

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РФ

Федеральное государственное бюджетное образовательное учреждение высшего образования

«ПЕНЗЕНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

Факультет

«Вычислительная техника»

(наименование)

Кафедра

«Системы автоматизированного проектирования»

(наименование)

Направление подготовки

09.03.01 «Информатика и вычислительная техника»

(код и наименование)

Профиль

«Системы автоматизированного проектирования»

(наименование)

БАКАЛАВРСКАЯ РАБОТА

на тему

Разработка кроссплатформенного
статического анализатора кода C#

Студент

Кобцев, 2.06.17
(подпись, дата)

Кобцев А.В.

(фамилия, инициалы)

Руководитель

Гузков, 2.06.17
(подпись, дата)

Гузков А.А.

(фамилия, инициалы)

Нормоконтролёр

инф. каф. САПР ПГУ
(должность, место работы)

Гузков, 2.06.17
(подпись, дата)

Гузков А.А.

(фамилия, инициалы)

Работа допущена к защите (протокол заседания кафедры от

№

Заведующий кафедрой

Гузков
(подпись)

Гузков А.А.

(фамилия, инициалы)

Работа защищена с отметкой

(протокол заседания ГЭК от №)

Секретарь ГЭК

(подпись)

(фамилия, инициалы)

Пенза 2017

РЕФЕРАТ

Пояснительная записка содержит листов, рисунков, источников, приложений.

СТАТИЧЕСКИЙ АНАЛИЗ, ОБНАРУЖЕНИЕ ОШИБОК ПРОГРАММ, АВТОМАТИЗАЦИЯ ПОИСКА ОШИБОК, ЯЗЫК ПРОГРАММИРОВАНИЯ, .NET STANDART, C#, ПРОГРАММА, КРОССПЛАТФОРМЕННАЯ РАЗРАБОТКА.

Объектом исследования является статический анализ исходного кода.

Цель работы – разработать программу статического анализа исходного кода, написанного на языке C#.

В процессе разработки проводилось изучение принципов статического анализа исходного кода и создания кроссплатформенных приложений с графическим пользовательским интерфейсом.

Полученные результаты: создан кроссплатформенный статический анализатор исходного кода, написанного на языке C#.

Разработанную программу следует применять в процессе разработки и тестирования приложений для поиска ошибок в коде на языке C#.

Для разработки программы применялся язык программирования C#, среда разработки Microsoft Visual Studio 2017 и операционная система Windows 10.

					ВКР 090301.07 17 81 01				
Изм	Лист	№ докум.	Подпись	Дата	Разработка кроссплатформенного статического анализатора кода С#. Пояснительная записка	Лит.	Лист	Листов	
Разраб.		Кобцев А.В.							
Провер.		Гудков А.А.					3		
						ФВТ, каф. САПР, группа			
Утверд.		Бершадский А.М.				13ВВ2			

СОДЕРЖАНИЕ

Перечень условных обозначений	5
Введение	6
1 Анализ предметной области	8
1.1 Актуальность статического анализа кода	8
1.2 Обзор готовых решений	10
2 Анализ технического задания	14
2.1 Анализ требований к разработке	14
2.2 Анализ функциональных требований	16
2.3 Анализ нефункциональных требований	20
3 Используемые технологии и средства разработки	22
3.1 Язык программирования	22
3.2 Инструментальные средства проектирования	22
3.3 Интегрированная среда разработки	22
3.4 Программная платформа	23
3.5 Средства создания графического интерфейса	24
4 Проектирование и разработка	27
4.1 Графический интерфейс пользователя	27
4.2 Статический анализатор	33
4.3 Разработка диагностических правил	37
5 Функциональное тестирование	42
Заключение	49
Список использованной литературы	50
Приложение. Листинг программы	53

ПЕРЕЧЕНЬ УСЛОВНЫХ ОБОЗНАЧЕНИЙ

ПО – программное обеспечение;

API – (англ. Application Programming Interface) – интерфейс программирования приложений;

UML – (англ. Unified Modeling Language) – унифицированный язык моделирования;

XAML – (англ. eXtensible Application Markup Language) – расширяемый язык разметки для графических приложений;

WPF – (англ. Windows Presentation Foundation) – графическая (презентационная) подсистема в составе .NET Framework, использующая язык XAML;

MVVM (англ. Model-View-ViewModel) – шаблон проектирования приложений с графическим пользовательским интерфейсом.

ВВЕДЕНИЕ

Даже самые высококвалифицированные разработчики ПО совершают ошибки. Ошибки могут быть различными: от простейших опечаток и до логических ошибок в алгоритмах. Чем раньше ошибки в коде приложений будут обнаружены, тем дешевле стоит их исправление.

Обзор кода – один из самых старых и надежных методов выявления дефектов. Он заключается в совместном внимательном чтении исходного кода и высказывании рекомендаций по его улучшению. В процессе чтения кода выявляются ошибки или участки кода, которые могут стать ошибочными в будущем. Единственный существенный недостаток методологии обзора кода, это крайне высокая цена: необходимо регулярно собирать нескольких программистов для обзора нового кода или повторного обзора кода после внесения рекомендаций. Получается, что с одной стороны хочется регулярно осуществлять обзор кода, а с другой – это слишком дорого. Компромиссным решением являются инструменты статического анализа кода, способные помочь разработчикам ПО автоматизировать процесс поиска дефектов в программном коде.

Существует целый класс инструментов статического анализа кода, позволяющих выявить на этапе кодирования значительный процент логических ошибок и простых опечаток [1, 2]. Эти инструменты осуществляют более высокоуровневую проверку кода, опираясь на знание некоторых паттернов кодирования, используют эвристические алгоритмы и имеют гибкую систему настройки.

Статический анализ кода – это процесс выявления ошибок и недочетов в исходном коде программ [1]. Его можно рассматривать как автоматизированный процесс обзора кода. Использование специализированных инструментов статического анализа приносит большую пользу для процесса разработки: они помогают бороться со многими

ошибками и позволяют контролировать качество процесса разработки в целом [2].

Способы использования полученной в ходе анализа информации также различны – от выявления мест, возможно содержащих ошибки, до формальных методов, позволяющих математически доказать какие-либо свойства программы (например, соответствие поведения спецификации).

В данной выпускной квалификационной работе требуется реализовать программу статического анализа исходного кода на языке программирования C#. Актуальность работы заключается в том, что C# является одним из самых востребованных языков и находится в списке самых популярных языков программирования [3]. В настоящее время, с появлением таких технологий как Mono, и .NET Core, разработка программного обеспечения на языке C# возможна не только в операционной системе Microsoft Windows, она распространилась и набирает популярность на других операционных системах. Однако, существующие инструменты анализа кода на языке C# не являются пригодными для внедрения в процесс разработки на операционных системах, отличных от Microsoft Windows. Эти анализаторы не являются кроссплатформенными приложениями и не имеют прямых аналогов на других ОС. Из этого следует, что разработка кроссплатформенного статического анализатора – вопрос актуальный и требующий скорейшего разрешения.

1 АНАЛИЗ ПРЕДМЕТНОЙ ОБЛАСТИ

1.1 Актуальность статического анализа кода

1.1.1 Предпосылки использования статического анализа

Разработчики ПО тратят на поиск и исправление ошибок гораздо больше времени, чем думают. Инструменты статического анализа могут существенно экономить усилия разработчиков, выявляя часть таких ошибок на ранних этапах процесса разработки [4].

Искать обыкновенные ошибки в коде столь же важно и нужно, как и алгоритмические ошибки. Простые ошибки могут приводить к серьезным последствиям [4]. Лучше находить их как можно больше с помощью статического анализатора, а не с огромной тратой времени в отладчике. Вовремя найденная ошибка предотвращает худший вариант: ваше приложение содержит уязвимость, но оно уже установлено на десятках тысяч компьютеров. Полезно найти как можно больше простых ошибок с помощью инструментов, чтобы посвятить себя поиску дефектов в алгоритмах и созданию новой функциональности.

1.1.2 Диагностика всех ветвей программы

Динамический анализ – анализ программного обеспечения, выполняемый при помощи выполнения программ на реальном или виртуальном процессоре (в отличие от статического анализа). Динамический анализ на практике не может покрыть все ветвления программы, особенно те участки кода, которые обрабатывают нестандартные или аварийные ситуации. Этот минус динамических анализаторов заключается в необходимости выполнить код, который содержит ошибку. А для многих участков кода сделать это может быть невозможно. В отличие от динамического анализа статический анализ производится без реального выполнения исследуемых программ [5].

1.1.3 Масштабируемость

Чтобы регулярно проверять большие проекты динамическими методами, необходимо создать специальную инфраструктуру. Необходимо наличие специальных тестов, реализация которых является сложной и объемной работой. На практике, тесты покрывают не более 80% программного кода. Нужен параллельный запуск нескольких экземпляров приложения с различными входными данными. Статический анализ масштабируется в несколько раз проще [6]. Как правило, достаточно предоставить программе, осуществляющей статический анализ, машину с большим количеством ядер. Современные инструментальные средства статического анализа кода предоставляют много информации о выявленных дефектах в удобном для восприятия формате.

1.1.4 Более высокоуровневый анализ

Статический анализ проводит более высокоуровневый анализ, чем динамический [7]. И это позволяет ему искать такие вещи, которые с точки зрения динамического анализа корректны. Теоретически, статический анализатор обладает информацией о коде, а значит способен находить больше ошибок, чем динамический анализатор. На практике, возможности статических анализаторов ограничены только объемом доступной памяти и приемлемым временем работы.

Первостепенная задача компилятора состоит в получении бинарного кода из компилируемых исходных файлов. Скорость компиляции – одна из важных его характеристик, поэтому на статический анализ исходного кода отводится не так много времени, чтобы проводить глубокий анализ или оперировать большим количеством диагностических правил. Поэтому компиляторы сообщают только о самых распространённых проблемных конструкциях в коде.

Будучи развивающимися в одном направлении, специализированные инструменты статического анализа являются более гибкими и развитыми в

сфере анализа исходного кода программ [7]. В отличие от компиляторов, статические анализаторы предоставляют больше диагностических правил, многие из которых диагностируют очень нестандартные и не самые распространённые ошибки.

1.2 Обзор готовых решений

1.2.1 ReSharper (R#)

Коммерческое (платное) дополнение, разработанное компанией JetBrains для повышения продуктивности работы в Microsoft Visual Studio [8]. Проводит статический анализ кода в масштабе всего решения, предусматривает дополнительные средства автозаполнения, навигации, поиска, подсветки синтаксиса, форматирования, оптимизации и генерации кода, предоставляет 40 автоматизированных рефакторингов, упрощает юнит-тестирование.

Из недостатков можно заметить отсутствие локализации продукта для российского рынка. А также то, что диагностики существуют внутри процесса Visual Studio, являющегося 32-битным, в результате чего накладываются серьезные ограничения на объем используемой памяти. В некоторых случаях это критично и не позволит провести глубокий анализ больших проектов.

1.2.2 FxCop

Бесплатный инструмент для статического анализа кода от Microsoft, проверяющий сборки .NET на соответствие рекомендациям по проектированию библиотек .NET Framework [9]. Выполняет анализ кода для проверки соответствия стандартам программирования и соглашениям именования, и позволяет удостовериться, что соответствующие правила используются в написанной программе.

Из недостатков следует отметить поверхностные диагностики исходного кода и отсутствие локализации продукта для российского рынка –

интерфейс и отчеты на английском языке, отсутствие материалов и документации на продукт на русском языке.

1.2.3 Coverity

Пакет ПО, состоящий из статического и динамического анализаторов кода, принадлежащий компании Synopsys [10]. ПО ищет ошибки, и недочеты в безопасности исходных кодах программ, написанных на Си, C++, Java, C# и JavaScript.

Из недостатков следует отметить отсутствие локализации продукта для российского рынка, жесткую лицензионную и ценовую политику и отсутствие автономного приложения с дружелюбным пользователю интерфейсом для анализа кода на языке C#.

1.2.4 PVS-Studio

Инструмент представляет собой статический анализатор кода, выявляющий ошибки и другие недостатки в исходных текстах программ, написанных на языках C, C++ и C# [11]. Анализатор ориентирован на разработчиков Windows приложений.

Из недостатков следует отметить сложность внедрения в процесс разработки ПО и отсутствие кроссплатформенного автономного приложения с графическим интерфейсом пользователя для анализа кода на языке C#.

1.2.5 Результат сравнения аналогов

Ошибки, обнаруживаемые статическими анализаторами весьма разнообразны. Некоторые анализаторы специализируются на определенной области или типах дефектов. Другие, поддерживают определенные стандарты кодирования. Область статического анализа активно развивается, появляются новые диагностические правила и стандарты, некоторые правила устаревают. Поэтому нет смысла пытаться сравнить анализаторы, на основании списков обнаруживаемых дефектов. Единственный способ сравнить инструменты по функциональности – это проверить с их помощью набор проектов и посчитать найденных ими количество настоящих ошибок.

В результате анализа информации, представленной на официальных веб-сайтах компаний-разработчиков рассматриваемых статических анализаторов кода, была составлена табл. 1.1.

Таблица 1.1 – Результаты сравнения статических анализаторов

Параметр	ReSharper	FxCop	Coverity	PVS-Studio
Платность	Да	Нет	Да	Да
Кроссплатформенность	Нет	Нет	Нет	Нет
Интегрируемость	Простая	Простая	Сложная	Сложная
Русская локализация	Нет	Нет	Нет	Да
Продвинутость анализа ¹	5	4	10	7
Кол-во предупреждений	544	420	81	198
Кол-во реальных ошибок	121	42	74	121

Представленные аналоги были проверены на проекте с открытым исходным кодом на языке программирования C#. Проверяемый проект содержит около 5 тысяч строк кода.

Количество строк кода – это метрика программного обеспечения, используемая для измерения его объёма с помощью подсчёта количества строк в тексте исходного кода. Как правило, этот показатель используется для прогноза трудозатрат на разработку конкретной программы на конкретном языке программирования, либо для оценки производительности труда уже после того, как программа написана.

Результаты проверки представлены общим количеством выдаваемых анализатором предупреждений и количеством реальных ошибок, отобранных из этих предупреждений (см. табл. 1.1). Исходя из оценки продвинутости анализаторов, представленной по десятибалльной шкале, и соотношения количества реальных обнаруженных анализаторами ошибок к общему

¹ По 10-балльной шкале

количеству выданных предупреждений, можно сказать, что самыми лучшими анализаторами из представленных, по результатам проверки, оказались только Coverity и PVS-Studio, для которых процент ложных срабатываний оказался в пределах 10-15%.

Из результата сравнения становится ясно, что в большинстве аналогов отсутствует локализация продукта для российского рынка, также большинство продуктов платны (некоторые даже ведут довольно жесткую лицензионную и ценовую политику для российского рынка), половина из них сложны для интеграции в текущий процесс разработки ПО. Большинство этих продуктов имеет большой процент ложных срабатываний, по причине реализации только поверхностных диагностик исходного кода. А также, все эти продукты не имеют кроссплатформенного автономного приложения с графическим интерфейсом пользователя.

Таким образом, из вышеперечисленного выше можно сделать вывод, что создание кроссплатформенного автономного статического анализатора кода C# является актуальной задачей.

2 АНАЛИЗ ТЕХНИЧЕСКОГО ЗАДАНИЯ

2.1 Анализ требований к разработке

Техническое задание на разрабатываемую программу содержит требования:

- программа должна быть написана на языке C#;
- реализована поддержка кроссплатформенности – возможности запуска как в среде .NET Framework, так и Mono.
- автономная работа программы с поддержкой графического пользовательского интерфейса;
- возможность кастомизации диагностических правил;
- возможность интегрирования с существующими системами сборки проектов.

Кроссплатформенность – одно из основных требований для ПО на рынке [12]. Основным источником эффективности ПО – это возможность поставлять его более широкому кругу клиентов. Также важное значение имеют возможность масштабирования в рамках одного предприятия и максимально похожий графический интерфейс на разных ОС.

В настоящее время разработка программных продуктов под платформы, совместимые с .NET, возможна не только под операционной системой Microsoft Windows [13]. Поэтому важно обеспечить таких разработчиков средствами для статического анализа.

Продукт следует разрабатывать как автономное приложение, так как, если оно будет плагином, встраиваемым в среду разработки, то оно будет иметь серьёзные ограничения на объём используемой памяти [14]. Как автономное приложение продукт получит больший контроль над обходом синтаксического дерева и семантической модели, а мы реализуем распараллеливание необходимым нам образом, тем самым больше контролируя процесс разбора и анализа кода.

Для лучшей диагностики следует обеспечить получение синтаксического дерева – базового элемента, необходимого для статического анализа [15]. Мы должны иметь наиболее полную синтаксическую информацию: не только синтаксические конструкции языка, но и конечные символы грамматики языка, а также дополнительную синтаксическую информацию (комментарии, директивы препроцессора, пробелы и т.п.). Это позволит увеличить продвинутость правил диагностики кода.

Для глубокого и качественного анализа нужна полная информация о всех типах, встретившихся в коде [15]. В большинстве диагностических правил недостаточно простого обхода узлов дерева, часто приходится обрабатывать типы выражений и получать информацию об анализируемых объектах. Для этого и нужна семантическая модель, которая должна быть корректной. Для достижения корректности семантической модели проект должен быть скомпилированным, все зависимости должны быть на месте.

При разработке диагностических правил важно предусмотреть все возможные, невозможные, а также ряд невероятных случаев, с которыми мы можем столкнуться в ходе анализа. Для этого необходимо разрабатывать большое количество юнит-тестов, как позитивных – фрагментов кода, где должна срабатывать диагностика, так и негативных – тех фрагментов, на которые не стоит выдавать предупреждения.

При разработке продукта следует обеспечить модульный API для возможной интеграции кастомизированных правил диагностики и управления их списком, а также упрощения интегрируемости с существующими системами сборки проектов.

Следует обеспечить локализацию продукта для российского рынка – интерфейс и отчеты, материалы и документация на продукт должны быть на русском языке.

2.2 Анализ функциональных требований

Единственный пользователь разрабатываемой программы – это разработчик, специализирующийся на языке программирования C#. Разрабатываемая программа (далее анализатор) является моделируемой системой, благодаря возможностям которой разработчик может получить конкретный, измеримый и нужный ему результат.

Разработчик должен иметь следующие возможности:

- возможность выбрать нужные модули с диагностиками из списка имеющихся;
- возможность подключить сторонний модуль с кастомными диагностиками;
- возможность выбрать разрабатываемый им проект для статического анализа;
- возможность запустить статический анализ,
- возможность прервать процесс анализа
- возможность получить отчёт по окончании анализа;
- возможность детально просмотреть отчёт в дополнительном окне программы-анализатора.

Описание основных вариантов использования, выявленных в результате анализа функциональных требований, представлено в табл. 2.1.

Таблица 2.1 – Варианты использования

Действ. лицо	Цель	Краткое описание
Разработчик	Выбор диагностик	Настройка списка диагностик статического анализа
Разработчик	Выбор анализируемого проекта	Выбор проекта, на котором будет проводится процесс статического анализа с применением выбранных диагностик
Разработчик	Управление процессом анализа	Запуск и прерывание процесса статического анализа, а также получение сведений о его прогрессе
Разработчик	Просмотр отчёта	Просмотр списка предупреждений, выданных анализатором по окончанию процесса статического анализа

Таким образом, на основе данных, описанных в табл. 2, можно составить UML диаграмму вариантов использования, представленную на рисунке 2.1.



Рисунок 2.1 – Диаграмма вариантов использования

Для лучшего понимания взаимодействия прецедентов друг с другом наилучшим вариантом будет построение диаграмм деятельности [17]. Наиболее важным с точки зрения использования является прецедент «Управление процессом анализа», диаграмма деятельности которого представлена на рисунке 2.2.

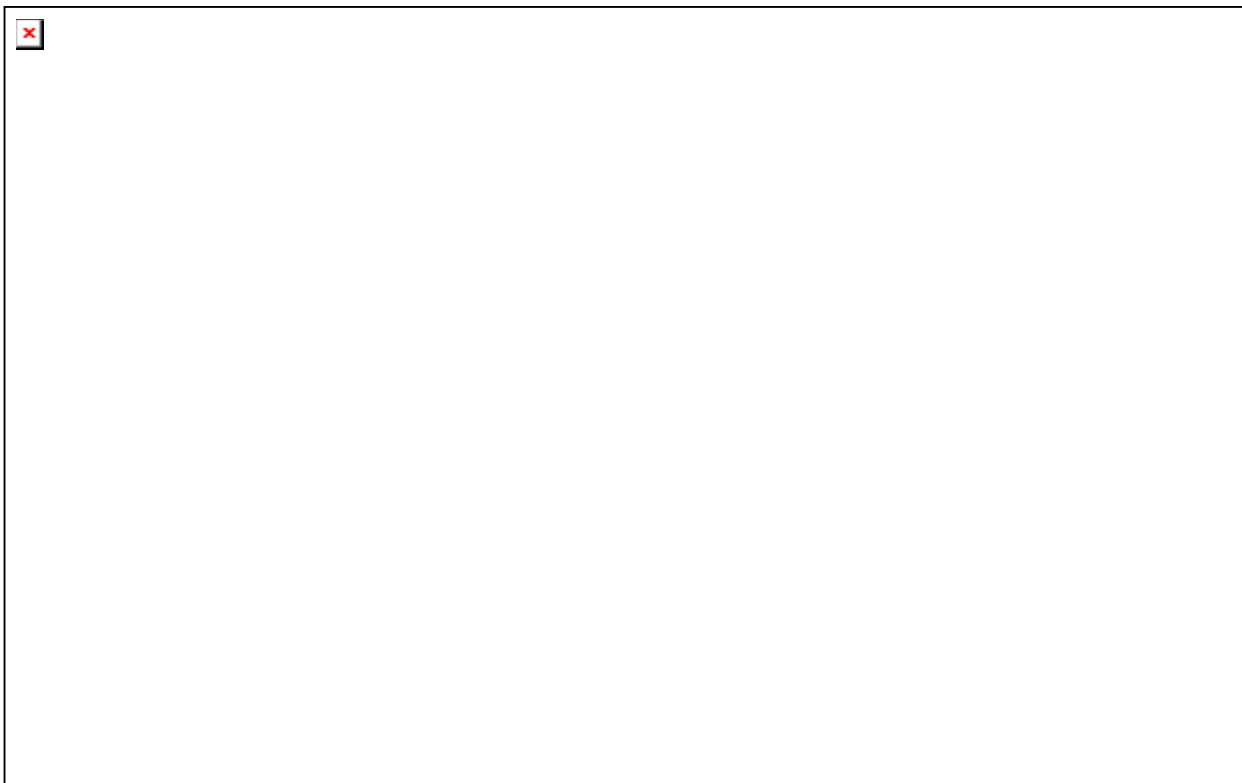


Рисунок 2.2 – Диаграмма деятельности варианта использования «Управление процессом анализа»

Для более детального анализа взаимодействий разработанных на этапе анализа предметной области сущностей удобно использовать диаграммы последовательности [17]. Так как при работе с анализатором следует учитывать, что время процесса анализа может быть довольно большим, будет разумным построить диаграмму последовательности для варианта использования «Управление процессом анализа». Данная диаграмма отображена на рисунке 2.3.

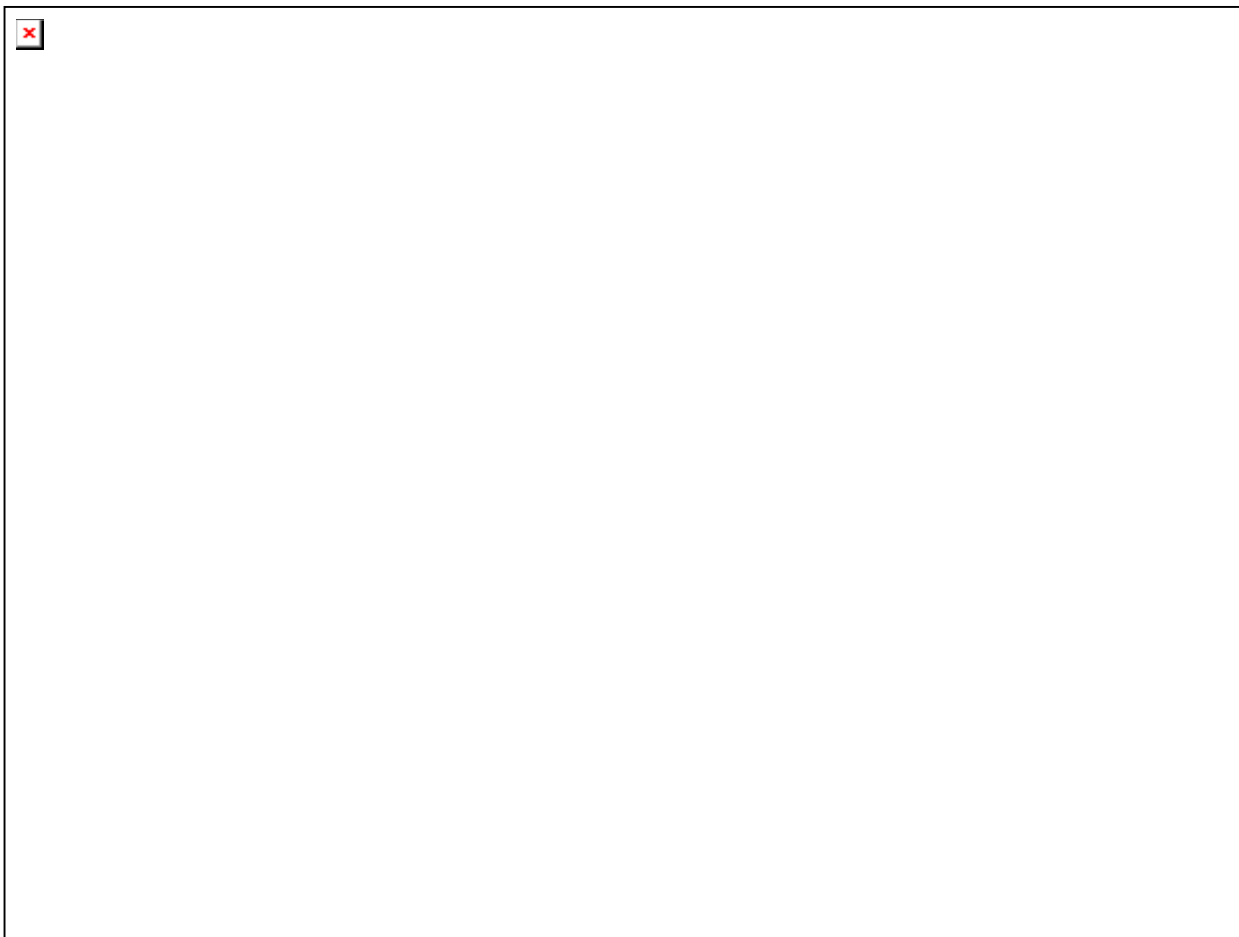


Рисунок 2.3 – Диаграмма последовательности варианта использования
«Управление процессом анализа»

Как видно на данной диаграмме (см. рисунок 2.3), процесс статического анализа является фоновым, т.е. анализатор должен ожидать команд пользователя в асинхронном режиме и только потом переходить к управлению процессом анализа. Сигналом к началу процесса служит команда запуска анализа. Имеет смысл сначала проверить список выбранных диагностик и доступность проверяемого проекта, чтобы заранее определить, сможет ли процесс анализа продолжаться. После проверки становится понятно, можно ли переходить к анализу или же нет.

2.3 Анализ нефункциональных требований

Основные нефункциональные требования предъявляются к производительности и составу аппаратной платформы, на которой

производится запуск анализатора. Таким образом, для функционирования анализатора, необходим персональный компьютер:

- работающий под управлением операционной системы, поддерживающей одну из настольных .NET платформ;
- имеющий процессор, работающий на частоте не менее 2 ГГц;
- имеющий не менее 4 Гб ОЗУ;
- подключенный к монитору с разрешением не менее 1024x768 пикселей.

Кроме того, к разрабатываемому анализатору предъявляются дополнительные требования:

- требования к производительности: анализатор должен быть максимально производительным приложением, по возможности распараллеливая процесс статического анализа;
- требования к переносимости: анализатор должен быть кроссплатформенным автономным приложением;
- требования к расширяемости: возможность добавлять дополнительные модули с диагностиками;
- требования к удобству использования: анализатор должен иметь дружелюбный к пользователю графический интерфейс;
- требования к конфигурируемости: анализатор должен предоставлять настройки, обеспечивающие кастомизацию списка диагностических правил и окна программы [20].

3 ИСПОЛЬЗУЕМЫЕ ТЕХНОЛОГИИ И СРЕДСТВА РАЗРАБОТКИ

3.1 Язык программирования

В качестве языка для разработки программы выбран C#, так как он является языком объектно-ориентированного программирования, позволяет проектировать сложные программные комплексы, при этом сохраняя оптимальную производительность. Кроме того, разрабатывая графические приложения именно с использованием языка C#, программисту не нужно заботиться о выделении/освобождении памяти и работе с указателями, а можно сосредоточиться на решении поставленной задачи [16].

3.2 Инструментальные средства проектирования

Для разработки моделей и диаграмм на этапе анализа функциональных требований и проектирования будет использован программный продукт Microsoft Visio. Данное ПО поддерживает стандарт UML 2.0, является простым в использовании и широко распространено. UML – язык графического описания для объектного моделирования в области разработки программного обеспечения, моделирования бизнес-процессов, системного проектирования и отображения организации структур [17]. Основное назначение UML – помочь разработчикам достигнуть соглашения в графических обозначениях для представления общих понятий (например, класс, компонент и т.д.) и больше сконцентрироваться на проектировании и архитектуре проекта.

3.3 Интегрированная среда разработки

В качестве среды разработки статического анализатора будет использована интегрированная среда разработки Microsoft Visual Studio 2017. Данная среда поддерживает командную разработку, включает в себя средство контроля версий (TFS), позволяет разрабатывать и поддерживать модульные тесты [18]. Таким образом, данная среда идеально подходит для

проведения полного спектра работ по реализации, тестированию и контролю качества разрабатываемого программного средства.

3.4 Программная платформа

Для достижения кроссплатформенности продукт следует разрабатывать на языке C# с возможностью запуска на платформах .NET Framework и Mono. Поэтому, в качестве технологии разработки были выбраны .NET-платформы, удовлетворяющие .NET Standard Library – формальному набору спецификаций общих интерфейсов других платформ: .NET Core, .NET Framework, Mono/Xamarin и остальных [19].

Библиотеки, удовлетворяющие спецификациям .NET Standard, могут быть использованы на различных платформах .NET. Фактически, он гарантирует возможность использования библиотек в разных средах исполнения и приносит универсальность в программную экосистему.

.NET Standard одной и той же версии покрывает идентичное множество кода этих платформ, что отражено на рисунке 3.1.

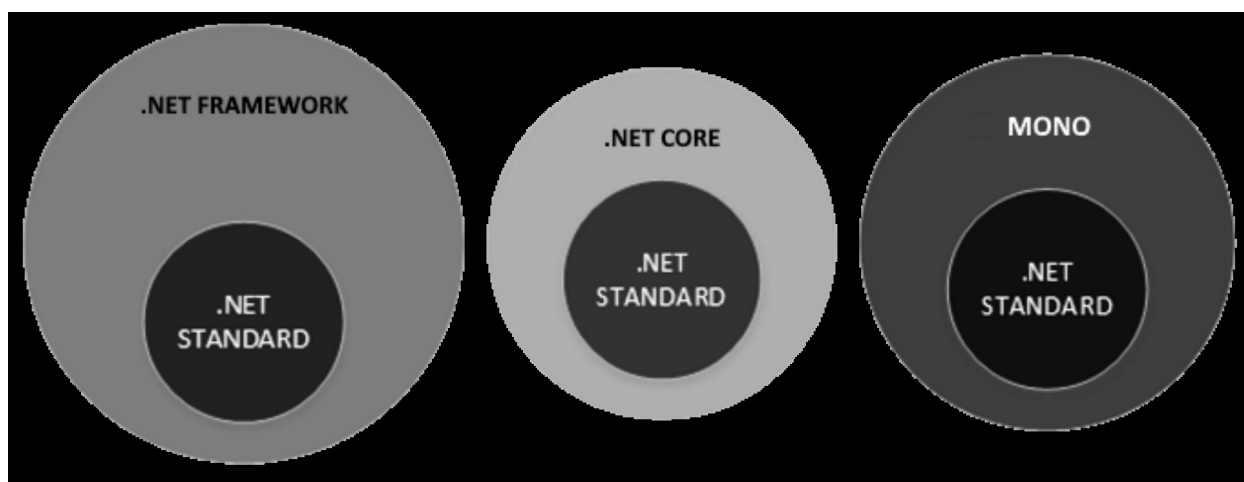


Рисунок 3.1 – Покрытие кода .NET платформ

Назначение .NET Framework – служить средой для поддержки разработки и выполнения сильно распределенных компонентных приложений [19]. Она обеспечивает совместное использование разных языков программирования, а также безопасность, переносимость программ и общую модель программирования для платформы Windows.

.NET Core является кроссплатформенным (Windows, Mac, Linux) аналогом .NET Framework, с открытым исходным кодом [19]. Он содержит среду CoreCLR – кроссплатформенную реализацию CLR, в которую входит JIT-компилятор – RyuJIT. Также .NET Core включает в себя CoreFX – набор библиотек, функциональность которых частично совпадает со стандартной библиотекой классов: API .NET Core предоставляет подмножество API .NET Framework, но также и предоставляет ряд собственных API.

Mono – программная платформа, позволяющая разработчикам достаточно просто создавать развитые кроссплатформенные приложения. Технология Mono разрабатывается в виде открытой реализации технологии .NET Framework от Microsoft и поддерживает стандарты ECMA-334 (стандарт языка C#) и ECMA-335 (стандарт среды исполнения (Common Language Runtime, CLI) [19]. Открытость указанных стандартов позволяет уменьшить проблемы создания кроссплатформенных приложений.

3.5 Средства создания графического интерфейса

3.5.1 Графические библиотеки

Технология WPF (Windows Presentation Foundation) является частью экосистемы платформы .NET и представляет собой подсистему для построения графических интерфейсов.

Фреймворк AvaloniaUI сделан по тем же принципам, что и WPF, т. е. используется XAML, привязка данных и шаблонизированные элементы управления. На текущий момент, использование AvaloniaUI – это единственный способ сделать кроссплатформенный графический интерфейс на настоящем XAML, используя шаблон проектирования MVVM, который будет работать на Windows, OS X и Linux.

3.5.2 Язык разметки XAML

XAML – разработанный Microsoft расширяемый декларативный язык разметки для приложений, основанный на XML. Декларативные языки позволяют описать состояние, которого нужно добиться, но не требуют

описать как прийти в это состояние. XAML файлы можно создавать и редактировать при помощи инструментов визуального конструирования, таких как: Microsoft Expression Blend, Microsoft Visual Studio. Также, их можно создавать при помощи стандартного текстового редактора кода.

3.5.3 Шаблон проектирования MVVM

При проектировании архитектуры графического приложения применяется шаблон проектирования MVVM, ориентированный на современные платформы разработки, такие как WPF и UWP от компании Microsoft и свободный кроссплатформенный фреймворк AvaloniaUI, которые основаны на применении языка разметки XAML.

Основные преимущества MVVM заключаются в увеличении таких показателей как:

- модульность;
- гибкость;
- тестируемость;
- поддерживаемость.

Используя шаблон проектирования MVVM, мы можем отвязать графическое представление от модельного. На рисунке 3.2 отображена схема взаимодействия компонентов этого шаблона:

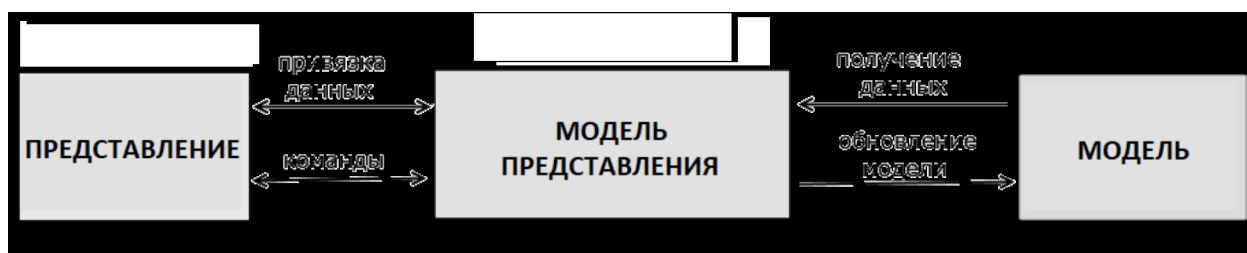


Рисунок 3.2 – Схема шаблона проектирования MVVM

При использовании этого паттерна, представление должно иметь ссылку на источник данных, которым в данном случае является модель представления. Элементы представления связаны с соответствующими свойствами и событиями модели представления. Под моделью понимается

часть, содержащая в себе функциональную бизнес-логику приложения. Модель должна быть полностью независима от остальных частей программы и не должна знать об элементах дизайна, и каким образом будут отображаться содержащиеся в ней данные. Достигается результат при помощи модели представления, в которой содержится вся логика построения графического интерфейса.

4 ПРОЕКТИРОВАНИЕ И РАЗРАБОТКА

4.1 Графический интерфейс пользователя

4.1.1 Диалог настройки списка правил

В результате анализа функциональных требований (см. табл. 2.1) было выявлено, что для прецедента «Выбор диагностик» требуется специализированный диалог, который должен реализовывать возможности:

- подключения сторонних модулей с правилами анализа;
- удаления сторонних модулей с правилами анализа;
- возможность выбора модулей для процесса анализа.

Макет для прототипа разрабатываемого диалога представлен на рисунке 4.1.

Назад Добавить модуль	
☒ Встроенный модуль правил	
☐ Сторонний модуль 1	X
☒ Сторонний модуль 2	X
☐ Сторонний модуль 3	X

Рисунок 4.1 – Макет диалога настройки списка правил

Код спроектированного макета написан на языке XAML и представлен на рисунке 4.2.

```

<Grid>
  <Grid.RowDefinitions>
    <RowDefinition Height="*" />
    <RowDefinition Height="40" />
  </Grid.RowDefinitions>
  <ListBox ItemsSource="{StaticResource RuleModules}" HorizontalContentAlignment="Stretch"
    ScrollViewer.CanContentScroll="False">
    <ListBox.ItemTemplate>
      <ItemContainerTemplate>
        <Grid Background="LightGray" >
          <Expander Header="{Binding ModuleName}">
            <ListBox ItemsSource="{Binding InnerRules}">
              <ListBox.ItemTemplate>
                <ItemContainerTemplate>
                  <CheckBox IsChecked="{Binding IsRuleEnabled}" Content="{Binding RuleName}" />
                </ItemContainerTemplate>
              </ListBox.ItemTemplate>
            </ListBox>
          </Expander>
          <Button Command="{Binding RemoveModuleCommand}"
            Visibility="{Binding IsInternal, Converter={StaticResource ConvertTrueToHidden}}"
            Width="20" Height="20" Margin="0" Content="x"
            HorizontalAlignment="Right" VerticalAlignment="Top" />
        </Grid>
      </ItemContainerTemplate>
    </ListBox.ItemTemplate>
  </ListBox>
  <StackPanel Grid.Row="1" Orientation="Horizontal" HorizontalAlignment="Center">
    <Button Margin="5" Height="23" Width="150" Command="{Binding LoadModuleCommand}">Добавить модуль</Button>
    <Button Margin="5" Height="23" Width="150" Command="{Binding CloseCommand}">OK</Button>
  </StackPanel>
</Grid>

```

Рисунок 4.2 – XAML-код диалога настройки списка правил

4.1.2 Диалоги выбора модулей и проектов

Для таких прецедентов как «Выбор анализируемого проекта» и «Добавление модулей с диагностиками» (см. рисунок 2.1) используются стандартный диалог открытия файла из фреймворка AvaloniaUI, настроенный на необходимые фильтры и расширения. Диалог выбора файла анализируемого проекта изображен на рисунке 4.3.

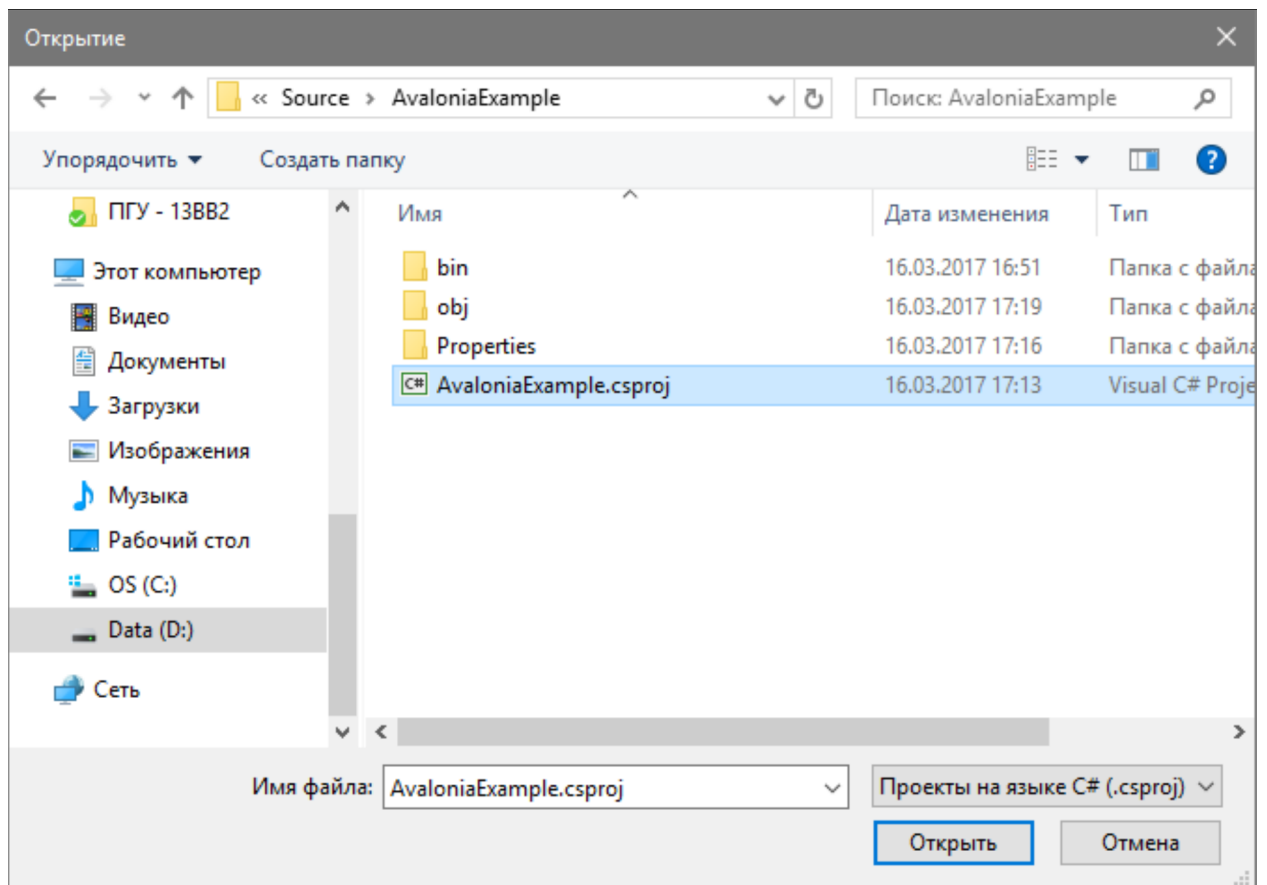


Рисунок 4.3 – Диалог выбора анализируемого проекта

Схематичный код для настройки и использования файлового диалога для выбора анализируемого проекта представлен на рисунке 4.4.

```

/// <summary>
/// Обработчик команды загрузки нового проекта.
/// </summary>
/// <param name="parameter">Неиспользуемый параметр.</param>
private void LoadProject(Object parameter)
{
    var openFileDialog = new OpenFileDialog()
    {
        AddExtension = true,
        CheckFileExists = true,
        DefaultExt = ".csproj",
        Filter = "Проекты на языке C# (.csproj)|*.csproj",
    };

    if (openFileDialog.ShowDialog() == true)
    {
        // TODO Загрузка проекта.
    }
}

```

Рисунок 4.4 – Использование файлового диалога для выбора анализируемого проекта

4.1.3 Панель управления процессом анализа

Для прецедента «Управление процессом анализа» (см. табл. 2.1) была спроектирована панель управления процессом анализа. Макет панели управления представлен на рисунке 4.5.

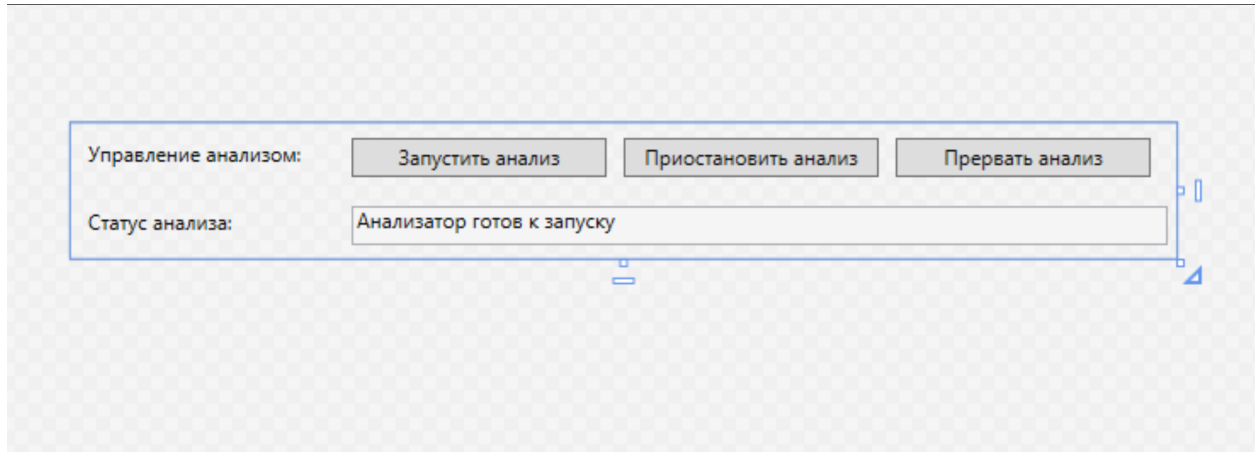


Рисунок 4.5 – Макет панели управления анализом

Код на языке XAML для макета панели управления анализом представлен в рисунке 4.6.

```
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition />
        <RowDefinition />
    </Grid.RowDefinitions>
    <StackPanel Orientation="Horizontal" Grid.Row="0">
        <Label Width="150" Margin="5" Height="30">Управление анализом:</Label>
        <Button Width="150" Margin="5" Height="23" Command="{Binding RunAnalysisCommand}">Запустить анализ</Button>
        <Button Width="150" Margin="5" Height="23" Command="{Binding PauseAnalysisCommand}">Приостановить анализ</Button>
        <Button Width="150" Margin="5" Height="23" Command="{Binding BreakAnalysisCommand}">Прервать анализ</Button>
    </StackPanel>
    <DockPanel Grid.Row="1">
        <Label Margin="5" Width="150" Height="30" DockPanel.Dock="Left">Статус анализа:</Label>
        <TextBox Margin="5" Height="23" IsReadOnly="True" Background="WhiteSmoke" Text="{Binding AnalysisStatus}" />
    </DockPanel>
</Grid>
```

Рисунок 4.6 – XAML-код макета панели управления анализом

Панель управления процессом анализа содержит кнопки, привязанные к соответствующим асинхронным командам анализатора, и отображает прогресс процесса анализа с помощью статусной строки.

4.1.4 Отображение отчёта анализатора

В процессе статического анализа анализатор добавляет найденные предупреждения в отчёт, который может быть отображён в специальном

списке. Макет списка предупреждений статического анализатора представлен на рисунке 4.7, код на языке XAML представлен на рисунке 4.8.

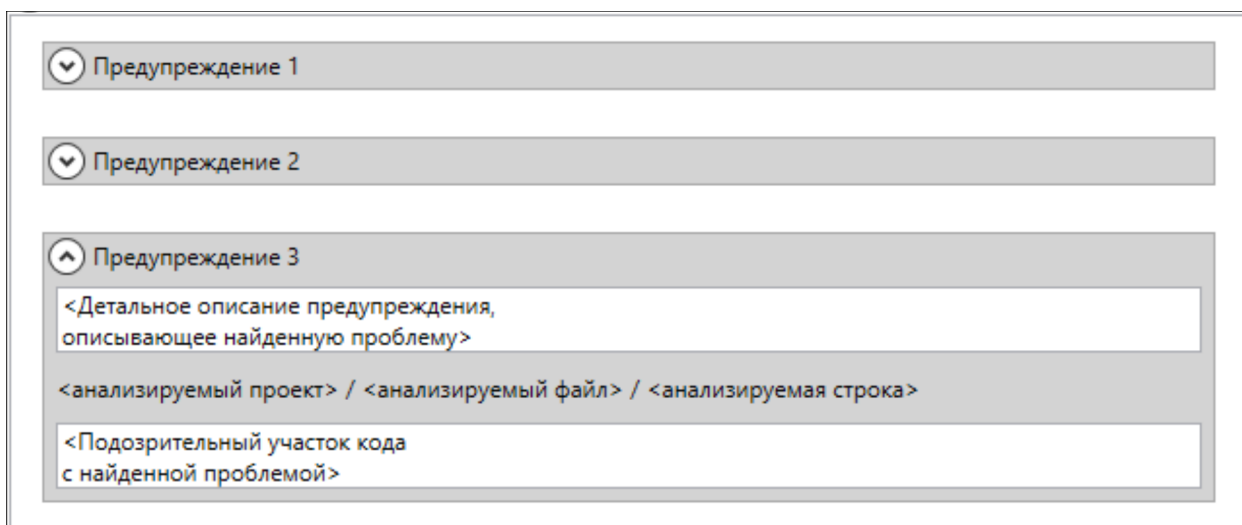


Рисунок 4.7 – Макет списка предупреждений анализатора

```
<Grid>
  <ListBox Grid.Row="1" ItemsSource="{StaticResource Warnings}" HorizontalContentAlignment="Stretch"
    ScrollViewer.CanContentScroll="False">
    <ListBox.ItemTemplate>
      <ItemContainerTemplate>
        <Border Background="LightGray" BorderBrush="DarkGray" BorderThickness="1" Margin="10">
          <Expander Header="{Binding WarningName}">
            <StackPanel Orientation="Vertical">
              <TextBox IsReadOnly="True" Text="{Binding WarningDescription, Mode=OneWay}" Margin="5"/>
              <TextBlock Text="{Binding TargetPath, Mode=OneWay}" Margin="5"/>
              <TextBox IsReadOnly="True" Text="{Binding SuspiciousCode, Mode=OneWay}" Margin="5"/>
            </StackPanel>
          </Expander>
        </Border>
      </ItemContainerTemplate>
    </ListBox.ItemTemplate>
  </ListBox>
</Grid>
```

Рисунок 4.8 – XAML-код макета списка предупреждений анализатора

4.1.5 Диаграмма классов интерфейса

На рисунке 4.9 представлена общая спроектированная диаграмма классов интерфейса, включая классы диалога настройки списка правил, панели управления процессом анализа и классы списка предупреждений анализатора.

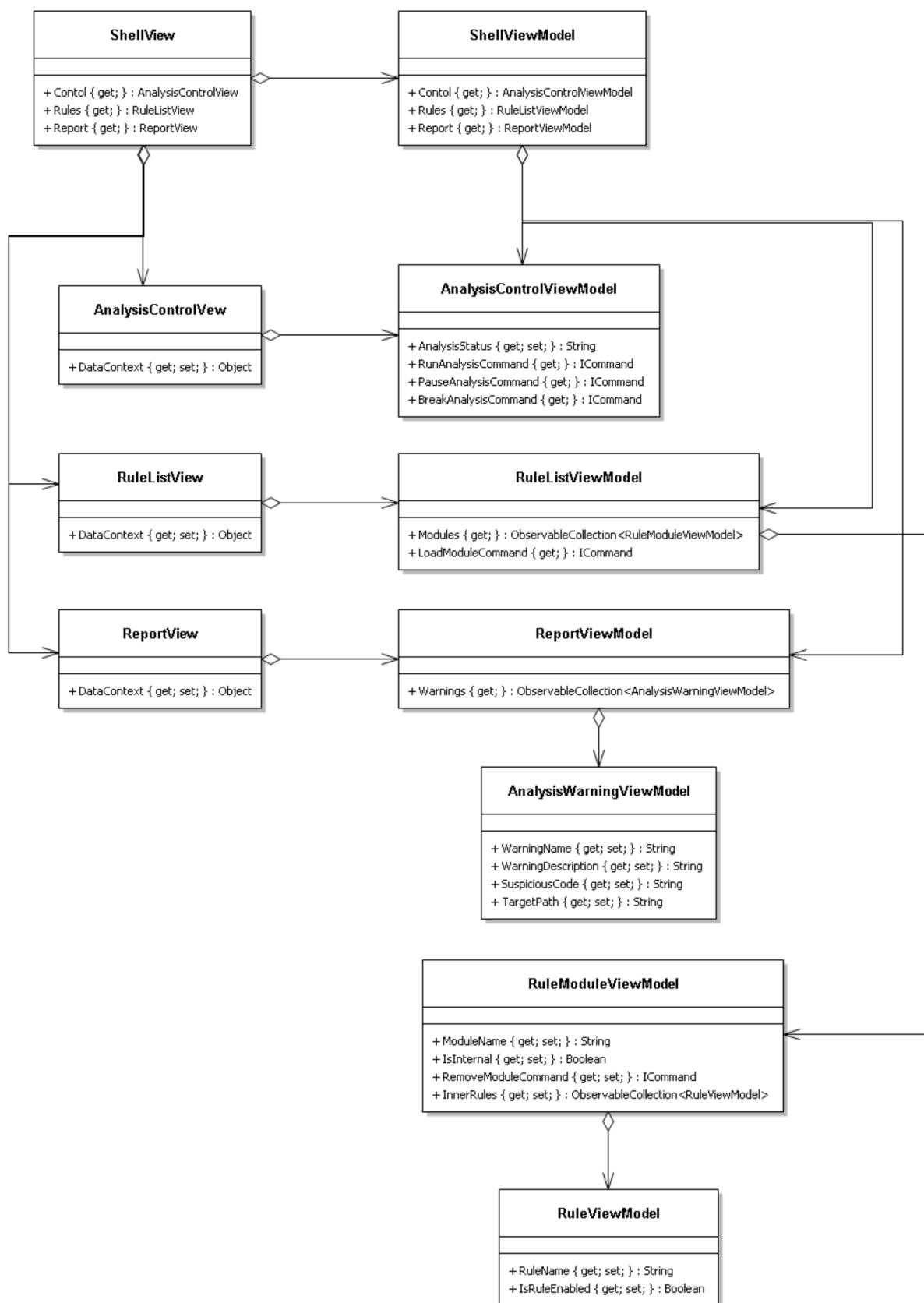


Рисунок 4.9 – Диаграмма классов интерфейса пользователя

4.2 Статический анализатор

Для того, чтобы выполнить статический анализ, необходимо получить список файлов, для которых будет проверяться исходный код, синтаксическое дерево для каждого файла, а также информацию о типах, необходимую для корректного анализа. В таблице 4.1 приведены описания классов, необходимых для анализа.

Таблица 4.1 – Классы, необходимые для статического анализа

Класс	Описание
Solution	Решение, содержащее проекты на языке C#
Project	Проект, содержащий файлы на языке C#
Document	Файл исходного кода на языке C#
Compilation	Компиляция, используется для получения семантической модели
SyntaxTree	Абстрактное синтаксическое дерево
SemanticModel	Информация о типах
SyntaxNode	Абстрактный класс, потомки которого представляют собой узлы синтаксического дерева

Для процесса статического анализа необходим рекурсивный обход абстрактного синтаксического дерева. Для выполнения такого обхода наиболее пригодна архитектура анализатора, реализующая шаблон проектирования «посетитель» [21]. Диаграмма классов этого шаблона представлена на рисунке 4.10.

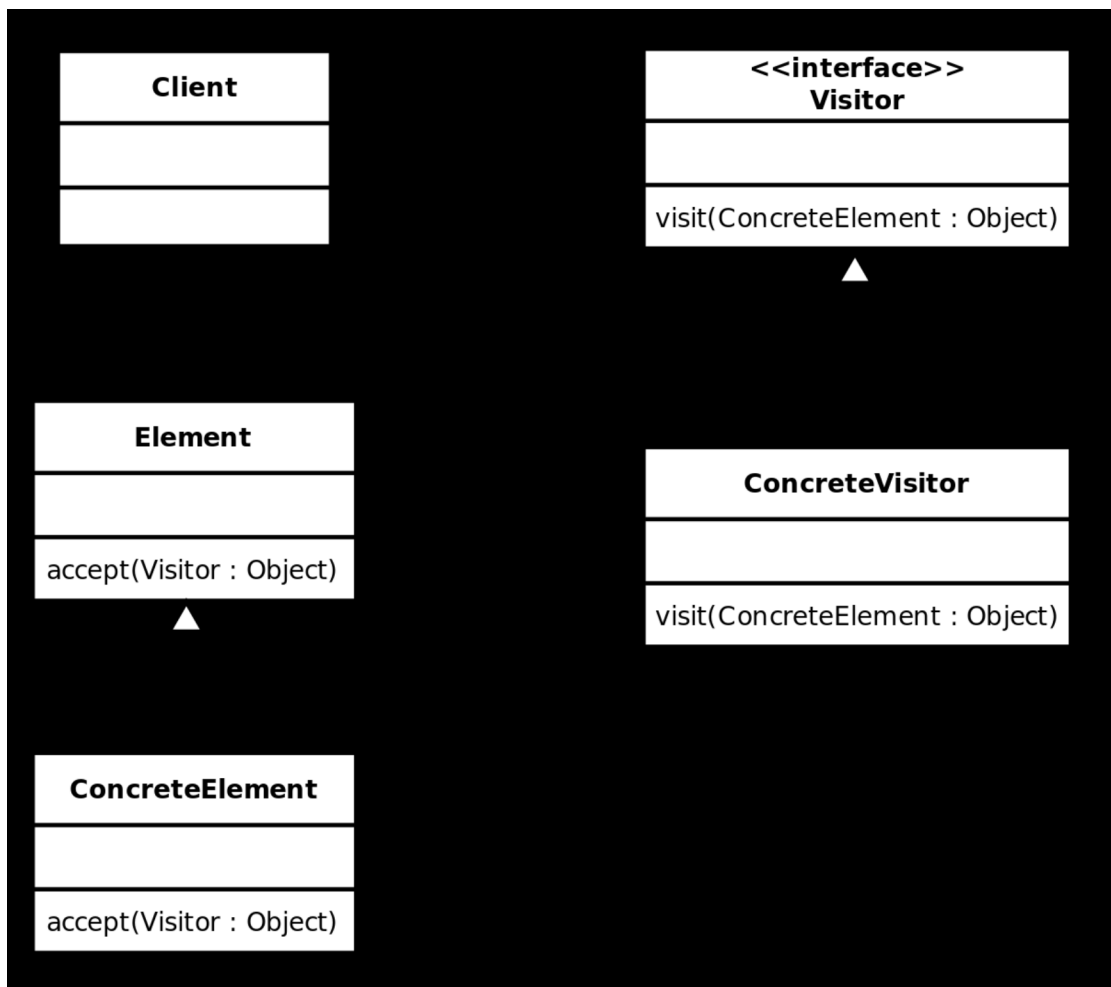


Рисунок 4.10 – Диаграмма классов шаблона проектирования «посетитель»

«Посетитель» относится к поведенческим паттернам проектирования. Эти паттерны решают задачи эффективного и безопасного взаимодействия между объектами программы. «Посетитель» обходит связанную структуру: один объект за другим, и, в зависимости от типа посещённого объекта, выполняет над ним какое-то действие. Такое поведение приносит следующие преимущества:

- упрощение добавления новых операций над всей связанной структурой объектов;
- объединение родственных операции в одном классе;
- «посетитель» может накапливать состояние при обходе структуры.

Описания классов шаблона проектирования посетитель представлены в таблице 4.2.

Таблица 4.2 – Описание классов шаблона проектирования «посетитель»

Класс	Описание
Visitor	Посетитель объявляет общий интерфейс для всех типов посетителей. Здесь должны быть объявлены методы посещения, по одному на каждый класс конкретных компонентов
ConcreteVisitor	Конкретный посетитель реализует все операции, объявленные в интерфейсе посетителя. Каждый конкретный посетитель реализует какое-то особенное поведение над компонентами.
Element	Элемент определяет метод получения посетителя. Этот метод должен принимать в параметрах любой из объектов посетителей.
ConcreteElement	Конкретный компонент реализует метод получения посетителя, который принимает объект посетителя как аргумент, вызывает его метод посещения, соответствующий классу компонента, передавая туда ссылку на самого себя.

Каждый узел, представляющий ту или иную конструкцию языка, имеет свой тип, определяющий ряд свойств, упрощающих навигацию по дереву и получение необходимых данных. Класс-посетитель, который будет обходить синтаксическое дерево и проводить анализ, должен определять для каждого типа узла метод, выполняющий обход узлов подобного типа и вызывающий

обработчики (диагностические правила) позволяющие анализировать только интересные нас конструкции языка.

Проектирование архитектуры программы заключается в разработке детализированной диаграммы классов. Исходя из анализа предметной области, была спроектирована диаграмма классов, которая представлена на рисунке 4.11.

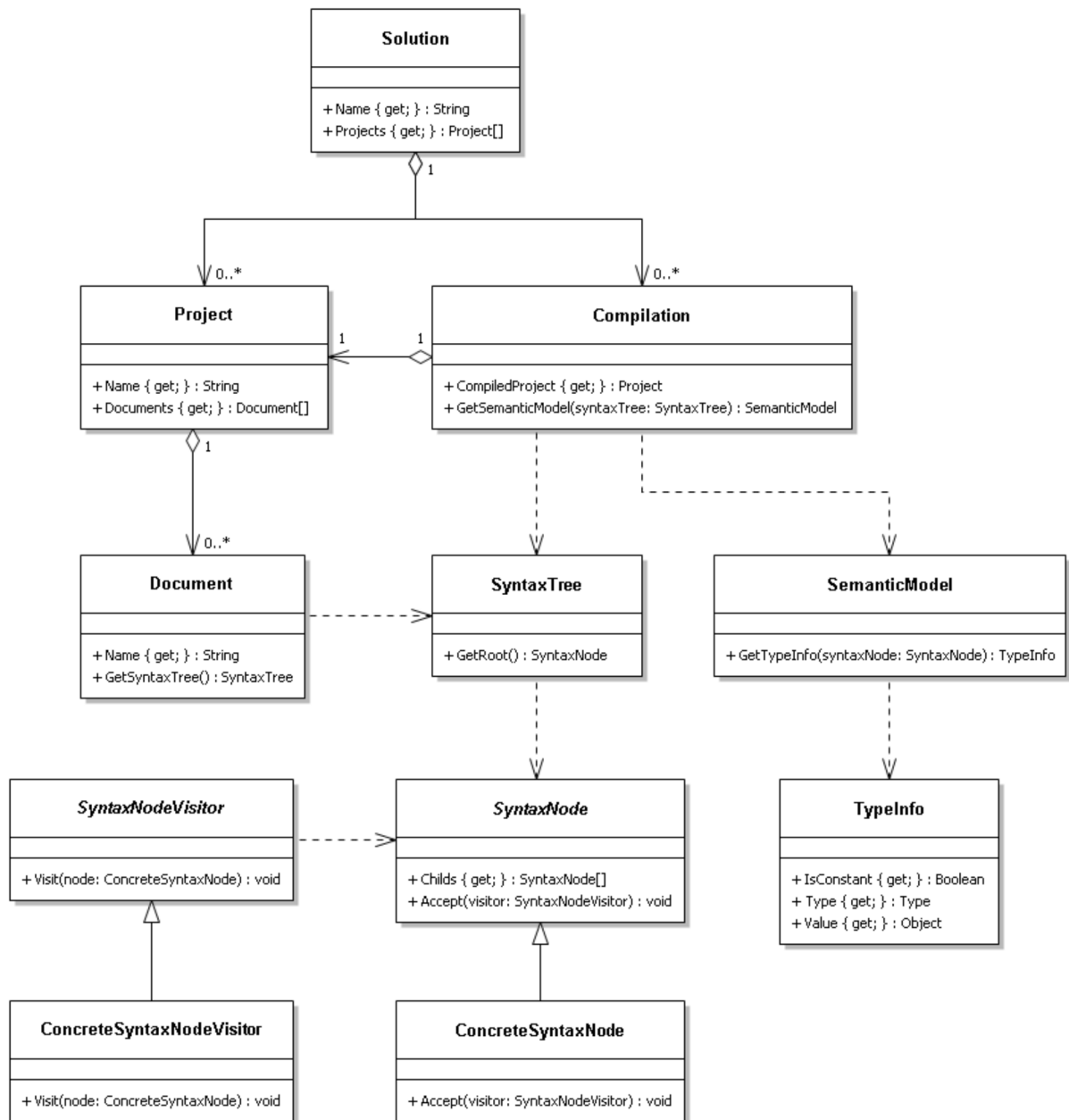


Рисунок 4.11 – Диаграмма классов архитектуры анализатора

4.3 Разработка диагностических правил

Поиск ошибок, осуществляемый статическим анализатором, реализуется за счёт различных правил. Для реализации всех правил существует набор общих действий, выделив которые, можно получить общий алгоритм написания диагностики.

Таблица 4.3 – Алгоритм разработки правил анализа

Номер этапа	Название этапа	Описание
1	Формулировка	Необходимо сформулировать суть правила: необходимо обдумать, на какие фрагменты кода стоит выдавать предупреждения, а на какие – нет.
2	Юнит-тесты	Реализация наборов позитивных и негативных тестов. На позитивных тестах должна срабатывать диагностика, а база негативных юнит-тестов будет снижать количество ложных срабатываний.
3	Реализация	Реализация диагностики на языке программирования.
4	Тестирование на реальных проектах	Тестирование диагностического правила на реальных проектах позволит обнаружить ложные срабатывания (а может и падения) и расширить базу юнит-тестов.
5	Доработка	После тестирования на реальных проектах скорее всего придётся дорабатывать диагностику. После внесения необходимых изменений правило снова тестируется.

Для реализации диагностических правил необходимы несколько ключевых классов, представляющих контекст анализа и контейнер реализации правил.

Контекст анализа должен хранить и накапливать состояние при рекурсивном обходе анализатором абстрактного синтаксического дерева. В

его задачу входит регистрация обработчиков, выполняющихся в методах обхода тех или иных узлов синтаксического дерева, при помощи которых мы можем анализировать только интересующие нас конструкции языка. Он должен быть способен при соответствующем запросе подгрузить часть семантической модели, ответственной за текущий узел синтаксического дерева, что позволит нам узнать семантическую информацию об анализируемых объектах и их типах, а также получить константные значения для константных полей и литералов. Также контекст должен предоставлять возможность публикации диагностического сообщения в отчёт анализатора.

Были реализованы два ключевых класса, совокупность которых представляет контекст анализа:

- `AnalysisContext` – позволяет зарегистрировать обработчики для методов обхода конструкций языка;
- `AnalysisActionContext` – абстрактный класс, наследники которого реализуются для каждой из конструкций языка, хранят в себе соответствующую синтаксическую и семантическую информацию и предоставляют возможность публикации диагностических сообщений.

Контейнер реализации правил должен хранить в себе обработчики для методов обхода интересующих нас (для конкретной диагностики) конструкций языка и предоставлять возможность корректно зарегистрировать их в контексте анализа. По сути, контейнер реализации должен является абстрактным классом, наследование которого – необходимое и достаточное условие для реализации конкретного диагностического правила.

Для удовлетворения этих требований был реализован абстрактный класс `DiagnosticAnalyzer`, имеющий абстрактный метод `Initialize`, определив который в классах-наследниках, мы можем зарегистрировать необходимые нам обработчики. Также был реализован класс `Diagnostic`, представляющий

диагностическое сообщение, которое можно опубликовать в контексте анализа при достижении условий, задаваемых в наших обработчиках.

Диаграмма классов, представляющих контекст анализа и контейнер реализации диагностического правила, представлена на рисунке 4.12.

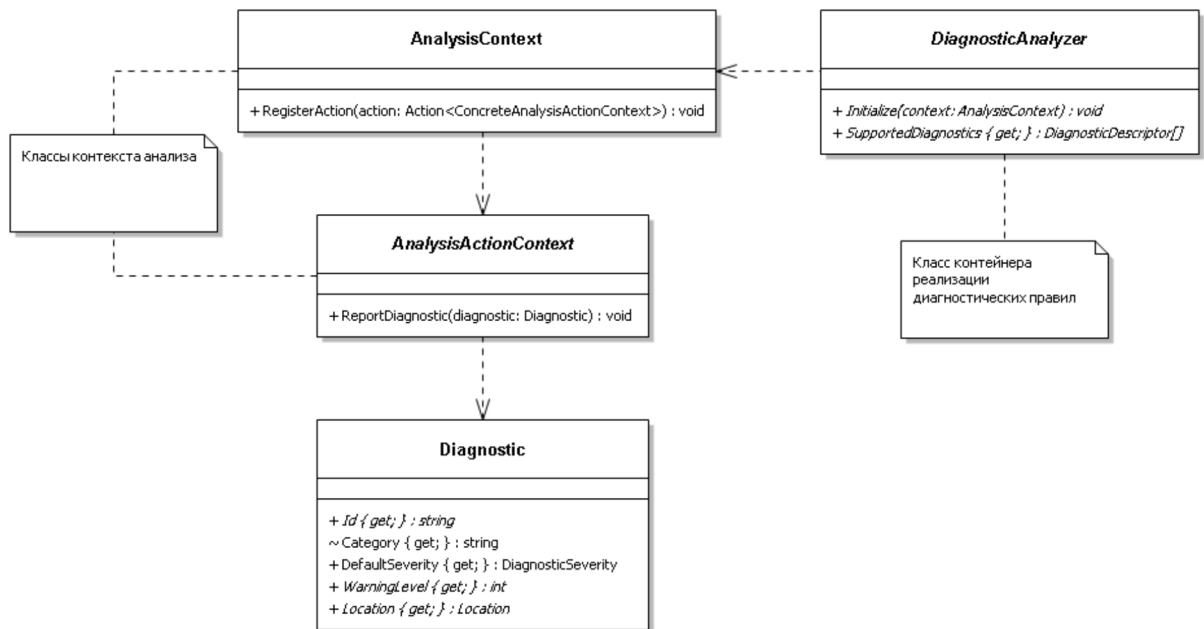


Рисунок 4.12 – Диаграмма классов, необходимых для реализации диагностических правил

Рассмотрим пример реализации простейшего диагностического правила на основе реализации конкретного наследника класса `DiagnosticAnalyzer`. Наша цель – вывести ошибку, которая появляется при записи недопустимого шаблона регулярного выражения в статическом методе `Regex.Match`. Этот метод возвращает первую подстроку, соответствующую шаблону регулярного выражения во входной строке.

Пример строки, которая вызывает `Regex.Match` с недопустимым шаблоном регулярного выражения:

```
Regex.Match("my text", @"pXXX");
```

Первым аргументом является проверяемая строка, вторым – шаблон регулярного выражения, который необходимо проверить. Наследуем класс `DiagnosticAnalyzer` и реализуем необходимую функциональность:

```
public class RegexPatternArgumentAnalyzer : DiagnosticAnalyzer
{
    /// <summary>
    /// Метаданные для диагностического правила.
    /// </summary>
    internal static DiagnosticDescriptor s_rule = new DiagnosticDescriptor(
        "Regex",
        "Ошибка разбора строкового аргумента",
        "Ошибка в регулярном выражении: {0} ",
        "StaticAnalysis.Syntax",
        DiagnosticSeverity.Error
    );

    /// <summary>
    /// Список реализуемых диагностик.
    /// </summary>
    public override ImmutableArray<DiagnosticDescriptor> SupportedDiagnostics
    {
        get { return ImmutableArray.Create(s_rule); }
    }

    /// <summary>
    /// Инициализация и регистрация обработчиков.
    /// </summary>
    /// <param name="context">Контекст анализа.</param>
    public override void Initialize(AnalysisContext context)
    {
        context.RegisterSyntaxNodeAction(
            AnalyzeNode, SyntaxKind.InvocationExpression);
    }

    /// <summary>
    /// Обработчик, проверяющий паттерн регулярного выражения
    /// на корректность.
    /// </summary>
    /// <param name="context">Контекст анализа.</param>
    private void AnalyzeNode(SyntaxNodeAnalysisContext context)
    {
        var invocationExpr = (InvocationExpressionSyntax)context.Node;

        // поиск нужного узла
        var regexLiteral = invocationExpr.Arguments[1].Expression
            as LiteralExpressionSyntax;
        if (regexLiteral == null) { return; }

        // получение константного выражения аргумента,
        // используя семантическую модель
        var regexOpt = context.SemanticModel.GetConstantValue(regexLiteral);
        if (!regexOpt.HasValue) { return; }

        var regex = regexOpt.Value as String;
        if (regex == null) { return; }

        try
```

```

    {
        // проверка выражения на корректность
        System.Text.RegularExpressions.Regex.Match("", regex);
    }
    catch (ArgumentException e)
    {
        // публикация диагностического сообщения
        var diagnostic = Diagnostic.Create(
            s_rule, regexLiteral.GetLocation(), e.Message);
        context.ReportDiagnostic(diagnostic);
    }
}
}

```

Класс `RegexPatternArgumentAnalyzer` является наследником класса `DiagnosticAnalyzer`. В определяемом методе `Initialize` происходит регистрация обработчика `AnalyzeNode` для обхода конструкций языка, ответственных за вызов методов. В обработчике реализован код поиска нужного элемента для анализа (в нашем случае второго аргумента метода `Regex.Match`). Далее, используя семантическую модель, идет получение константного значения анализируемого элемента и непосредственная проверка на корректность. Если проверка показывает, что анализируемый элемент некорректен, происходит публикация диагностического сообщения в контекст анализа.

5 ФУНКЦИОНАЛЬНОЕ ТЕСТИРОВАНИЕ

По итогам разработки было проведено функциональное тестирование программы. План тестирования приведён в таблице 5.1.

Таблица 5.1 – Функциональное тестирование

Номер теста	Название теста	Описание	Результат выполнения
1	Проверка работы программы	Запускаем программу	Программа запускается; пользователю показывается начальная форма с предложением настроить правила анализа и выбрать объект для анализа (см. рисунок 5.1)
2	Настройка правил анализа	Открываем диалог настройки правил анализа при помощи кнопки «Настройка правил»	Пользователю отображается диалог со списком модулей с правилами анализа; пользователь имеет возможности для добавления нового модуля и отключения или удаления уже существующих в списке (см. рисунок 5.2)
3	Проверка открытия решения для анализа	Открываем файл с расширением .sln при помощи кнопки «Выбрать»	Программа переходит в состояние готовности анализа проектов, входящих в решение; пользователю показывается форма управления процессом анализа. (см. рисунок 5.3)

Продолжение таблицы 5.1

Номер теста	Название теста	Описание	Результат выполнения
4	Проверка открытия проекта для анализа	Открываем файл с расширением .csproj при помощи кнопки «Выбрать»	Программа переходит в состояние готовности анализа проекта; пользователю показывается форма управления процессом анализа.
5	Проверка открытия документа для анализа	Открываем файл с расширением .cs при помощи кнопки «Выбрать»	Программа переходит в состояние готовности анализа C#-файла; пользователю показывается форма управления процессом анализа.
6	Запуск анализа	При помощи кнопки «Запустить» инициируем процесс статического анализа для выбранного объекта	Программа переходит в состояние процесса анализа; пользователю становится доступна кнопка «Прервать»
7	Прерывание процесса анализа	При помощи кнопки «Прервать» останавливаем процесс анализа	Пользователю становится доступна кнопка «Просмотр отчёта»
8	Завершение процесса анализа	Дожидаемся завершения процесса анализа	Пользователю становится доступна кнопка «Просмотр отчёта»
9	Просмотр отчёта	Нажимаем кнопку «Просмотр отчёта»	Пользователю отображается диалог со списком предупреждений для ошибок, обнаруженных анализатором (см. рисунок 5.4)

Далее представлены скриншоты программы, упомянутые на различных этапах функционального тестирования в таблице 5.1.

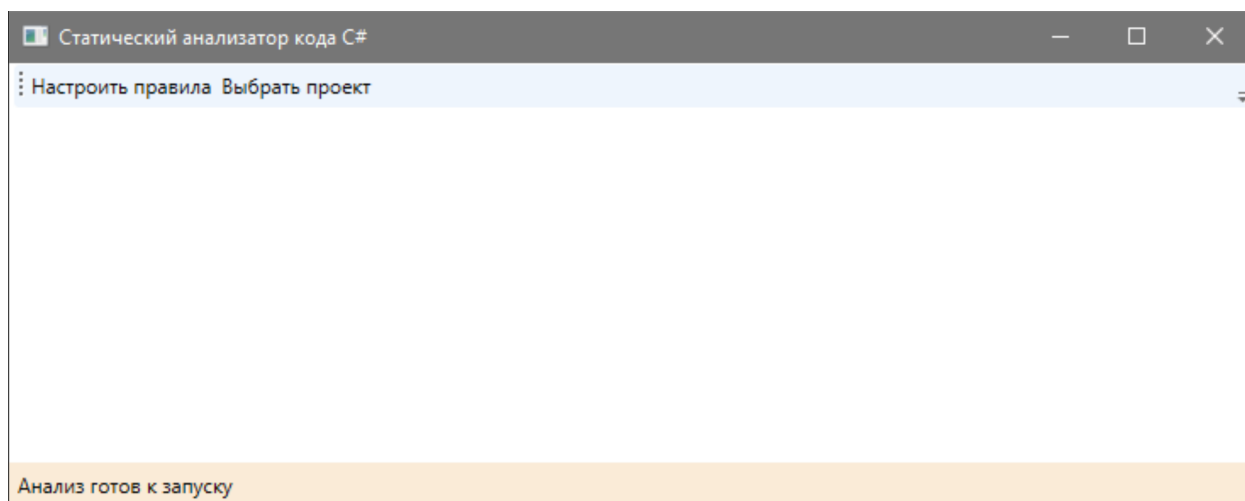


Рисунок 5.1 – Начальная форма программы



Рисунок 5.2 – Диалог настроек правил анализа

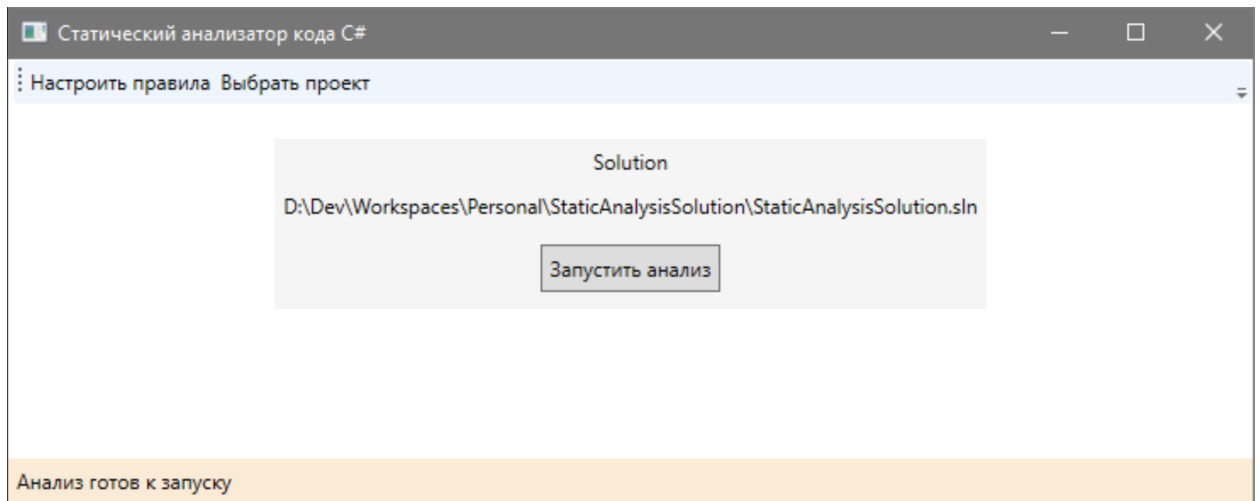


Рисунок 5.3 – Форма управления процессом анализа

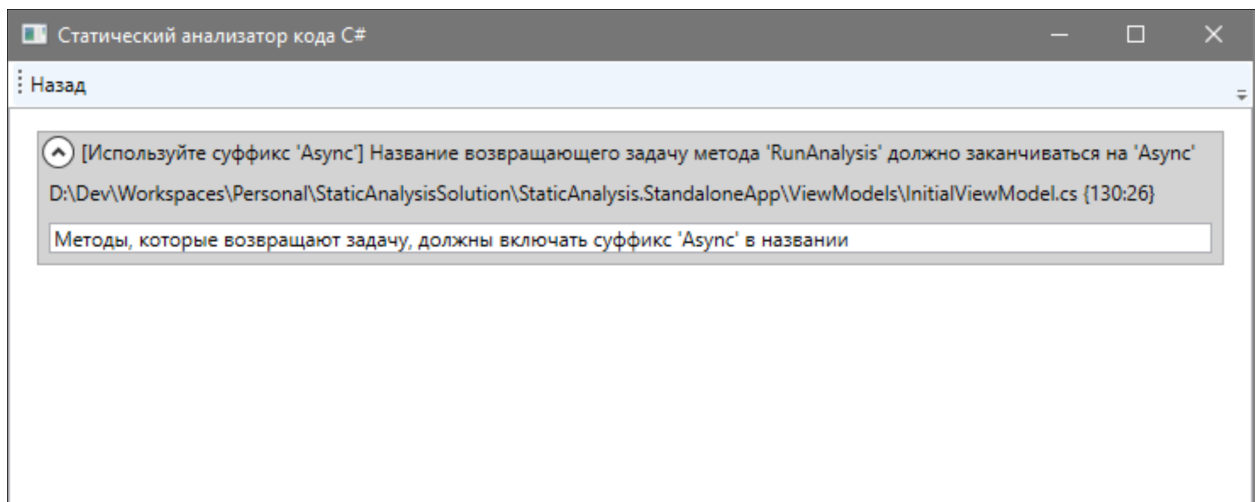


Рисунок 5.4 – Диалог просмотра отчёта анализатора

При помощи юнит-тестов, были проведены проверки корректного срабатывания диагностик на типовых фрагментах исходного кода. **Некоторые** из них будут приведены ниже.

Далее приведён фрагмент кода, на котором срабатывает диагностика, следящая за тем, чтобы такие ресурсы как потоки, файлы и дескрипторы были обязательно освобождены при помощи реализации интерфейса `IDisposable`, что поможет избежать утечек памяти:

```
public class ResourceHolder
{
    private FileStream fs; // здесь предупреждение:
                          // Dispose или Close никогда не вызываются
}
```

```

public void OpenResource(string path)
{
    this.fs = new FileStream(path, FileMode.Open);
}

public void WriteToFile(string path, string text)
{
    var fs = new FileStream(path, FileMode.Open); // файловый поток
                                                    // не закрывается
    var bytes = Encoding.UTF8.GetBytes(text);
    fs.Write(bytes, 0, bytes.Length);
}
}

```

Далее приведён фрагмент кода, на котором срабатывает диагностика, проверяющая, чтобы указатели и дескрипторы использовались только безопасным способом, что поможет избежать утечек памяти и повысит качество кода:

```

static void Main(string[] args)
{
    System.Reflection.FieldInfo fieldInfo = ...;
    SafeHandle handle = (SafeHandle)fieldInfo.GetValue(rKey);
    IntPtr dangerousHandle = handle.DangerousGetHandle(); // предупреждение
}

```

Далее приведён фрагмент кода, на котором срабатывает диагностика, проверяющая, чтобы рекурсия обязательно имела условия выхода, что поможет программисту не упустить ошибку в реализации различных алгоритмов:

```

int Pow(int num, int exponent) // предупреждение
{
    num = num * Pow(num, exponent-1);
    return num; // рекурсия никогда не прерывается
}

void InternalRecursion(int i)
{
    start:
        goto end;
    end:
        goto start; // предупреждение
}

```

Далее приведен фрагмент кода, на котором срабатывает диагностика, проверяющая, чтобы в конструкторах класса не вызывались виртуальные

члены, которые могут быть перегружены в наследниках этого класса, так как это может привести к неопределенному поведению:

```
public class Parent
{
    public Parent()
    {
        DoSomething(); // предупреждение
    }

    public virtual void DoSomething() // метод может быть перегружен
    {
        // ...
    }
}
```

Далее приведен фрагмент кода, на который срабатывает диагностика, проверяющая, чтобы операторы перехода использовались только совместно с условными конструкциями, что должно помочь программисту с правильной реализацией различных алгоритмов:

```
for (var i = 0; i < 10; i++)
{
    Console.WriteLine(i);
    break; // цикл выполняется только однажды
}

for (var i = 0; i < 10; i++)
{
    continue;
    Console.WriteLine(i); // это никогда не выполнится
}

for (var i = 0; i < 10; i++)
{
    Console.WriteLine(i);
    continue; // эта конструкция не нужна
}
```

Далее приведен фрагмент кода, на который срабатывает диагностика, проверяющая сохранность стека вызовов при повторном создании исключения, что помогает сохранить качество кода и простоту отладки:

```
try
{}
catch (ExceptionType1 exc)
```

```

{
    Console.WriteLine(exc);
    throw exc; // предупреждение; stacktrace сброшен
}
catch (ExceptionType2 exc)
{
    throw new Exception("My custom message", exc); // корректно
}

```

Далее приведен фрагмент кода, на котором срабатывает диагностика, проверяющая использование нулевых ссылок:

```

object o = null;
if (condition)
{
    M1(o.ToString()); // предупреждение, всегда null
}
else
{
    o = new object();
}
M2(o.ToString());

```

Все разработанные диагностики корректно отработали на этих и других фрагментах кода, предложенных юнит-тестами, что показывает их правильную и качественную реализацию.

ЗАКЛЮЧЕНИЕ

В ходе выполнения выпускной квалификационной работы были проанализированы статические анализаторы кода на языке программирования C#. В результате анализа выяснилось, что у подобных решений присутствует ряд недостатков.

Был разработан кроссплатформенный статический анализатор кода C#, отвечающий требованиям технического задания, в котором были устранены недостатки аналогичных решений и реализованы все функциональные требования.

Выполненное функциональное тестирование разработанного статического анализатора показало, что программа полностью работоспособна и доказало корректность её работы.

Разработанную программу следует применять в процессе современной разработки и тестирования приложений на платформе .NET для поиска ошибок в программном коде на языке программирования C#.

СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ

1. Глухих, М.И. Программная инженерия: Обеспечение качества программных средств методами статического анализа [Текст] / М.И. Глухих, В.М. Ицыксон. – СПб: Изд. Политехн. ун-та, 2011. – 149 с.
2. Методы анализа и обеспечения качества ПО [Электронный ресурс] / Кафедра компьютерных систем и программных технологий СПбПУ; Ицыксон В.М. – Режим доступа: <http://kspt.icc.spbstu.ru/course/QA>, свободный. (дата обращения 24.03.2017 г.).
3. Индекс ТЮВЕ: Популярность языка программирования C# [Электронный ресурс]. – Режим доступа: <https://www.tiobe.com/tiobe-index>, свободный. (дата обращения 25.03.2017 г.).
4. Простая ошибка при кодировании – не значит нестрашная ошибка [Электронный ресурс] / Блог разработчиков PVS-Studio. – Режим доступа: <https://habrahabr.ru/company/pvs-studio/blog/326846>, свободный. (дата обращения 26.03.2017 г.).
5. Статический анализ кода [Электронный ресурс] / Словари и энциклопедии на Академике. – Режим доступа: <http://dic.academic.ru/dic.nsf/ruwiki/129701>, свободный. (дата обращения 22.02.2017).
6. Статический анализ кода [Электронный ресурс] / Intel Corp. – Режим доступа: <https://software.intel.com/ru-ru/articles/Static-code-analysis-ru>, свободный. (дата обращения 22.02.2017).
7. Разрушаем мифы о статическом анализе кода [Электронный ресурс] / Блог разработчиков PVS-Studio. – Режим доступа: <https://habrahabr.ru/company/pvs-studio/blog/131551>, свободный. (дата обращения 26.03.2017 г.).
8. ReSharper – расширение для Microsoft Visual Studio [Электронный ресурс] / JetBrains: Профессиональные инструменты разработки программного

- обеспечения. – Режим доступа: <http://jetbrains.ru/products/resharper/>, свободный. (дата обращения 27.03.2017 г.).
9. FxCop – инструмент для проверки кода [Электронный ресурс] / Microsoft Corp. – Режим доступа: [https://msdn.microsoft.com/ru-ru/library/bb429476\(v=vs.80\).aspx](https://msdn.microsoft.com/ru-ru/library/bb429476(v=vs.80).aspx), свободный. (дата обращения 27.03.2017 г.).
10. Coverity – инструмент для статического анализа кода [Электронный ресурс] / Synopsys, Inc – Режим доступа: <https://www.synopsys.com/software-integrity/security-testing/static-analysis-sast.html>, свободный. (дата обращения 27.03.2017 г.).
11. PVS-Studio: статический анализатор кода для C, C++ и C# [Электронный ресурс] / Компания PVS-Studio – Режим доступа: <https://www.viva64.com/ru/pvs-studio>, свободный. (дата обращения 27.03.2017 г.).
12. Кроссплатформенные приложения против нативных: сравнение и выбор подходов [Электронный ресурс]. – Режим доступа: <https://vc.ru/p/cross-native>, свободный. (дата обращения 29.03.2017 г.).
13. Microsoft открывает исходный код .NET и делает его кроссплатформенным [Электронный ресурс]. – Режим доступа: <https://techcrunch.com/2014/11/12/microsoft-takes-net-open-source-and-cross-platform>, свободный. (дата обращения 29.03.2017 г.).
14. Существует ли 64-битная версия Visual Studio? Ограничения 32-битной версии и затраты на портирование [Электронный ресурс]. – Режим доступа: <https://blogs.msdn.com/ricom/2009/06/10/visual-studio-why-is-there-no-64-bit-version-yet/>, свободный. (дата обращения 29.03.2017 г.).
15. Введение в Roslyn. Использование для разработки инструментов статического анализа [Электронный ресурс]. – Режим доступа: <https://www.viva64.com/en/b/0399/>, свободный. (дата обращения 29.03.2017 г.).

- 16.Рихтер, Д. CLR via C#. Программирование на платформе Microsoft.NET Framework 4.5 на языке C# [Текст] / Д. Рихтер. – СПб:Питер, 2017. – 896 с.
- 17.Гради, Б. Объектно-ориентированный анализ и проектирование с примерами приложений [Текст] / Б. Гради – СПб: Питер, 2012. – 720 с.
- 18.IDE Visual Studio [Электронный ресурс]. – Режим доступа: <https://www.visualstudio.com/ru/>, свободный. (дата обращения 30.03.2017 г.).
- 19.Введение в .NET Core [Электронный ресурс]. / Блог компании Microsoft – Режим доступа: <https://habrahabr.ru/company/microsoft/blog/245901>, свободный. (дата обращения 30.03.2017 г.).
- 20.Нефункциональные требования к программному обеспечению [Электронный ресурс]. – Режим доступа: <https://habrahabr.ru/post/231961>, свободный (дата обращения 31.03.2017 г.).
- 21.Приемы объектно-ориентированного проектирования [Текст] / Э. Гамма, Р. Хелм, Р. Джонсон, Дж. Влссидес – СПб: Питер, 2016 – 366 с.;

ПРИЛОЖЕНИЕ

Листинг программы

Analyzer.cs

```
using System;
using System.Collections.Generic;
using System.Collections.Immutable;
using System.Diagnostics;
using System.IO;
using System.Linq;
using System.Reflection;
using System.Threading;
using System.Threading.Tasks;
using Core.CodeAnalysis;
using Core.CodeAnalysis.CSharp;
using Core.CodeAnalysis.Diagnostics;
using Core.CodeAnalysis.MSBuild;
using Newtonsoft.Json;

namespace StaticAnalysis.Core
{
    public partial class Analyzer
    {
        #region Поля

        private readonly Dictionary<String, StyleViolation> _styleViolations;
        private readonly CompilationWithAnalyzersOptions _options;
        private readonly Dictionary<String, AnalyzerModule> _modules = new
Dictionary<String, AnalyzerModule>();
        private IEnumerable<DiagnosticAnalyzer> _diagnostics =>
            _modules.Values.Where(m => m.IsEnabled).SelectMany(m =>
m.Diagnostics);

        #endregion Поля

        public Analyzer(IEnumerable<AnalyzerModuleDefinition>
moduleDefinitions)
        {
            _styleViolations = ReadStyleViolations().ToDictionary(v => v.Id);

            _options = new CompilationWithAnalyzersOptions(
                new AnalyzerOptions(default(ImmutableArray<AdditionalText>)),
                OnAnalyzerException,
                false,
                false
            );

            AddModuleInternal(typeof(Internal.Rules.ModuleUnit).Assembly).IsInternal =
true;

            AddModuleInternal(typeof(Internal.AsyncUsage.ModuleUnit).Assembly).IsInternal
= true;

            Initialize(moduleDefinitions);
        }

        public Analyzer Initialize(IEnumerable<AnalyzerModuleDefinition>
moduleDefinitions)
        {
```

```

        foreach (var module in Modules.Where(m => !m.IsInternal))
        {
            RemoveModule(module.ModuleName);
        }

        foreach (var moduleDef in moduleDefinitions)
        {
            AddModule(moduleDef);
        }

        return this;
    }

    #region Модули

    public IEnumerable<AnalyzerModule> Modules =>
    _modules.Values.ToArray();

    public void AddModule(AnalyzerModuleDefinition definition)
    {
        var moduleAssembly = Assembly.LoadFile(definition.ModulePath);

        AddModuleInternal(moduleAssembly);
    }

    internal AnalyzerModule AddModuleInternal(Assembly moduleAssembly)
    {
        var analyzerModule = new AnalyzerModule(moduleAssembly);

        _modules.Add(analyzerModule.ModuleName, analyzerModule);

        return analyzerModule;
    }

    public void RemoveModule(String moduleName)
    {
        if (_modules.TryGetValue(moduleName, out var module))
        {
            if (!module.IsInternal)
            {
                _modules.Remove(module.ModuleName);
            }
        }
    }

    #endregion Модули

    #region Запуск анализа

    public async Task<SolutionReport> AnalyzeSolutionAsync(String
solutionPath, CancellationToken cancellationToken)
    {
        if (String.IsNullOrEmpty(solutionPath))
        {
            throw new ArgumentNullException(nameof(solutionPath));
        }

        if (!File.Exists(solutionPath))
        {
            throw new FileNotFoundException();
        }
    }

```

```

        var workspace = MSBuildWorkspace.Create();
        workspace.WorkspaceFailed += (o, e) =>
        {
            Debug.WriteLine(e.Diagnostic.Message);
        };

        var currentSolution = await
workspace.OpenSolutionAsync(solutionPath, cancellationTokens);

        return await AnalyzeSolutionInternalAsync(currentSolution,
cancellationTokens);
    }

    internal async Task<SolutionReport>
AnalyzeSolutionInternalAsync(Solution solution, CancellationTokens
cancellationTokens)
    {
        var projectReports = new List<ProjectReport>();

        foreach (var project in solution.Projects)
        {
            projectReports.Add(await AnalyzeProjectInternalAsync(project,
cancellationTokens));
        }

        return new SolutionReport(solution.FilePath, projectReports);
    }

    public async Task<ProjectReport> AnalyzeProjectAsync(String
projectPath, CancellationTokens cancellationTokens)
    {
        if (String.IsNullOrEmpty(projectPath))
        {
            throw new ArgumentNullException(nameof(projectPath));
        }

        if (!File.Exists(projectPath))
        {
            throw new FileNotFoundException();
        }

        var workspace = MSBuildWorkspace.Create();
        var project = await workspace.OpenProjectAsync(projectPath,
cancellationTokens);

        return await AnalyzeProjectInternalAsync(project,
cancellationTokens);
    }

    internal async Task<ProjectReport>
AnalyzeProjectInternalAsync(Project project, CancellationTokens
cancellationTokens)
    {
        var compilation = await
project.GetCompilationAsync(cancellationTokens);
        var diagnostics = await AnalyzeInternalAsync(compilation,
cancellationTokens);
        var issues = diagnostics.Select(CreateIssue);

        return new ProjectReport(project.Name, issues);
    }

```

```

        public async Task<DocumentReport> AnalyzeDocumentAsync(String
filePath, CancellationToken cancellationToken)
        {
            if (String.IsNullOrEmpty(filePath))
            {
                throw new ArgumentNullException(nameof(filePath));
            }

            if (!File.Exists(filePath))
            {
                throw new FileNotFoundException();
            }

            var csharpSource = File.ReadAllText(filePath);
            var compilation = await
GetCompilationForSingleDocumentAsync(filePath, csharpSource,
cancellationToken);
            var diagnostics = await AnalyzeInternalAsync(compilation,
cancellationToken);
            var issues = diagnostics.Select(CreateIssue);

            return new DocumentReport(issues);
        }

        internal async Task<IEnumerable<Diagnostic>>
AnalyzeInternalAsync(Compilation compilation, CancellationToken
cancellationToken)
        {
            var parseDiagnostics = compilation.GetParseDiagnostics();

            if (_diagnostics.Any())
            {
                var compilationWithAnalyzers = new
CompilationWithAnalyzers(compilation, _diagnostics.ToImmutableArray(),
_options);
                var analyzerDiagnostics = await
compilationWithAnalyzers.GetAnalyzerDiagnosticsAsync(cancellationToken);

                return
parseDiagnostics.Concat(analyzerDiagnostics).ToImmutableArray();
            }

            return parseDiagnostics;
        }

        /// <summary>
        /// Функция обратного вызова для статического анализатора.
        /// Вызывается при исключении, произошедшем при выполнении одной из
диагностик.
        /// </summary>
        /// <param name="exception">Исключение, произошедшее при
анализе.</param>
        /// <param name="analyzer">Анализатор со сбоем.</param>
        /// <param name="diagnostic">Собранная диагностика.</param>
        private void OnAnalyzerException(Exception exception,
DiagnosticAnalyzer analyzer, Diagnostic diagnostic)
        {
            // Отправить слушателям отладочного вывода.
            // И ничего не делать.
            Debug.WriteLine(exception);
            Debug.Flush();
        }

```

```

#endregion Запуск анализа

#region УТИЛИТЫ

/// <summary>
/// Получить компиляцию для C#-документа.
/// </summary>
/// <param name="filePath">Путь к файлу.</param>
/// <param name="csharpSource">Исходный код документа.</param>
/// <param name="cancellationToken">Токен отмены асинхронной
операции.</param>
/// <returns></returns>
private async Task<Compilation>
GetCompilationForSingleDocumentAsync(String filePath, String csharpSource,
CancellationTokens cancellationToken)
{
    if (String.IsNullOrEmpty(filePath))
    {
        throw new ArgumentNullException(nameof(filePath));
    }

    if (String.IsNullOrEmpty(csharpSource))
    {
        throw new ArgumentException("C#-документ пуст.",
nameof(csharpSource));
    }

    return await Task.Factory.StartNew(() =>
    {
        SyntaxTree tree = CSharpSyntaxTree.ParseText(csharpSource,
path: filePath);

        Compilation compilation = CSharpCompilation.Create("target")
.AddReferences(MetadataReference.CreateFromFile(typeof(Object).Assembly.Location))
.AddSyntaxTrees(tree);

        return compilation;
    },
cancellationToken);
}

/// <summary>
/// Создать предупреждение из результата диагностики.
/// </summary>
private Issue CreateIssue(Diagnostic diagnostic)
{
    var file = diagnostic.Location.SourceTree.FilePath;
    var lineSpan = diagnostic.Location.GetLineSpan();
    var line = lineSpan.StartLinePosition.Line;
    var column = lineSpan.StartLinePosition.Character;

    var id = diagnostic.Id;
    var title = diagnostic.Descriptor.Title.ToString();
    var category = diagnostic.Descriptor.Category;
    var message = diagnostic.GetMessage();
    var description = diagnostic.Descriptor.Description.ToString();

    var severity = Severity.Info;

```

```

        switch (diagnostic.DefaultSeverity)
        {
            case DiagnosticSeverity.Hidden:
                severity = Severity.Hidden;
                break;
            case DiagnosticSeverity.Info:
                severity = Severity.Info;
                break;
            case DiagnosticSeverity.Warning:
                severity = Severity.Warning;
                break;
            case DiagnosticSeverity.Error:
                severity = Severity.Error;
                break;
        }

        if (_styleViolations.ContainsKey(id))
        {
            severity = _styleViolations[id].Severity;
        }

        return new Issue(category, id, title, message, description,
severity, file, line, column);
    }

    /// <summary>
    /// Прочитать стилевые настройки для предупреждений.
    /// </summary>
    /// <returns></returns>
    private StyleViolation[] ReadStyleViolations()
    {
        try
        {
            String violationsJson =
File.ReadAllText(GetPathToFile(@"violations.json"));

            return
JsonConvert.DeserializeObject<StyleViolation[]>(violationsJson);
        }
        catch (FileNotFoundException)
        {
            return new StyleViolation[0];
        }
    }

    /// <summary>
    /// Получить абсолютный путь к файлу.
    /// </summary>
    /// <param name="file">Относительный путь к файлу.</param>
    /// <returns></returns>
    private String GetPathToFile(String file)
    {
        String exeFilePath = Assembly.GetExecutingAssembly().Location;
        String exeFolder = Path.GetDirectoryName(exeFilePath);

        return Path.Combine(exeFolder, file);
    }

    #endregion Утилиты
}
}

```

AnalyzerModule.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Reflection;
using Core.CodeAnalysis.Diagnostics;

namespace StaticAnalysis.Core
{
    public class AnalyzerModule
    {
        internal AnalyzerModule(Assembly assembly)
        {
            Assembly = assembly;
            IsEnabled = true;
            Diagnostics = assembly.GetTypes()
                .Where(t => t.IsAbstract == false &&
typeof(DiagnosticAnalyzer).IsAssignableFrom(t))
                .Select(t => Activator.CreateInstance(t) as
DiagnosticAnalyzer)
                .Where(t => t != null).ToArray();
        }

        public Boolean IsInternal { get; internal set; }

        public Boolean IsEnabled { get; set; }

        public String ModuleName => Assembly.GetName().Name;

        internal Assembly Assembly { get; }

        internal IEnumerable<DiagnosticAnalyzer> Diagnostics { get; }
    }
}
```

Issue.cs

```
using System;

namespace StaticAnalysis.Core
{
    public class Issue
    {
        internal Issue(
            String category, String id,
            String title, String message, String description,
            Severity severity,
            String filePath, Int32 line, Int32 column
        )
        {
            Category = category;
            Id = id;
            Title = title;
            Message = message;
            Description = description;
            Severity = severity;
            FilePath = filePath;
            LineNumber = line;
            Column = column;
        }

        public String Category { get; }
    }
}
```

```

        public String Id { get; }

        public String Title { get; }

        public String Message { get; }

        public String Description { get; }

        public Severity Severity { get; }

        public String ProjectName { get; internal set; }

        public String FilePath { get; }

        public Int32 LineNumber { get; }

        public Int32 Column { get; }
    }
}

```

ReportBase.cs

```

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

namespace StaticAnalysis.Core
{
    public abstract class ReportBase : IEnumerable<Issue>
    {
        public abstract IEnumerable<Issue> Issues { get; }

        public IEnumerator<Issue> GetEnumerator() => Issues.GetEnumerator();

        IEnumerator IEnumerable.GetEnumerator() => Issues.GetEnumerator();
    }
}

```

Severity.cs

```

namespace StaticAnalysis.Core
{
    public enum Severity
    {
        Error = 3,
        Warning = 2,
        Info = 1,
        Hidden = 0
    }
}

```

InitialViewModel.cs

```

using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Threading;
using System.Threading.Tasks;
using Caliburn.Micro;
using Microsoft.Win32;
using StaticAnalysis.Core;

namespace StaticAnalysis.StandaloneApp.ViewModels
{

```

```

/// <summary>
/// Модель представления, показываемого при старте программы.
/// </summary>
public class InitialViewModel : Screen
{
    private String _status;
    private Boolean _competed;
    private Boolean _running;
    private IEnumerable<Issue> _issues;

    /// <summary>
    /// Конструктор <see cref="ShellViewModel"/>.
    /// </summary>
    public InitialViewModel(ShellViewModel shell)
    {
        Shell = shell;
        Status = "Анализ готов к запуску";
    }

    /// <summary>
    /// Ссылка на главную модель представления.
    /// </summary>
    public ShellViewModel Shell { get; }

    /// <summary>
    /// Путь до объекта анализа.
    /// </summary>
    public String AnalysisItem { get; private set; }

    /// <summary>
    /// Тип объекта анализа.
    /// </summary>
    public AnalysisItemType AnalysisItemType
    {
        get
        {
            if (String.IsNullOrEmpty(AnalysisItem))
            {
                return AnalysisItemType.Unknown;
            }

            if (AnalysisItem.EndsWith(".sln"))
            {
                return AnalysisItemType.Solution;
            }

            if (AnalysisItem.EndsWith(".csproj"))
            {
                return AnalysisItemType.Project;
            }

            if (AnalysisItem.EndsWith(".cs"))
            {
                return AnalysisItemType.Document;
            }

            return AnalysisItemType.Unknown;
        }
    }

    /// <summary>
    /// Готов ли объект анализа.

```

```

    /// </summary>
    public Boolean IsAnalysisReady
    {
        get
        {
            return !String.IsNullOrEmpty(AnalysisItem)
                && File.Exists(AnalysisItem)
                && AnalysisItemType != AnalysisItemType.Unknown;
        }
    }

    public Boolean CanRunAnalysis => Shell.Initial.IsAnalysisReady &&
!IsAnalysisRunning;

    /// <summary>
    /// Статус анализатора.
    /// </summary>
    public String Status
    {
        get => _status;
        private set
        {
            _status = value;
            NotifyOfPropertyChanged();
        }
    }

    public Boolean IsAnalysisCompeted
    {
        get => _competed;
        private set
        {
            _competed = value;
            NotifyOfPropertyChanged();
        }
    }

    public Boolean IsAnalysisRunning
    {
        get => _running;
        private set
        {
            _running = value;
            NotifyOfPropertyChanged();
        }
    }

    public CancellationTokenSource Cancellor { get; private set; }

    public void Exit()
    {
        Shell.DeactivateItem(this, true);
    }

    /// <summary>
    /// Команда запуска анализа.
    /// </summary>
    public async Task RunAnalysis()
    {
        try
        {
            Cancellor = new CancellationTokenSource();

```

```

        Status = "Анализ запущен";
        IsAnalysisCompeted = false;
        IsAnalysisRunning = true;
        NotifyOfPropertyChanged(() => CanRunAnalysis);

        _issues = Enumerable.Empty<Issue>();

        switch (Shell.Initial.AnalysisItemType)
        {
            case AnalysisItemType.Solution:
                _issues = await
Shell.Analyzer.AnalyzeSolutionAsync(Shell.Initial.AnalysisItem,
CancellationToken);
                break;
            case AnalysisItemType.Project:
                _issues = await
Shell.Analyzer.AnalyzeProjectAsync(Shell.Initial.AnalysisItem,
CancellationToken);
                break;
            case AnalysisItemType.Document:
                _issues = await
Shell.Analyzer.AnalyzeDocumentAsync(Shell.Initial.AnalysisItem,
CancellationToken);
                break;
        }

        IsAnalysisCompeted = true;
        Status = "Анализ завершен";
    }
    catch (AggregateException ex)
    {
        throw ex.InnerException;
    }
    catch (OperationCanceledException)
    {
        Status = "Анализ был отменен";
    }
    catch (Exception ex)
    {
        Status = $"Анализ завершился с ошибкой '{ex.Message}';
    }
    finally
    {
        IsAnalysisRunning = false;
        NotifyOfPropertyChanged(() => CanRunAnalysis);
    }
}

/// <summary>
/// Команда прерывания анализа.
/// </summary>
public void BreakAnalysis()
{
    Cancellation.Cancel(true);
}

public void GotoReport()
{
    Shell.Report.Initialize(_issues);
    Shell.ActivateItem(Shell.Report);
}

```

```

    /// <summary>
    /// Обработчик команды загрузки нового проекта.
    /// </summary>
    public void LoadProject()
    {
        var openFileDialog = new OpenFileDialog()
        {
            AddExtension = true,
            CheckFileExists = true,
            Filter = "Файлы исходных кодов C# |*.sln; *.csproj; *.cs"
        };

        if (openFileDialog.ShowDialog() == true)
        {
            AnalysisItem = openFileDialog.FileName;
            NotifyOfPropertyChanged(() => AnalysisItem);
            NotifyOfPropertyChanged(() => AnalysisItemType);
            NotifyOfPropertyChanged(() => IsAnalysisReady);
            NotifyOfPropertyChanged(() => CanRunAnalysis);
            IsAnalysisCompleted = false;
        }
    }

    /// <summary>
    /// Обработчик команды настройки списка диагностических правил.
    /// </summary>
    public void SelectModules()
    {
        Shell.ActivateItem(Shell.ModuleList);
    }
}

```

IssueViewModel.cs

```

using System;
using System.IO;
using Caliburn.Micro;
using StaticAnalysis.Core;

namespace StaticAnalysis.StandaloneApp.ViewModels
{
    /// <summary>
    /// Модель представления предупреждения анализатора.
    /// </summary>
    public class IssueViewModel : PropertyChangedBase
    {
        private Issue _issue;

        /// <summary>
        /// Конструктор <see cref="IssueViewModel"/>.
        /// </summary>
        /// <param name="issue">Ссылка на предупреждение анализатора.</param>
        public IssueViewModel(Issue issue)
        {
            _issue = issue;
        }

        /// <summary>
        /// Категория диагностики.
        /// </summary>
        public String Category => _issue.Category;
    }
}

```

```

    /// <summary>
    /// Идентификатор диагностики.
    /// </summary>
    public String Id => _issue.Id;

    /// <summary>
    /// Сообщение диагностики.
    /// </summary>
    public String Message => _issue.Message;

    /// <summary>
    /// Название диагностики.
    /// </summary>
    public String Title => _issue.Title;

    /// <summary>
    /// Описание диагностики.
    /// </summary>
    public String Description => _issue.Description;

    /// <summary>
    /// Имя файла с обнаруженной проблемой.
    /// </summary>
    public String FilePath => _issue.FilePath;

    /// <summary>
    /// Имя проекта с обнаруженной проблемой.
    /// </summary>
    public String ProjectName => _issue.ProjectName;

    /// <summary>
    /// Уровень проблемы.
    /// </summary>
    public Severity Severity => _issue.Severity;

    /// <summary>
    /// Номер строки.
    /// </summary>
    public Int32 LineNumber => _issue.LineNumber;

    /// <summary>
    /// Номер символа в строке.
    /// </summary>
    public Int32 Column => _issue.Column;

    /// <summary>
    /// Название диагностики вместе с идентификатором.
    /// </summary>
    public String FullName => $"{ Title} { Message} ";

    /// <summary>
    /// Позиция в файле.
    /// </summary>
    public String Location => $"{ FilePath} {{{ LineNumber} :{ Column}}} ";
}

```

ModuleListViewModel.cs

```

using System;
using System.Collections.Generic;
using System.Linq;
using Caliburn.Micro;

```

```

using Microsoft.Win32;

namespace StaticAnalysis.StandaloneApp.ViewModels
{
    /// <summary>
    /// Модель представления списка модулей с диагностическими правилами.
    /// </summary>
    public class ModuleListViewModel : Screen
    {
        /// <summary>
        /// Конструктор <see cref="ModuleListViewModel"/>.
        /// </summary>
        public ModuleListViewModel(ShellViewModel shell)
        {
            Shell = shell;
        }

        /// <summary>
        /// Ссылка на главную модель представления.
        /// </summary>
        public ShellViewModel Shell { get; }

        /// <summary>
        /// Перечисление модулей.
        /// </summary>
        public IEnumerable<ModuleViewModel> Modules =>
            Shell.Analyzer.Modules.Select(m => new ModuleViewModel(m));

        /// <summary>
        /// Обработчик команды загрузки нового модуля.
        /// </summary>
        public void LoadModule()
        {
            var openFileDialog = new OpenFileDialog()
            {
                AddExtension = true,
                CheckFileExists = true,
                DefaultExt = ".dll",
                Filter = "Модули с правилами анализа |*.dll",
            };

            if (openFileDialog.ShowDialog() == true)
            {
                Shell.Analyzer.AddModule(new
                    Core.AnalyzerModuleDefinition(openFileDialog.FileName));
                NotifyOfPropertyChanged(() => Modules);
            }
        }

        public void Exit()
        {
            Shell.DeactivateItem(this, true);
        }

        /// <summary>
        /// Обработчик команды удаления модуля.
        /// </summary>
        /// <param name="parameter">Неиспользуемый параметр.</param>
        private void RemoveModule(Object parameter)
        {
            throw new NotImplementedException();
        }
    }
}

```

```

    }
}

```

ModuleViewModel.cs

```

using System;
using Caliburn.Micro;
using StaticAnalysis.Core;

namespace StaticAnalysis.StandaloneApp.ViewModels
{
    /// <summary>
    /// Модель представления модуля с диагностическими правилами.
    /// </summary>
    public class ModuleViewModel : PropertyChangedBase
    {
        private AnalyzerModule _module;

        /// <summary>
        /// Конструктор <see cref="ModuleViewModel"/>.
        /// </summary>
        /// <param name="module">Ссылка на модуль анилизатора.</param>
        public ModuleViewModel(AnalyzerModule module)
        {
            _module = module;
        }

        /// <summary>
        /// Название модуля.
        /// </summary>
        public String ModuleName => _module.ModuleName;

        /// <summary>
        /// Является ли модуль встроенным.
        /// </summary>
        public Boolean IsInternal => _module.IsInternal;

        /// <summary>
        /// Включен ли модуль в процесс анализа.
        /// </summary>
        public Boolean IsEnabled
        {
            get => _module.IsEnabled;
            set
            {
                _module.IsEnabled = value;
                NotifyOfPropertyChange();
            }
        }
    }
}

```

ReportViewModel.cs

```

using System.Collections.Generic;
using System.Collections.ObjectModel;
using System.Linq;
using Caliburn.Micro;
using StaticAnalysis.Core;

namespace StaticAnalysis.StandaloneApp.ViewModels
{
    /// <summary>
    /// Модель представления отчёта анализатора.

```

```

/// </summary>
public class ReportViewModel : Screen
{
    /// <summary>
    /// Конструктор <see cref="ModuleViewModel"/>.
    /// </summary>
    public ReportViewModel(ShellViewModel shell)
    {
        Shell = shell;
        Issues = new ObservableCollection<IssueViewModel>();
    }

    public ReportViewModel Initialize(IEnumerable<Issue> issues)
    {
        Issues = new ObservableCollection<IssueViewModel>(
            issues
                .Where(i => i.Severity != Severity.Hidden)
                .Select(i => new IssueViewModel(i))
        );

        NotifyOfPropertyChange(() => Issues);

        return this;
    }

    /// <summary>
    /// Ссылка на главную модель представления.
    /// </summary>
    public ShellViewModel Shell { get; }

    /// <summary>
    /// Список предупреждений.
    /// </summary>
    public ObservableCollection<IssueViewModel> Issues { get; private
set; }

    public void Exit()
    {
        Shell.DeactivateItem(this, true);
    }
}

```

ShellViewModel.cs

```

using System.Linq;
using Caliburn.Micro;
using StaticAnalysis.Core;

namespace StaticAnalysis.StandaloneApp.ViewModels
{
    /// <summary>
    /// Тип объекта анализа.
    /// </summary>
    public enum AnalysisItemType { Solution, Project, Document, Unknown }

    /// <summary>
    /// Модель представления, показываемого при старте программы.
    /// </summary>
    public class ShellViewModel : Conductor<Screen>.Collection.OneActive
    {
        /// <summary>
        /// Конструктор <see cref="ShellViewModel"/>.
    }
}

```

```

    /// </summary>
    public ShellViewModel()
    {
        DisplayName = "Статический анализатор кода C#";

        Analyzer = new
Analyzer(Enumerable.Empty<AnalyzerModuleDefinition>());
        Initial = new InitialViewModel(this);
        ModuleList = new ModuleListViewModel(this);
        Report = new ReportViewModel(this);

        ActivateItem(Initial);
    }

    /// <summary>
    /// Ссылка на анализатор.
    /// </summary>
    public Analyzer Analyzer { get; }

    /// <summary>
    /// Модель представления управления процессом анализа.
    /// </summary>
    public InitialViewModel Initial { get; }

    /// <summary>
    /// Модель представления списка модулей с диагностическими правилами.
    /// </summary>
    public ModuleListViewModel ModuleList { get; }

    /// <summary>
    /// Модель представления отчёта анализатора.
    /// </summary>
    public ReportViewModel Report { get; }
}
}

```

Bootstrapper.cs

```

using System;
using System.Collections.Generic;
using System.Windows;
using Caliburn.Micro;
using StaticAnalysis.StandaloneApp.ViewModels;

namespace StaticAnalysis.StandaloneApp
{
    public class Bootstrapper : BootstrapperBase
    {
        private SimpleContainer _container;

        public Bootstrapper()
        {
            Initialize();
        }

        protected override void Configure()
        {
            _container = new SimpleContainer();
            _container.Singleton<IWindowManager, WindowManager>();
            _container.PerRequest<ShellViewModel>();
        }

        protected override void OnStartup(Object sender, StartupEventArgs e)

```

```

    {
        DisplayRootViewFor<ShellViewModel>();
    }

    protected override Object GetInstance(Type service, String key)
    {
        return _container.GetInstance(service, key);
    }

    protected override IEnumerable<Object> GetAllInstances(Type service)
    {
        return _container.GetAllInstances(service);
    }

    protected override void BuildUp(Object instance)
    {
        _container.BuildUp(instance);
    }
}

```

DelegateCommand.cs

```

using System;
using System.Windows.Input;

namespace StaticAnalysis.StandaloneApp.Commands
{
    public class DelegateCommand : ICommand
    {
        private readonly Predicate<object> _canExecute;
        private readonly Action<object> _execute;

        public DelegateCommand(Action<object> execute) : this(execute, null)
        { }

        public DelegateCommand(Action<object> execute, Predicate<object>
canExecute)
        {
            _execute = execute;
            _canExecute = canExecute;
        }

        public event EventHandler CanExecuteChanged;

        public virtual bool CanExecute(object parameter)
        {
            return _canExecute?.Invoke(parameter) ?? true;
        }

        public virtual void Execute(object parameter)
        {
            _execute?.Invoke(parameter);
        }

        public void RaiseCanExecuteChanged()
        {
            CanExecuteChanged?.Invoke(this, EventArgs.Empty);
        }
    }
}

```