Санкт-Петербургский
Государственный Университет

Кафедра Системного Программирования

Программная инженерия

Мирошников Владислав Игоревич

Инструмент для сравнения .NET сборок в интегрированной среде разработки JetBrains Rider

Отчёт по производственной практике

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

Консультант:
технический руководитель Rider в ООО «Интеллиджей Лабс»,
к.т.н. И.Е. Мигалёв

Санкт-Петербург
2022

SAINT-PETERSBURG STATE UNIVERSITY

Department of System Programming

Software Engineering

Miroshnikov Vladislav Igorevich

A diff tool for comparing .NET assemblies in the JetBrains Rider IDE

Course Work

Scientific supervisor:
docent of the department of SP, C. Sc. Yurii Litvinov

Consultant:
Rider Technical Lead at «JetBrains», C. Sc. Ivan Migalev

Saint-Petersburg
2022

Содержание

Введение

Платформа .NET [1] — это программная платформа компании Microsoft для разработки различных приложений. Одной из отличительных особенностей данной платформы является возможность разработки приложений с использованием нескольких языков программирования, например, `C#`, `F#`, `Visual Basic .NET`, благодаря общеязыковой среде исполнения `Common Language Runtime`. В настоящее время существуют две различных платформы от компании Microsoft: `.NET Framework` и `.NET Core/.NET`. `.NET Framework` является более старой версией и поддерживает только операционную систему Windows, что и отличает его от `.NET Core/.NET`, который уже является кроссплатформенным, что соответствует современным стандартам индустрии.

Нельзя отрицать растущую популярность платформы .NET. Так, согласно исследованиям `Stack Overflow`, за 2021 год `.NET Core/.NET` и `.NET Framework` входят в тройку наиболее популярных фреймворков [2], а также являются «наиболее любимыми» фреймворками, не предназначенными для веб-разработки, среди опрошенных респондентов как в 2021 году, так и в 2020 и 2019 годах.

Одной из базовых, структурных и функциональных единиц платформы .NET является **сборка**, на уровне которой проходит контроль версий, развертывание и конфигурация приложения. В процессе разработки на .NET время от времени появляется необходимость в сравнении версий скомпилированных программ. Так, иногда это могут быть зависимости нашей программы, иногда — наши собственные бинарные файлы, для которых хочется понять, какой версии кода соответствует какой-либо из ранее опубликованных бинарных файлов.

Существуют различные способы, позволяющие сравнивать .NET сборки. Например, можно применить любой декомпилятор для получения исходного кода и инструмент для сравнения текста. Однако подобные решения не очень удобны, поскольку не являются единым инструментом. Существуют и полноценные приложения, обеспечивающие сравнение сборок. Однако они имеют свои существенные недостатки, например, отсутствие поддержки сравнения `.NET Core/.NET` сборок.

Одной из существующих сред разработки для платформы .NET является **Rider** [3] — кроссплатформенная среда разработки компании JetBrains, являющаяся одной из «наиболее любимых» IDE [4] согласно все тем же исследованиям `Stack Overflow` за 2021 год.

В данной работе предлагается расширить возможности для работы с платформой .NET в Rider, а именно — добавить возможность сравнения скомпилированных .NET сборок. При этом необходимо учесть недостатки существующих

решений и улучшить возможности сравнения. В данной работе рассматриваются различные способы для сравнения сборок, а также предлагается реализация одного из них в рамках среды разработки Rider.

1. Цели и задачи

Целью данной работы является добавление инструмента для сравнения скомпилированных .NET сборок в Rider. Для достижения цели были поставлены следующие задачи.

1. Провести обзор различных видов .NET сборок, возможность сравнения которых необходимо поддержать, а также обзор структуры и содержания .NET сборки.

2. Провести обзор существующих решений, позволяющих сравнивать .NET сборки.

3. Рассмотреть различные способы для сравнения сборок, а также выбрать наиболее подходящий для дальнейшей реализации.

4. Согласно выбранному способу интегрировать в Rider инструмент для сравнения .NET сборок, обеспечивающий сравнение исходного кода, представленного в виде декомпилированного кода на `C#`.

5. Провести апробацию решения с целью получения обратной связи от пользователей.

2. Обзор предметной области

Введем определение сборки .NET. Так, **.NET сборка** — это коллекция типов и ресурсов, собранных для совместной работы и образующих логически функциональную единицу. Сборки могут быть созданы из одного или нескольких файлов исходного кода. Для простоты понимания, их можно рассматривать как архивы, которые имеют определенную структуру и, в частности, содержат данные об исходном коде. Обычно скомпилированная сборка представлена на диске в виде файла с расширением `.exe` или `.dll`.

2.1. Различные виды .NET сборок

Согласно официальной документации платформы .NET от Microsoft [7], сборки делятся на два вида:

- Исполняемые сборки — сборки, представленные в виде исполняемого файла с расширением `.exe`. Далее для краткости изложения данный вид сборок именуется как «Exe-сборка».
- Сборки, представленные в виде файла библиотеки динамической компоновки — файлы с расширением `.dll`. Далее для краткости изложения данный вид сборок именуется как «Dll-сборка».

Кроме того, также существует разделение сборок на однофайловые и многофайловые сборки. Однофайловая сборка является простейшим типом сборки, все ее содержимое размещено внутри единственного файла `*.exe` или `*.dll`. Многофайловая же сборка состоит из набора модулей .NET, которые развертываются в виде одной логической единицы и снабжаются одним номером версии. При этом один из модулей называется «главным модулем», содержащим в том числе и точку входа. Такие сборки можно создавать с помощью компиляторов командной строки, однако из самой IDE создание такого вида сборок затруднительно или не представляется возможным. Например, в IDE-среде Visual Studio не предусмотрено отдельного шаблона проекта для создания многофайловой сборки на языке `C#`. Одно из главных преимуществ такого вида сборок состоит в том, что они позволяют объединять модули, написанные на различных .NET-совместимых языках программирования.

Разберем более детально Exe и Dll-сборки, а также их различные подвиды.

2.1.1 Exe-сборка

Стандарт `ECMA-335` [8] — это спецификация общеязыковой инфраструктуры и платформы .NET, определяющий архитектуру исполнительной системы .NET, устройство различных библиотек, структуру и виды сборок, описание промежуточного языка `CIL` и другое.

В соответствии с данным стандартом, отличительными особенностями Exe-сборки является следующее:

- Возможность «прямого» вызова — пользователь может напрямую запустить .NET приложение с расширением `.exe`.
- Наличие собственного адресного пространства и области памяти — поскольку Exe-сборка является исполняемой, то ее можно запустить как отдельный процесс операционной системы с собственным адресным пространством и областью памяти.

Как описывалось выше, существуют две основные платформы .NET: `.NET Framework` и `.NET Core/.NET`. Каждая из данных платформ предоставляет свой вариант реализации Exe-сборок:

1. `.NET Framework`

   В данной платформе существует единственный и, так называемый, «классический» вид Exe-сборки. В результате компиляции мы получаем один файл с расширением `.exe`, который остается запустить пользователю. При этом необходимо учитывать ограничение в виде зависимостей от других сборок.

2. `.NET Core/.NET`

   В данной платформе существует несколько видов Exe-сборок:

   - Native host сборка — в результате компиляции получается как файл с расширением `.exe`, так и файл динамической библиотеки с расширением `.dll`. При этом существует и **ограничение**: запуск Exe-приложения невозможен без наличия соответствующего файла динамической библиотеки, поскольку во время запуска происходит загрузка Dll-библиотеки, содержащей исходный код приложения.

- Single File сборка (или, так называемая, standalone сборка) — сборка, представленная в виде единого файла, позволяет объединить все зависящие от приложения файлы, ресурсы, другие сборки в единую сборку. Данный вид сборки значительно упрощает развертывание и распространение приложения, однако размер такого файла будет большим, так как он будет включать в себя среду выполнения и библиотеки платформы.

2.1.2 Dll-сборка

В соответствии с упомянутым выше стандартом `ECMA-335` [8] отличительной особенностью данного вида сборки является отсутствие возможности «прямого» запуска, вследствие чего Dll-сборка не имеет своего адресного пространства или области памяти. Такая сборка является динамической в том смысле, что она может быть загружена или подключена только внутри какого-либо другого процесса, например, консольного приложения или веб-приложения. Dll-сборку также именуют библиотекой классов, ведь фактически такая сборка содержит исходный код, но не имеет собственной «точки входа».

Стоит отметить, что в `.NET Framework`, что в `.NET Core/.NET` существует единственный вид Dll-сборок.

Подводя итог обзору различных видов .NET сборок следует также сказать, что сравнение всех описанных выше видов должно быть поддержано в инструменте для сравнения.

2.2. Структура и содержание .NET сборки

Для реализации инструмента сравнения .NET сборок необходимо также понимать, из чего состоят данные сборки и какую структуру они имеют. Для этого обратимся к официальной документации от Microsoft [9].

Так, любая .NET сборка, как Exe, так и Dll, состоит из следующего:

- Манифест сборки — ключевой компонент сборки, без которого невозможен запуск исходного кода. Включает в себя как имя, версию, так и список всех файлов сборки, список ссылок на другие сборки, список ссылок на типы, используемые сборкой. Манифест позволяет системе определить все файлы, входящие в сборку, сопоставить ссылки на типы, ресурсы, сборки с их файлами, управлять контролем версий.

- Метаданные типов — используются для определения местоположения типов в файле приложения.

Рис. 1: Содержание .NET сборки

- Код приложения на промежуточном языке CIL. Так, файл сборки не содержит исходного кода на языке `C#` или другом .NET-совместимом языке, а вместо этого содержит так называемый IL-код (байт-код) — язык виртуальной машины .NET.

- Ресурсы сборки. Это могут быть как различные JSON, XML файлы, так и изображения, аудиозаписи.

Дополнительно стоит отметить, что существует такое понятие, как **формат .NET сборки** [10] — формат двоичного файла. Формат полностью определен и стандартизирован в спецификации `ECMA-335` [8]. Все компиляторы и среды выполнения .NET используют данный формат. Например, он определяет, что сборка не зависит от процессора и операционной системы.

Наличие подобного формата позволяет сделать инструмент для сравнения сборок **платформо-независимым** в том смысле, что нет необходимости в каких-либо проверках того, на каком процессоре или операционной системе была получена та или иная сборка. Благодаря единому формату сборок становится возможным сравнивать любые сборки, даже если одна из них была получена, например, на операционной системе Windows, а другая — на операционной системе macOs.

2.3. Обзор существующих решений

Прежде чем переходить к реализации инструмента сравнения .NET сборок, необходимо понимать, существуют ли инструменты, обеспечивающие подобную функциональность. Для выполнения данной задачи был осуществлен поиск существующих решений. Были произведены следующие запросы: «.net assemblies diff», «assembly diff tool for .net», «how to compare .net assemblies»,

«compare compiled .net assemblies», «compare dll/exe files», «dll/exe diff». По каждому из запросов были изучены первые 2-3 страницы выдачи поисковой системы `Microsoft Bing`. Все найденные решения были рассмотрены на предмет соответствия данной предметной области в рамках системного обзора литературы. Для определения того, подходит ли найденное решение, был выдвинут следующий **критерий выборки**:

Найденное решение/способ должен обеспечивать возможность сравнения .NET сборок с возможностью просмотра разницы исходного кода, представленного в виде декомпилированного кода на `C#`.

Данный критерий выборки обусловлен целью работы, объявленной в разделе 1., поскольку итоговый инструмент сравнения сборок в Rider должен обеспечивать возможность сравнения исходного кода, представленного в виде декомпилированного кода на `C#`. Очевидно, что сравнение исходного кода в формате промежуточного `IL` кода, то есть на языке высокоуровневых машинных команд платформы .NET, или же на любом другом высокоуровневом .NET-совместимом языке не имеет настолько практической пользы, поскольку именно `C#` среди всех .NET языков является наиболее распространенным.

Найденные решения, удовлетворяющие **критерию выборки**, можно разделить на две группы:

- Решения, позволяющие сравнивать сборки с использованием нескольких различных инструментов.
- Полноценные решения, позволяющие сравнивать сборки напрямую без использования каких-либо сторонних инструментов.

2.3.1 Решения с использованием нескольких различных инструментов

Данный вид решений нельзя считать своего рода конкурентами, поскольку они не являются единым и полноценным инструментом.

Решения подобного вида основаны на использовании двух различных инструментов и обеспечивают достижение цели следующим образом:

1. К каждой из двух сборок применяется .NET декомпилятор — инструмент, позволяющий восстановить, то есть декомпилировать из промежуточного `IL` кода исходный код на языке `C#`. Примерами таких декомпиляторов являются:
    - Ildasm

- IlSpy
- .NET Reflector
- dotPeek

Далее полученный исходный код на языке `C#` необходимо сохранить в текстовом виде.

2. К полученному исходному коду необходимо применить любой, отвечающий за сравнение текста, инструмент. Примерами таких инструментов могут быть различные IDE, которые обеспечивают сравнение файлов.

2.3.2 Полноценные инструменты сравнения

Данный вид решений объединяет полноценность — все они представлены в виде единого инструмента, обеспечивающего и декомпиляцию, и показ разницы исходного кода.

2.3.2.1 BitDiffer

BitDiffer — инструмент для сравнения .NET сборок с открытым исходным кодом на GitHub [12]. Графически сравнение сборок представлено в виде дерева, в котором показываются различия в названии сборок, версии, атрибутов сборки. Также в дереве отображены пространства имен, классы, интерфейсы и внутренние типы, такие как методы, поля, свойства и другие типы платформы .NET.

Отличительной особенностью данного инструмента, как и существенным недостатком, является отсутствие возможности полного просмотра декомпилированного кода. Так, в дереве можно увидеть добавленный или удаленный метод, или же, например, пометку о том, что реализация метода изменилась. Однако непосредственно саму реализацию просмотреть возможности нет, то есть **BitDiffer** декомпилирует только сигнатуру типа (модификаторы доступа, название типа, параметры).

Ключевые недостатки:

- Отсутствие возможности полного просмотра декомпилированного кода.
- Данный инструмент позволяет сравнивать сборки только платформы `.NET Framework` и полностью не поддерживает сборки платформы `.NET Core/.NET`.

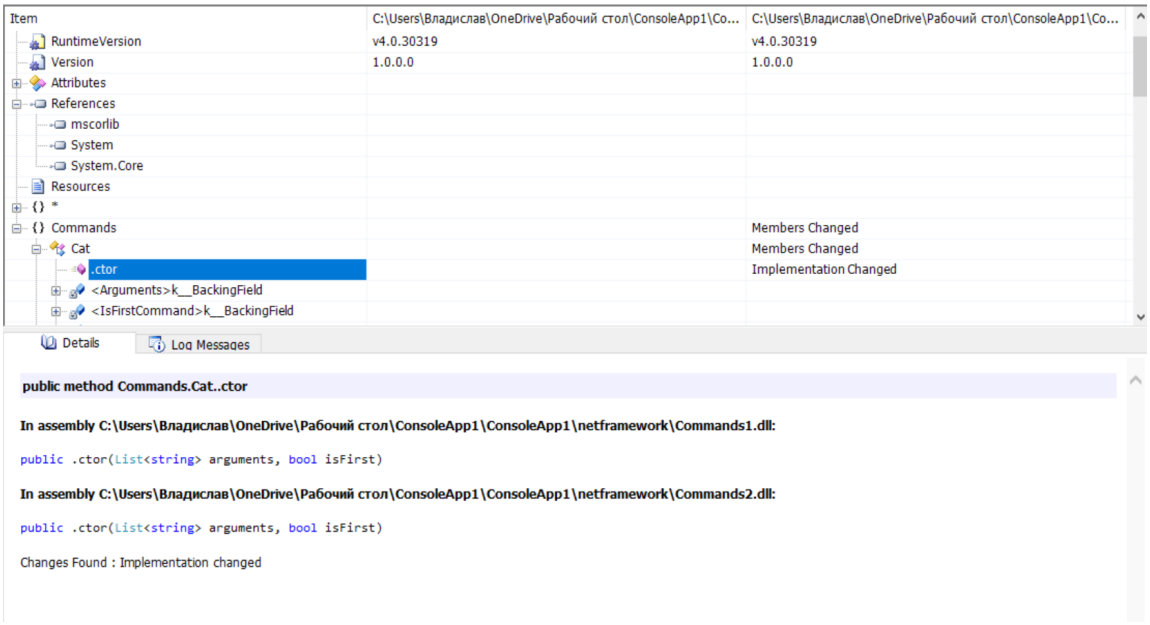

Рис. 2: Инструмент для сравнения сборок `BitDiffer`

Этот фактор является наиболее главным недостатком и фактически свидетельствует об «устаревании» данного инструмента, что делает его неактуальным в настоящее время. Об этом также свидетельствует и дата выпуска последней версий продукта — май 2015 года.

2.3.2.2 JustAssembly

`JustAssembly` [13] — приложение для сравнения .NET сборок компании `Telerik` с открытым исходным кодом. Для декомпиляции `IL` кода в `C#`-представление используется собственный декомпилятор, основанный на библиотеке `Mono.Cecil` [14] — библиотека с открытым исходным кодом, позволяющая работать со сборкой как с массивом байтов. С ее помощью можно как создавать свои собственные сборки, так и модифицировать уже существующие.

По аналогии с `BitDiffer`, сравнение сборок представляется в виде дерева, в котором также можно увидеть узлы, отвечающие за пространство имен типов, внутри которых расположены различные классы, интерфейсы и другие типы. Благодаря такому представлению можно понять, какие классы или другие сущности были добавлены, удалены или же изменены каким-либо образом. Так-

же помимо этого в дереве показывается сравнение ресурсов сборки — можно увидеть, какие, например, фотографии или JSON-файлы были добавлены или удалены, что, безусловно, является достоинством инструмента `JustAssembly`. Однако просмотреть непосредственно сам контент ресурсов не представляется возможным.

Одним из важных достоинств также является возможность просмотра разницы декомпилированного кода при нажатии на соответствующий узел дерева. Сравнение кода в данном инструменте основано на разностном алгоритме Майерса [16] — одном из алгоритмов, использующихся для сравнения текста.

Ключевые недостатки:

- Отсутствие поддержки сравнения Exe-сборок платформы `.NET Core/.NET` — не поддерживаются как `Native host` Exe-сборки, так и `Single File` Exe-сборки. Для платформы `.NET Core/.NET` есть возможность сравнения только Dll-сборок.

- При определенных случаях дерево строится не совсем корректно. Например, в «старой» сборке определен какой-либо метод, а в «новой» сборке, то есть в новой версии, в сигнатуру этого метода был добавлен какой-либо новый параметр. В таком случае хотелось бы увидеть узел, показывающий, что метод изменился, ведь изменилась его сигнатура. Однако в данном случае в дереве будет показано два различных узла, значащих, что «старый» метод был полностью удален, а «новый» метод добавлен, даже если кроме сигнатуры у них не изменилась реализация. Резюмируя, любые типы, имеющие разную сигнатуру, считаются разными.

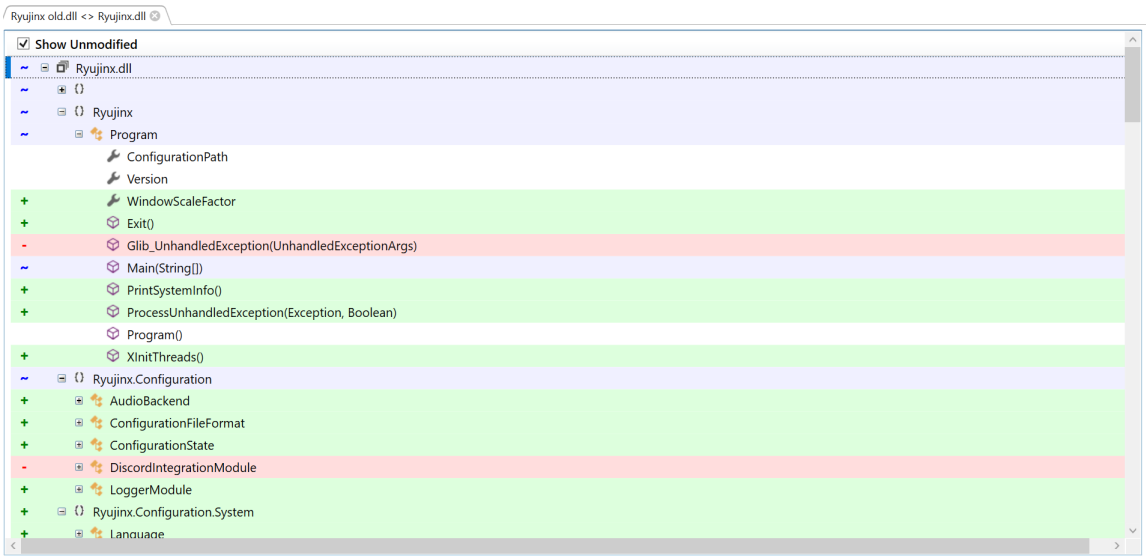

Рис. 3: Разница сборок в виде дерева в `JustAssembly`

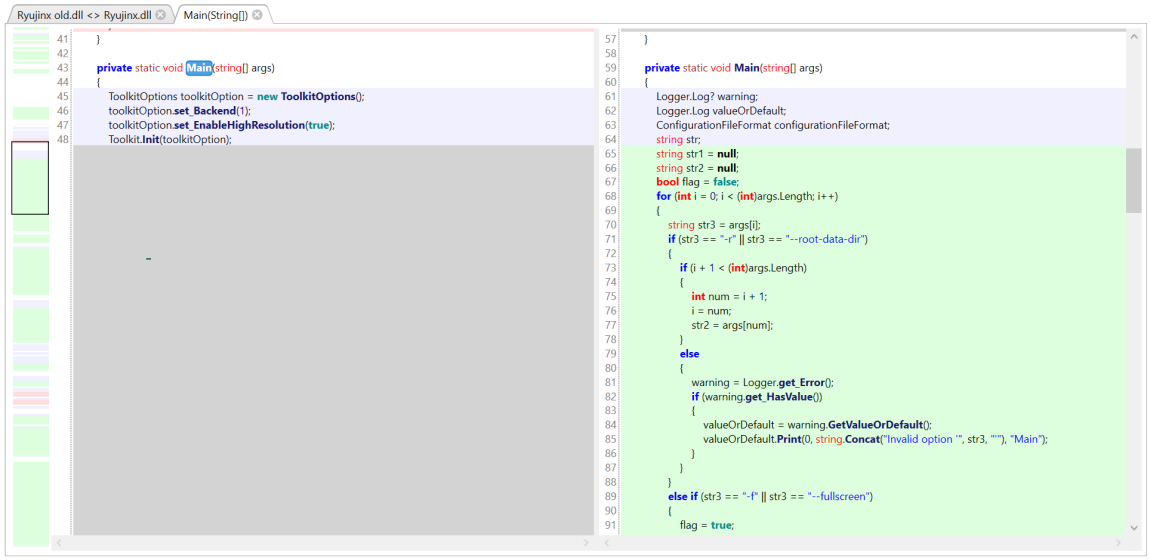

Рис. 4: Разница декомпилированного кода в `JustAssembly`

2.4. Архитектура Rider

Для достижения цели данной работы необходимо также понимать архитектуру Rider, поскольку она напрямую влияет на реализацию инструмента сравнения сборок.

Как говорилось ранее, Rider [3] — кроссплатформенная среда разработки для платформы .NET от компании JetBrains. Для понимания внутреннего устройства и архитектуры среды Rider обратимся к соответствующей статье [11], опубликованной в журнале `CODE Magazine`. Так, среда разработки Rider состоит из двух основных параллельно запущенных процессов операционной системы:

- «Фронтенд»-процесс — отвечает за отрисовку пользовательского интерфейса, основан на платформе `IntelliJ` [6] и работает на виртуальной машине Java.

- «Бэкенд»-процесс — процесс без пользовательского интерфейса, отвечающий за анализ кода, код-инспекции, форматирование и другую логику. Основан на продукте `ReSharper` [5], представляющем собой расширение к среде разработки Visual Studio от компании JetBrains, и работает на виртуальной машине .NET.

Один из ключевых моментов в работе среды разработки Rider — «общение» двух процессов. Для данной цели используется специально разработанный «Reactive Distributed» протокол [15], представленный в виде библиотеки с открытым исходным кодом. Данный протокол обеспечивает принцип реактивности: он предоставляет сущности, которые могут реагировать на изменения через механизм подписки на события. Также данный протокол гарантирует консистентность состояния системы в каждый момент времени и, что немаловажно, позволяет работать в распределённых системах. Именно благодаря ему можно обеспечивать обмен информацией между двумя процессами.

Схематично архитектура Rider изображена на рисунке 5 ниже.

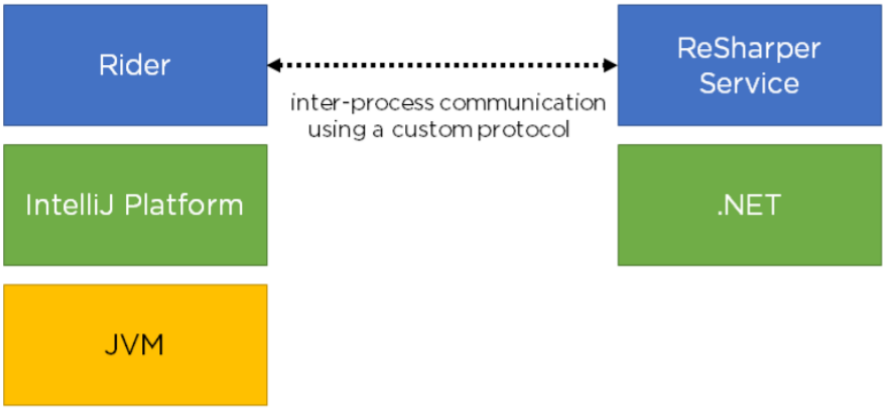

Рис. 5: Архитектура среды разработки Rider (из соответствующей статьи [11])

3. Архитектура инструмента сравнения

3.1. Декомпозиция задачи

Высокоуровнево задача реализации инструмента сравнения разделяется на три ключевые подзадачи:

- «Разобрать» сборку и найти объекты, которые различаются и содержат исходный код. Данная подзадача обусловлена тем, что сборка может состоять из нескольких исходных файлов и имеет определенную структуру, описанную в разделе 2.2. В связи с этим среди различных таблиц метаданных, ресурсов сборки и другого содержимого необходимо найти именно объекты исходного кода.

- Декомпилировать найденные объекты обратно в `C#` код, поскольку в сборке они будут представлены в виде промежуточного `IL` кода.

- Посчитать и отобразить разницу, представленную в виде кода на `C#`.

Таким образом, принимая во внимание архитектуру, было принято следующее решение:

- «Разбор» сборки и поиск объектов исходного кода, как и процесс декомпиляции в `C#`, должен выполняться на «бэкенд»-процессе с использованием возможностей `ReSharper`.

- Вычисление и отображение разницы должно выполняться на «фронтенд»-процессе с использованием возможностей платформы `IntelliJ`.
- Обмен информацией между двумя процессами должен обеспечиваться протоколом RD, описанном в разделе 2.4.

3.2. Реализация, основанная на сравнении директорий/каталогов платформы `IntelliJ`

В платформе `IntelliJ` уже реализована возможность сравнения произвольных каталогов и архивов. В интегрированной среде разработки `IntelliJ IDEA` также присутствует возможность сравнивать JAR-файлы, представляющие собой некий аналог .NET сборки. JAR-файлы являются архивами, содержащими исходный код, файлы манифеста и различные ресурсы, например, аудиофайлы. Инструмент сравнения JAR-файлов подсвечивает разницу между декомпилированными версиями Java-классов внутри них.

На основе имеющихся возможностей платформы `IntelliJ IDEA` было принято решение сделать первую версию инструмента сравнения .NET сборок, пример работы которого приведен на рисунке 6.

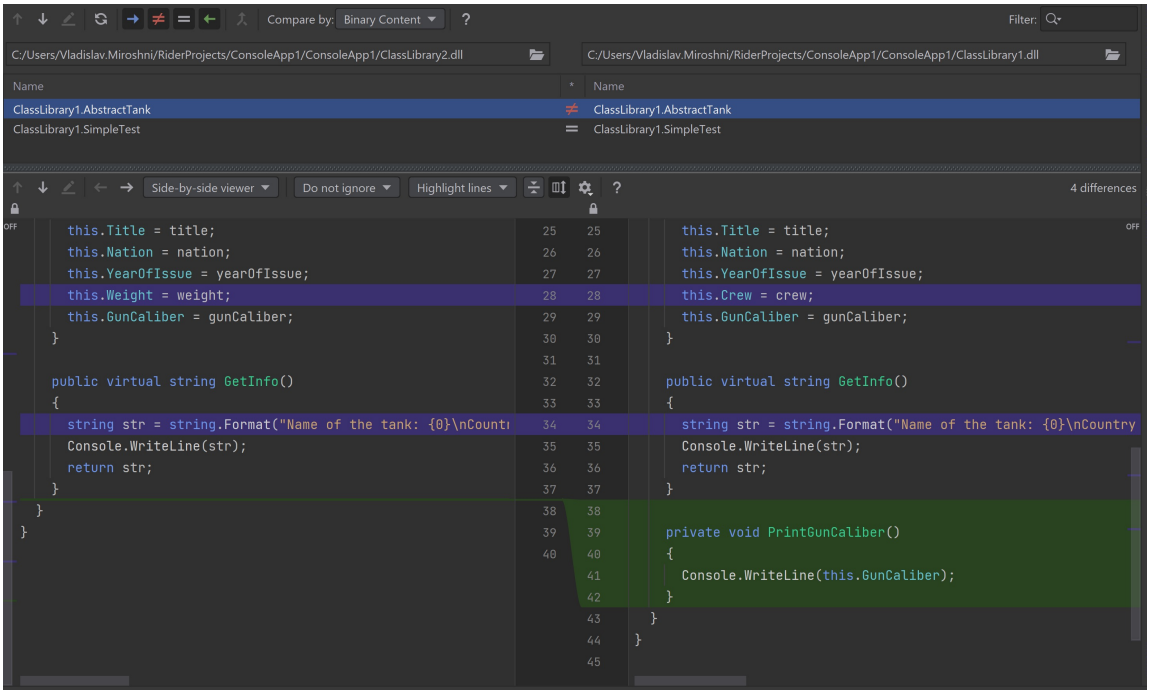

Рис. 6: Пример работы инструмента сравнения .NET сборок, основанного на сравнении каталогов

Однако выбранный способ имеет свои недостатки. Например, представление, основанное на сравнении каталогов, не позволит просматривать разницу зависимостей сборок и ресурсных файлов, а также не позволит интегрировать данный инструмент в другие продукты `ReSharper` и `dotPeek` из-за наличия зависимости от платформы `IntelliJ`. Таким образом, было принято решение рассмотреть альтернативный вариант. Для этого проведем обзор подсистемы Rider — `Assembly Explorer`.

3.3. Подсистема `Assembly Explorer`

`Assembly Explorer` — существующая в среде разработки Rider подсистема, предоставляющая широкие возможности по изучению содержимого сборок. Сборка представляется в виде дерева, в корневом узле которого указано название сборки, версия, платформа, например, `.NET Framework v.4.8` или `.NET Core v5.0`. В самом дереве при раскрытии корневого узла пользователь может увидеть следующую информацию:

- Метаданные сборки — в данном поддереве представлена вся информация о метаданных, различные таблицы метаданных, специальные токены типов и другое.

- Зависимости сборки — в данном поддереве содержится информация о зависимостях сборки, то есть на какие другие сборки, пакеты, модули ссылается текущая сборка.

- Ресурсы сборки — в данном поддереве находятся различные ресурсы, например, аудиофайлы, изображения, XML документы. При двойном нажатии на узел ресурса можно просмотреть его содержимое. Например, при нажатии на узел JSON файла произойдет открытие содержимого в новом окне.

- Пространства имен и вложенные типы — данное поддерево отвечает за исходный код. Можно просматривать классы, его внутренние части, такие как методы, поля, конструкторы и другое. При двойном нажатии на узел происходит декомпиляция исходного кода в `C#`, открытие в новом окне и навигация в документе.

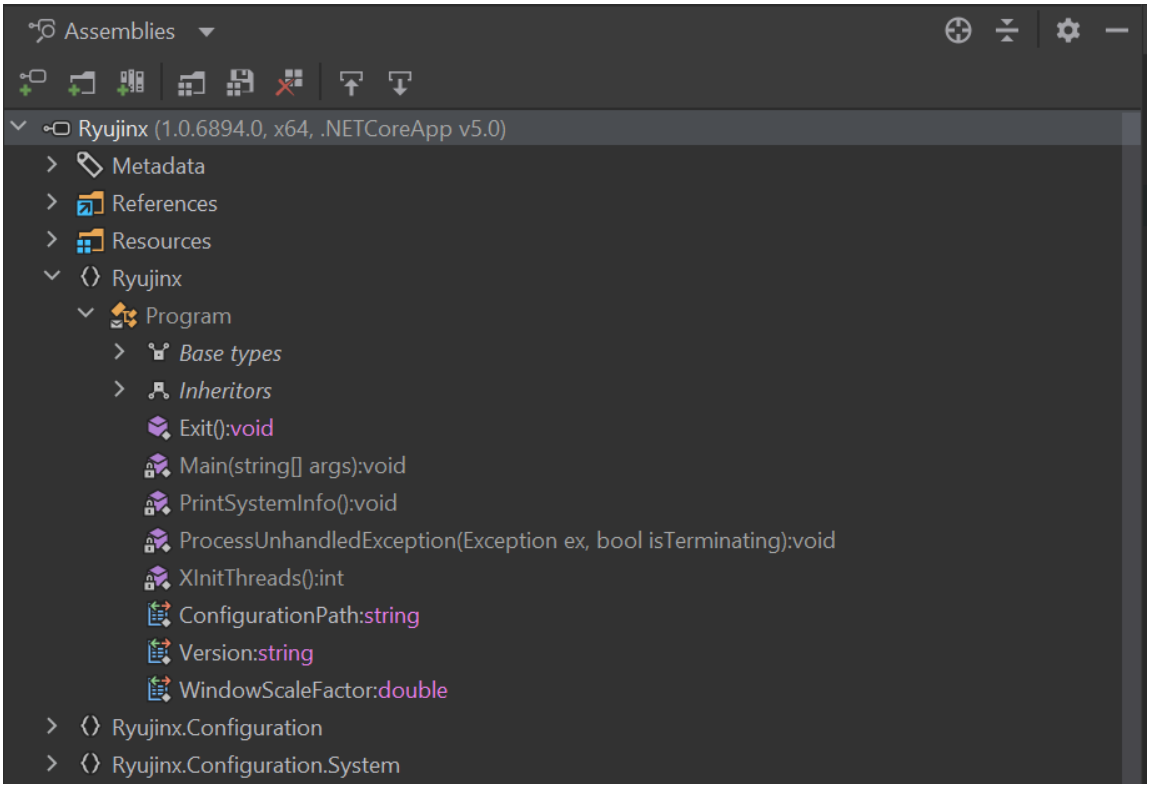

Рис. 7: Дерево сборки в `Assembly Explorer`

Стоит также отметить, что данная подсистема внедрена и в другие продукты .NET экосистемы JetBrains — `ReSharper` и `dotPeek`.

Принимая во внимание возможности подсистемы `Assembly Explorer`, было принято следующее **ключевое** решение — использовать в реализации инструмента сравнения представление в виде дерева по следующим причинам:

- Представление сравнения сборок в виде дерева значительно расширяет функциональность инструмента в отличие от версии, основанной на сравнении каталогов. Подобное дерево сравнения сборок позволит сравнивать не только разницу исходного кода, но также и ресурсов, зависимостей от других сборок.

- Сравнение сборок в виде дерева также позволит расширить и **область применения** данного инструмента. Так, интеграция с подсистемой `Assembly Explorer` в дальнейшем позволит внедрить функциональность сравнения

сборок в другие продукты — `ReSharper` и `dotPeek`, поскольку общая кодовая база `Assembly Explorer` находится в `ReSharper`. По данной причине выбор способа, основанного на сравнении каталогов, значительно снизит область применения данной функциональности и возможные сценарии работы, поскольку позволит сравнивать сборки только в Rider.

Учитывая выбранный способ, предлагается следующее:

- Интегрироваться с подсистемой `Assembly Explorer` и реализовать свое дерево сравнения сборок, в котором узлы будут отвечать за добавленный, удаленный или измененный тип .NET сборки — класс, интерфейс, метод и другие сущности. Подобное представление помимо просмотра разницы исходного кода позволит пользователю понимать, какие сущности изменились от одной версии сборки к другой.

- Добавить возможность сравнения зависимостей от других сборок и ресурсов.

- При двойном нажатии на узел дерева, отвечающий за сущность исходного кода, в соответствующем окне показывать разницу декомпилированного кода на `C#`. При этом должна обеспечиваться и навигация. Например, при нажатии на узел метода необходимо навигироваться в декомпилированном коде к данному методу.

4. Реализация инструмента сравнения сборок

4.1. Общий принцип работы

Решение, основанное на сравнении сборок в виде дерева, работает следующим образом:

1. При запросе пользователя «сравнить две сборки» на фронтенд-процессе происходит инициализация фронтенд-дерева, а также различных дополнительных компонентов. В это же время на бэкенд посылается асинхронный запрос для «чтения» сборок и выполняется инициализация «внутреннего бэкенд»-дерева, которое создается отдельно на каждый запрос. При разборе сборок среди различной метаинформации и другого содержимого происходит поиск объектов и разделение на 3 поддерева: поддерево зависимостей, ресурсов и типов сборки. При этом выполняется сопоставление соответствующих пар для слияния в один узел дерева (например один и тот же метод с изменившейся реализацией от одной версии сборки к другой), а также применяются различные алгоритмы, например, семантического сравнения .NET-типов, о котором будет рассказано в разделе 4.3. Найденные данные пересылаются на фронтенд-процесс с использованием протокола, описанного в разделе 2.4., и отображаются в виде итогового дерева сравнения сборок. Стоит отметить, что именно благодаря реактивному протоколу обеспечивается консистентность данных в любой момент времени между фронтенд и бэкенд-деревьями.

2. При двойном нажатии на узел дерева, отвечающий за .NET-тип (класс, поле, метод и т.д.) посылается запрос на бэкенд, находится нужный узел бэкенд-дерева и при помощи декомпилятора выполняется трансляция IL-кода в `C#` с дальнейшей пересылкой декомпилированного кода на фронтенд. При этом обеспечивается и навигация: на фронтенд также посылается необходимый параметр, на который нужно сдвинуть каретку в соответствующем окне.

3. На фронтенд-стороне вычисляется разница `C#` кода с применением алгоритмов Майерса и LCS [16, 17], сдвигается каретка и выполняется отображение пользователю с подсветкой синтаксиса языка `C#`.

4. Для «общения» между процессами используется специально созданная протокольная модель на Kotlin DSL, описывающая контейнеры/протокольные типы, которые содержат какие-либо данные (например, декомпилиро-

ванный код класса или пути к сборкам). Данные протокольные типы затем сериализуются и десериализуются при взаимодействии двух процессов.

Схематично архитектура инструмента сравнения сборок изображена на рисунке 8.

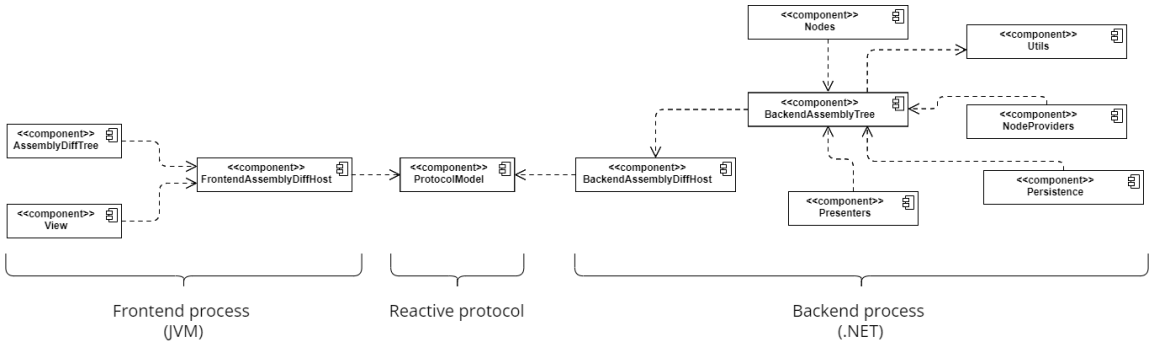

Рис. 8: Диаграмма компонентов инструмента сравнения сборок

Стоит дополнительно отметить, что данный инструмент поддерживает сравнение как `.NET Framework`, так и `.NET Core/.NET` сборок, включая их подвиды, описанные в разделе 2.1., что выделяет данный инструмент в сравнении с существующими решениями по количеству поддерживаемых видов сборок.

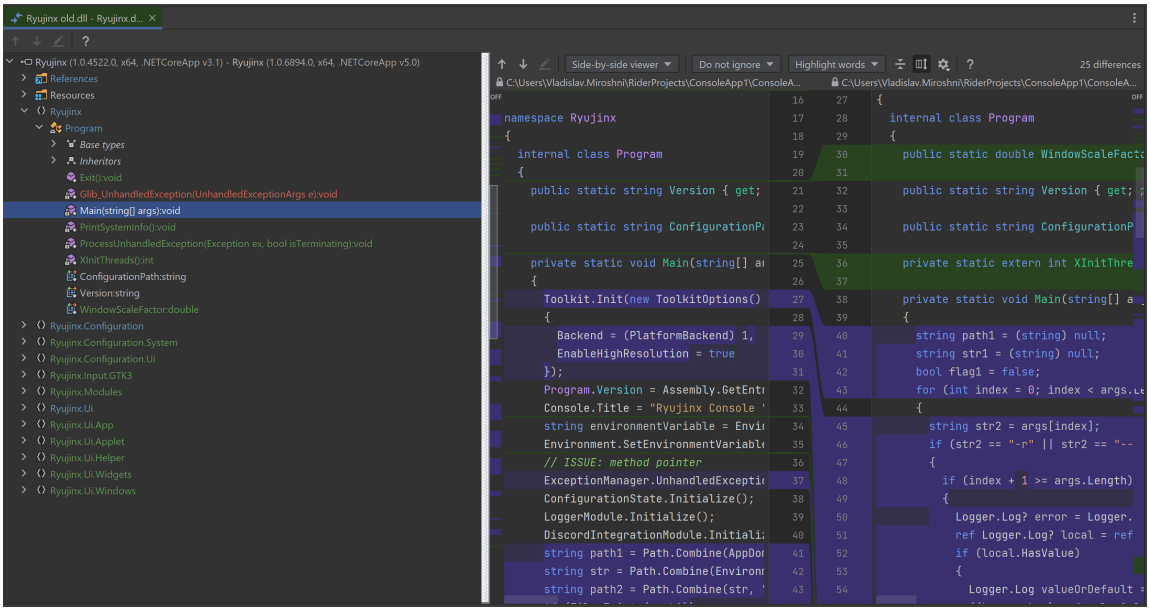

Рис. 9: Пример работы инструмента сравнения сборок

Рассмотрим структуру и алгоритм построения дерева сравнения.

4.2. Алгоритм построения дерева сравнения сборок

Дерево сравнения сборок — **ключевой компонент** в инструменте сравнения, позволяющий просматривать разницу в ресурсах, зависимостях, самих .NET-типах, а также разницу декомпилированного кода на `C#`.

На бэкенд-процессе при поступлении соответствующего запроса создается отдельный экземпляр `AssemblyDiffTreeHost`, отвечающего за построение и обновление дерева.

Алгоритм построения дерева состоит в следующем:

1. После получения пары путей сборок происходит загрузка сборок и создается корневой узел дерева, содержащий информацию о двух сборках, различиях в названии, версиях, платформах, архитектуре, на которой была получена сборка.

2. После загрузки сборок среди различной метаинформации и другого содержимого ищутся так называемые «точки входа» и от корневого узла

нулевого уровня создается первый уровень дерева, содержащий узел зависимостей, узел ресурсов и узлы, представляющие контейнеры для .NET-типов — пространства имен. Таким образом, дерево разделяется на три поддерева, которые обрабатываются и строятся независимо в асинхронном режиме.

3. Для каждого уровня дерева справедливо следующее: глобально есть три группы сущностей: узлы, провайдеры и презентеры. Каждый уровень дерева строится следующим образом: для каждого узла вызывается соответствующий провайдер, который строит поддерево следующего уровня, вычисляя детей и порождая нужные экземпляры узлов. Можно говорить, что провайдер n-1-го уровня порождает узлы n-го уровня дерева. Например, при посещении узла, отвечающего за класс, вызовется соответствующий провайдер, который будет порождать детей, то есть строить следующий уровень, содержащий типы данного класса: методы, поля, события, вложенные классы и т.д. Таким образом, дерево сравнения сборок строится с применением поиска в ширину для обхода дерева в порядке уровней. При этом на каждом уровне дерева необходимо выполнять поиск и сопоставление сущности старой сборки с новой и слияние в один узел дерева (в случае, если пары нет, то объект считается удаленным или добавленным). Для каждого из поддеревьев ресурсов, зависимостей и типов применяются собственные алгоритмы сопоставления.

4. Презентеры применяются для составления отображения узла: генерация текста, выставление стилей и цветов, выбор соответствующей иконки. Далее данные узла оборачиваются в специальный протокольный контейнер и пересылаются на фронтенд для распаковки и отображения пользователю.

Рассмотрим подробнее принцип работы отдельных поддеревьев.

4.2.0.1 Поддерево зависимостей

Данное поддерево содержит узлы двух типов: `AssemblyReference` (зависимость от другой сборки, например, `System.Runtime`) и `ModuleReference` (это могут быть ссылки на какую-либо стороннюю библиотеку). Для всех зависимостей старой и новой сборки производится объединение и дальнейшая группировка по имени без учета версии и дополнительных токенов. Таким образом, каждая найденная пара «старая-новая зависимость» будет объединяться в один узел и для нее будет вычислена разница в версиях для показа пользователю.

Если для какой-то зависимости нет пары, то она считается добавленной или удаленной.

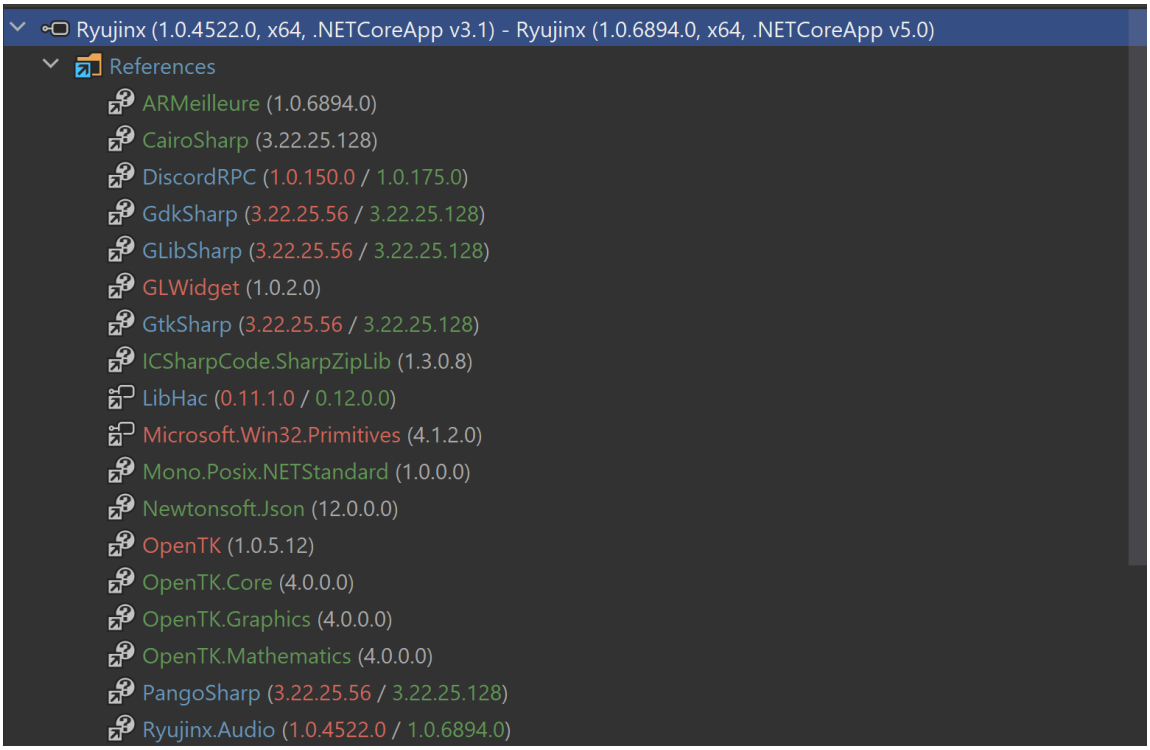

Рис. 10: Пример поддерева зависимостей

4.2.0.2 Поддерево ресурсов

Данное поддерево содержит различные узлы ресурсов: фото, XML, JSON-файлы и т.д. По аналогии с поддеревом зависимостей происходит объединение и группировка по имени ресурса, затем поиск и слияние пар. Таким образом, пользователь сможет определять, какие ресурсные файлы были добавлены или удалены от одной версии сборки к другой.

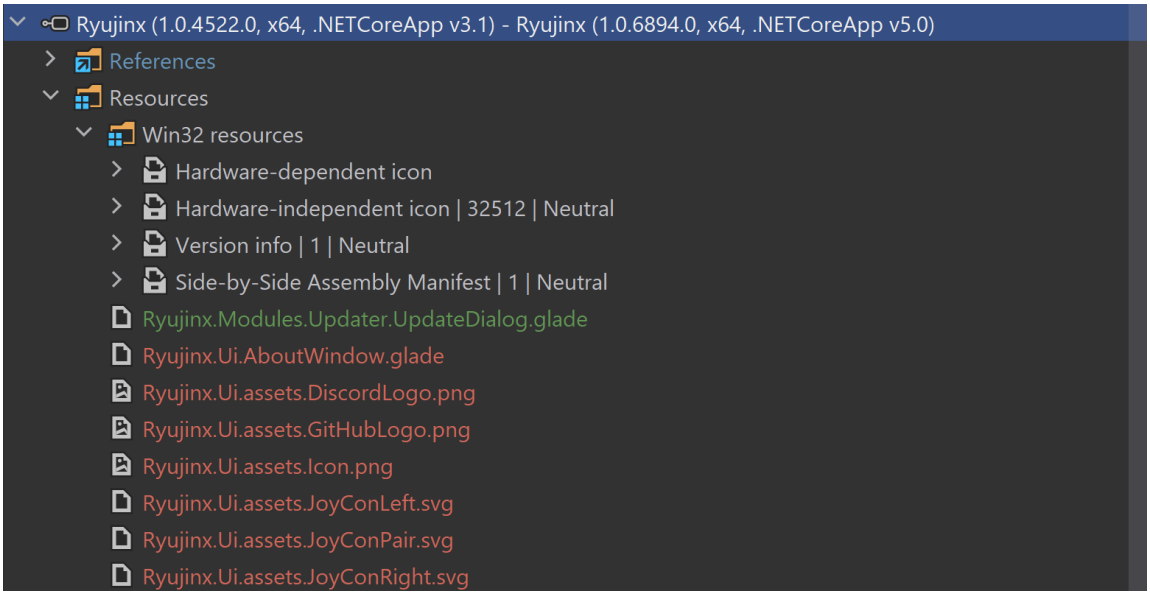

Рис. 11: Пример поддерева ресурсов

4.2.0.3 Поддерево типов сборки

Данное поддерево содержит пространства имен и вложенные в них типы: классы, интерфейсы, перечисления и т.д. со своими собственными внутренними типами.

При построении этого поддерева несколько меняется логика и общий подход. Алгоритм следующий:

1. На самом первом этапе поиска ищутся пространства имен из обеих сборок и группируются по уникальному для CLR `QualifiedName`, при этом стоит отметить, что в одной сборке не может быть двух пространств имен с одинаковым полным именем. Таким образом, у нас будут две группы: в первой будут представлены пространства имен без пары, то есть удаленные или добавленные пространства имен. Во второй группе будут пары «старое-новое пространство имен», означающие, что мы нашли нужную пару для слияния в один узел дерева. Для определения состояния такого узла (изменен/не изменен) происходит следующее: сначала производится быстрая проверка — сравнивается количество внутренних типов для пары пространств имен. Например, если не совпадает количество классов внутри

старого и нового пространства имен, то такой узел считается измененным. Если быстрая проверка не прошла, то производится полная проверка: для обоих пространств имен собираются CLRName вложенных типов и выполняется их сравнение как множеств. При этом возможен один крайний случай — если пространства имен не отличаются по своим вложенным типам, а различие есть только на более глубоких уровнях (например изменилось тело метода внутри класса). В таком случае состояние узла пространства имени будет динамически обновлено с помощью специального триггера.

2. На втором этапе для каждого пространства имён строятся свои поддеревья независимо друг от друга. Этот уровень будет содержать «детей» пространств имен — классы, интерфейсы, структуры, перечисления и записи. При этом также произойдет группировка по имени типа и получатся две группы. В первой группе будут узлы без пары, то есть узлы, означающие, что данный тип был удален или добавлен в зависимости от своего расположения (из старой сборки или из новой). Во второй группе будут пары, необходимые для слияния в один узел дерева. При этом состояние узла изменено/не изменено определяется следующим образом:

   (a) Сперва происходит быстрая проверка двух типов — сравнение их сигнатур. Например, если в старой сборке класс был помечен при помощи модификатора видимости `public`, а в новой при помощи `private`, то такой узел определяется изменившимся.

   (b) В случае, если сигнатуры совпадают, происходит полная проверка — для данных двух типов производится рендер IL-кода. Например, для двух классов мы сгенерируем весь их IL-код и далее сравним его для определения состояния узла. В данном случае рендер IL-кода — ключевой момент в определении состояния узла. Можно было рассмотреть другое решение и генерировать для классов их код на языке `C#` и сравнивать его, однако данный подход был бы более медленным, нежели рендер IL-кода, что и было выбрано для достижения наибольшего быстродействия в построении дерева.

3. При построении следующих уровней мы работаем с «детьми» классов, интерфейсов, структур, перечислений, записей, то есть с полями, методами, свойствами, вложенными классами и т.д. Общая логика здесь отчасти совпадает с пунктом 2: для каждого типа, например класса, мы собираем его старые и новые внутренние типы из сборок и также производим группировку по имени типа. Однако здесь возможны три группы:

(a) Первая группа состоит из типов, для которых не нашлось других типов с таким же именем, то есть они обрабатываются как удаленные или добавленные в зависимости от своего расположения.

(b) Вторая группа состоит из пар, то есть у нас нашлись пары типов с одинаковым названием. В данном случае логика меняется: сначала нужно определить, принадлежат ли оба типа какой-то одной сборке. Если это так, это значит, что типы не нужно сливать в один узел, а это два разных типа и соответственно два разных узла дерева. Это возможно за счет механизма перегрузок в платформе .NET. Например, в старой сборке в классе было два перегруженных метода с именем `Start`, а в новой сборке в этом же классе подобных методов нет. В таком случае оба метода считаются удаленными и рассматриваются как разные узлы дерева. Если же каждый из двух типов принадлежит разной сборке, то у нас происходит слияние в один узел и выполняется определение состояния. Также сначала производится быстрая проверка — сравнение сигнатур. Если данная проверка не прошла, то происходит полная проверка — рендер IL-кода. Однако здесь есть свои особенности в зависимости от проверяемого .NET-типа. Например, если происходит работа со свойством, то необходимо помимо рендера IL-кода найти и зарендерить IL-код соответствущих `Getter` и `Setter`, так как в самой сборке они расположены отдельно от свойства. Аналогичное поведение и для событий, где необходимо искать соответствующие `Adder` и `Remover`. Отдельного внимания заслуживает метод: если метод является итератором и в нем используется `yield return` или же если метод помечен ключевым словом `async`, то в таком случае в IL-коде в отдельном compiler-generated классе генерируется так называемая `StateMachine`. В таком случае, чтобы понять, изменился метод или нет, необходимо помимо рендера IL-кода самого метода выполнять поиск среди различных метаданных и атрибутов данной `StateMachine` и далее производить рендер. Аналогичная ситуация и с использованием лямбда-выражений: в таком случае при компиляции создаются отдельные compiler-generated методы, а иногда и целые классы, если есть замыкание.

(c) В третьей группе будут кортежи из 3-х и более типов с одинаковым именем. Здесь будут только типы, которые могут быть перегружены: методы и операторы. Данная группа представляет наибольший, в том числе и исследовательский, интерес. Здесь сразу возникает проблема сопоставления типов: представим, что в старой сборке в классе у нас

есть три перегруженных метода `Start` с разными сигнатурами и в новой сборке есть три перегруженных метода `Start` с разными сигнатурами. Возникает вопрос: как именно сопоставить данные методы для дальнейшего слияния в один узел? Проблема могла бы решаться с помощью специального токена, который выдается каждому типу при компиляции. Если бы он был уникален и не изменялся, мы бы смогли точно найти нужные пары методов по этому токену, однако он меняется при перекомпиляции. В данном случае нужно выбрать некоторый другой подход. Для этого был разработан специальный алгоритм семантического сопоставления и сравнения .NET-типов, о котором будет рассказано в разделе 4.3.

Также на каждом уровне после завершения работы провайдеров вычисляется представление для узла с помощью соответствующего презентера. Особого внимания заслуживает генерация текста. Так, для узлов на каждом уровне этого поддерева (исключая уровень пространства имен как ненужный случай) применяется алгоритм вычисления разницы сигнатур. Этот алгоритм вычисляет и идентифицирует добавление/удаление параметров, изменение типа параметра или возвращаемого значения, а также изменение generic параметров. Разница сигнатур будет показана пользователю в дереве, и ему не придется дополнительно просматривать декомпилированный код.

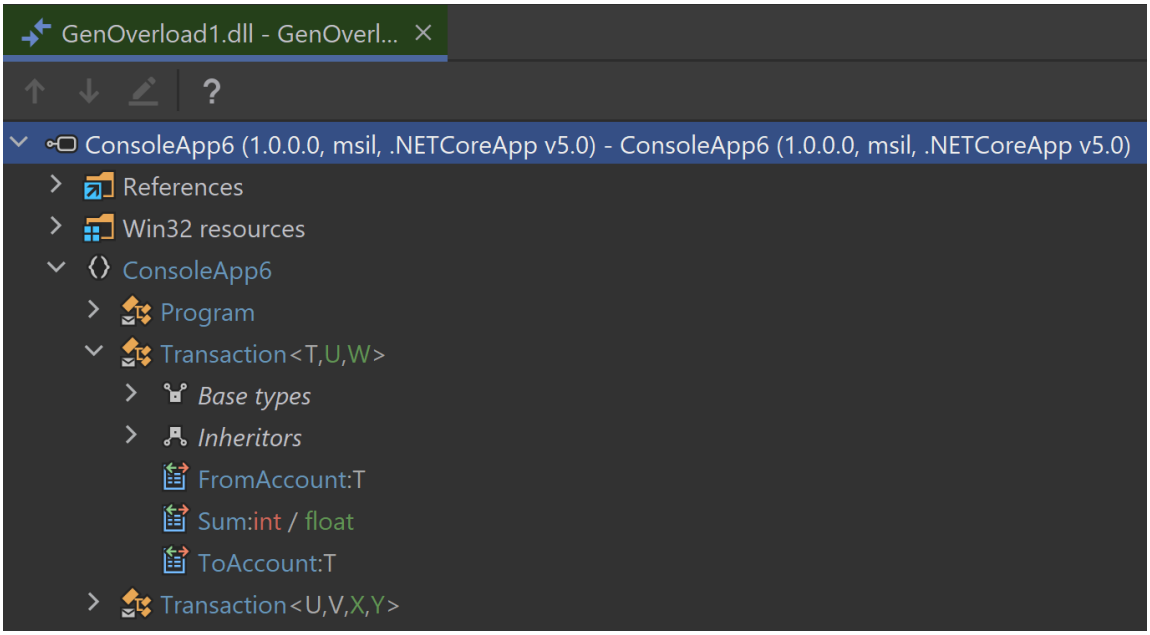

Рис. 12: Пример поддерева типов сборок (включая разницу сигнатур)

Данный алгоритм построения дерева является в некотором смысле уникальным, поскольку он предоставляет широкую функциональность в изучении разницы сборок и включает разницу не только .NET-типов, но и разницу зависимостей и ресурсов, а также использует специально разработанные алгоритмы для сравнения сигнатур и учета семантических особенностей при сопоставлении типов.

Стоит отметить, что в существующих решениях нет подобного поведения при построении дерева и используются не самые точные подходы. Например, в некоторых существующих решениях все типы, имеющие разную сигнатуру, считаются разными. В таком случае представим, что в старой сборке есть какой-то метод `Add`, и считаем, что в новой сборке в данном методе добавился только один параметр в сигнатуру, а само тело метода не изменилось. Тогда в дереве будут два разных узла: метод `Add` со старой сигнатурой будет считаться полностью удаленным, а с новой сигнатурой полностью добавленным, несмотря на то, что реализация метода совсем не изменилась. Однако в данном случае выглядит более корректным показывать пользователю в дереве один узел с состоянием изменено и добавленным параметром в сигнатуру.

4.3. Алгоритм семантического сравнения .NET-типов

Для сопоставления перегруженных типов (методов и операторов) был разработан специальный алгоритм, при этом вводится следующий **инвариант**: методы с разными сигнатурами не считаются разными (в отличие от того, как реализовано в некоторых существующих решениях).

Предположим следующую ситуацию. В старой сборке есть класс с двумя перегруженными методами:

```
class OverloadListing
{
  public void Add(int id, int age)
  {
    // Method body
  }
  // Other methods ...
  public void Add(string name, string email)
  {
    // Method body
  }
}
```

В новой сборке есть такой же класс с двумя перегруженными методами, но данные методы имеют уже другие сигнатуры:

```
class OverloadListing
{
  public void Add(string name, string[] emails)
  {
    // Method body
  }
  // Other methods ...
  public void Add(int id, float age)
  {
    // Method body
  }
}
```

В данном примере алгоритм не считает правильным показывать в дереве два удаленных метода и два добавленных. В случае нескольких перегруженных методов, сигнатуры которых в новой сборке поменялись, алгоритм полагает наи-

более корректным выполнять сопоставление методов с наиболее похожими сигнатурами. Таким образом, необходимо для старого метода искать его наиболее близкий новый метод с учетом семантических различий.

При реализации такого алгоритма наиболее важно предложить подходящую метрику, определяющую близость сигнатур. В данном примере очевидно, что сигнатура из старой сборки `public void (int, int)` ближе к сигнатуре `public void (int, float)`, чем к `public void (string, string[])`, так как в первом случае поменялся тип одного параметра, а во втором — двух. Метрика алгоритма работает следующем образом: для выбранной пары .NET-типов создается `similarityCounter` — счетчик, аккумулирующий данные о близости сигнатур. Далее выполняются множественные проверки: сравнение `CLRName` для данных двух типов, generic параметров (если имеются), сравнение модификаторов доступа, ключевых слов, типов возвращаемого значения, также сравниваются типы передаваемых параметров, их количество, название параметров, наличие дополнительных модификаторов ref/in/out, выполняется `CLRTypeConversion` — определение близости типов данных (например, определяется, что тип float ближе к типу double или int, чем к string) и другие проверки. Для каждой проверки в случае выполнения есть свой вес, на который увеличивается счетчик `similarityCounter`. Например, для ключевых слов есть следующие проверки:

```
public int CompareByMostCommonParams(Element first,
   Element second)
{
    //...
    if (first.IsStatic == second.IsStatic)
        similarityCounter++;

    if (first.IsVirtual == second.IsVirtual)
        similarityCounter++;

    if (first.IsOverride == second.IsOverride)
        similarityCounter++;

    //other keywords checks (sealed, abstract, etc.)
}
```

Используя данную метрику, выполняется сравнение всех сигнатур из первой сборки со всеми сигнатурами из второй и применяется жадный алгоритм: сортируются всевозможные пары и берутся наиболее близкие пары сигнатур, далее

наиболее близкие из оставшихся и т.д. При этом, если в старой сборке в классе три перегруженных метода, а в новой сборке пять, то будут найдены только три сопоставления, а два метода будут считаться добавленными. Вполне возможна ситуация, когда есть пары с одинаковым показателем близости (например, были сигнатуры (int) и (double), а стали (string) и (object)), тогда паросочетание выбирается произвольно. Сравнение тела методов и рендер IL-кода в данной ситуации не выполняется, таким образом уменьшаются накладные расходы. Данный алгоритм в некоторым смысле эвристический — безусловно, можно определить и рассмотреть другие метрики близости.

5. Апробация решения

В качестве апробации решения и получения обратной связи от пользователей использовался сервис YouTrack [18] — система отслеживания ошибок. При помощи данного сервиса как пользователи, опробовавшие инструмент сравнения сборок в EAP версии Rider (Early Access Program), так и непосредственно разработчики, завели 17 соответствующих issue, большая часть из которых была успешно исправлена. Общая выборка разработчиков и пользователей составила не менее 10 человек. В качестве примеров исправленных ошибок можно привести проблемы, связанные с некорректным отображением статуса узла в дереве при определенных сценариях работы и некоторые недочеты с показом разницы сигнатур.

6. Результаты

Благодаря работе над производственной практикой были достигнуты следующие результаты.

1. Проведен обзор различных видов .NET сборок, структуры и содержания .NET сборки.
2. Проведен обзор существующих решений, позволяющих сравнивать .NET сборки.
3. Рассмотрены различные способы для сравнения сборок, а также выбран наиболее подходящий для дальнейшей реализации.
4. Интегрирован в Rider инструмент для сравнения .NET сборок в виде дерева сравнения сборок, обеспечивающего сравнение исходного кода, представленного в виде декомпилированного кода на `C#`.
5. Проведена апробация решения с получением обратной связи от пользователей.

Благодаря полученным результатам, виден дальнейший вектор работы над проектом со следующими задачами:

1. Расширить сравнение ресурсов сборки: добавить возможность просмотра разницы содержимого ресурсов.
2. Интегрировать инструмент сравнения сборок в `dotPeek` и `ReSharper`.

Список литературы

[1] .NET platform from Microsoft:
https://dotnet.microsoft.com/ (дата обращения: 01.11.2021).

[2] Stack Overflow frameworks and platforms survey for 2021 year:
https://insights.stackoverflow.com/survey/2021#most-popular-technologies-misc-tech (дата обращения: 01.11.2021).

[3] JetBrains Rider IDE:
https://www.jetbrains.com/ru-ru/rider/ (дата обращения: 02.11.2021).

[4] Stack Overflow most loved collaboration tools survey for 2021 year:
https://insights.stackoverflow.com/survey/2021#most-loved-dreaded-and-wanted-new-collab-tools-love-dread (дата обращения: 02.11.2021).

[5] JetBrains ReSharper:
https://www.jetbrains.com/ru-ru/resharper/ (дата обращения: 01.11.2021).

[6] JetBrains IntelliJ Platform:
https://www.jetbrains.com/ru-ru/opensource/idea/ (дата обращения: 02.11.2021).

[7] .NET assemblies types according to Microsoft documentation:
https://docs.microsoft.com/en-us/dotnet/standard/assembly/ (дата обращения: 02.11.2021).

[8] Standard ECMA-335: Common Language Infrastructure (CLI), 6th edition, June 2012:
https://www.ecma-international.org/wp-content/uploads/ECMA-335_6th_edition_june_2012.pdf (дата обращения: 02.11.2021).

[9] .NET Assembly contents according to Microsoft documentation:
https://docs.microsoft.com/en-us/dotnet/standard/assembly/contents (дата обращения: 02.11.2021).

[10] .NET Assembly format according to Microsoft documentation:
https://docs.microsoft.com/en-us/dotnet/standard/assembly/file-format (дата обращения: 03.11.2021).

[11] Chris Woodruff and Maarten Balliauw. Building a .NET IDE with JetBrains Rider, CODE Magazine, 2018:
https://www.codemag.com/Article/1811091/Building-a-.NET-IDE-with-JetBrains-Rider (дата обращения: 03.11.2021).

[12] BitDiffer application to compare .NET assemblies:
https://github.com/bitdiffer/bitdiffer (дата обращения: 03.11.2021).

[13] JustAssembly diff tool to compare .NET assemblies:
https://www.telerik.com/justassembly (дата обращения: 03.11.2021).

[14] Mono.Cecil library for .NET assemblies:
https://www.mono-project.com/docs/tools+libraries/libraries/Mono.Cecil/ (дата обращения: 04.11.2021).

[15] Reactive Distributed communication framework:
https://github.com/JetBrains/rd (дата обращения: 04.11.2021).

[16] Eugene W. Myers. An O(ND) Difference Algorithm and Its Variations:
http://www.xmailserver.org/diff2.pdf (дата обращения: 08.11.2021).

[17] J. W. Hunt, M. D. McIlroy. An Algorithm for Differential File Comparison:
https://www.cs.dartmouth.edu/~doug/diff.pdf (дата обращения: 08.11.2021).

[18] JetBrains YouTrack:
https://www.jetbrains.com/ru-ru/youtrack/ (дата обращения: 13.04.2022).