012: Visual C++/CLI, C#, Visual Basic .NET и F#. Таблица поддержки операционных систем представлена на рисунке 2.1. Необходимость использования низкоуровневых возможностей Visual C++/CLI в разрабатываемом ПО отсутствует, следовательно, данный язык можно исключить из списка кандидатов. Visual Basic .NET уступает по удобству использования двум другим кандидатам из нашего списка. Оставшиеся два языка программирования C# и F# являются первостепенным, элегантными, мультипарадигменными языками программирования для платформы Microsoft .NET. Таким образом, с учетом вышеперечисленных факторов, целесообразно остановить выбор на следующих технологиях:

– операционная система Windows 10;

– платформа разработки Microsoft .NET;

– языки программирования C# и F#.

Поддержка операционных систем приведена на рисунке 2.1



Рисунок 2.1 – Таблица поддержки операционных систем

Для реализации поставленной задачи нет необходимости в использовании каких-либо прикладных библиотек для создания настольных или веб-приложений, достаточно использовать стандартные библиотеки указанных выше языков. Поддержка платформой Microsoft .NET различных языков программирования позволяет использовать язык, который наиболее просто и «красиво» позволяет решить возникающую задачу. Разрабатываемое программное обеспечение в некоторой степени использует данное преимущество платформы. Язык C# больше подходит для создания высокоуровнего дизайна приложения (иерархия классов и интерфейсов, организация пространств имен и публичного программного интерфейса), язык F# – для реализации логики приложения, функций и методов, прототипирования различных идей. В разрабатываемом программном продукте C# используется для предоставления удобного программного интерфейса. Далее проводится характеристика используемых технологий и языков программирования более подробно.

## 1.1 Программная платформа Microsoft .NET

Программная платформа Microsoft .NET является одной из реализаций стандарта ECMA-335 и является современным инструментом создания клиентских и серверных приложений для операционной системы Windows [2]. Первая общедоступная версия .NET Framework вышла в феврале 2002 года. С тех пор платформа активно эволюционировала и на данный момент было выпущено шесть версии данного продукта. На данный момент номер последней версии .NET Framework 4.7. Платформа Microsoft .NET была призвана решить некоторые наболевшие проблемы, скопившиеся на момент ее выхода, в средствах разработки приложений под Windows. Ниже перечислены некоторые из них:

* сложность создания надежных приложений;
* сложность развертывания и управления версиями приложений и библиотек;
* сложность создания переносимого ПО;
* отсутствие единой целевой платформы для создателей компиляторов;
* проблемы с безопасным исполнением непроверенного кода;
* великое множество различных технологий и языков программирования, которые не совместимы между собой.

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

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

– загрузка и исполнение управляемого кода;

– управление памятью при размещении объектов;

– изоляция памяти приложений;

– проверка безопасности кода;

– преобразование промежуточного языка в машинный код;

– доступ к расширенной информации от типов – метаданным;

– обработка исключений, включая межъязыковые исключения;

– взаимодействие между управляемым и неуправляемым кодом (в том числе и COM-объектами);

– поддержка сервисов для разработки (профилирование, отладка и т.д.).

Программы, написанные для Microsoft .NET представляют из себя набор типов, взаимодействующих между собой. Microsoft .NET имеет общую систему типов (Common Type System, CTS). Данная спецификация описывает определения и поведение типов, создаваемых для Microsoft .NET [3]. В частности, в данной спецификации описаны возможные члены типов, механизмы сокрытия реализации, правила наследования, типы-значения и ссылочные типы, особенности параметрического полиморфизма и другие возможности, предоставляемые CLI. Общая языковая спецификация (Common Language Specification, CLS) – подмножество общей системы типов. Это набор конструкций и ограничений, которые являются руководством для создателей библиотек и компиляторов в среде .NET Framework. Библиотеки, построенные в соответствии с CLS, могут быть использованы из любого языка программирования, поддерживающего CLS. Языки, соответствующие CLS (к их числу относятся языки C#, Visual Basic .NET, Visual C++/CLI), могут интегрироваться друг с другом. CLS – это основа межъязыкового взаимодействия в рамках платформы Microsoft .NET. [4]

Некоторые из возможностей, предоставляемых Microsoft .NET: верификация кода, расширенная информация о типах во время исполнения, сборка мусора, безопасность типов, невозможны без наличия подробных метаданных о типах из которых состоит исполняемая программа. Подробные метаданные о типах генерируются компиляторами и сохраняются в результирующих сборках. Сборка – это логическая группировка одного или нескольких управляемых модулей или файлов ресурсов, является минимальной единицей с точки зрения повторного использования, безопасности и управлениями версиями.

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

Ручное управление памятью всегда являлось очень кропотливой и подверженной ошибкам работой. Ошибки в управлении памятью являются одними из наиболее сложных в устранении типами программных ошибок, также эти ошибки обычно приводят к непредсказуемому поведению программы, поэтому в Microsoft .NET управление памятью происходит автоматически. Автоматическое управление памятью является механизмом поддержания иллюзии бесконечности памяти. Когда объект данных перестает быть нужным, занятая под него память автоматически освобождается и используется для построения новых объектов данных. Имеются различные методы реализации такого автоматического распределения памяти [3]. В Microsoft .NET для автоматического управления памятью используется механизм сборки мусора (garbage collection). Существуют различные алгоритмы сборки мусора со своими достоинствами и недостатками. В Microsoft .NET используется алгоритм пометок (mark and sweep) в сочетании с различными оптимизациями, такими как, например, разбиение всех объектов по поколениям и использование различных куч для больших и малых объектов.

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

* обеспечение многопоточного исполнения программы;
* поддержание модели памяти, принятой в CLR;
* поддержка двоичной сериализации;
* управление вводом и выводом;
* структурная обработка исключений;
* возможность размещения исполняющей среды внутри других процессов.

Как уже упоминалось выше, большую ценностью для Microsoft .NET представляет библиотека стандартных классов – соответствующая CLS-спецификации объектно-ориентированная библиотека классов, интерфейсов и системы типов (типов-значений), которые включаются в состав платформы Microsoft .NET. Эта библиотека обеспечивает доступ к функциональным возможностям системы и предназначена служить основой при разработке .NET-приложений, компонент, элементов управления [5].

## 1.2 Язык программирования C#

C# – объектно-ориентированный, типо-безопасный язык программирования общего назначения. Язык создавался с целью повысить продуктивность программистов. Для достижения этой цели в языке гармонично сочетаются простота, выразительность и производительность промежуточного кода, получаемого после компиляции. Главным архитектором и идеологом языка с первой версии является Андрес Хейлсберг (создатель Turbo Pascal и архитектор Delphi). Язык C# является платформенно нейтральным, но создавался для хорошей работы с Microsoft .NET [6]. Этот язык сочетает простой синтаксис, похожий на синтаксис языков C++ и Java, и полную поддержку всех современных объектно-ориентированных концепций и подходов. В качестве ориентира при разработке языка было выбрано безопасное программирование, нацеленное на создание надежного и простого в сопровождении кода.

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

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

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

3 Свойства, методы и события. В чистой объекто-ориентированной парадигме все функции являются методами. В C# методы являются лишь одной из возможных разновидностей членов типа, в C# типы также могут содержать свойства, события и другие члены. Свойство – это такая разновидность функций, которая инкапсулирует часть состояния объекта. Событие – это разновидность функций, которые реагируют на изменение состояния объекта.

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

C# полагается на автоматическое управление памятью со стороны исполняющей среды, предоставляя совсем немного средств для управления жизненным циклом объектов. Несмотря на это, в языке все же присутствует поддержка работы с указателями. Данная возможность предусмотрена для случаев, когда критически важна производительность приложения или необходимо обеспечить взаимодействие с неуправляемым кодом [7].

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

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

Первая версия C# была похожа по своим возможностям на Java 1.4, несколько их расширяя: так, в C# имелись свойства (выглядящие в коде как поля объекта, но на деле вызывающие при обращении к ним методы класса), индексаторы (подобные свойствам, но принимающие параметр как индекс массива), события, делегаты, циклы foreach, структуры, передаваемые по значению, автоматическое преобразование встроенных типов в объекты при необходимости (boxing), атрибуты, встроенные средства взаимодействия с неуправляемым кодом (DLL, COM) и прочее [8].

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

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

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

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

4 Анонимные методы, обеспечивающие функциональность замыканий.

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

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

7 Поддержка 64-разрядных вычислений позволяет увеличить адресное пространство и использовать 64-разрядные примитивные типы данных.

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

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

2 Инициализацию объекта вместе с его свойствами: Customer c = new Customer(); c.Name = "James"; c.Age=30; можно записать как Customer c = new Customer { Name = "James", Age = 30 };

3 Лямбда-выражения: listOfFoo.Where(delegate(Foo x) { return x.size > 10; }); теперь можно записать как listOfFoo.Where(x => x.size > 10);

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

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

x = "hello"; .

6 Безымянные типы: var x = new { Name = "James"}; .

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

8 Автоматические свойства: компилятор сгенерирует закрытое поле и соответствующие аксессор и мутатор для кода вида public string Name { get; private set; }.

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

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

– возможность использования позднего связывания;

– именованные и опциональные параметры;

– новые возможности COM interop;

– ковариантность и контрвариантность интерфейсов и делегатов;

– контракты в коде (Code Contracts).

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

## 1.3 Язык программирования F#

F# – это мультипарадигменный язык программирования, разработанный в подразделении Microsot Research и предназначенный для исполнения на платформе Microsoft .NET. Он сочетает в себе выразительность функциональных языков, таких как OCaml и Haskell с возможностями и объектной моделью Microsoft .NET. История F# началась в 2002 году, когда команда разработчиков из Microsoft Research под руководством Дона Сайма решила, что языки семейства ML вполне подходят для реализации функциональной парадигмы на платформе Microsoft .NET. Идея разработки нового языка появилась во время работы над реализацией обобщенного программирования для Common Language Runtime. Известно, что одно время в качестве прототипа нового языка рассматривался Haskell, но из-за функциональной чистоты и более сложной системы типов потенциальный Haskell.NET не мог бы предоставить разработчикам простого механизма работы с библиотекой классов .NET Framework, а значит, не давал бы каких-то дополнительных преимуществ. Как бы то ни было, за основу был взят OCaml, язык из семейства ML, который не является чисто функциональным и предоставляет возможности для императивного и объектно-ориентированного программирования. Однако Haskell хоть и не стал непосредственно родителем нового языка, тем не менее, оказал на него некоторое влияние. Например, концепция вычислительных выражений (computation expressions или workflows), играющих важную роль для асинхронного программирования и реализации DSL на F#, позаимствована из монад Haskell.

Следует также отметить, что на данный момент F# является, пожалуй, единственным функциональным языком программирования, который продвигается одним из ведущих производителей в области разработки программного обеспечения. Он позволяет использовать множество уже существующих библиотек, писать приложения для самых разных платформ и что не менее важно делать все это в современной IDE.

Далее рассматриваются некоторые из возможностей, предоставляемых F#.

1. Функциональная парадигма. F# предоставляет полный набор инструментов функционального программирования: здесь есть алгебраические типы данных и функции высшего порядка, средства для композиции функций и неизменяемые структуры данных, а также частичное применение на пару с каррированием. Все функциональные возможности F# реализованы в конечном итоге поверх общей системы типов .NET Framework. Однако этот факт не обеспечивает удобства использования таких конструкций из других языков платформы. При разработке собственных библиотек на F# следует предусмотреть создание объектно-ориентированных оберток, которые будет проще использовать из C# или Visual Basic .NET.
2. Императивное программирование. В случаях, когда богатых функциональных возможностей не хватает, F# предоставляет разработчику возможность использовать в коде прелести изменяемого состояния. Это как непосредственно изменяемые переменные, поддержка полей и свойств объектов стандартной библиотеки, так и явные циклы, а также изменяемые коллекции и типы данных.
3. Объектно-ориентированная парадигма. Объектно-ориентированные возможности F#, как и многое другое в этом языке, обусловлены необходимостью предоставить разработчикам возможность использовать стандартную библиотеку классов .NET Framework. С поставленной задачей язык вполне справляется: можно как использовать библиотеки классов, реализованные на других .NET языках, так и разрабатывать свои собственные. Следует отметить, однако, что некоторые возможности ОО языков реализованы не самым привычным образом.
4. Система типов. Каждая переменная, выражение или функция в F# имеет тип. Можно считать, что тип – это некий контракт, поддерживаемый всеми объектами данного типа. К примеру, тип переменной однозначно определяет диапазон значений этой переменной; тип функции говорит о том, какие параметры она принимает и значение какого типа она возвращает; тип объекта определяет то, как этот объект может быть использован.
5. F# – статически типизированный язык. Это означает, что тип каждого выражения известен на этапе компиляции, и позволяет отслеживать ошибки, связанные с неправильным использованием объектов определенного типа, до запуска программы. Помимо этого, F# – язык программирования со строгой типизацией, а значит, в выражениях отсутствует неявное приведение типов. Попытка использовать целое число там, где компилятор ожидает увидеть число с плавающей точкой, приведет к ошибке компиляции.
6. Вывод типов. В отличие от большинства других промышленных языков программирования, F# не требует явно указывать типы всех значений. Механизм вывода типов позволяет определить недостающие типы значений, глядя на их использование. При этом некоторые значения должны иметь заранее известный тип. В роли таких значений, к примеру, могут выступать числовые литералы, так как их тип однозначно определяется суффиксом. На рисунке 2.2 приведен простой пример:

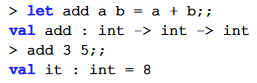


Рисунок 2.2 – Пример автоматического вывода типа функции

Функция add возвращает сумму своих параметров и имеет тип int -> int -> int. Если не смотреть на сигнатуру функции, то можно подумать, что она складывает значения любых типов, но это не так. Попытка вызвать ее для аргументов типа float или decimal приведет к ошибке компиляции. Причина такого поведения кроется в механизме вывода типов. Поскольку оператор + может работать с разными типами данных, а никакой дополнительной информации о том, как эта функция будет использоваться, нет, компилятор по умолчанию приводит ее к типу int -> int -> int.

В большинстве случаев автоматический вывод типов является очень удобным и способствует написанию полиморфный функций. Алгоритм, используемый компилятором F#, использует глобальный вывод типов и позволяет справляться даже с очень сложными типами. Для демонстрации возможностей вывода типа для полиморфных функций рассмотрим классический пример – комбинаторный базис SKI:

*I =* λ*x.x* (2.1)

*K =* λ*x y.x* (2.2)

*S =* λ*f g x.(fx)(gx)* (2.3)

Пример вывода интерактивного окружения F# приведен в листинге 2.2. Как можно заметить типы полученных функций довольно сложные, но компилятор смог их вывести.

На рисунке 2.3 приведен пример определения комбинаторного базиса SKI:

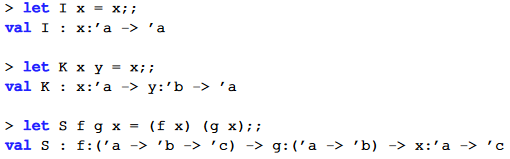


Рисунок 2.3 – Пример определения комбинаторного базиса SKI

Обычно при программировании на F# в функциональном стиле нет необходимости указывать типы явно, транслятор сам назначит выражению наиболее общий тип. Однако, иногда бывает полезно ограничить вывод типа. Подобная мера не заставит работать код, который до этого не работал, но может использоваться как документация для понимания его предназначения; также возможно использовать более короткие синонимы для сложных типов. Ограничение типа может быть задано в F# путем добавления аннотации типа после некоторого выражения. Аннотации типов состоят из двоеточия, за которым указан тип. Обычно расположение аннотаций не имеет значения; если они есть, то они заставляют компилятор использовать соответствующие ограничения.

1. Стратегии вычислений. Обычно выражения в F# вычисляются «энергично». Это означает, что значение выражения будет вычислено независимо от того, используется оно где-либо или нет. В противоположность жадному подходу существует стратегия ленивых вычислений, которая позволяет вычислять значение выражения только тогда, когда оно становится необходимо. Преимуществами такого подхода являются:

– производительность, поскольку неиспользуемые значения просто не вычисляются;

– возможность работать с бесконечными или очень большими последовательностями, так как они никогда не загружаются в память полностью;

– декларативность кода.

Использование ленивых вычислений избавляет программиста от необходимости следить за порядком вычислений, что делает код проще. Главный недостаток ленивых вычислений – плохая предсказуемость. В отличие от энергичных вычислений, где очень просто определить пространственную и временную сложность алгоритма, с ленивыми вычислениями все может быть куда менее очевидно. F# позволяет программисту самостоятельно решать, что именно должно вычисляться лениво, а что нет. Это в значительной степени устраняет проблему плохой предсказуемости, так как ленивые вычисления применяются только там, где это действительно необходимо, позволяя сочетать лучшее из обоих миров.

1. Сопоставление с образцом. Образец – это описание «формы» ожидаемой структуры данных: образец сам по себе похож на литерал структуры данных (т.е. он состоит из конструкторов алгебраических типов и литералов примитивных типов: целых, строковых и т.п.), однако может содержать метапеременные – «дырки», обозначающие: «значение, которое встретится в данном месте, назовем данным именем». Сопоставление с образцом – основной способ работы со структурами данных в F#. Эта языковая конструкция состоит из ключевого слова match, анализируемого выражения, ключевого слова with и набора правил. Каждое правило – это пара образец-результат. Все выражение сопоставления с образцом принимает значение того правила, образец которого соответствует анализируемому выражению. Все правила сопоставления с образцом должны возвращать значения одного и того же типа. В простейшем случае в качестве образцов могут выступать константы:

> let xor x y =

match x, y with

| true, true -> false

| false, false -> false

| true, false -> true

| false, true -> true ;;

val xor : bool -> bool -> bool

В правилах сопоставления с образцом можно использовать символ подчеркивания, если конкретное значение неважно. Если набор правил сопоставления не покрывает всевозможные значения образца, компилятор выдает предупреждение. На этапе исполнения, если ни одно правило не будет соответствовать образцу, будет сгенерировано исключение. Сопоставление с образцом очень мощный механизм, но иногда и его выразительности недостаточно для описания идеи. Язык F# вводит понятие активных шаблонов, когда шаблон может представлять из себя пользовательскую функцию, которая может содержать дополнительную логику обработки.

1. Вычислительные выражения. Среди нововведений F# можно особо выделить так называемые вычислительные выражения (computation expressions или workflows). Они являются обобщением генераторов последовательности и, в частности, позволяют встраивать в F# такие вычислительные структуры, как монады и моноиды. Также они могут быть применены для асинхронного программирования и создания DSL. Вычислительное выражение имеет форму блока, содержащего некоторый код на F# в фигурных скобках. Этому блоку должен предшествовать специальный объект, который называется еще построителем (builder). Общая форма следующая: builder { comp-expr }. Построитель определяет способ интерпретации того кода, который указан в фигурных скобках. Сам код вычисления внешне почти не отличается от обычного кода на F#, кроме того, что в нем нельзя определять новые типы, а также нельзя использовать изменяемые значения. Вместо таких значений можно использовать ссылки, но делать это следует с большой осторожностью, поскольку вычислительные выражения обычно задают некие отложенные вычисления, а последние не очень любят побочные эффекты.
2. Асинхронные потоки операций. Асинхронные потоки операций – это один из самых интересных примеров практического использования вычислительных выражений. Код, выполняющий какие-либо неблокирующие операции ввода-вывода, как правило сложен для понимания, поскольку представляет из себя множество методов обратного вызова, каждый из которых обрабатывает какой-то промежуточный результат и возможно начинает новую асинхронную операцию. Асинхронные потоки операций позволяют писать асинхронный код последовательно, не определяя методы обратного вызова явно. Для создания асинхронного потока операций используется построитель async.

## 1.4 Akka.Net

Это, по сути, безальтернативная реализация модели акторов в среде .NET. Есть API как для C#, так и для F#.

Краткий список возможностей библиотеки:

* базовые типы для создания акторов;
* большое количество готовых типов для обычных задач вроде маршрутизации сообщений, балансировки нагрузки между акторами и так далее;
* управление mailboxи разные варианты реализации для него;
* «среда выполнения» для акторов — Actor System;
* управление именами акторов (впрочем, всегда можно написать что-то свое);
* удаленный обмен сообщениями с другими акторами по сети;
* кластеризация системы акторов — возможность запустить приложение из коробки на нескольких машинах;
* встроенная очередь для dead letter;
* поддержка DI;
* поддержка персистентных акторов.

Сразу перечислю некоторые недостатки:

* Сложная и неочевидная конфигурация библиотеки:
* не все задокументированные настройки работают;
* некоторые не работают так, как задумано;
* некоторые вообще не работают;
* Непривычные подходы к именованию (dead letters, например) и концепции некоторых сущностей в библиотеке. Не все сразу понятно, документация тоже не всегда помогает. Благо, Akka.NET — это библиотека [с открытым исходным кодом](https://github.com/akkadotnet/akka.net) и всегда можно взять исходники, собрать из них библиотеку и посмотреть, что там и как внутри происходит.
* Крутая кривая обучения. Гайд по библиотеке ровно один, плюс есть несколько примеров (которые не всегда рабочие). Документация не всегда исчерпывающая.

В Akka.NET есть ряд встроенных типов, которые можно использовать для поочередной отправки сообщений группам акторов. Эти типы называют Routers. В библиотеке есть несколько реализаций для самых частых сценариев:

* Round-robin — последовательная отправка сообщений акторам.
* Hash key — адресация по ключу в сообщении.
* Random — адресат выбирается случайным образом.
* Weighted round robin — то же, что и Round-robin, но с помощью весовых коэффициентов можно настраивать частоту отправки сообщений конкретным адресатам.
* Broadcast — сообщение отправляется всем акторам из группы.

Перейдем теперь к реальному примеру использования Akka.NET. Есть некоторая система, которая обрабатывает информацию с датчиков. Как уже было сказано, датчики не подключены напрямую к серверу системы из-за довольно скромных возможностей для автономной связи на дальние расстояния и особенностей топологии их размещения.

Информация с оконечных устройств поступает в систему через стационарные шлюзы.

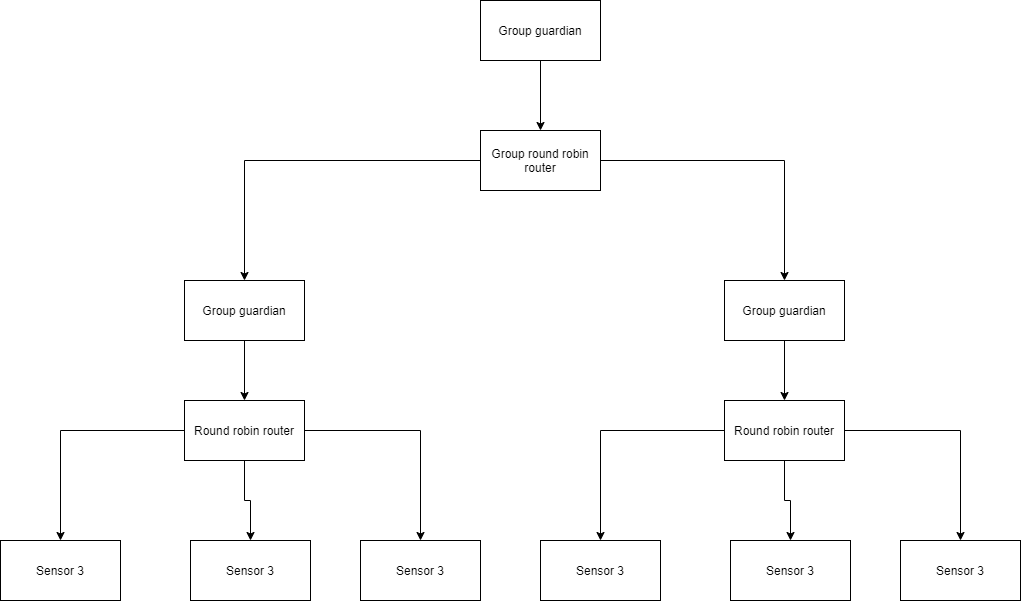
Представим каждый шлюз в виде актора. Тогда актор шлюза может создавать акторы оконечных устройств, которые будут к нему подключены. Таким образом у нас получилась небольшая иерархия устройств.

Добавим в эту иерархию актор, который будет обрабатывать ошибки оконечных устройств — между шлюзом и датчиком. Он будет периодически опрашивать датчик на предмет его состояния и отправлять команду перезагрузки, если что-то с оконечным устройством не так. Добавим также актор над шлюзом, который возьмет на себя управление нестабильным TCP-соединением и будет заниматься буферизацией, повторной отправкой сообщений от шлюза на сервер при необходимости.

На минутку перестанем добавлять акторы, остановимся и осмотримся. Как видно, основной способ расширения системы — это добавление в нее новых акторов. Они могут брать на себя произвольные функции — как расширение поведения существующих акторов, так и полностью новые сценарии в системе. Это позволяет расширять систему с минимальными затратами на внедрение нового функционала.

Вернемся к нашему примеру со шлюзом и оконечными устройствами, которые к нему подключены. Усложняем задачу: у нас есть 6 устройств, которые измеряют наличие или отсутствие группы химических соединений в жидкости в двух параллельных трубопроводах.

Устройства производят измерения по команде — погружают измерительный стержень, а затем каким-то образом читают показания. Сами устройства дублируют друг друга (группами по 3) и разделены на две группы. Нет смысла постоянно запрашивать измерения со всех шести датчиков — это нецелесообразно, потому что реактивы не дешевые. Вместо этого мы можем по очереди опрашивать каждую группу и отправлять команду на измерение с помощью Round-robin router.



Это был пример базового сценария использования Akka.NET. Перейдем теперь к более сложному. Естественный вопрос, который хочется задать следующим — как мне масштабировать систему? Как сделать её более отказоустойчивой?

Здесь также на одном уровне находится библиотека user32.dll. И хотя выше говорилось, что WPF не использует эту библиотеку для рендеринга и визуализации, однако для ряда вычислительных задач (не включающих визуализацию) данная библиотека продолжает использоваться.

Прежде чем затрагивать вопросы про масштабируемость, нужно понять, как акторы на разных физических машинах обмениваются сообщениями. За счет ссылочной прозрачности и асинхронной природы обмена сообщениями в модели акторов легко сделать переход от in-process коммуникации до сетевой. Это принципиально отличает модель акторов от, например, DCOM или RPC (в виде WCF к примеру), где за синхронными вызовами методов стояли асинхронные вызовы по сети.

В Akka.NET уже есть расширение для обмена сообщениями — Akka.Remoting. Что умеет Akka.Remoting:

отправлять сообщения на удаленные акторы;

развертывать акторы удаленно (на принимающей стороне);

есть возможность расширения для поддержки других сетевых протоколов, не только TCP и UDP.

Важные замечания при использовании Akka.Remoting

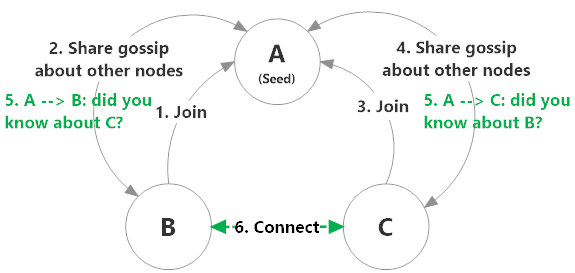
При использовании Akka для сетевой коммуникации важно ДО того, как проект начинает разрастаться, продумать, как будут сериализоваться сообщения.

Сериализатор практически невозможно поменять после того, как ваш проект достигнет определенных размеров. Вам нужно будет поддерживать две параллельные копии системы с разными сериализаторами для апгрейда — Akka.NET (впрочем, как и любая другая распределенная система) не сможет десериализовать сообщения, если они были упакованы другим способом. Никакой метаинформации про тип транспортного протокола с сообщениями не передается. Впрочем, всегда можно написать свой сериализатор, который будет абстракцией над механизмом сериализации.

Если мы хотим объединить множество акторов, размещенных на разных машинах, в один логический кластер, то для этого существует специальное расширение — Akka.Cluster. Оно реализует логику работы кластера акторов поверх Akka.Remoting. Поговорим о том, как работает кластер в Akka.NET. Для работы кластера акторов необходимо, чтобы выполнялись следующие ограничения.

На каждой машине система акторов (ActorSystem) должна иметь одинаковое название. Не обязательно, чтобы в каждой системе был один и тот же набор акторов, важно именно название. Название системы принимает участие в построении адреса к актору внутри кластера.

В кластере обязательныо должны присутствовать seed nodes. Akka.Cluster реализован поверх протокола Gossip (рис. 3). Назначение протокола — в per-to-peer обмене информацией между узлами сети или, в нашем случае — информацией о машинах в кластере. Сам по себе протокол никак не регламентирует добавление новых участников сети в кластер. Чтобы обойти это, в Akka.Cluster есть seed nodes — это точки входа в кластер. Они указываются как доменные имена/адреса узлов в соответствующей секции конфигурационного файла приложения. Если в кластере нет seed node или она вышла из строя, то новые машины не смогут присоединиться к кластеру. Поэтому имеет смысл держать 2 и более seed nodes для обеспечения отказоустойчивости кластера.

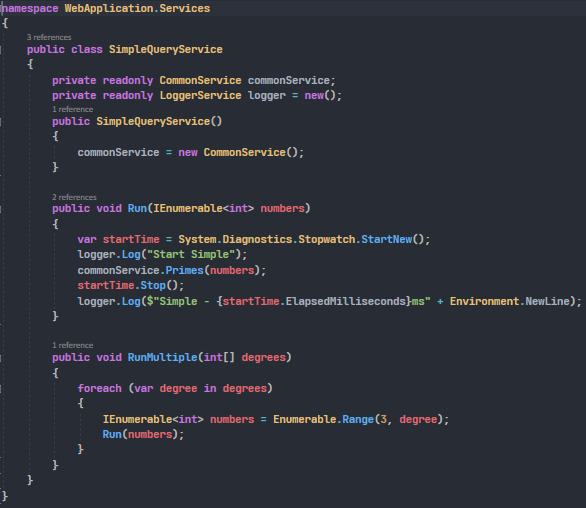


Akka.NET — это неплохая альтернатива конвенциональным подходам для построения распределенных систем. Из коробки вы получаете большое количество расширяемой и модифицируемой функциональности. Модули в библиотеке достаточно независимы друг от друга. Используя предоставляемую функциональность как фундамент, можно ускорить разработку IoT-решения и упростить его поддержку. К тому же базовые концепции библиотеки довольно простые, и при правильном подходе к разделению ответственности между акторами можно построить легко расширяемую систему.

# 2 АРХИТЕКТУРА И МОДУЛИ СИСТЕМЫ

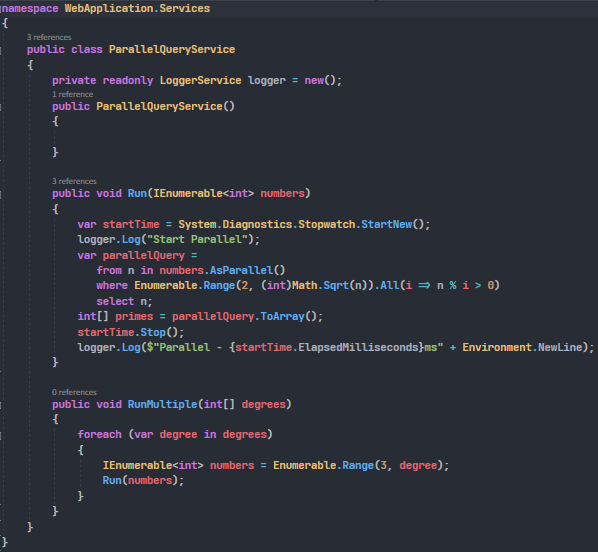
## 2.1 Однопоточный модуль

Самый простой модуль системы, призванный продемонстрировать решение проблемы «в лоб», с которым будут сравниваться последующие решения. В этом модуле будет производиться некоторый математический расчёт, сложностью которого будет изменяться. Таким образом будет построен график времени выполнения от сложности, который и будет основным источником информации.



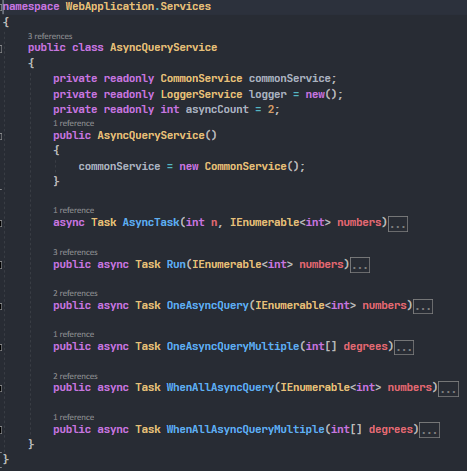
В методе Run мы вычисляем одинарно, логируя данные и время, в то время как в методе RunMultiple мы производим расчёт по нескольким уровням сложности, впоследствии используя эти данные для построения графиков и сравнения производительности.

## 2.2 Многопоточный модуль



В методе Run мы вычисляем одинарно, логируя данные и время, в то время как в методе RunMultiple мы производим расчёт по нескольким уровням сложности, впоследствии используя эти данные для построения графиков и сравнения производительности.

## 2.3 Асинхронный модуль



В данном модуле есть отдельные методы для асинхронного выполнения синхронных и многопоточных методов, выполнения подряд идущих и одновременно выполняемых асинхронных задач.

Асинхронный подход следует использовать при реализации операции службы, если реализация операции службы осуществляет блокирующий вызов, например операцию ввода-вывода. При реализации асинхронной операции попробуйте вызывать асинхронные операции и методы, чтобы, насколько это возможно, расширить функции асинхронного вызова. Например вызовите метод BeginOperationTwo() из метода BeginOperationOne().

* Асинхронный подход следует использовать в клиенте или вызывающем приложении в следующих случаях.
* Если операции вызываются из приложения промежуточного уровня. (Дополнительные сведения о таких сценариях см. в статье о [клиентских приложениях среднего уровня](https://docs.microsoft.com/ru-ru/dotnet/framework/wcf/feature-details/middle-tier-client-applications).)
* Если операции вызываются на страницах ASP.NET, следует использовать асинхронные страницы.
* Если операции вызываются из любого однопоточного приложения, например приложения Windows Forms или WCF (Windows Presentation Foundation). При использовании модели асинхронных вызовов на основе событий результирующее событие создается в потоке пользовательского интерфейса, в результате чего приложение получает возможность реагирования на действия пользователя, но при этом не требуется управлять несколькими потоками.
* В общем случае при выборе между синхронным и асинхронным вызовом следует выбирать асинхронный вызов.

Реализация асинхронной операции службы

Асинхронные операции могут быть реализованы с помощью одного из трех следующих методов:

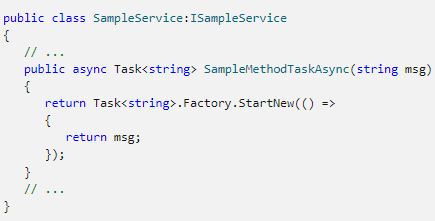
асинхронная модель на основе задач;

асинхронная модель на основе событий;

асинхронная модель IAsyncResult.

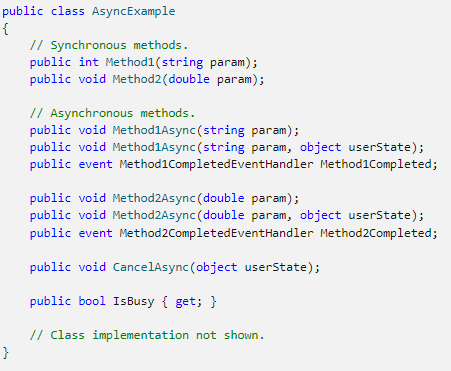
Асинхронная модель на основе задач

Асинхронная модель на основе задач — это предпочтительный способ реализации асинхронных операций в силу его чрезвычайного удобства и простоты. Чтобы использовать этот метод, просто реализуйте операцию службы и укажите тип возвращаемого значения задачи <t>, где t — это тип, возвращаемый логической операцией. Пример:



Служба, поддерживающая асинхронную модель на основе событий, будет содержать одну или несколько операций с именем MethodNameAsync. Эти методы могут копировать синхронные версии, выполняющие ту же операцию в текущем потоке. Этот класс также может содержать событие MethodNameCompleted, а также метод MethodNameAsyncCancel (или просто CancelAsync). Клиент, вызывающий операцию, определяет обработчик событий, вызываемый после завершения операции.

В следующем фрагменте кода показано объявление асинхронных операций с помощью асинхронной модели на основе событий.



Операцию службы можно реализовать асинхронно с помощью платформа .NET Framework шаблона асинхронного программирования и пометить <Begin> метод [AsyncPattern](https://docs.microsoft.com/ru-RU/dotnet/api/system.servicemodel.operationcontractattribute.asyncpattern) свойством, имеющим значение true . В этом случае асинхронная операция доступна в метаданных так же, как и синхронная операция: она предоставляется в виде одной операции с сообщением запроса и согласованным с ним сообщением ответа. В этом случае имеется возможность выбора одной из двух моделей программирования клиента. Этот шаблон может быть представлен в них в виде синхронной или асинхронной операции, поскольку при вызове службы имеет место обмен сообщениями "запрос-ответ".

В целом в связи с асинхронной природой систем полагаться на потоки не следует. Самый надежный способ передачи данных на различные этапы обработки диспетчеризации операций — это использование расширений.

Чтобы определить операцию контракта X, которая выполняется асинхронно независимо от того, как она вызывается в клиентском приложении, выполните следующие действия:

* Определите два метода, используя шаблон BeginOperation и EndOperation.
* Метод BeginOperation включает параметры in и ref для операции и возвращает значение типа [IAsyncResult](https://docs.microsoft.com/ru-RU/dotnet/api/system.iasyncresult).
* Метод EndOperation включает параметр [IAsyncResult](https://docs.microsoft.com/ru-RU/dotnet/api/system.iasyncresult), а также параметры out и ref, и возвращает результат возвращаемого типа операции.

Например, следующий метод:

int DoWork(string data, ref string inout, out string outonly)

Для создания асинхронной операции эти два метода должны иметь следующий вид.

[OperationContract(AsyncPattern=true)]

IAsyncResult BeginDoWork(string data,

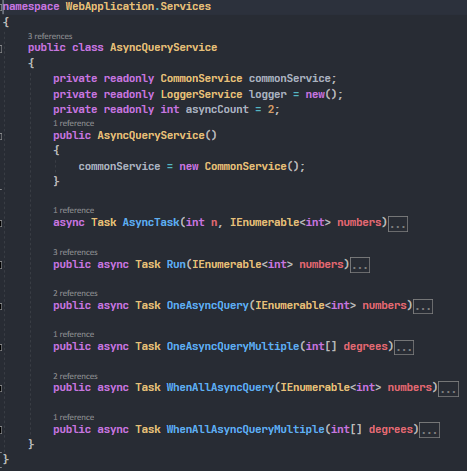
ref string inout,

AsyncCallback callback,

object state);

int EndDoWork(ref string inout, out string outonly, IAsyncResult result);

## 2.4 Акторный модуль

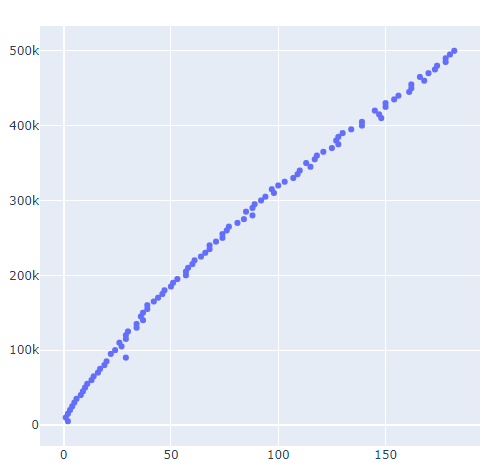


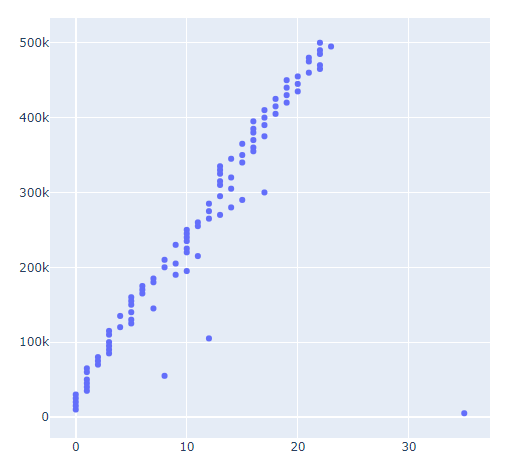
В данном модуле есть отдельные методы для асинхронного выполнения синхронных и многопоточных методов, выполнения подряд идущих и одновременно выполняемых асинхронных задач.

# 3 АНАЛИЗ ДАННЫХ

## 3.1 Сравнительные таблицы

## 3.2 Графическое представление





## 3.3 Комплексный вывод

# ЗАКЛЮЧЕНИЕ

В данной дипломной работе был рассмотрен вопрос написания библиотеки для работы с телеметрией, поступающей со спутника и созданием приложения для ее корректного отображения и анализа.

Были получены удовлетворительные результаты в виде программного средства для сбора и анализа телеметрии с использованием MVVM-паттерна и методов реактивного программирования. Такой подход оставляет широкую возможность для расширения приложения, увеличения количества полезных функций и возможного переноса проекта на другую операционную систему.

Программа осуществляет показ телеметрии в десктоп приложении, отвечая требованиям выдачи информации в комфортном для конечного потребителя виде, а также оставляя возможность для комфортного добавления дополнительных модулей программы.

Проведен обзор существующих методов создания настольных приложений для работы с большим объемом информации.

Разработаны и проанализированы существующие подходы и архитектурные решения в создании реактивного, кроссплатформенного настольного приложения для показа телеметрии.

# СПИСОК ИСПОЛЬЗУЕМЫХ ИСТОЧНИКОВ

1. Джейсон Visual C# .NET. Полное руководство / Джейсон, Майк Прайс; , Гандэрлой. - М.: Корона Принт, 2004. - 960 c.  
2. Нейгел, К. C# 2005 для профессионалов / К. Нейгел. - М.: Вильямс, 2006. - 476 c.  
3. Рихтер CLR via C#. Программирование на платформе Microsoft .NET Framework 2.0 на языке C# / Рихтер, Джефри. - М.: Питер, 2007. - 656 c.  
4. Робинсон, С. C# для профессионалов / С. Робинсон, О. Корнес, Д. Глинн, и др.. - М.: ЛОРИ, 2005. - 930 c.

# ПРИЛОЖЕНИЕ А

**(обязательное)**

**Листинг программы**

//MainWindow.xaml

<Window x:Class="Medusa.Analyze1553B.UI.MainWindow"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:d="http://schemas.microsoft.com/expression/blend/2008"

xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"

xmlns:local="clr-namespace:Medusa.Analyze1553B.UI"

mc:Ignorable="d"

Title="{Binding Title}" Height="450" Width="1200">

<Grid>

<Grid.ColumnDefinitions>

<ColumnDefinition/>

<ColumnDefinition/>

<ColumnDefinition/>

<ColumnDefinition/>

</Grid.ColumnDefinitions>

<Grid.RowDefinitions>

<RowDefinition Height="auto"/>

<RowDefinition/>

<RowDefinition/>

<RowDefinition/>

<RowDefinition/>

</Grid.RowDefinitions>

<Grid Grid.Row="0" Grid.Column="1">

<Grid.ColumnDefinitions>

<ColumnDefinition/>

<ColumnDefinition/>

</Grid.ColumnDefinitions>

</Grid>

<Grid Grid.Row="1" Grid.Column="1">

<Grid.ColumnDefinitions>

<ColumnDefinition/>

<ColumnDefinition/>

</Grid.ColumnDefinitions>

</Grid>

<Grid Grid.Row="0" Grid.Column="0" Grid.ColumnSpan="4">

<Menu x:Name="MainMenu" DataContext="{Binding Commands}"

VerticalAlignment="Center">

<MenuItem Header="Menu ">

<MenuItem Header="AddTemplateRow" Command="{Binding CreateTestTableCommand}"

InputGestureText="Ctrl+T"/>

<MenuItem Header="Open" Command="{Binding OpenXmlForTableCreationCommand}"

InputGestureText="Ctrl+N"/>

<MenuItem Header="Save" Command="{Binding SaveXmlFromTableCommand}"

InputGestureText="Ctrl+S"/>

</MenuItem>

<MenuItem Header="Clear" Command="{Binding ClearTableCommand}" InputGestureText="Ctrl+L"/>

<MenuItem Header="Help" Command="{Binding ShowHelpInformationCommand}" InputGestureText="Ctrl+H"/>

</Menu>

</Grid>

<ListView Grid.Row="1" Grid.RowSpan="3" Grid.Column="0" Grid.ColumnSpan="3"

ItemsSource="{Binding dataRecordsList}">

<ListView.View>

<GridView>

<GridViewColumn DisplayMemberBinding="{Binding Path=Index}">Index</GridViewColumn>

<GridViewColumn DisplayMemberBinding="{Binding Path=MonitorTime}">MonitorTime</GridViewColumn>

<GridViewColumn DisplayMemberBinding="{Binding Path=Channel}">Channel</GridViewColumn>

<GridViewColumn DisplayMemberBinding="{Binding Path=Error}">Error</GridViewColumn>

<GridViewColumn DisplayMemberBinding="{Binding Path=Cw1.Value}">Cw1</GridViewColumn>

<GridViewColumn DisplayMemberBinding="{Binding Path=Cw2.Value}">Cw2</GridViewColumn>

<GridViewColumn DisplayMemberBinding="{Binding Path=Rw1.Value}">Rw1</GridViewColumn>

<GridViewColumn DisplayMemberBinding="{Binding Path=Rw2.Value}">Rw2</GridViewColumn>

<GridViewColumn DisplayMemberBinding="{Binding Path=Data.Length}">Length</GridViewColumn>

</GridView>

</ListView.View>

</ListView>

</Grid>

</Window>

// MainWindow.cs

using Medusa.Analyze1553B.UIServices;

using System;

using System.Globalization;

using System.IO;

using System.Text;

using System.Windows;

using System.Windows.Data;

using SynchronizationContext = System.Threading.SynchronizationContext;

namespace Medusa.Analyze1553B.UI

{

/// <summary>

/// Логика взаимодействия для MainWindow.xaml

/// </summary>

public partial class MainWindow : Window, ISynchronizationContextProvider, IDialogService

{

readonly Microsoft.Win32.OpenFileDialog openFileDlg;

readonly Microsoft.Win32.SaveFileDialog saveFileDialog;

string Filter = "Xml files (\*.xml)|\*.xml|All files (\*.\*)|\*.\*";

public SynchronizationContext SynchronizationContext { get; }

public MainWindow()

{

InitializeComponent();

SynchronizationContext = SynchronizationContext.Current;

openFileDlg = new Microsoft.Win32.OpenFileDialog

{

DefaultExt = ".xml",

Filter = this.Filter

};

saveFileDialog = new Microsoft.Win32.SaveFileDialog()

{

Filter = this.Filter

};

}

public MemoryStream ShowOpenFileDialog()

{

if (openFileDlg.ShowDialog() == true)

{

byte[] byteArray = Encoding.ASCII.GetBytes(openFileDlg.FileName);

MemoryStream stream = new MemoryStream(byteArray);

return stream;

}

else

{

return new MemoryStream();

}

}

public MemoryStream ShowSaveFileDialog()

{

if (saveFileDialog.ShowDialog() == true)

{

byte[] byteArray = Encoding.ASCII.GetBytes(openFileDlg.FileName);

MemoryStream stream = new MemoryStream(byteArray);

return stream;

}

else

{

return new MemoryStream();

}

}

public void CreateCustomWindow()

{

MessageBox.Show("Здесь будет кастомное окно!");

}

public void Message(string message, string title = null)

{

MessageBox.Show(this, message, title, MessageBoxButton.OK, MessageBoxImage.Information);

}

}

}

//Commands.cs

using System;

using System.Reactive.Concurrency;

using System.Threading.Tasks;

using System.Windows.Input;

using System.Data;

using System.Xml;

using System.Text;

using System.Xml.Schema;

using System.Xml.Linq;

using System.IO;

using ReactiveUI;

using Medusa.Analyze1553B.UIServices;

using Medusa.Analyze1553B.VMServices;

namespace Medusa.Analyze1553B.VM

{

public partial class Commands

{

private readonly IScheduler scheduler;

private readonly IVmObject vmObject;

private readonly IDialogService dialogService;

private readonly IDataService dataService;

#region Commands

public ICommand TransactionsCommand { get; }

public ICommand DepositCommand { get; }

public ICommand WithdrawalCommand { get; }

public ICommand OpenXmlForTableCreationCommand { get; }

public ICommand SaveXmlFromTableCommand { get; }

public ICommand ShowHelpInformationCommand { get; }

#endregion

public Commands(ISynchronizationContextProvider syncContext, IVmObject vmObject, IDialogService dialogService, IDataService dataService)

//public Commands(ISynchronizationContextProvider syncContext, IDialogService dialogService)

{

this.vmObject = vmObject;

this.dialogService = dialogService;

this.dataService = dataService;

this.scheduler = new SynchronizationContextScheduler(syncContext.SynchronizationContext);

//TransactionsCommand = CreateCommand(Transactions);

//DepositCommand = CreateCommand(Deposit);

//WithdrawalCommand = CreateCommand(Withdrawal);

OpenXmlForTableCreationCommand = CreateCommand(OpenXmlForTableCreation);

SaveXmlFromTableCommand = CreateCommand(SaveXmlFromTable);

ShowHelpInformationCommand = CreateCommand(ShowHelpInformation);

}

#region Command Implementation

private void Transactions()

{

\_ = this.vmObject.\_account.Transactions();

}

private void Deposit()

{

\_ = this.vmObject.\_account.Deposit(this.vmObject.depositedAmount);

}

private void Withdrawal()

{

\_ = this.vmObject.\_account.Withdrawal(this.vmObject.withdrawnAmount);

}

private void OpenXmlForTableCreation()

{

StreamReader reader = new StreamReader(dialogService.ShowOpenFileDialog());

string path = reader.ReadToEnd();

}

private void SaveXmlFromTable()

{

// convert stream to string

StreamReader reader = new StreamReader(dialogService.ShowSaveFileDialog());

string path = reader.ReadToEnd();

}

private void ShowHelpInformation()

{

dialogService.CreateCustomWindow();

}

#endregion

ICommand CreateCommand(Func<Task> command) => ReactiveCommand.CreateFromTask(command, outputScheduler: scheduler);

ICommand CreateCommand<T>(Func<T, Task> command) => ReactiveCommand.CreateFromTask(command, outputScheduler: scheduler);

ICommand CreateCommand(Action command) => ReactiveCommand.Create(command, outputScheduler: scheduler);

ICommand CreateCommand<T>(Action<T> command) => ReactiveCommand.Create(command, outputScheduler: scheduler);

}

}

//TypeResolver.cs

using System;

using StructureMap;

namespace Medusa.Analyze1553B.DI

{

public class TypeResolver

{

public static object Resolve(Type type) => Default.Resolve(type);

public static T Resolve<T>() where T : class => Default.Resolve<T>();

public static IContainer Container => Default?.Container;

public static TypeResolverInstance CreateNew()

{

return new TypeResolverInstance(Default.Container);

}

private static readonly TypeResolverInstance Default = new TypeResolverInstance(null);

}

public class TypeResolverInstance : IDisposable

{

internal TypeResolverInstance(IContainer parentContainer)

{

Container = parentContainer?.CreateChildContainer() ?? new Container();

}

public object Resolve(Type type)

{

return Container.GetInstance(type);

}

public T Resolve<T>() where T : class

{

return Container.GetInstance<T>();

}

public IContainer Container { get; }

public void Dispose()

{

Container.Dispose();

}

}

}

# ПРИЛОЖЕНИЕ Б

**(рекомендуемое)**

**Входной файл**

22019040 0 0000 0000 0 A3C1 FFFF A000 FFFF 684E FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF 0000 0000

22019376 0 0400 0000 0 A7C1 FFFF A000 FFFF 684E FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF # 0001 0001

22019696 0 8000 0000 1 A3C1 FFFF A000 FFFF 7FD8 FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF # 0002 0002

22019968 0 8400 0000 1 A7C1 FFFF A000 FFFF 7FD8 FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF # 0003 0003

22539376 0 0000 0000 0 A0C5 FFFF A000 FFFF 0001 9D92 259E 0100 C431 FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF # 0004 0004

22549632 0 0400 0000 0 A454 FFFF A000 FFFF 0001 0000 0000 0000 FFFF FFFF 0000 800F A000 A001 A002 A003 A004 A005 A006 A007 A008 A009 A00A 604E FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF # 0005 0005

22559680 0 0000 0000 0 A094 FFFF A000 FFFF 0001 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 9D92 259E 0000 0000 C331 FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF # 0006 0006

22669632 0 0000 0000 0 A072 FFFF A000 FFFF 0001 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 9D92 259E 0000 0000 C331 FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF # 0007 0007

22670400 0 0000 0000 0 A0B8 FFFF A000 FFFF C001 0000 0000 0000 0000 0000 0000 0000 0000 9D92 259E 0000 0000 0000 0000 0000 0000 0000 0000 0000 9D92 259E 0000 4663 FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF # 0008 0008

22671232 0 0000 0000 0 A138 FFFF A000 FFFF C001 0000 0000 0000 0000 0000 0000 0000 0000 9D92 259E 0000 0000 0000 0000 0000 0000 0000 0000 0000 9D92 259E 0000 4663 FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF # 0009 0009

22769648 0 0000 0000 0 A072 FFFF A000 FFFF 0002 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 9D92 259E 0000 0000 C332 FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF # 000A 000A

22770416 0 0000 0000 0 A0B8 FFFF A000 FFFF C002 0000 0000 0000 0000 0000 0000 0000 0000 9D92 259E 0000 0000 0000 0000 0000 0000 0000 0000 0000 9D92 259E 0000 4664 FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF # 000B 000B

22771248 0 0000 0000 0 A138 FFFF A000 FFFF C002 0000 0000 0000 0000 0000 0000 0000 0000 9D92 259E 0000 0000 0000 0000 0000 0000 0000 0000 0000 9D92 259E 0000 4664 FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF # 000C 000C

22869616 0 0000 0000 0 A072 FFFF A000 FFFF 0003 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 9D92 259E 0000 0000 C333 FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF # 000D 000D

22870368 0 0000 0000 0 A0B8 FFFF A000 FFFF C003 0000 0000 0000 0000 0000 0000 0000 0000 9D92 259E 0000 0000 0000 0000 0000 0000 0000 0000 0000 9D92 259E 0000 4665 FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF # 000E 000E

22871216 0 0000 0000 0 A138 FFFF A000 FFFF C003 0000 0000 0000 0000 0000 0000 0000 0000 9D92 259E 0000 0000 0000 0000 0000 0000 0000 0000 0000 9D92 259E 0000 4665 FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF # 000F 000F

22969632 0 0000 0000 0 A072 FFFF A000 FFFF 0004 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 9D92 259E 0000 0000 C334 FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF # 0010 0010

22970400 0 0000 0000 0 A0B8 FFFF A000 FFFF C004 0000 0000 0000 0000 0000 0000 0000 0000 9D92 259E 0000 0000 0000 0000 0000 0000 0000 0000 0000 9D92 259E 0000 4666 FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF # 0011 0011

22971216 0 0000 0000 0 A138 FFFF A000 FFFF C004 0000 0000 0000 0000 0000 0000 0000 0000 9D92 259E 0000 0000 0000 0000 0000 0000 0000 0000 0000 9D92 259E 0000 4666 FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF # 0012 0012

23069616 0 0000 0000 0 A072 FFFF A000 FFFF 0005 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 9D92 259E 0000 0000 C335 FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF # 0013 0013

23070400 0 0000 0000 0 A0B8 FFFF A000 FFFF C005 0000 0000 0000 0000 0000 0000 0000 0000 9D92 259E 0000 0000 0000 0000 0000 0000 0000 0000 0000 9D92 259E 0000 4667 FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF # 0014 0014

23071184 0 0000 0000 0 A138 FFFF A000 FFFF C005 0000 0000 0000 0000 0000 0000 0000 0000 9D92 259E 0000 0000 0000 0000 0000 0000 0000 0000 0000 9D92 259E 0000 4667 FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF # 0015 0015

23169616 0 0000 0000 0 A072 FFFF A000 FFFF 0006 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 9D92 259E 0000 0000 C336 FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF # 0016 0016