Санкт-Петербургский государственный университет

Математическое обеспечение и администрирование информационных систем
Системное программирование

Бережных Алексей Владимирович

# Изолированный запуск поставщиков типов для компилятора F#

Бакалаврская работа

Научный руководитель:
доц. кафедры СП, к. т. н. Литвинов Ю. В.

Рецензент:
доц. кафедры информатики, к.ф.-м. н. Григорьев Д. А.

Санкт-Петербург
2020

SAINT-PETERSBURG STATE UNIVERSITY

Software and Administration of Information Systems
Software Engineering

Berezhnykh Aleksey Vladimirovich

# Hosting F# type providers out-of-process

Bachelor's Thesis

Scientific supervisor:
Assistant Professor, C.Sc. Litvinov Y. V.

Reviewer:
Assistant Professor, Ph.D. Grigoryev D. A.

Saint-Petersburg
2020

# Оглавление

**Введение** 4

**1. Обзор** 8

1.1. Сервисы компилятора . . . 8

1.2. Поставщики типов . . . 9

1.3. Предоставляемые типы . . . 10

1.4. Среда разработки JetBrains Rider . . . 13

1.5. Библиотека RD . . . 13

**2. Описание реализации** 15

2.1. API изолированных поставщиков типов . . . 15

2.1.1. Основная идея . . . 15

2.1.2. Реализация в Rider . . . 17

2.2. Жизненный цикл изолированного процесса . . . 23

2.3. Управление кэшированием . . . 24

2.3.1. Кэширование на стороне Rider . . . 24

2.3.2. Кэширование на стороне изолированного процесса 26

2.3.3. Инвалидация и очистка кэшей . . . 27

2.4. Поддержка старых версий базовой библиотеки поставщиков типов . . . 28

2.5. Тестирование и апробация . . . 29

2.6. Замечания . . . 30

**Заключение** 31

**Список литературы** 33

# Введение

Современное программирование процветает на пространствах с большими данными и сложными задачами по их обработке. Можно наблюдать, что из-за распространенности многоядерных процессоров все чаще требуется писать параллельный код. Точно так же и в облачных системах важно уметь распределить выполнение задачи по параллельным потокам вычислений. К сожалению, с помощью традиционных императивных и объектно-ориентированных языков очень сложно писать распараллеливаемый код без ошибок из-за обилия разделяемых состояний и функций с побочными эффектами. Сейчас для решения этой проблемы всё чаще и чаще применяется функциональная парадигма программирования, которая заведомо требует строить программные системы таким образом, чтобы в них не было неявной зависимости по данным.

В 2005 году компания Microsoft представила F# [5] – строго типизированный, кроссплатформенный язык программирования, позволяющий решать сложные задачи с помощью функциональной парадигмы. С точки зрения бизнеса основная роль F# заключается в сокращении времени на разработку и развертывание аналитических программных компонентов в современных программных системах. Его совместимость со всеми языками и библиотеками .NET и способность справляться со сложностью обработки и анализа больших данных предлагают привлекательные возможности для бизнеса. Сейчас F# – это достаточно зрелый язык, который очень эффективно используется для решения задач обработки данных.

Поставщики типов являются расширениями для компилятора F# и решают серьезную проблему: данные в большинстве источников слабо структурированы, в то время как большинство промышленных языков программирования – со строгой статической типизацией. Традиционно для доступа к данным, будь то СУБД или веб-сервис, приходилось переписывать вручную или с помощью инструментов генерировать программный код для слоя данных. Однако практика кодогенерации в

большинстве случаев считается неприемлемой из-за сложности в понимании и поддержке сгенерированного кода. Поставщики типов позволили упростить получение данных, предоставляя необходимые типы данных «на лету», подхватывая структуру данных из внешнего описания или схемы и упрощая доступ ко всевозможным источникам данных.

Языковую поддержку F# в популярных интегрированных средах разработки обеспечивает программная библиотека сервисов компилятора F# [3] с открытым исходным кодом[1], предоставляемая самими разработчиками языка. В частности, именно данная библиотека обеспечивает возможность использования поставщиков типов разработчиками в своих проектах. Однако текущая реализация механизма поставщиков типов в сервисах компилятора F# имеет особенность: поставщики типов являются непосредственной частью сервисов компилятора и загружаются в тот же процесс, что и они, а сервисы компилятора загружаются в корневой процесс интегрированных сред разработки. И это приводит к следующим последствиям:

- поставщики типов выполняются в той же среде исполнения, что и сервисы компилятора. Это может привести к конфликтам окружения, если поставщик типов разрабатывался для среды, отличной от среды, в которой запускаются сервисы компилятора;
- в случае критической ошибки или длительного ожидания при выполнении кода поставщика типов возможна угроза работе всей среды разработки;
- сервисы компилятора при загрузке библиотеки динамической компоновки с поставщиком типов блокируют её на запись, что означает невозможность перекомпиляции кода поставщика типов при его разработке без закрытия всех проектов, где используется данный поставщик типов.

Одной из популярных IDE, полностью поддерживающих F#, явля-

[1]Сервисы компилятора – библиотека языковой поддержки F#. URL: `https://github.com/dotnet/fsharp` (дата обращения: 25.05.2020).

ется JetBrains Rider[2]. Главной мотивацией данной бакалаврской работы является переход кодовой базы JetBrains Rider на среду выполнения .NET Core: поскольку поставщики типов являются пользовательскими расширениями компилятора и не все из них могут исполняться на .NET Core, то после перехода Rider на .NET Core полностью или частично прекратится поддержка проектов, использующих несовместимые поставщики типов. Поэтому необходимо решение, позволяющее запускать поставщики типов в том же программном окружении, что используется при сборке пользовательского проекта, и дающее возможность управлять жизненным циклом поставщиков типов для решения вышеуказанных проблем.

Данная бакалаврская работа выполнялась под руководством специалистов компании JetBrains, поскольку актуальной является задача модернизации механизма взаимодействия поставщиков типов и сервисов компилятора F# с целью изоляции их работы от работы JetBrains Rider.

## Постановка задачи

Целью данной бакалаврской работы является реализация механизма изолированного от JetBrains Rider управления поставщиками типов. Для этого необходимо:

- изолировать порождение поставщиков типов в отдельный от работы компилятора процесс;
- разработать и интегрировать в Rider протокол взаимодействия между поставщиками типов и сервисами компилятора:
  - описать формат передаваемых по протоколу данных;
  - описать и реализовать модель взаимодействия между зонами внутри процесса Rider и изолированного процесса для передачи информации от поставщиков типов между ними;

[2] JetBrains Rider – интегрированная среда разработки для .NET. URL: `https://www.jetbrains.com/ru-ru/rider` (дата обращения: 18.02.2020).

    - обеспечить кэширование передаваемых данных;
    - обеспечить возможность запуска поставщиков типов в пользовательской среде выполнения для .NET.

- создать тестовое покрытие для функциональности изолированных поставщиков типов и произвести апробацию полученного решения.

# 1. Обзор

## 1.1. Сервисы компилятора

Пакет сервисов компилятора F# [3] – программная библиотека с открытым исходным кодом, основанная на исходном коде компилятора для F#, предоставляющая дополнительные функциональные возможности для использования редакторами кода и средами интегрированной разработки.

Библиотека активно разрабатывается и поддерживается F# Software Foundation и предоставляет набор программных интерфейсов для получения готовой к использованию информации из кода проектов: данных о типах и методах, деревьев разбора кода, контекстных подсказок и тд., – всё, что необходимо для навигации по исходному коду, отображения всплывающих подсказок, анализа кода или применения рефакторинга, а также реализации других функций поверх сервисов компилятора.

Сервисы компилятора используются в большинстве существующих реализаций поддержки языка F# в таких инструментах для разработчиков, как Visual F# (плагин для Visual Studio), FsAutoComplete (используется плагинами Ionide для редакторов кода Visual Studio Code и Atom) и JetBrains Rider. Поскольку библиотека активно поддерживается сообществом F#, то никакой существенной дополнительной работы не требуется в будущем для поддержки более новых версий языка, потому что она делится своей кодовой базой с актуальным компилятором F# и, как следствие, последними обновлениями языка.

Поддержка поставщиков типов обеспечивается набором закрытых для использования извне библиотеки программных модулей, которые содержат реализацию предоставляемых типов, порождаемых поставщиками типов и позволяющих получать готовую к использованию информацию из кода проектов с поставщиками, а также описывают логику управления жизненным циклом поставщиков типов.

## 1.2. Поставщики типов

Поставщик типов [2] – реализация метапрограммирования в языке F#, компонент для сервисов компилятора, предоставляющий типы, свойства и методы внешнего источника данных для использования в своём коде.

Например, поставщик типов для SQL[3] может формировать типы, представляющие таблицы и столбцы в реляционной базе данных, позволяя работать с ними напрямую из кода на F# (Рис. 1).

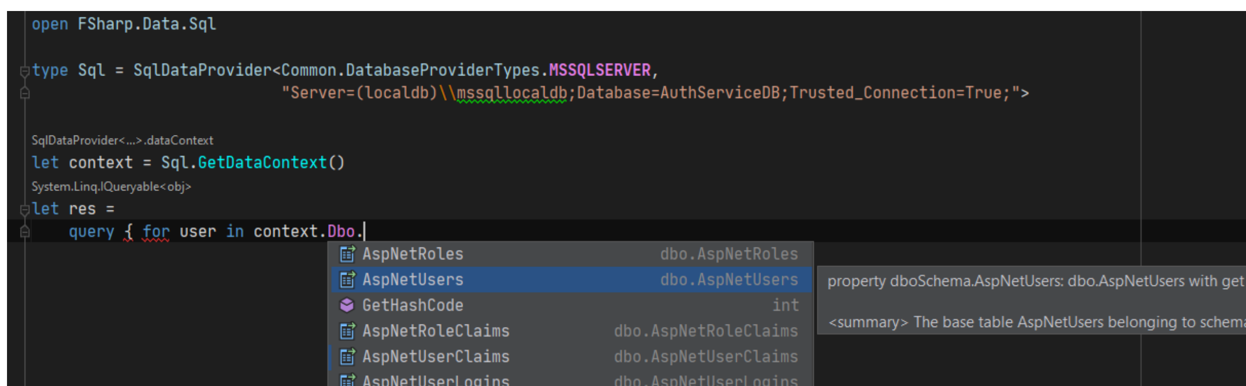


Рис. 1: Поставщик типов SQL базы данных

Разработчику необходимо лишь создать в коде своего проекта экземпляр поставщика типов, передать ему в качестве параметра строку подключения к базе данных, после чего сервисы компилятора запросят у поставщика предоставляемые им данные, и во время написания кода будет доступен строго типизированный набор всех сущностей, находящихся в указанной базе данных. Среда разработки работает с полученными из поставщика типов классами и методами так же, как если бы работала с обычными типами – предлагая контекстные подсказки и статический анализ кода. Обращение к предоставляемым данным через поставщиков типов выполняется лишь во время написания кода и во время выполнения программы не требуется, поскольку вся информация о поставляемых типах добавляется в библиотеку динамической компоновки на этапе компиляции.

[3]Поставщиков типов для SQL. URL: https://fsprojects.github.io/SQLProvider (дата обращения: 16.05.2020).

С точки зрения компилятора каждый поставщик типов представляет из себя реализацию программного интерфейса ITypeProvider, указанного в спецификации языка F#, контракт которого требует описания возвращаемых предоставляемых типов, реализации логики получения этих данных и их инвалидации (например поставщик типов базы данных может отправить событие инвалидации, когда поменялась схема базы данных, и после получения события инвалидации от поставщика типов сервисы компилятора выполнят заново вывод типов в проекте).

При необходимости разработчик может создать собственные поставщики типов, используя официальную документацию[4] и программную библиотеку поставщиков типов, содержащую базовую реализацию интерфейса ITypeProvider[5].

## 1.3. Предоставляемые типы

Для получения программистом метаданных (таких как список методов и полей класса) обо всех обьектах, используемых в его приложении на .NET, используются типы среды выполнения[6]. Для любого типа из загруженной в память приложения библиотеки динамической компоновки можно создать экземляр его типа среды выполнения и получить обьектно-ориентированное представление всех его метаданных в среде исполнения .NET.

Предоставляемые поставщиками типы с точки зрения компилятора являются обёртками над экземплярами настоящих типов среды выполнения .NET и позволяют получить информацию, необходимую для отображения программисту при написании им кода, такую как:

- предоставляемые пространства имён – реализации интерфейса виртуального пространства имён, позволяющие получить все

[4]Создание поставщика типов. URL: `https://docs.microsoft.com/ru-ru/dotnet/fsharp/tutorials/type-providers/creating-a-type-provider` (дата обращения: 24.05.2020).

[5]Библиотека поставщиков типов. URL: `https://github.com/fsprojects/FSharp.TypeProviders.SDK#the-f-type-provider-sdk` (дата обращения: 12.03.2020).

[6]Класс Type. URL: `https://docs.microsoft.com/ru-ru/dotnet/api/system.type?view=netcore-3.1` (дата обращения: 14.02.2020).

классы в пространстве и ответить, есть ли в нём запрашиваемый компилятором класс. Позволяют программисту использовать классы, предоставленные поставщиками типов;

- предоставляемые классы – обёртки над классами, содержащие методы, поля, свойства, подклассы, библиотеку динамической компоновки, в которой определён тип, и другие дополнительные атрибуты классов. Могут быть генеративными (добавляются при компиляции в библиотеку динамической компоновки проекта, в котором созданы, и могут использоваться в других проектах) и затираемыми (остаются после компиляции только в том проекте, в котором были созданы);
- предоставляемые библиотеки динамической компоновки – обёртки над виртуальными библиотеками, содержащие в памяти или на диске предоставляемые классы;
- предоставляемые конструкторы – обёртки над конструкторами классов, содерщащие список параметров, статические аргументы, декларирующий тип и другие дополнительные атрибуты конструкторов. Также предоставляемый конструктор может быть конструктором поставщика типов, в таком случае при передаче в него константных параметров поставщик типов порождает предоставляемый класс;
- предоставляемые методы – обёртки над описаниями функций, содерщащие список параметров, статические аргументы, декларирующий тип, тип возвращаемого значения и другие дополнительные атрибуты функций. Предоставляемый метод при передаче ему параметров во время редактировани кода в среде разработки может в качестве результата породить предоставляемый класс;
- предоставляемые параметры – обёртки над параметрами функций, содерщащие тип возвращаемого значения, значение по умолчанию и другие дополнительные атрибуты параметров;

- предоставляемые свойства – обёртки над свойствами класса, содержащие методы для чтения и записи свойства, тип возвращаемого значения, значение по умолчанию и другие дополнительные атрибуты свойств;

- предоставляемые поля – обёртки над полями класса, содержащие методы для чтения и записи поля, тип возвращаемого значения, значение по умолчанию и другие дополнительные атрибуты полей;

- предоставляемые события – обёртки над событиями в .NET[7], содержащие методы подписки и отписки и тип обработчика;

- предоставляемые выражения – обёртки над цитированиями кода F#[8]. Когда программист во время написания кода с использованием поставщиков типов вызывает предоставляемый метод, передавая в него в качестве параметров выражения различной сложности (константы, создание обьектов, анонимные функции, выражения, использующие данные других поставщиков типов и тд.) сервисы компилятора запрашивают у соответствующего поставщика типов информацию о полученном после применения параметров к вызванному методу выражении. Используется сервисами компилятора для вывода необходимой перегрузки предоставляемого метода;

- предоставляемые переменные – обёртки над переменными в цитированиях кода.

Текущая реализация предоставляемых типов в сервисах компилятора имеет значительный недостаток: код компилятора и предоставляемых типов сильно связан и имеет непосредственный доступ к их деталям реализации, опираясь на обёрнутый тип среды выполнения,

[7]Механизм событий в .NET. URL: https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/keywords/event (дата обращения: 23.12.2019).

[8]Цитирование кода в F#. URL: https://docs.microsoft.com/ru-ru/dotnet/fsharp/language-reference/code-quotations (дата обращения: 17.04.2020).

который вернул поставщик типов. Это препятствует возможной виртуализации и переопределению поведения предоставляемых типов, а также мешает потенциальной передаче данных из предоставляемых типов между процессами, поскольку типы среды выполнения создаются в процессе выполнения программы, их внутренняя реализация сложна и не допускает переопределений поведения, а самая большая проблема – нет возможности создать экзепляры типов среды выполнения без загрузки содержащей их библиотеки динамической компоновки в память приложения.

## 1.4. Среда разработки JetBrains Rider

JetBrains Rider – кроссплатформенная интегрированная среда разработки для языковой платформы .NET, основанная на JetBrains IntelliJ Platform и JetBrains ReSharper. В Rider включена полная поддержка языка F# [1], реализованная в виде программного расширения[9].

Хотя поддержка большинства языков в ReSharper и IntelliJ была реализована с нуля, поддержка F# была реализована с помощью собственной версии библиотеки сервисов компилятора – это позволило повторно использовать существующие инструменты, которые уже есть в других интегрированных средах разработки, и, в то же время, дополнить их собственной функциональностью. При открытии проекта на F# сервисы компилятора загружаются в хост-процесс среды вместе со всеми поставщиками типов, используемыми в проекте.

## 1.5. Библиотека RD

RD [4] – библиотека[10] для коммуникации между процессами, разработанная компанией JetBrains:

- позволяет создать процесс с выбранной средой исполнения для .NET;

[9] Языковая поддержка F# в JetBrains Rider. URL: `https://github.com/JetBrains/fsharp-support` (дата обращения: 25.05.2020).

[10] RD – реактивная распределенная коммуникационная платформа для .NET. URL: `https://github.com/JetBrains/rd`

- позволяет описать формат передаваемых данных и удалённо вызываемых процедур на языке Kotlin и сгенерировать клиент-серверный код на C#/Kotlin/C++;
- в случае возникновения исключения передаёт отладочную информацию между процессами;
- позволяет вести непрерывную запись метрик производительности и системной информации о передаваемых данных;
- автоматически выбирает свободный порт для установки соединения между процессами;
- механизм сердцебиений[11], позволяющий установить обрыв соединения между родительским и дочерним процессом и аварийно завершить дочерний процесс.

[11] Протокол сердцебиений. URL: https://en.wikipedia.org/wiki/Heartbeat_(computing) (дата обращения: 10.02.2020).

# 2. Описание реализации

## 2.1. API изолированных поставщиков типов

### 2.1.1. Основная идея

Чтобы изолировать работу поставщиков типов от работы сервисов компилятора, необходимо переопределить методы получения и создания экземпляров поставщиков типов, однако сервисы компилятора не предоставляют таких возможностей из-за инкапсуляции программных модулей. Поэтому была создана точка расширения в компиляторе (API изолированных поставщиков типов), позволяющая переопределить её из внешних библиотек и программ.

Интерфейс API изолированных поставщиков типов содержит методы, позволяющие переопределить:

- порождение поставщиков типов – сервисы компилятора передают данные, необходимые для создания поставщиков, а реализация должна создать и вернуть в компилятор их экземпляры. Экземпляры поставщиков типов представляют из себя реализацию интерфейса, описанного в спецификации языка F#;
- порождение предоставляемых классов – компилятор запрашивает у предоставляемого поставщиком типов пространства имён список экземпляров всех классов, в нём содержащихся;
- вывод предоставляемых классов – компилятор запрашивает у предоставляемого пространства имён класс по имени, пространство имён возвращает экземпляр типа или пустой результат, если такого типа нет в данном пространстве;
- идентификация поставщиков типов – когда компилятору необходимо вывести пользователю сообщение об ошибке, произошедшей в поставщике типов, ему необходимо указать название поставщика;

- порождение предоставляемых цитирований кода – когда сервисы компилятора делают подстановку параметров в функции, они просят поставщик типов предоставить выражение, необходимое для вывода необходимой перегрузки метода. Для этого сервисы компилятора передают API изолированных поставщиков типов в качестве аргументов предоставляемый метод и переданные в него для подстановки предоставляемые переменные.

Для того, чтобы точку расширения можно было реализовать извне кода сервисов компилятора, инкапсулированные раннее модули были открыты, а сама точка расширения составлена таким образом, чтобы оперировать только предоставляемыми типами, которые можно дальнейшими изменениями в сервисах компилятора виртуализировать и иметь возможность передавать по протоколу. Поэтому все места в сервисах компилятора, где от поставщиков типов и предоставляемых пространств имён требуются типы времени выполнения, которые позже оборачиваются сервисами компилятора в собственную реализацию предоставляемых типов, были заменены на вызовы методов API изолированных поставщиков типов.

Данный интерфейс был реализован по умолчанию для сервисов компилятора, и на данный момент публикация[12] предложения в репозитории сервисов компилятора находится на стадии обсуждения. Разработчикам языка необходимо оценить риски из-за расширения зоны ответственности, к которой добавилась предложенная точка расширения. Однако, в случае непринятия в официальный репозиторий сервисов компилятора, API изолированных поставщиков типов и сопутствующие изменения в компиляторе, реализованные в данной работе, будут добавлены в собственную версию сервисов компилятора F# от команды языковой поддержки F# в Rider.

[12] API изолированных поставщиков типов. URL: `https://github.com/dotnet/fsharp/pull/8235` (дата обращения: 24.05.2020).

### 2.1.2. Реализация в Rider

Главная идея реализации состоит в том, чтобы хранить настоящие поставщики типов и предоставляемые данные в изолированном процессе, а обращаться к ним посредством прокси-обьектов на стороне Rider и сервисов компилятора, выбрав в качестве транспортного канала между процессами библиотеку RD.

Для реализации API изолированных поставщиков типов были созданы собственные прокси-реализации поставщика типов и предоставляемых типов[13], для них на языке Kotlin описаны протокольные модели, по которым сгенерированы клиенты, используемые в прокси-предоставляемых типах (Рис. 2):

- протокольная модель предоставляемого типа (поставщика типов), в которую конвертируется оригинальный предоставляемый тип для передачи от изолированного процесса в процесс Rider. Содержит:
  - идентификатор оригинального предоставляемого типа (поставщика типов), хранимого в изолированном процессе;
  - легковесный набор данных, такой как поля и флаги (чтобы оптимизировать передачу и хранения большого количество булевых полей из различных предоставляемых типов, для каждой модели, у которой суммарный обьём таких полей занимает более 4 байт, т.е. размер 4 булевых полей, все поля передаются в виде флагового перечисления[14], занимающего 1 байт).
- протокольная модель хоста предоставляемого типа (поставщика типов) – набор удалённых процедур для каждой категории предоставлямых типов, возвращающая результат для прокси-предоставляемых типов по хранимому идентификатору

[13] Реализация механизма изолированных поставщиков типов в JetBrains Rider. URL: https://github.com/JetBrains/fsharp-support/pull/91 (дата обращения: 24.05.2020).

[14] Класс Enum. URL: https://docs.microsoft.com/ru-ru/dotnet/api/system.flagsattribute?view=netcore-3.1 (дата обращения: 18.02.2020).

оригинального предоставляемого типа (поставщика типов) в изолированном процессе и дополнительным параметрам вызова.

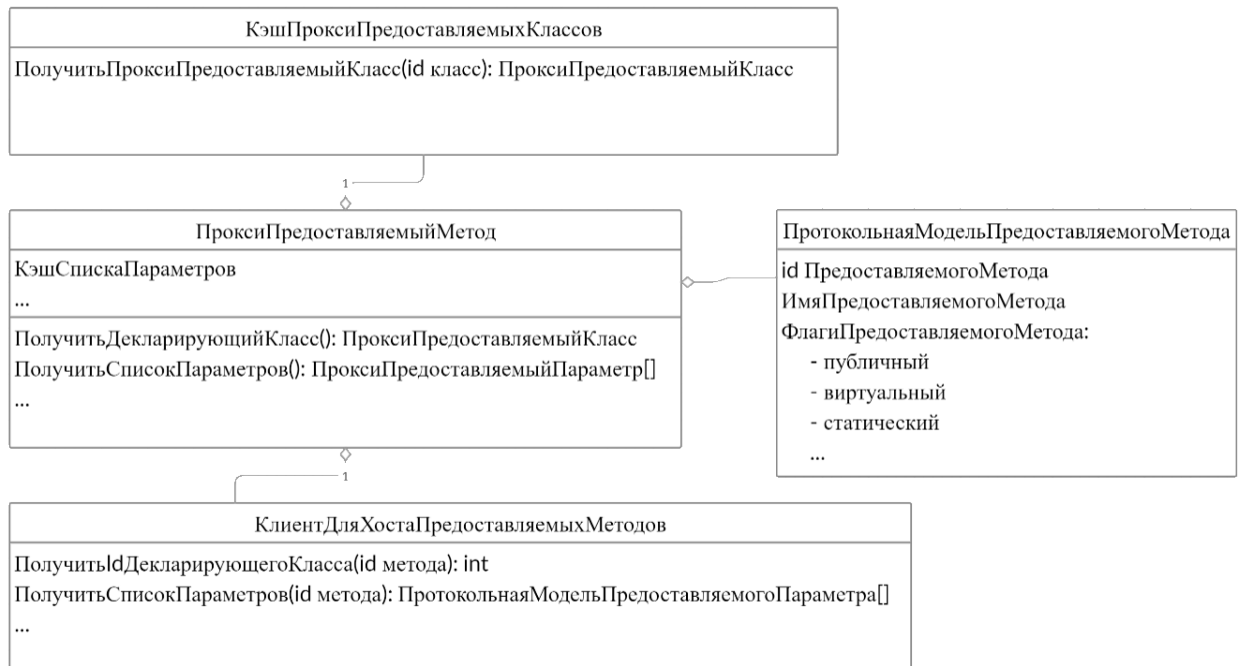


Рис. 2: Пример реализации прокси-предоставляемого метода

Протокольные модели предназначены для энергичной передачи данных о предоставляемых типах, в то время как модель хоста необходима для ленивого вычисления обьёмных массивов данных из предоставляемых типов по требованию (например, получить список параметров у метода, поскольку они сами по себе являются предоставляемыми типами).

Поскольку изначально не все данные можно было передать по протоколу и не все детали реализации предоставляемых типов можно было переопределить в коде Rider, в сервисах компилятора были сделаны и предложены соответствующие исправления и модификации:

- общая инкапсуляция типов времени выполнения в предоставляемых классах[15] – позволила передавать по протоколу полные данные о предоставляемых классах (принято разработчиками сервисов компилятора);

[15]Инкапсуляция типов времени выполнения в предоставляемых классах. URL: https://github.com/dotnet/fsharp/pull/8656 (дата обращения: 23.05.2020).

- инкапсуляция типов времени выполнения в контексте генеративных классов[16] – позволила осуществить поддержку генеративных классов (находится на рассмотрении разработчиками сервисов компилятора);

- инкапсуляция типов времени выполнения в предоставляемых выражениях[17] – позволила передавать между процессами все данные о предоставляемых выражениях (принято разработчиками сервисов компилятора);

- инкапсуляция метода конвертации предоставляемого класса в переменную[18] (принято разработчиками сервисов компилятора);

- виртуализация предоставляемых типов[19] – необходима для полного переопределения поведения предоставляемых типов и возможности создать свои реализации в виде прокси-предоставляемых типов. Будет предложена к публикации в сервисы компилятора в случае принятия авторами языка API изолированных поставщиков типов.

Для каждой отдельной категории предоставляемых типов (пространств имён, классов, методов и других) при создании изолированного процесса инициализируются отдельные протокольные хосты (Рис. 3). Протокольный хост живёт в изолированном процессе на протяжении всей его жизни и отвечает за координацию между прокси-предоставляемым типом на стороне процесса Rider (и сервисов компилятора, в частности) и его проксируемым предоставляемым типом, живущим на стороне изолированного процесса. Протокольный хост для предоставляемого типа является серверной частью в реализованной ар-

[16]Инкапсуляция типов времени выполнения в контексте генеративных классов. URL: `https://github.com/dotnet/fsharp/pull/9235` (дата обращения: 23.05.2020).

[17]Инкапсуляция типов времени выполнения в предоставляемых выражениях. URL: `https://github.com/dotnet/fsharp/pull/8809` (дата обращения: 23.05.2020).

[18]Инкапсуляция метода конвертации предоставляемого класса в переменную. URL: `https://github.com/dotnet/fsharp/pull/9005` (дата обращения: 23.05.2020).

[19]Виртуализация предоставляемых типов. URL: `https://github.com/DedSec256/fsharp/tree/ber.a/TypeProviders` (дата обращения: 23.05.2020).

хитектуре, обращения к которому выполняются между процессами посредством клиента, сгенерированного инструментами библиотеки RD.

Для каждой отдельной категории предоставляемых типов реализована фабрика предоставляемых моделей, которая при запросе от хоста с переданным экземпляром предоставляемого типа сохраняет её в своём кэше, присваивает уникальный идентификатор и конвертирует в протокольную модель предоставляемого типа для дальнейшей отправки по протоколу в процесс Rider. Кэши также создаются по одному на каждую категорию предоставляемых типов и являются общими для хостов и фабрик, при этом доступны хостам только для чтения, а фабрикам – для чтения и записи.

После получения протокольной модели на стороне Rider создаётся прокси-предоставляемый тип, которому передаётся соответствующая протокольная модель, соответствующий клиент протокольной модели хоста предоставляемого типа и кэш прокси-предоставляемых классов текущего поставщика типов.

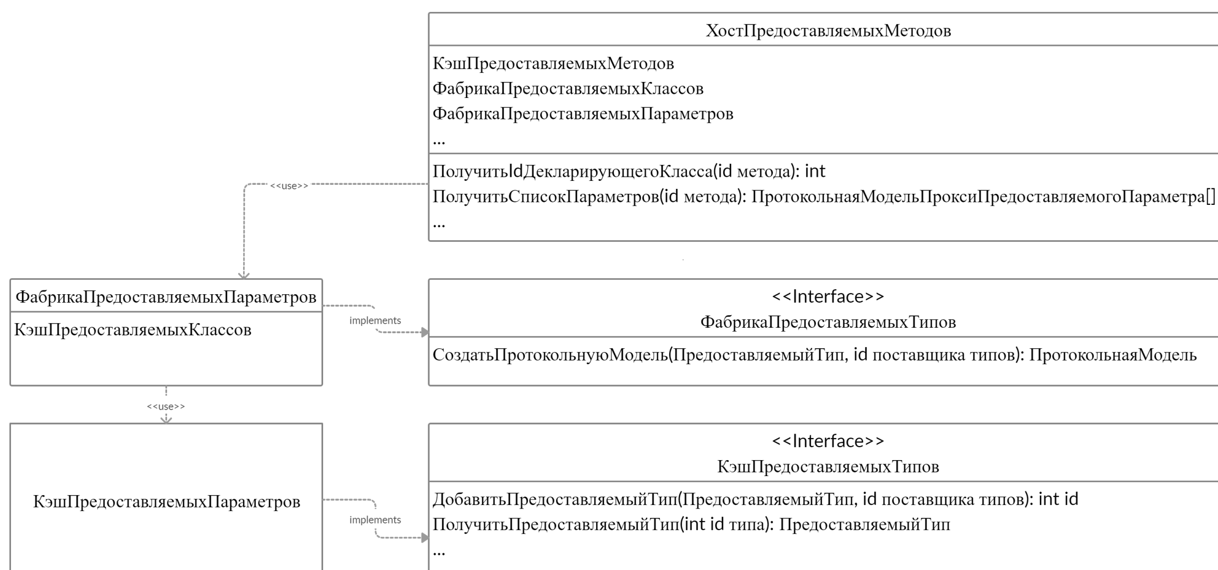


Рис. 3: Пример хоста для предоставляемых методов в изолированном процессе

Рассмотрим непосредственно реализацию описанной выше точки расширения в Rider:

- порождение поставщиков типов (Рис. 4) – сервисы компилятора

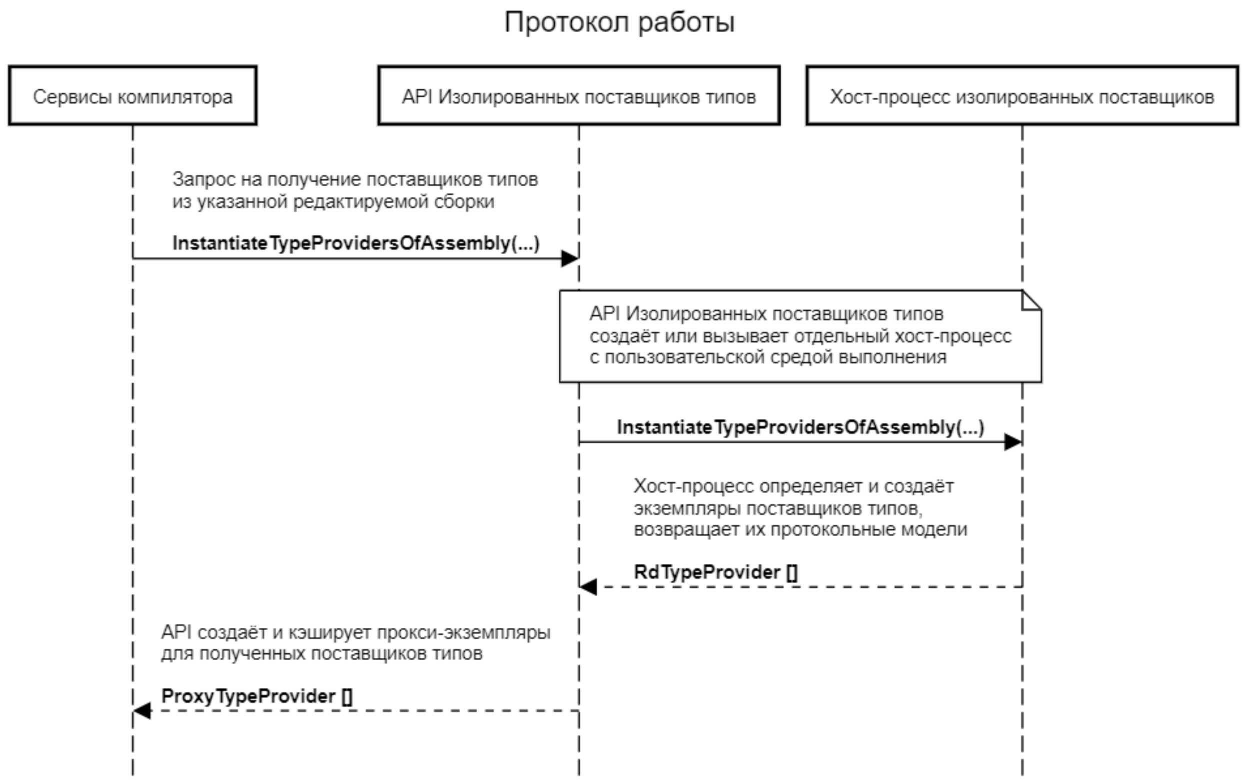


Рис. 4: Создание поставщиков типов

передают данные, необходимые для создания поставщиков, и обращаются к API изолированных поставщиков. API создаёт при первом таком обращении новый хост-процесс с пользовательской средой выполнения .NET, конвертирует переданные параметры и отправляет между процессами посредством RD, запрашивая у клиента хоста поставщиков типов данные о поставщиках типов. Хост поставщиков типов с помощью страндартной реализации точки расширения в сервисах компилятора создаёт поставщики типов, кэширует и конвертирует их с помощью фабрики поставщиков типов в модель, содержащую идентификатор поставщика типов, его полное и короткое имя, и возвращает в процесс Rider. После чего на стороне Rider создаются прокси-поставщики типы, которым передаются полученные протокольные модели поставщиков. Созданные прокси-поставщики инициализируют кэши предоставляемых классов и пространств имён и возвращаются в сервисы компилятора в качестве результата. Аналогично, при запросе пространств имён у прокси-поставщика, он через клиент своего хо-

ста запросит у настоящего поставищка типов данные о пространствах имён;

- порождение предоставляемых классов – прокси-предоставляемое пространство имён запрашивает у хоста предоставляемых пространств имён идентификаторы предоставляемых классов, после чего запрашивает данные о классах в кэше предоставлемых классов текущего поставщика типов и кэширует полученный список локально;

- вывод предоставляемых классов – прокси-предоставляемое пространство имён ищет класс по имени среди классов в своём локальном кэше;

- порождение предоставляемых цитирований кода – сервисы компилятора передают поставщику типов в качестве параметров для порождения прокси-предоставляемого выражения прокси-предоставляемый метод и прокси-предоставляемые переменные. Прокси-поставщик получает из изолированного процесса необходимое предоставляемое выражение, передав хосту поставщиков типов индентификаторы полученных прокси-предоставляемого метода и прокси-предоставляемых переменных;

- идентификация поставщиков типов – в реализации по умолчанию используется имя поставщика типов из его типа времени выполнения, которое в случае прокси-поставщиков будет всегда одинаковым, поэтому в переопределённой версии метода имя оригинального поставщика типа передаётся в прокси-поставщик типов через протокольную модель поставщика типов.

Когда прокси-предоставляемый тип или поставщик типов запрашивает данные у изолированного процесса (Рис. 5), например, прокси-предоставляемый метод с полученным идентификатором 1 запрашивает через клиент хоста предоставляемых методов список своих параметров, хост, получив запрос, достаёт из кэша предоставленных методов

оригинальный предоставленный метод и идентификатор его поставщика типов, получает у метода список параметров и конвертирует их с помощью фабрики предоставляемых параметров. Фабрика добавляет полученные параметры в кэш предоставляемых параметров в раздел их поставщика типов по его идентификатору и возвращает сконвертированные, пригодные для транспортировки легковесные модели хосту. Так работает для всех предоставляемых типов, кроме предоставляемых классов – в случае с ними из изолированного процесса возвращаются только их идентификаторы, по которым настощие данные о класах запрашиваются из кэша предоставляемых классов конкретного поставщика типов на стороне Rider (причины такой реализации описаны в главе про кэширование передаваемых данных).

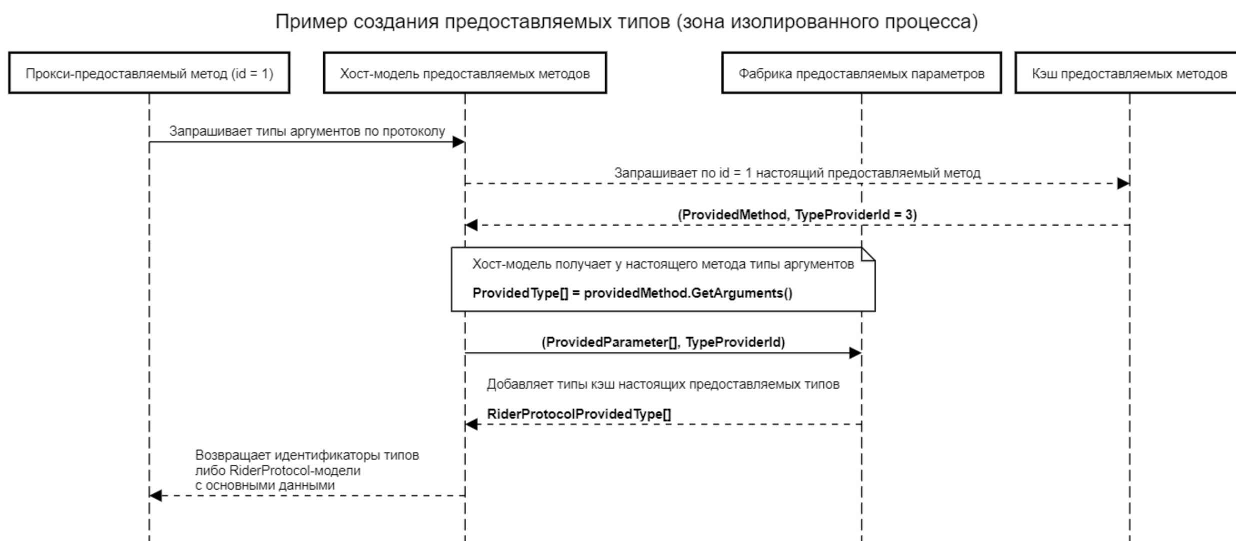


Рис. 5: Получение списка параметров прокси-предоставляемым методом

## 2.2. Жизненный цикл изолированного процесса

В реализации данной работы для всех проектов в открытом решении порождается единый изолированный процесс. Он создаётся при первом обращении компилятора к API изолированных поставщиков и продолжает существовать вплоть до закрытия пользователем решения в Rider. Управление жизненным циклом процесса реализовано с по-

мощью общепринятого в компании JetBrains паттерна управления ресурсами Lifetime[20], который позволяет вынести ответственность за контроль за временем жизни сущности из самой сущности. Реализация API изолированных поставщиков получает с помощью контейнера зависимостей извне экземпляр Lifetime, привязанный к времени жизни открытого решения, и создаёт из него дочерний объект Lifetime, который передаёт в RD при создании процесса. Дочерний Lifetime автоматически уничтожается при уничтожении родительского экзмепляра Lifetime, после чего RD автоматически отправляет изолированному процессу сигнал о закрытии, при получении которого процесс завершает работу.

Однако архитектура позволяет при необходимости для определённых групп поставщиков типов создавать дополнительные процессы, поскольку каждый отдельный экземпляр прокси-поставщика типов в своей реализации принимает клиент для обращения к хостам предоставляемых типов из API изолированных поставщиков и не полагается на принадлежность клиента конкретному процессу, оставляя это деталью реализации API изолированных поставщиков.

## 2.3. Управление кэшированием

Кэширование данных выполняется как на стороне процесса Rider, так и на стороне изолированного процесса поставщиков типов, однако политики кэширования в каждом из процессов отличаются. Отличаются они так же для кэширования предоставляемых классов и остальных предоставляемых типов.

### 2.3.1. Кэширование на стороне Rider

Прокси-поставщики типов кэшируются после создания в API изолированных поставщиков типов в хэш-таблице, где ключом является имя динамически подключаемой библиотеки, из которой создавались

[20] Lifetime – паттерн управления ресурсами. URL: https://www.jetbrains.com/help/resharper/sdk/Platform/Lifetime.html (дата обращения: 23.02.2020).

экзмепляры поставщиков типов. Предоставляемые пространства имён локально кэшируются самим прокси-поставщиком после запроса об их получении.

Поскольку предоставляемые классы являются наиболее тяжеловесным по количеству данных и потенциально могут часто встречаться (например, один и тот же предоставляемый класс может встречаться в качестве возвращаемого типа у методов или как тип параметров), то каждый прокси-поставщик типов содержит собственный кэш предоставляемых классов, который передаётся всем прокси-предоставляемым типам поставщика, создаваемым позже. Данный кэш принимает идентификатор предоставляемого класса, полученный после запроса какого-либо прокси-предоставляемого типа у изолированного процесса, и с помощью вызова специального метода запрашивает через RD протокольную модель предоставляемого класса, из которой создаёт, кэширует и возвращает прокси-предоставляемый класс. Такой подход позволяет передавать по протоколу в большинстве случаев только целочисленный индентификатор типа, а при необходимости получить информацию о предоставляемом типе всего один раз. Поскольку все хосты возвращают вместо протокольной модели класса его индентификатор, то гарантируется получение данных о классе только через кэш прокси-предоставляемых классов. Так, например, прокси-предоставляемому методу (Рис. 6) достаточно закэшировать полученные идентификаторы прокси-предоставляемых классов, а позже запросить их данные по идентификаторам из кэша предоставляемых-классов текущего поставщика типов.

Известно, что все предоставляемые типы достаточно уникальны для каждого предоставляемого класса: у разных классов разные методы, у разных методов разные наборы параметров, и т.д., – это значит, что встретить одинаковые по содержанию типы в различных классах маловероятно, поэтому было принято решение не следить за уникальностью таких данных. После того, как экземпляр прокси-предоставляемого типа получает по протоколу от изолированного процесса дополнительные данные, он кэширует их в своём локальном хэше. Локальный кэш пред-

Рис. 6: Кэширование прокси-предоставляемых классов внутри процесса Rider

ставляет из себя лениво вычислимую функцию с сохранением результата. Для этого сначала использовалась стандартная реализация Lazy[21], однако позже выяснилось, что она не подходит, поскольку при возникновении исключения при вычислении она кэширует исключение и при последующих обращениях возвращает его. Это означает, что при любом неудавшемся запросе к хосту на стороне изолированного процесса (например, при истёкшем времени запроса), получить данные второй раз станет невозможным, что приведёт к ошибкам вывода типов в файле. Поэтому использовалась собственная реализация, которая не содержит логики кэширования исключения.

### 2.3.2. Кэширование на стороне изолированного процесса

Реализация кэширования на стороне изолированного процесса в данной работе исходит из предположения, что большинство данных кэшируются на стороне процесса Rider и уже будет уникальным, поэтому фабрики предоставляемых моделей кэшируют все модели, переданные им от хостов, не следя за их уникальностью. Это позволяет облегчить

[21]Класс Lazy. URL: `https://docs.microsoft.com/ru-ru/dotnet/api/system.lazy-1?view=netcore-3.1` (дата обращения: 22.04.2020).

реализацию кэшей и не требует дополнительных ресурсов на проверку наличия добавляемой модели в кэше.

Однако отдельная реализация кэша используется для предоставляемых классов. Ввиду уникальности полного имени предоставляемого класса, его можно использовать для определения эквивалентности двух классов, что и использует кэш предоставляемых классов, добавляя к себе лишь не встречавшийся ранее класс.

Кэши на стороне изолированного процесса реализованы как составная хэш-таблица, где первичным ключём является идентификатор предоставляемого типа. Также кэши позволяют получить список всех предоставляемых типов по идентификатору поставщика типов, к которому относится предоставляемый тип.

### 2.3.3. Инвалидация и очистка кэшей

Настоящий поставщик типов, загруженный на стороне изолированного процесса, может запросить инвалидацию – например, поставщик типов базы данных может отправить событие инвалидации, когда поменялась схема базы данных. После получения события инвалидации от поставщика типов, сервисы компилятора выполняют заново вывод типов в проекте, в таком случае требуется инвалидация кэшей как у настоящего поставщика-типов, так и у его прокси. Полная очистка кэшей в данной работе рассмотрена и реализована для случая, когда разработчик закрывает всё решение со своим проектом.

На стороне процесса Rider корневой ссылкой на все предоставляемые типы является сам поставщик типов, посколько он кэширует у себя список предоставляемых им пространств имён и содержит в себе ссылку на кэш предоставляемых классов. Поэтому очистка кэшей при закрытии всего решения выполняется очисткой хэш-таблицы с поставщиками типов на стороне API изолированных поставщиков (на стороне изолированного процесса очистка произойдёт автоматически из-за жизненного цикла изолированного процесса), а инвалидация кэшей конкретного экземпляра поставщика типов выполняется после получения соответствующего сигнала: на стороне Rider прокси-поставщик

сам очищает свои кэши, а на стороне изолированного процесса поставщик типов очищает в общих кэшах с предоставляемыми типами набор обьектов по своему идентификатору.

## 2.4. Поддержка старых версий базовой библиотеки поставщиков типов

Для создания пользовательских поставщиков типов используется базовая программная библиотека, с её помощью разработчику необходимо наследовать свой класс с реализацией поставщика типа от базовой реализации и описать предоставляемые типы и логику их создания. Данная библиотека используется в большинстве используемых поставщиков и имеет некоторые особенности.

Библиотека имеет зависимость от версии библиотеки сервисов компилятора, в то время как исполняемый код изолированного процесса зависит от последней версии библиотеки сервисов компилятора, используемой в Rider. Это приводило к конфликту версий зависимостей во время создания поставщиков типов, что было устранено написанием конфигурации для переадресации[22] версий библиотеки сервисов компилятора.

Также в ходе выполнения данной работы мы столкнулись с одной неочевидной особенностью некоторых поддерживаемых версий базовой библиотеки поставщиков (например, используемой в реализации поставщика Yaml-конфигураций[23]) – для создания экземпляров поставщиков типов необходимо передать в конструктор базовой реализации поставщика типов набор путей к файлам библиотек динамической компоновки различных зависимостей, однако, ввиду особенностей управления ссылками на обьекты внутри сервисов компилятора, этот набор зависимостей передаётся в конструктор не напрямую, а посредством передаваемого в конструктор лямбда-выражения, из иерархии обьектов

[22]Перенаправление версий сборки. URL: `https://docs.microsoft.com/ru-ru/dotnet/framework/configure-apps/redirect-assembly-versions` (дата обращения: 15.02.2020).

[23]Поставщик типов для Yaml-конфигураций. URL: `https://fsprojects.github.io/FSharp.Configuration/YamlConfigProvider.html` (дата обращения: 4.03.2020).

внутри замыкания которого и достаётся набор зависимостей с помощью отражения[24]. В .NET лямбда-выражения компилируются в обьекты [6], а механизм замыканий лямбда-выражений основывается на инициализации закрытого поля внутри компилируемого по лямбда-выражению обьекта. Передавать такой обьект или код функции по протоколу нецелесообразно, поэтому в данной работе вместо данной функции (которой является лямбда-выражение) в конструктор создаваемого на стороне изолированного процесса поставщика типов передаётся искусственно созданная функция, вызов которой вызывает переданную из сервисов компилятора лямбду на стороне процесса Rider. И если получение с помощью отражения замыканий лямбда-выражений работает при передаче функции в качестве аргумента конструктора внутри одного процесса, то это невозможно при передаче между процессами.

Поэтому для решения данной проблемы был написан отдельный программный модуль, который с помощью отражения получает из замыкания лямбда-выражения необходимые данные, представленные набором строк, после чего отправляет по протоколу, а на стороне изолированного процесса обратно запаковывает в нужной последовательности в нужные поля искусственно созданных обьектов, после чего замыкает в искуственно созданной функции на стороне изолированного процесса и передаёт эту функцию в конструктор поставщика типов.

## 2.5. Тестирование и апробация

Для тестирования функциональности полученного решения был написан собственный поставщик типов и тесты, проверяющие изолированный запуск, вывод предоставляемых типов и их атрибутов и отображение базовых ошибок при некорректном использовании поставщиков типов.

Также была произведена апробация результата на нескольких популярных поставщиках типов[25]:

[24] Отражение в .NET. URL: https://docs.microsoft.com/ru-ru/dotnet/csharp/programming-guide/concepts/reflection (дата обращения: 23.02.2020).

[25] Поставщики типов для обработки данных. URL: https://fsharp.github.io/FSharp.Data (дата

- поставщиках типов для SQL, YAML, XML, JSON;
- поставщике типов для регулярных выражений;
- поставщике типов для Swagger RESTful API.

После добавления реализованного в данной работе решения в следующий выпуск JetBrains Rider планируется провести A/B-тестирование на пользователях среды разработки.

## 2.6. Замечания

При просмотре файлов журналирования RD-протокола было обнаружено, что большое количество передаваемых по протоколу предоставляемых классов являются не уникальными классами поставщиков типов, а обёртками над стандартными типами из основных бибилиотек .NET, которые используются поставщиками типов при предоставлении данных. В дальнейшем может иметь смысл научиться не хранить и не передавать такие классы между процессами, а создавать их на лету из известных базовых библиотек .NET, что, в случае возможности реализации, позволило бы сократить количество потребляемой памяти и трафика, передаваемого между процессами.

обращения: 20.05.2020).

# Заключение

## Полученные результаты

В ходе выполнения данной работы были достигнуты следующие результаты:

- изолировано порождение поставщиков типов в отдельный от работы компилятора процесс:
  - предложен к публикации в компилятор и реализован в JetBrains Rider программный интерфейс, позволяющий изолировать работу поставщиков типов;
- разработан протокол взаимодействия между поставщиками типов и сервисами компилятора на основе библиотеки RD:
  - описана и реализована модель передачи предоставляемых типов между процессами;
  - реализованы и опубликованы в компиляторе исправления инкапсуляции типов времени выполнения для предоставляемых типов;
  - реализован механизм кэширования передаваемых данных;
- создано тестовое покрытие для функциональности изолированных поставщиков типов и произведена апробация полученного решения на нескольких популярных поставщиках типов.

## Благодарность

Я выражаю большую благодарность моему консультанту из компании JetBrains – Адучучинку Е. П., который помогал мне на протяжении всей работы над дипломом и обеспечил всеми необходимыми удобствами для комфортного исследования и разработки.

Также я хочу выразить глубокую признательность своим преподавателям по программированию: Григорьеву Д. А. и Литвинову Ю. В. за

привитие мне любви к разработке программного обеспечения в целом и платформе .NET в частности, что значительно повлияло на мой выбор темы дипломной работы и дало окончательную уверенность в выборе пути программного инженера.

# Список литературы

[1] Auduchinok Evgeniy. Implementation of F# language support in JetBrains Rider IDE. — 2017. — URL: http://se.math.spbu.ru/SE/diploma/2017/pi/Auduchinok.pdf (дата обращения: 14.12.2019).

[2] Foundation F# Software. The F# 4.1 Language Specification. — 2019. — URL: https://fsharp.org/specs/language-spec/4.1/FSharpSpec-4.1-latest.pdf (дата обращения: 17.02.2020).

[3] Foundation F# Software. F# Compiler Services. — 2019. — URL: https://fsharp.github.io/FSharp.Compiler.Service/ (дата обращения: 07.05.2020).

[4] JetBrains. ReSharper DevGuide. — 2018. — URL: https://www.jetbrains.com/help/resharper/sdk/Products/Rider.html#protocol-extension (online; accessed: 21.04.2020).

[5] Syme Don. The Early History of F#. — 2018. — URL: https://fsharp.org/history/hopl-draft-1.pdf (дата обращения: 14.12.2019).

[6] Рихтер Джеффри. CLR via C#. Программирование на платформе Microsoft .NET Framework 4.5 на языке C, глава 17. — СПб.: Питер, 2013.