

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение высшего образования

«ПЕНЗЕНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

Факультет

вычислительной техники

Кафедра

«Вычислительная техника»

Направление подготовки: 09.04.01 «Информатика и вычислительная техника»

Магистерская программа: «Вычислительные машины, комплексы, системы и сети»

МАГИСТЕРСКАЯ ДИССЕРТАЦИЯ

на тему

**Аналитическая система поиска избыточных
функциональных объектов в исполняемых файлах,
разработанных для платформы Microsoft .NET**

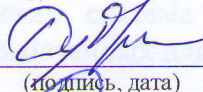
Студент



(подпись, дата)

Сафронов Дмитрий Викторович
(ФИО полностью)

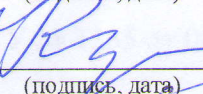
Руководитель



(подпись, дата)

А. В. Дубравин
(фамилия, инициалы)

Нормоконтролёр



(подпись, дата)

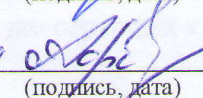
1.07.19 А.В. Кучин
(фамилия, инициалы)

Рецензент

исс. студент

ООО «АТТ», «Краткосрочный»

(должность, место работы)

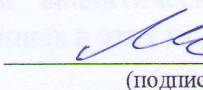


(подпись, дата)

01.07.19 Арсенов А.А.
(фамилия, инициалы)

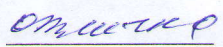
Работа допущена к защите (протокол зас. кафедры от 31.05.19 № 38-09-14)

Заведующий кафедрой

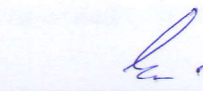


(подпись)

М.А. Митрохин
(фамилия, инициалы)

Работа защищена с отметкой  (протокол заседания ГЭК от 02.07.2019г. № 12)

Секретарь ГЭК



(подпись)

Е.И. Гурин
(фамилия, инициалы)

Пенза 2019

Федеральное государственное бюджетное образовательное учреждение
высшего образования
«ПЕНЗЕНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

Факультет

вычислительной техники

Кафедра

«Вычислительная техника»

«Утверждаю»

Заведующий кафедрой

М.А. Митрохин

«23» апреля 20 19 г.

**ЗАДАНИЕ
НА МАГИСТЕРСКУЮ ДИССЕРТАЦИЮ**

1. Студент Сафронов Дмитрий Викторович гр. 17ВВм1 факультета вычислительной техники.

2. Тема работы: «Аналитическая система поиска избыточных функциональных объектов в исполняемых файлах, разработанных для платформы Microsoft .NET».

Тема утверждена приказом ПГУ № 1255 от 08.12.2017 г.

3. Руководитель магистерской диссертации к.т.н. доцент кафедры ВТ Дубравин А.В.

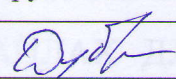
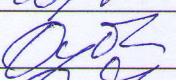
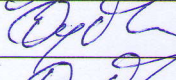
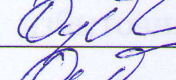
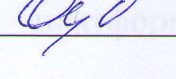
4. Задание на работу (назначение разработки, исходные данные и т.п.).

Назначение разработки — аналитическая система, предназначенная для выявления избыточных функциональных объектов в исполнимых файлах, предназначенных для выполнения под управлением платформы Microsoft .NET. Для некоторой совокупности исполнимых программных модулей аналитическая система должна выполнять поиск функциональных объектов, определённых в этих модулях, к которым отсутствуют обращения из этих модулей.

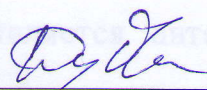
5. Перечень подлежащих разработке вопросов.

Требуется провести анализ предметной области, разработать программные модули аналитической системы, которые будут обеспечивать выявление избыточных функциональных объектов. Произвести отладку и тестирование, оформить работу.

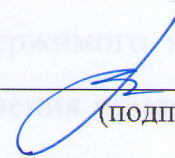
6. Календарный график выполнения работы

№ п/п	Наименование этапов работы	Объем работы	Срок выполнения	Подпись руководителя
1	Анализ предметной области	10	07.05.2019	
2	Определение модулей программы	5	14.05.2019	
3	Разработка основных функций	25	21.05.2019	
4	Отладка и тестирование	20	28.05.2019	
5	Оформление работы	30	02.06.2019	

Дата выдачи задания " 08 " декабре 2018г.

Руководитель магистерской диссертации  Дубравин А.В.
(подпись, дата)

Задание к исполнению принял студент  Сафронов Д.В.
(подпись, дата)

Работу к защите допустить
Декан факультета  Фионова Л.Р.
(подпись, дата)

ОТЗЫВ

НА ДИССЕРТАЦИОННУЮ РАБОТУ

**САФРОНОВА ДМИТРИЯ ВИКТОРОВИЧА «АНАЛИТИЧЕСКАЯ СИСТЕМА ПОИСКА
ИЗБЫТОЧНЫХ ФУНКЦИОНАЛЬНЫХ ОБЪЕКТОВ В ИСПОЛНЯЕМЫХ ФАЙЛАХ,
РАЗРАБОТАННЫХ ДЛЯ ПЛАТФОРМЫ MICROSOFT .NET», ПРЕДСТАВЛЕННУЮ НА
СОИСКАНИЕ СТЕПЕНИ МАГИСТРА ТЕХНИКИ И ТЕХНОЛОГИИ ПО НАПРАВЛЕНИЮ
09.04.01 «ИНФОРМАТИКА И ВЫЧИСЛИТЕЛЬНАЯ ТЕХНИКА»**

С полной уверенностью можно сказать, что на сегодняшний день системы автоматизированного анализа программного кода являются жизненно необходимыми в процессе разработки современных информационных систем и программных продуктов. Без всестороннего анализа невозможно достичь высокого качества программного продукта. Особняком стоят задачи контроля отсутствия избыточности программного кода в процессе разработки программных систем. В виду их малой распространенности, решаемая задача является наиболее актуальной в настоящее время. Цель данной работы состояла в создании методов и подходов к разработке аналитической системы, предназначенной для автоматизации рутинных задач, выполняемых при анализе сложных программных продуктов.

Для достижения поставленной цели в работе формулируются и решаются сложные научно-технические задачи по разработке универсальных методов автоматизации рутинных операций выявления избыточного кода в разрабатываемых программных информационных системах.

В процессе решения поставленных задач автором получены новые значимые теоретические и практические результаты. Наиболее важными результатами исследования можно считать:

- разработаны критерии отнесения функциональных объектов к разряду избыточных;
- разработаны алгоритмы выявления избыточных функциональных объектов в совокупности исполняемых файлов для платформы Microsoft .NET;
- разработана аналитическая система, выполняющая автоматизированный поиск избыточного кода и методика применения аналитической системы на практике.

Диссертация представляет собой завершенную научно-квалификационную работу на актуальную тему и выполнена на высоком теоретическом уровне с применением современных методов исследования. Основные результаты достаточно полно представлены в опубликованных печатных работах. Диссертационная работа Сафронова Д.В. содержит решение задачи, имеющей большое значение в области автоматизации анализа сложных программных систем.

За время научно-исследовательской работы Сафронов Д.В. показал себя грамотным специалистом, проявил добросовестность, трудолюбие и самостоятельность. Поэтому считаю, что диссертация заслуживает отличной оценки, а ее автор заслуживает присуждения степени магистра техники и технологии по направлению 09.04.01 «Информатика и вычислительная техника».

Научный руководитель

к.т.н., доцент Дубравин А.В.

РЕЦЕНЗИЯ

НА ДИССЕРТАЦИОННУЮ РАБОТУ

**САФРОНОВА ДМИТРИЯ ВИКТОРОВИЧА «АНАЛИТИЧЕСКАЯ СИСТЕМА ПОИСКА
ИЗБЫТОЧНЫХ ФУНКЦИОНАЛЬНЫХ ОБЪЕКТОВ В ИСПОЛНЯЕМЫХ ФАЙЛАХ,
РАЗРАБОТАННЫХ ДЛЯ ПЛАТФОРМЫ MICROSOFT .NET»**

Диссертационная работа Сафронова Дмитрия Викторовича посвящена исследованию и разработке методов и подходов к созданию аналитической системы, предназначенной для автоматизации рутинных задач, выполняемых при анализе сложных программных продуктов.

Применение предложенных в работе методов автоматизации позволяет существенно ускорить процесс анализа программного кода и минимизировать использование ручных операций в процессе выявления избыточного кода, содержащегося в сложных программных системах.

В работе автором был проведен глубокий анализ существующих средств автоматизированного анализа кода, рассмотрены отечественные и зарубежные аналоги.

Сафроновым Д.В. разработаны критерии отнесения функциональных объектов к разряду избыточных, алгоритмы выявления избыточных функциональных объектов в совокупности исполняемых файлов для платформы Microsoft .NET. На основе полученных результатов была разработана аналитическая система, выполняющая автоматизированный поиск избыточного кода, а также предложена методика применения аналитической системы на практике.

На мой взгляд, работа не имеет недостатков.

Пояснительная записка оформлена качественно, технически грамотно. Исследования автора изложены ясно и чётко, между разделами существует

логическая взаимосвязь. Считаю, что диссертация заслуживает отличной оценки, а Сафронов Д.В. присуждения степени магистра техники и технологии по направлению 09.04.01 «Информатика и вычислительная техника».



Начальник отдела
ООО "НТП "Криптософт"

Афонин А.Ю.

СОДЕРЖАНИЕ

Введение	7
1. Анализ задачи выявления избыточного кода в приложениях, предназначенных для выполнения под управлением платформы Microsoft .NET	8
1.1. Определение и виды избыточного кода. Предпосылки возникновения избыточного кода	8
1.2. Подходы к выявлению избыточного кода	10
1.3. Особенности структурирования кода приложений, предназначенных для выполнения в среде Microsoft .NET	12
1.3.1. Модули и сборки	12
1.3.2. Метаданные	15
1.3.3. Механизмы взаимодействия между различными сборками. Рефлексия	19
1.4. Определение и виды функциональных объектов. Способы использования функциональных объектов	27
1.4.1. Варианты использования методов	28
1.4.2. Варианты использования классов	31
1.5. Обзор существующих программных средств, позволяющих выявлять избыточные функциональные объекты	32
1.5.1. Вывод	40
1.6. Выводы по главе 1	40
2. Разработка аналитической системы выявления избыточности в управляемых приложениях	42
2.1. Постановка задачи на разработку автоматизированного средства выявления избыточности	42
2.2. Выбор подхода к решению задачи	43
2.3. Алгоритмы решения задачи	44
2.4. Реализация аналитической системы	53

2.5. Выводы по главе 2	53
3. Методика выявления избыточности с применением аналитической системы.....	54
3.1. Подготовка входных данных. Проведение анализа	54
3.2. Результаты работы аналитической системы. Устранение избыточности и проведение повторного анализа	54
3.3. Ограничения на применение аналитической системы. Направления дальнейшего развития	56
3.4. Выводы по главе 3	57
Заключение	58
Список использованных источников	59
Приложение. Презентация	62

ВВЕДЕНИЕ

В настоящей работе рассматривается задача разработки программного средства, предназначенного для автоматизированного выявления избыточных функциональных объектов в составе приложений, предназначенных для выполнения под управлением платформы Microsoft .NET.

Поскольку избыточный код не отражается на функциональности разрабатываемого программного обеспечения, зачастую допускается его присутствие в выпускаемом программном продукте. Наличие такого кода, однако, может приводить к снижению таких характеристик приложения как производительность и безопасность.

В ходе работы планируется проведение анализа задачи выявления избыточного кода в приложениях, предназначенных для выполнения под управлением платформы Microsoft .NET, будет дано определение и приведена классификация избыточного кода. Планируется рассмотрение подходов к выявлению избыточного кода.

В рамках работы будут изучены особенности структурирования кода приложений, предназначенных для выполнения в среде Microsoft .NET, рассмотрены различные виды функциональных объектов и варианты их использования.

Планируется разработка аналитической системы поиска избыточных функциональных объектов в исполняемых файлах, разработанных для платформы Microsoft .NET, способной в автоматическом режиме выявлять неиспользуемые функциональные объекты в некоторой совокупности сборок, составляющих исследуемое программное обеспечение.

Также планируется разработка методики применения аналитической системы для проведения поиска избыточных функциональных объектов в исполняемых файлах, предназначенных для выполнения под управлением платформы Microsoft .NET.

1. АНАЛИЗ ЗАДАЧИ ВЫЯВЛЕНИЯ ИЗБЫТОЧНОГО КОДА В ПРИЛОЖЕНИЯХ, ПРЕДНАЗНАЧЕННЫХ ДЛЯ ВЫПОЛНЕНИЯ ПОД УПРАВЛЕНИЕМ ПЛАТФОРМЫ MICROSOFT .NET

Среда Microsoft .NET в настоящее время является популярной платформой для разработки приложений самого широкого спектра. В процессе активной разработки у программистов не всегда остаётся возможность тщательного обзора кода на предмет избыточности. Поскольку избыточный код не отражается на функциональности разрабатываемого программного обеспечения, зачастую допускается его присутствие в выпускаемом программном продукте.

Таким образом, представляется актуальной задача разработки программного средства, предназначенного для автоматизированного выявления избыточного кода в составе приложений, предназначенных для выполнения под управлением платформы Microsoft .NET.

1.1. Определение и виды избыточного кода. Предпосылки возникновения избыточного кода

Под *избыточным кодом* в настоящей работе понимаются фрагменты исходных текстов либо скомпилированного исполнимого кода программы, не задействованные при штатном выполнении программы. При этом штатным выполнением программы считаются не только основные ветви алгоритма, но и обработчики ошибок и исключительных ситуаций.

Очевидным критерием отнесения кода к категории избыточного является свойство неизменности алгоритма выполнения программы в результате модификации либо полного удаления этого кода.

Следует отметить, что при неизменности алгоритма — основной характеристики — удаление избыточного кода может привести к изменению дополнительных качественных и количественных характеристик. К примеру, в результате может измениться схема попадания кода в кэш исполняющего

процессора, что может привести к изменению времени выполнения программы. Кроме того, исключение фрагментов кода приводит к уменьшению возможности проведения атак, что повышает степень безопасности информационной системы.

В англоязычной литературе для обозначения рассматриваемого понятия применяется термин *недостижимый код* (*unreachablecode*). При этом его отличают от других видов избыточного (*redundant*) кода:

- кода, выполняющего повторные вычисления;
- кода, выполняющего вычисления, не влияющие на результат работы программы (*deadcode*);
- дублирующегося кода (*duplicate code*).

Такие виды избыточного кода алгоритмически связаны с остальным программным кодом, и их обнаружение и устранение требует глубокого анализа алгоритмов работы программы. В рамках данной работы такие виды избыточного кода не рассматриваются.

Можно выделить следующие основные причины появления избыточного кода:

- ошибки при разработке алгоритма;
- вспомогательный код, например, отладочная диагностика, тесты;
- эволюция программного кода, в результате которой фрагмент кода может быть заменён новым фрагментом — более производительным, либо обладающим отличной функциональностью, — но не удалён;
- написание кода «для полноты», особенно характерное для программирования в объектно-ориентированном стиле и активного применения шаблонов проектирования.

В двух последних случаях применительно к библиотекам программного кода (а не к самодостаточным приложениям) избыточность кода должна изучаться особенно тщательно, поскольку функции библиотеки могут использоваться в различных приложениях в разном объёме. Даже

сохранение «устаревшего» в результате эволюции кода может быть оправдано требованиями совместимости.

В ряде случаев для определённых сочетаний *язык программирования* — *целевая платформа* компиляторы способны самостоятельно выявлять фрагменты кода, к которым отсутствуют обращения, и не включать их в итоговый исполнимый файл. Кроме того, директивы условной компиляции позволяют явно указать, что определённый фрагмент не используется при заданных условиях компиляции. В таких случаях избыточность, присутствующая в исходных текстах программ, автоматически исключается на этапе компиляции и отсутствует в непосредственно исполняемом коде.

Следует отметить, что при компиляции приложений для платформы Microsoft .NET, которые являются объектом исследований в данной работе, автоматическое исключение избыточного кода не выполняется по причине наличия механизма рефлексии (см. раздел 1.3.3).

1.2. Подходы к выявлению избыточного кода

Поскольку программный код в процессе разработки приложений существует в двух ипостасях — исходный текст и исполнимый код, — можно выделить два подхода к выявлению и устранению его избыточности.

Анализ и модификация исходных текстов приложения проводится путём лексического и синтаксического разбора исходных текстов и построения *абстрактного синтаксического дерева (дерева разбора)*. Затем выполняется построение связей по управлению между функциональными объектами и анализ этих связей на предмет выявления функциональных объектов, на которые управление не передаётся. В дальнейшем производится удаление соответствующих узлов из дерева разбора и сохранение модифицированной версии исходных текстов.

Подход на основе исполнимого кода предполагает разбор и модификацию уже скомпилированных файлов. Очевидно, что исполнимый файл должен содержать достаточное количество информации, необходимой

для восстановления его семантической структуры, либо сопровождаться дополнительным файлом *символьной информации*. (Такую информацию используют, например, отладчики.) Основные шаги анализа при этом те же, что и при анализе по исходным текстам.

Дополнительно можно рассмотреть подход, основанный на результатах динамического анализа покрытия кода, при котором выполняется ряд тестовых запусков исследуемого приложения и фиксируется перечень функциональных объектов, которые получали управление. Такой подход, однако, для получения достоверных результатов требует при выполнении полного покрытия всех ветвей кода, включая обработчики исключительных ситуаций. Поэтому он применим только при определённых методиках разработки программного обеспечения, при которых написание полностью покрывающего комплекта тестов является неотъемлемой частью процесса разработки.

Подход на основе данных динамического анализа ввиду его трудоёмкости и не универсальности в рамках данной работы не рассматривается.

Наиболее универсальным является подход на основе исходных текстов. Такой подход позволяет проводить анализ программ на любых языках программирования, вне зависимости от способа трансляции (интерпретация или компиляция) и целевой платформы.

Во многих случаях, однако, анализ исходных текстов является весьма трудоёмкой задачей. Помимо разбора исходного кода от анализатора требуется определение вызываемого варианта перегруженной функции, учёт всех неявных вызовов, а в случае, например, языка C++—раскрытие шаблонов. Автоматизированное средство такого анализа по сложности сопоставимо с полноценным компилятором, а его применение требует организации среды, эквивалентной среде разработки.

Анализ на основе исполнимых файлов применим только в тех случаях, когда восстановление семантической информации, отбрасываемой в

процессе компиляции, не представляет особой сложности. В качестве преимущества такой подход предлагает чётко структурированный в виде функциональных объектов исполнимый файл и уже настроенные связи между функциональными объектами— адреса, указатели, токены и т. п.

1.3. Особенности структурирования кода приложений, предназначенных для выполнения в средеMicrosoft .NET

Приложения, предназначенные для выполнения в среде Microsoft .NET, называются *управляемыми приложениями*. Среда выполнения контролирует большинство аспектов функционирования приложений, таких как разделение кода и данных, доступ к памяти и т. п. Необходимым средством для такого контроля является, в том числе, обязательная структурированность исполнимого кода приложений и наличие дополнительных данных о его семантической структуре — метаданных.

1.3.1. Модули и сборки

Исполняемые файлы, предназначенные для выполнения под управлением платформы Microsoft .NET, называются модулями. Модули .NET, как и обычные исполняемые файлы Windows, могут содержать исполняемый код и ресурсы. Как правило, модули .NET содержат код на промежуточном языке `CommonIntermediateLanguage (CIL)`, но могут содержать и машинный код. Ресурсы представляют собой различные данные, такие как текстовые строки, иконки, звуки и т. п. В отличие от обычных исполняемых файлов Windows модули .NET содержат еще и метаданные, описывающие типы, определяемые исполняемым кодом.

Отдельные модули могут разрабатываться на различных языках программирования. Возможность взаимодействия обеспечивается применением общей системы типов (`CommonTypeSystem`).

С точки зрения формата модули .NET являются исполняемыми файлами формата `PortableExecutable (PE)`. Как и обычные исполняемые файлы Windows, модули .NET могут быть файлами EXE и DLL и точно

также файлы EXE используются для хранения модулей, непосредственно запускаемых на выполнение, а файлы DLL используются для хранения библиотек. Но есть одно важное отличие: для того, чтобы модуль .NET мог быть загружен и выполнен CLR, он должен входить в состав сборки.

Сборка — это логическое объединение файлов. В состав сборки могут входить как модули .NET, так и произвольные файлы. Объединение файлов в сборку выполняется с помощью манифеста, который представляет собой совокупную информацию определенных таблиц метаданных. Манифест размещается в одном (и только одном) модуле сборки, который называется главным модулем. Наличие у сборки главного модуля является обязательным.

Сборки могут быть однофайловыми и многофайловыми. Однофайловая сборка состоит только из одного файла — главного модуля. Если сборка является библиотекой, то главный модуль является файлом DLL и, как правило, имеет расширение .dll. Если сборка предназначена для непосредственного запуска на выполнение, то главный модуль является файлом EXE. Остальные модули сборки могут иметь произвольные расширения и всегда являются файлами DLL. Произвольные файлы, включенные в состав сборки, по определению имеют произвольные форматы и расширения.

Произвольные файлы могут входить в состав сборки в качестве внешних ресурсов или просто как файлы данных. Если файл является внешним ресурсом, то, помимо стандартных средств, доступ к его содержимому может быть осуществлен посредством классов из пространства имен `System.Resources` или методов `GetManifestResourceXXX` класса `Assembly`.

Сборка может быть подписана (signed) двумя независимыми и дополняющими друг друга способами: присвоением строгого имени (strong name) и присвоением цифрового сертификата.

Присвоение сборке строгого имени обеспечивает уникальность имени сборки, а также защищает все файлы сборки от несанкционированного изменения. Присвоение сборке строгого имени (подписание строгим именем — strong name signing) осуществляется следующим образом.

Для каждого файла сборки вычисляется контрольная сумма (хеш) по алгоритму, указанному в манифесте. Как правило, для подсчета хешей в .NET Framework используется алгоритм SHA1 (Secure Hash Algorithm).

Хеши всех файлов, кроме хеша главного модуля, помещаются в манифест. Хеш главного модуля шифруется по алгоритму RSA (Rivest Shamir Adleman). Этот алгоритм использует пару криптографических ключей (открытый и закрытый ключ). Шифрование можно осуществить только с помощью закрытого ключа, дешифрование — только с помощью открытого ключа. Шифрованное значение хеша главного модуля, называемое сигнатурой строгого имени (strong name signature), помещается в манифест. Это значение и является цифровой подписью. Открытый ключ помещается в манифест.

Такой механизм обеспечивает возможность проверки целостности сборки. Для проверки подлинности сборки подписываются с помощью сертификата.

Присвоенный файлу сертификат содержит цифровую подпись, обеспечивающую проверку целостности файла по тому же принципу, что и подписание строгим именем (содержит шифрованный на закрытом ключе хеш содержимого файла), и открытый ключ для дешифрования хеша. Кроме того, сертификат содержит сведения об издателе и цифровую подпись для проверки целостности самого сертификата (шифрованный на закрытом ключе хеш содержимого сертификата), но не содержит открытого ключа для дешифрования этого хеша.

Проверка целостности сертификата файла может быть выполнена лишь с помощью корневого сертификата. Корневой сертификат содержит цифровую подпись для проверки целостности самого корневого сертификата

(шифрованный на закрытом ключе хеш содержимого корневого сертификата) и открытый ключ для дешифрования этого хеша. Этот же открытый ключ используется для дешифрования хеша содержимого сертификата файла. Корневые сертификаты расположены в хранилище сертификатов операционной системы и могут быть помещены в него только администратором. Таким образом, файл защищен от несанкционированного повторного присвоения сертификата, что гарантирует подлинность файла.

Сценарий совместного использования строгого имени и цифрового сертификата выглядит следующим образом: сначала выполняется подписание сборки строгим именем, а затем выполняется присвоение сертификата главному модулю сборки. Сертификат гарантирует подлинность и целостность главного модуля, а строгое имя — целостность главного модуля и всех файлов сборки.

Строгое имя сборки хранится в манифесте и состоит из следующих компонентов:

- дружественное имя (friendly name);
- версия;
- региональные установки;
- открытый ключ — открытая часть пары криптографических ключей, используемых для формирования и проверки цифровой подписи сборки.

Имя сборки используется для ссылки (reference) на сборку при ее использовании другой сборкой. Ссылки на используемые сборки хранятся в манифесте. Каждая ссылка содержит имя используемой сборки. Если используемая сборка имеет строгое имя, то ссылка содержит либо открытый ключ используемой сборки, либо его хеш, рассчитываемый по алгоритму SHA.

1.3.2. Метаданные

Основными элементами файла формата PE являются:

заголовки, которые содержат служебную информацию, используемую операционной системой для загрузки исполняемого файла и поиска других его элементов;

секции, которые являются контейнерами для кода и данных программы, а также для различной служебной информации.

Основную информацию об исполнимом файле содержит заголовок `IMAGE_NT_HEADERS`. Он состоит из сигнатуры, заголовка файла и расширенного заголовка — `IMAGE_OPTIONAL_HEADER`.

Помимо кода и данных программы исполняемый файл содержит множество служебных данных, требующихся операционной системе для поддержки выполнения файла. Чтобы упростить поиск каких-либо специфичных данных в исполняемом файле, данные организуются в директории данных по типовому признаку. Тип директории данных определяет назначение и структуру хранящейся в ней информации. Это похоже на то, как директория на диске содержит файлы, относящиеся к определенному приложению. Поиск директорий данных в исполняемом файле осуществляется с помощью списка директорий данных.

Список директорий данных представлен полем `DataDirectory` заголовка `IMAGE_OPTIONAL_HEADER` и представляет собой массив структур `IMAGE_DATA_DIRECTORY`. Тип директории данных является индексом элемента в этом массиве.

Для хранения метаданных .NET используется директория `IMAGE_DIRECTORY_ENTRY_COM_DESCRIPTOR`. Эта директория содержит данные, представляющие собой структуру `IMAGE_COR20_HEADER`. Адрес и размер метаданных указаны в поле `MetaData` этой структуры. Общая структура метаданных приведена на рисунке 1:

Заголовок STORAGESIGNATURE				
Заголовок STORAGEHEADER				
Заголовок STORAGESTREAM				
Системные строки (куча строк)	Различные данные (куча блобов)	Значения GUID (куча GUID)	Строки пользователя (куча блобов)	Заголовок METADATA_TABLE_HEADER
				Таблица числа строк таблиц
				Таблицы метаданных
Поток #Strings	Поток #Blob	Поток #GUID	Поток #US	Поток #~

Рисунок 1 — Общая структура метаданных

Логически метаданные организованы в виде набора именованных потоков, каждый из которых содержит определенную категорию метаданных. Местонахождение потоков внутри файла описывается несколькими заголовками, расположенными в начале метаданных. Содержимое потоков можно условно разделить на два типа: кучи метаданных и таблицы метаданных.

Кучи содержат последовательности элементов, доступ к которым осуществляется по смещению от начала кучи. В общем случае элементы имеют различную длину и формат, но могут быть и однотипными элементами, например, строками, а также элементами одинаковой длины, например, значениями GUID. Существует три вида куч метаданных:

- куча строк; содержит строки в кодировке UTF-8;
- куча GUID; содержит массив 128-битных значений GUID;
- куча блобов (BinaryLargeObject, BLOB); содержит различные двоичные данные.

Таблицы содержат данные, организованные в виде записей (строк таблицы) с определенным количеством полей (колонок таблицы). Для каждой таблицы, которая может присутствовать в метаданных, определена

схема, описывающая имя и номер таблицы, количество и порядок расположения колонок таблицы, их имена и типы, на какие таблицы и кучи могут ссылаться те или иные колонки.

Всего в исполняемом файле .NETFramework может присутствовать до 45 таблиц. Краткое описание основных таблиц приведено в таблице **Ошибка! Не указано имя закладки..**

Таблица 1 — Таблицы метаданных .NETFramework

Номер таблицы	Имя таблицы	Назначение таблицы
0x00	Module	Описание текущего модуля.
0x01	TypeRef	Описание типов, определенных в других модулях или сборках, на которые ссылается текущий модуль.
0x02	TypeDef	Описание типов, определенных в текущем модуле.
0x04	FieldDef	Описание полей типов.
0x06	MethodDef	Описание методов типов.
0x08	ParamDef	Описание параметров методов типов.
0x0A	MemberRef	Описание полей и методов типов, определенных в других модулях или сборках, на которые ссылается текущий модуль. Содержит также описания методов с переменным числом параметров независимо от того, в текущем модуле или нет, определен метод. Кроме того, содержит описания и методов и полей инстанцированных обобщенных типов, независимо от того, в текущем модуле или нет, определен обобщенный тип.
0x14	EventDef	Описание событий.
0x17	PropertyDef	Описание свойств типов.
0x1A	ModuleRef	Описание модулей, на которые ссылается текущий модуль.
0x20	Assembly	Описание текущей сборки. Присутствует только в модуле, содержащем манифест.
0x23	AssemblyRef	Описание сборок, на которые ссылается текущий модуль.

Кучи и таблицы метаданных хранятся в потоках. Всего в файле .NETFramework может присутствовать до 5 потоков метаданных(см. таблицу **Ошибка! Не указано имя закладки.**).

Таблица 2 — Потоки метаданных .NETFramework

Имя потока	Описание
#Strings	Содержит кучу строк. В куче находятся системные строки: имена типов, методов, полей и т.д. Строки хранятся в кодировке UTF-8 и оканчиваются 0. Первая строка кучи всегда пустая, т.е. содержит только завершающий 0.
#Blob	Содержит кучу блобов. Первый элемент кучи всегда пустой, т.е. содержит только поле длины и длина равна 0. Блобы содержат сигнатуры полей и методов типов, дескрипторы маршалинга и параметры атрибутов.
#GUID	Содержит кучу значений GUID, представляющую собой массив 128-битных значений GUID.
#US	Содержит кучу блобов. Первый элемент кучи всегда пустой, т.е. содержит только поле длины и длина равна 0. Блобы содержат пользовательские строки, т.е. строковые константы, определенные в тексте программы. Строки содержат символы в кодировке UTF-16. Строки не оканчиваются нулем. Последний байт строки содержит 1, если старший байт любого символа строки не равен 0 или его младший байт имеет значение 0x01–0x08, 0x0E–0x1F, 0x27, 0x2D, 0x7F. В противном случае последний байт строки содержит 0.
#~	Содержит оптимизированные таблицы метаданных.

Код IL ссылается на пользовательские строки (поток #US) и таблицы метаданных (поток #~), таблицы метаданных ссылаются друг на друга и на данные других потоков. Данные потока #Blob могут ссылаться на таблицы метаданных.

Для ссылок на данные в кучах используются индексы. Данные в таблицах идентифицируются *токенами метаданных*. Токен представляет собой 32-битное значение, старший байт которого содержит идентификатор таблицы, а младшие байты — идентификатор записи (RecordIdentifier — RID), т. е. номер строки таблицы.

1.3.3. Механизмы взаимодействия между различными сборками.

Рефлексия

Сборка может не только состоять из нескольких файлов, но и использовать в процессе выполнения другие сборки. Ссылки на используемые сборки хранятся в манифесте и в IL-коде. Используемая сборка идентифицируется по имени. Подключение используемых сборок выполняет загрузчик прозрачно для приложения, поэтому такая загрузка называется *неявной*.

Кроме автоматической загрузки сборок средствами CLR существует возможность загрузки необходимых сборок из кода приложения. Такой способ называется динамической или *явной загрузкой* сборок. Для этого используется класс `System.Reflection.Assembly`. Он поддерживает три статических метода, позволяющих явно загрузить сборку: `Load`, `LoadFrom` и `LoadWithPartialName`.

Метаданные, содержащиеся в сборке, позволяют во время выполнения получить исчерпывающую информацию о типах, описанных в этой сборке. Механизм, реализующий эту возможность, называется отражением (reflection). Программный интерфейс отражения представлен классами пространства имен `System.Reflection`.

Помимо получения информации о типах рефлексия позволяет инстанцировать объекты по описателю типа, вызывать методы по дескриптору и т.д. С помощью рефлексии возможно обращение к закрытым (private), защищенным (protected) и внутренним (internal) членам класса.

1.3.3.1. Явная загрузка сборок и рефлексия типов

Явная загрузка сборок выполняется с помощью статического метода `Load` класса `System.Reflection.Assembly`. Этот метод является CLR-эквивалентом функции `LoadLibrary` Win32 API. Этот метод имеет несколько перегруженных вариантов, из которых наиболее часто используются следующие:

```
public class Assembly {
...
public static Assembly Load(AssemblyName assemblyRef);
public static Assembly Load(String assemblyString);
...
}
```

Реализация метода `Load` выполняет поиск сборки в глобальном кэше сборки (GAC), в папке приложения и в локальных папках сборки. Если выполняется загрузка по простому имени сборки (не по строке), поиск в глобальном кэше не выполняется.

Если метод `Load` находит заданную сборку, он возвращает ссылку на объект класса `Assembly`, который представляет загруженную сборку; в противном случае выбрасывается исключение `System.IO.FileNotFoundException`.

При создании программных инструментов, выполняющих анализ метаданных сборки посредством механизма отражения, для загрузки сборки применяются дополнительные методы `ReflectionOnlyLoad` и `ReflectionOnlyLoadFrom` класса `Assembly`, гарантирующие невозможность выполнения кода, содержащегося в загружаемых сборках:

```
public class Assembly {
...
public static Assembly ReflectionOnlyLoadFrom(
String assemblyFile);
public static Assembly ReflectionOnlyLoad(
String assemblyString);
...
}
```

Метод `ReflectionOnlyLoadFrom` загружает сборку, находящуюся по указанному пути; в других папках поиск этой сборки не производится.

Метод `ReflectionOnlyLoad` выполняет поиск сборки в глобальном кэше, папке приложения и в локальных папках сборки. Вот отличие от метода `Load`, метод `ReflectionOnlyLoad` не применяет политики подбора версий, поэтому загружается в точности та версия сборки, которая была запрошена.

Такое поведение можно изменить с помощью метода `ApplyPolicy` класса `AppDomain`.

Если сборка загружена с помощью методов `ReflectionOnlyLoadFrom` или `ReflectionOnlyLoad`, среда CLR запрещает выполнение любого кода в сборке; любая попытка выполнения кода в сборках, загруженных с помощью этих методов, приводит к возникновению исключения `InvalidOperationException`. Поэтому эти методы позволяют загружать сборки, которые иначе не могут быть загружены, например, предназначенные для другой процессорной архитектуры.

Зачастую при использовании механизма отражения для анализа сборок загруженных одним из этих методов требуется регистрация метода обратного вызова с помощью метода `ReflectionOnlyAssemblyResolve` класса `AppDomain` для ручной загрузки сборок, на которые ссылается загружаемая; среда CLR не выполняет такую загрузку автоматически. Когда этот метод вызывается, он должен вызвать метод `ReflectionOnlyLoadFrom` либо `ReflectionOnlyLoad` для явной загрузки используемой сборки и вернуть ссылку на эту сборку.

1.3.3.2. Обнаружение типов, объявленных в сборке

Механизм отражения часто используется для установления типов, определённых в сборке. Системная библиотека классов предлагает ряд методов для получения такой информации. Наиболее широко используется метод `GetExportedTypes` класса `Assembly`. На листинге **Ошибка! Не указано имя закладки.** приведён пример использования этого метода для обнаружения типов, объявленных в сборке.

```
using System;
using System.Reflection;
public static class Program {
    public static void Main() {
        String dataAssembly = "System.Data, version=4.0.0.0, " +
            "culture=neutral, PublicKeyToken=b77a5c561934e089";
        LoadAssemAndShowPublicTypes(dataAssembly);
    }
    private static void LoadAssemAndShowPublicTypes(
```

```
String assemId) {
// Explicitly load an assembly in to this AppDomain
Assembly a = Assembly.Load(assemId);
// Execute this loop once for each Type
// publicly-exported from the loaded assembly
foreach (Type t in a.GetExportedTypes()) {
// Display the full name of the type
Console.WriteLine(t.FullName);
}
}
}
```

Листинг1 — Обнаружение типов, определённых в сборке

1.3.3.3. Использование отражения для обнаружения членов типа

Возможность обнаружения и активации членов типа как правило используется при создании инструментов анализа сборок, таких как дизассемблер ILDasm, инструмент контроля качества кода FxCop и дизайнер Windows Forms, входящий в Visual Studio. Также некоторые библиотеки классов используют эту возможность для предоставления более удобной расширенной функциональности. Примерами таких библиотек являются библиотеки сериализации и привязки к данным.

В качестве членов некоторого типа могут быть определены поля, конструкторы, методы, свойства, события и вложенные типы. Системная библиотека типов содержит тип `System.Reflection.MemberInfo`, являющийся абстрактным базовым классом, который инкапсулирует набор свойств, общих для всех членов типа. Ряд классов, производных от `MemberInfo`, содержат свойства, специфичные для различных видов членов типа. Иерархия этих типов представлена на рисунке 2.

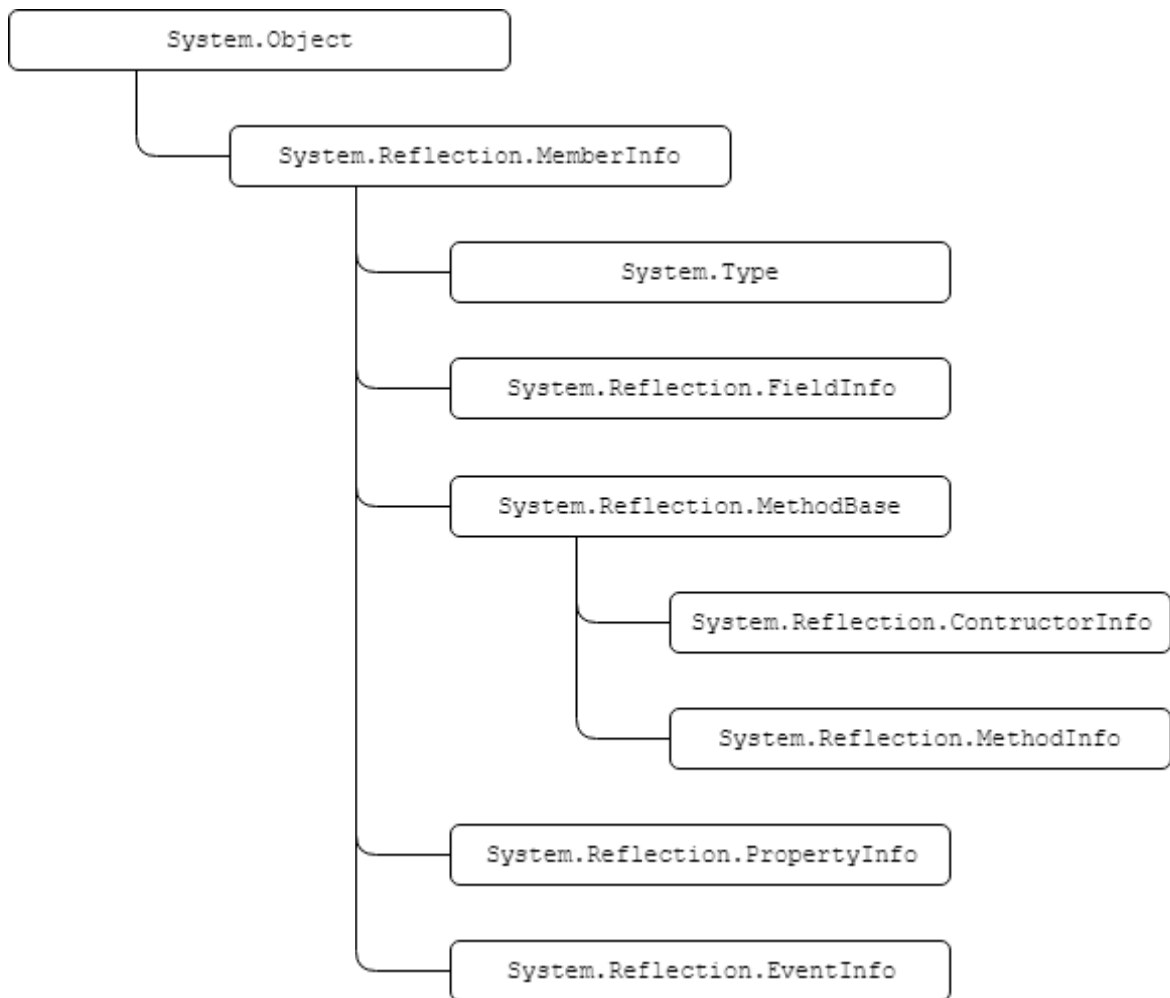


Рисунок 2 — Иерархия типов рефлексии, содержащих информацию о членах типа

На листинге **Ошибка! Не указано имя закладки.** приведён пример программы, перечисляющей члены всех типов, определённых в сборках. Код перечисляет все публичные типы, определённые во всех сборках, загруженных в текущий домен приложений. Для каждого типа вызывается метод `GetMembers`, который возвращает массив объектов, типы которых являются производными от класса `MemberInfo`; каждый объект ссылается на отдельный член, определённый в классе. Для каждого члена отображается его вид текстовое представление.

```

using System;
using System.Reflection;
public static class Program {
public static void Main() {

```

```

// Loop through all assemblies loaded in this AppDomain
Assembly[] assemblies =
AppDomain.CurrentDomain.GetAssemblies();
foreach (Assembly a in assemblies) {
WriteLine(0, "Assembly: {0}", a);
// Find Types in the assembly
foreach (Type t in a.GetExportedTypes()) {
WriteLine(1, "Type: {0}", t);
// Discover the type's members
const BindingFlags bf =
BindingFlags.DeclaredOnly |
BindingFlags.NonPublic |
BindingFlags.Public |
BindingFlags.Instance | BindingFlags.Static;
foreach (MemberInfo mi in t.GetMembers(bf)) {
String typeName = String.Empty;
if (mi is Type)
typeName = "(Nested) Type";
else if (mi is FieldInfo)
typeName = "FieldInfo";
else if (mi is MethodInfo)
typeName = "MethodInfo";
else if (mi is ConstructorInfo)
typeName = "ConstructoInfo";
else if (mi is PropertyInfo)
typeName = "PropertyInfo";
else if (mi is EventInfo)
typeName = "EventInfo";
WriteLine(2, "{0}: {1}", typeName, mi);
}
}
}
}
private static void WriteLine(Int32 indent, String format,
params Object[] args) {
Console.WriteLine(new String(' ', 3 * indent) + format,
args);
}
}

```

Листинг 1 — Получение информации о членах типа

1.3.3.4. Активация члена типа

После получения информации о члене типа, его можно активировать. Конкретные действия активации зависят от вида члена. Активация *поля* (*FieldInfo*) подразумевает установку или получение его значения, активация *конструктора* (*ConstructorInfo*) позволяет создавать экземпляры объектов, передавая параметры конструктору, активация *метода*

(MethodInfo) позволяет вызвать метод в заданными аргументами и получить возвращаемое значение, активация *свойства* (PropertyInfo) приводит к вызову соответствующего свойству метода установки или получения значения, а активация *события* (EventInfo) — к добавлению или удалению обработчика события.

Класс Type предоставляет метод InvokeMember, который позволяет активировать различные члены. Этот метод имеет ряд перегруженных вариантов, один из наиболее употребляемых представлен на листинге **Ошибка! Не указано имя закладки..**

```
public abstract class Type : MemberInfo, ... {
    public Object InvokeMember(
        String name, // Name of member
        BindingFlags invokeAttr, // How to look up members
        Binder binder, // How to match members and arguments
        Object target, // Object to invoke member on
        Object[] args, // Arguments to pass to method
        CultureInfo culture); // Culture used by some binders
    ...
}
```

Листинг 2— Объявление метода InvokeMember класса Type

При вызове метод InvokeMember выполняет поиск подходящего члена типа.

Если соответствие не найдено, выбрасывается исключение System.MissingMethodException, System.MissingFieldException либо System.MissingMemberException.

Если соответствие найдено, метод InvokeMember активирует его, если активация вернула какое-либо значение, возвращает это значение.

Если активация привела к исключению, метод InvokeMember перехватывает это исключение и выбрасывает новое исключение типа System.Reflection.TargetInvocationException, свойство InnerException которого содержит объект исключения, выброшенного в действительности.

Перед непосредственным выполнением вызова метод InvokeMember выполняет выбор соответствующего активируемого

члена. При вызове метода `InvokeMember` ему передаётся в качестве параметра строка с именем члена. Внутри типа, однако, может быть объявлено несколько членов с таким именем, например, одноимённые метод и поле, либо несколько перегруженных вариантов метода. Для уникального установления активируемого метода используются все параметры метода `InvokeMember`.

1.4. Определение и виды функциональных объектов. Способы использования функциональных объектов

Под функциональным объектом в данной работе понимается некоторый законченный фрагмент кода, необязательно атомарный, имеющий строго определённый интерфейс активации. Функциональными объектами могут являться глобальные функции, методы классов, а также сами классы. Интерфейсы, абстрактные классы и их методы не относятся к функциональным объектам, поскольку представляют собой только объявления, без связанной с ними реализации.

С каждым функциональным объектом связана некоторая область видимости, определяющая совокупность функциональных объектов, которые могут обращаться к заданному без применения специального механизма рефлексии (см. раздел 1.3.3). На языке C# область видимости задаётся следующими ключевыми словами (см. таблицу **Ошибка! Не указано имя закладки.**).

Таблица 3 — Область видимости функциональных объектов

Ключевое слово	Область видимости
<code>private</code>	для метода — только содержащий класс, для класса — только содержащая сборка
<code>protected</code>	для метода — содержащий класс и производные от него классы
<code>internal</code>	только содержащая сборка
<code>public</code>	не ограничена

Очевидно, что, приняв во внимание область видимости функционального объекта можно сузить область поиска обращений к этому объекту.

1.4.1. Варианты использования методов

Функциональные объекты-методы могут быть использованы следующими способами.

Прямой вызов (directcall). Непосредственный вызов конкретного метода у заданного объекта. Вызываемый метод определяется именем; набор параметров вызова идентифицирует один из перегруженных вариантов (если метод перегруженный). Прямой вызов может применяться как для экземплярных (instance), так и для статических (static) методов. На листингах **Ошибка! Не указано имя закладки.** и **Ошибка! Не указано имя закладки.** приведён пример кода, выполняющего вызов экземплярного метода (приведён фрагмент кода на языке C# и результат его дизассемблирования на языке CIL), а на листингах **Ошибка! Не указано имя закладки.** и **Ошибка! Не указано имя закладки.** — пример вызова статического метода.

```
DirectoryInfo folder = ...;
var fileList = folder.EnumerateFiles();
```

Листинг 3 — Вызов экземплярного метода на языке C#

```
.locals init (
    [0] class System.IO.DirectoryInfo folder,
    [1] class System.Collections.Generic.IEnumerator`1<
class System.IO.FileInfo>fileList
)
...
ldloc.0
callvirt instance class System.Collections.Generic.
    IEnumerable`1<class System.IO.FileInfo>
System.IO.DirectoryInfo::EnumerateFiles()
stloc.1
```

Листинг 4 — Вызов экземплярного метода на языке CIL

```
var relativePath = Path.GetRelativePath(rootPath,
fileInfo.Directory.FullName);
```

Листинг 5 — Вызов статического метода на языке C#

```
ldarg.0
```

```
ldfld      string Program::rootPath
ldarg.1
callvirt   instance class System.IO.DirectoryInfo
           System.IO.FileInfo::get_Directory()
callvirt   instance string
           System.IO.FileSystemInfo::get_FullName()
call       string System.IO.Path::GetRelativePath(string, string)
stloc.1
```

Листинг 6 — Вызов статического метода на языке CIL

Косвенный вызов (indirectcall). Вызов метода по идентификатору, предварительно помещённому в некоторую переменную. На листингах **Ошибка! Не указано имя закладки.** и **Ошибка! Не указано имя закладки.** приведён пример вызова метода по идентификатору на языке C++/CLI и результат его дизассемблирования на языке CIL.

```
int Parse(array<String^>^ tokens)
{
...
}

int main(array<String^> ^args)
{
    auto parseFn = &Parse;

    parseFn(args);

    return 0;
}
```

Листинг 7 — Вызов метода по идентификатору на языке C++/CLI

```
.method assembly static int32 main (string[] args) cil managed
{
    .locals (
        [0] int32,
        [1] class 'method System.Int32 *(System.String[])'
    parseFn
    )

    ldc.i4.0
    stloc.0
    ldftn      int32 '<Module>::Parse(string[])'
    stloc.1
    ldarg.0
    ldloc.1
    calli      int32 (string[])
    pop
    ldc.i4.0
```

```

    stloc.0
    ldloc.0
    ret
}

```

Листинг 8 — Вызов метода по идентификатору на языке CIL

Виртуальный вызов (virtualcall). Вызов метода с заданным именем и сигнатурой, определяемой набором параметров, объявленного в классе объявления объекта, реализация которого выбирается во время выполнения на основе фактического типа объекта.

Виртуальный вызов применяется в том случае, когда одна и та же функция, интерфейс которой декларируется в базовом классе, реализуется в производных классах различными способами. В этом случае объявляется объект с типом базового класса (интерфейса), который инициализируется вызовом конструктора одного из производных классов либо некоторым иным методом, возвращающим объект одного из производных классов. Синтаксически в операторе вызова указывается метод базового класса, однако в процессе выполнения вызывается фактическая реализация этого метода в производном классе.

Пример виртуального вызова приведён на листингах **Ошибка! Не указано имя закладки.** и **Ошибка! Не указано имя закладки.**

```

varconfigBuilder = newConfigurationBuilder();

// В командной строке задаются конфигурационные файлы для
различных аспектов анализа
Array.ForEach(args, arg => configBuilder.AddJsonFile(arg));
IConfigurationRoot launchSettings = configBuilder.Build();

var binFolder = launchSettings["binFolder"];

```

Листинг 10 — Виртуальный вызов метода на языке C#

Здесь виртуальный вызов присутствует неявно при обращении к индексируемому свойству интерфейса `IConfigurationRoot`; вызов метода `Build` также виртуальный.

```

.locals init (
    [0] class Microsoft.Extensions.Configuration.
ConfigurationBuilder configBuilder,

```

```

    [1] class Microsoft.Extensions.Configuration.
    IConfigurationRoot launchSettings,
    [2] string binFolder,
    ...
)

newobj      instance void Microsoft.Extensions.Configuration.
ConfigurationBuilder::.ctor()
stloc.0
...
ldloc.0
callvirt   instance class Microsoft.Extensions.Configuration.
    IConfigurationRoot Microsoft.Extensions.Configuration.
    ConfigurationBuilder::Build()
stloc.1
ldloc.1
ldstr      "binFolder"
callvirt   instance string Microsoft.Extensions.Configuration.
    IConfiguration::get_Item(string)
stloc.1
...

```

Листинг 9 — Виртуальный вызов метода на языке CIL

Вызов посредством механизма рефлексии. Вызов метода `Type.InvokeMember` с указанием имени и параметров заданного метода (см. раздел 1.3.3), либо вызов метода `MethodBase.Invoke` для объекта `MethodBase`, соответствующего заданному методу.

Неявные вызовы. Такие вызовы отсутствуют в коде приложения; они осуществляются средой выполнения. К таким вызовам относятся вызов стартовой функции управляемого приложения, вызов статических конструкторов классов, инициализирующих статические поля, и т. п. Среда выполнения загружает сборку и с помощью рефлексии получает ссылку на стартовую функцию, которую затем вызывает с помощью метода `Invoke`. Непосредственно в коде приложения вызов стартовой функции отсутствует.

1.4.2. Варианты использования классов

Наследование. Такой вариант использования не является непосредственно обращением к коду класса, однако обозначает, что класс некоторым образом задействован в приложении и при анализе не может быть целиком признан избыточным.

Создание объектов. Создание объекта предполагает выделение памяти под данные объекта и инициализацию экземпляра посредством вызова конструктора. Пример конструкции создания объекта приведён на листингах 10 и 11.

```
var listener = new TextWriterTraceListener(System.Console.Out);
```

Листинг 10 — Создание объекта на языке C#

```
.locals init (  
[0] System.Diagnostics.TextWriterTraceListener listener  
)  
...  
newobj instance void  
System.Diagnostics.TextWriterTraceListener  
::ctor(class System.IO.TextWriter)  
stloc.0
```

Листинг 11 — Создание объекта на языке CIL

Присутствие в коде оператора создания объекта означает, что из методов данного класса используются, как минимум, конструктор и методы, вызываемые из него.

Вызов статических методов и экземплярных методов. Такое использование соответствует рассмотренному в предыдущем пункте.

Использование класса в качестве обобщённого (generic) параметра для обобщённого класса или метода. Пример такого использования показан на листинге **Ошибка! Не указано имя закладки..**

```
var recordsList = newList<Record>();
```

Листинг 12 — Пример использования класса в качестве обобщённого параметра при создании объекта другого класса

Обращения посредством механизма рефлексии. Получение объекта типа `Type`, соответствующего классу, и дальнейшее обращение к методам также посредством механизма рефлексии.

1.5. Обзор существующих программных средств, позволяющих выявлять избыточные функциональные объекты

В настоящий момент для выявления избыточных функциональных объектов в основном используется неавтоматизированный анализ кода.

Современные средства разработки приложений содержат инструменты, помогающие в решении этой задачи. Помимо полнотекстового поиска с возможностью использования регулярных выражений, такие инструменты включают поиск символа с заданным именем, перечисление всех обращений к символу и т. п.

Например, среда разработки *VisualStudio 2019* при выборе символа и нажатии на правую клавишу мыши предлагает команду «Найти все ссылки». Пример результата выполнения этой команды показан на рисунке **Ошибка! Не указано имя закладки..**

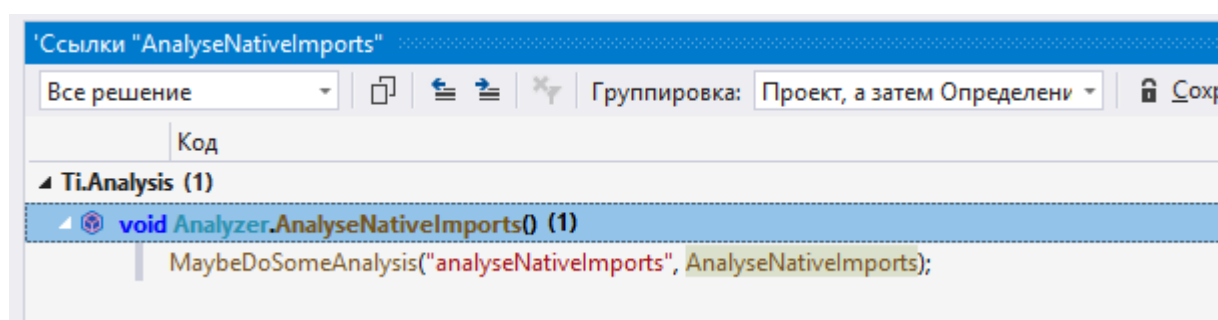


Рисунок 3 — Результат выполнения команды «Найти все ссылки»

Кроме того, среда *VisualStudio 2019*, а также редактор *VisualStudioCode* включают инструмент *CodeLens*, который показывает количество использований символа мелким шрифтом непосредственно в тексте программы (см. рисунок **Ошибка! Не указано имя закладки..**).

```

17 namespace Ti.Console
18 {
19     0 references
20     class Program
21     {
22         41 references
23         private static IConfigurationRoot launchSettings;
24
25         0 references
26         static void Main(string[] args)
27         {
28             Trace.Listeners.Add(new TextWriterTraceListener(System.Console.Out,
29             Trace.IndentSize = 2;

```

Рисунок 4 — Количество ссылок на символы, отображаемое инструментом CodeLens

На рисунке видно, что на класс Program и его метод Main в программном коде ссылки отсутствуют. В самом деле, метод Main является стартовым методом приложения. Такие методы, как было отмечено выше, непосредственно в программе не вызываются, их вызывает среда выполнения при инициализации приложения. По этой же причине отсутствуют ссылки на класс Program.

На рисунке 5 показан фрагмент объявления класса с указанием количества ссылок на различные методы этого класса.

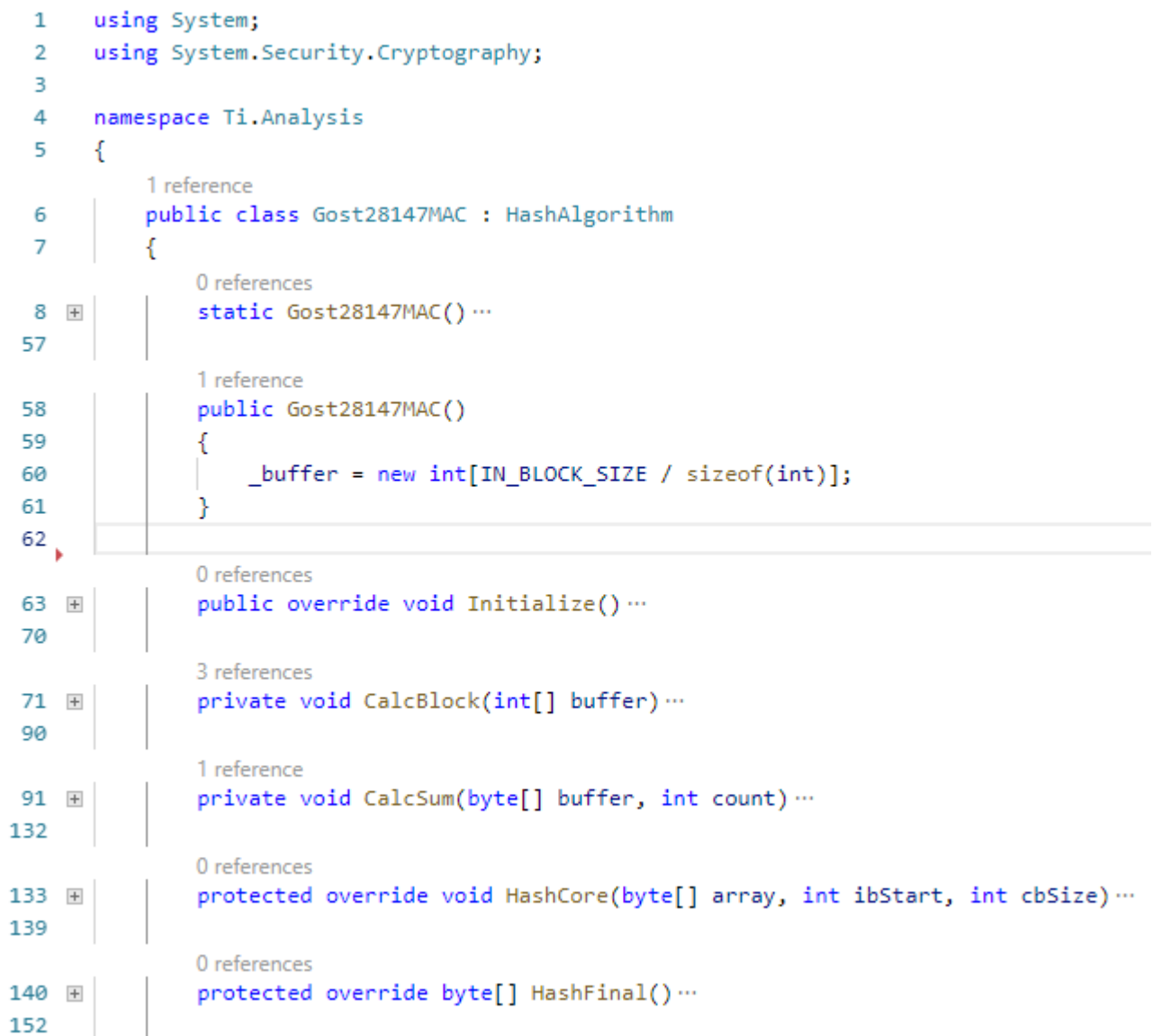


Рисунок 5 — Анализа кода с использованием CodeLens

Здесь показано, что на статический конструктор класса `Gost28147MAC` отсутствуют явные ссылки. Как было сказано выше, статические конструкторы вызываются средой выполнения во время инициализации класса.

При этом, к не статическому конструктору обращения имеются. Если щёлкнуть мышью по соответствующей надписи, эти обращения будут показаны (см. рисунок 6).

Закрытые методы CalcBlockи CalcSum вызываются внутри самого классаGost28147MAC, а виртуальные методыInitialize, HashCoreи



Рисунок 6 — Анализ обращения к конструктору с помощью CodeLens

HashFinal — показанные на рисунке как невызываемые, — на самом деле вызываются внутренними методами базового класса HashAlgorithm.

Таким образом, использование механизма CodeLens предполагает интерактивный анализ, а показания CodeLens не всегда соответствуют действительности.

Возможность автоматизированного поиска неиспользуемых классов и методов присутствует в расширении для VisualStudio— программе *ReSharper*. Программа используется в том числе для проверки качества кода и содержит множество правил для выявления проблем в коде приложения, сгруппированных по языкам программирования и по категориям.

На рисунках **Ошибка! Не указано имя закладки.** и **Ошибка! Не указано имя закладки.** приведён пример использования ReSharper. На рисунке **Ошибка! Не указано имя закладки.** показано окно выбора правил,

в котором для программ на языке C# отмечен набор правил *RedundanciesinSymbolDeclarations*, соответствующий различным вариантам избыточности. На рисунке **Ошибка! Не указано имя закладки.** показан результат поиска: найден один неиспользуемый класс и два неиспользуемых метода.

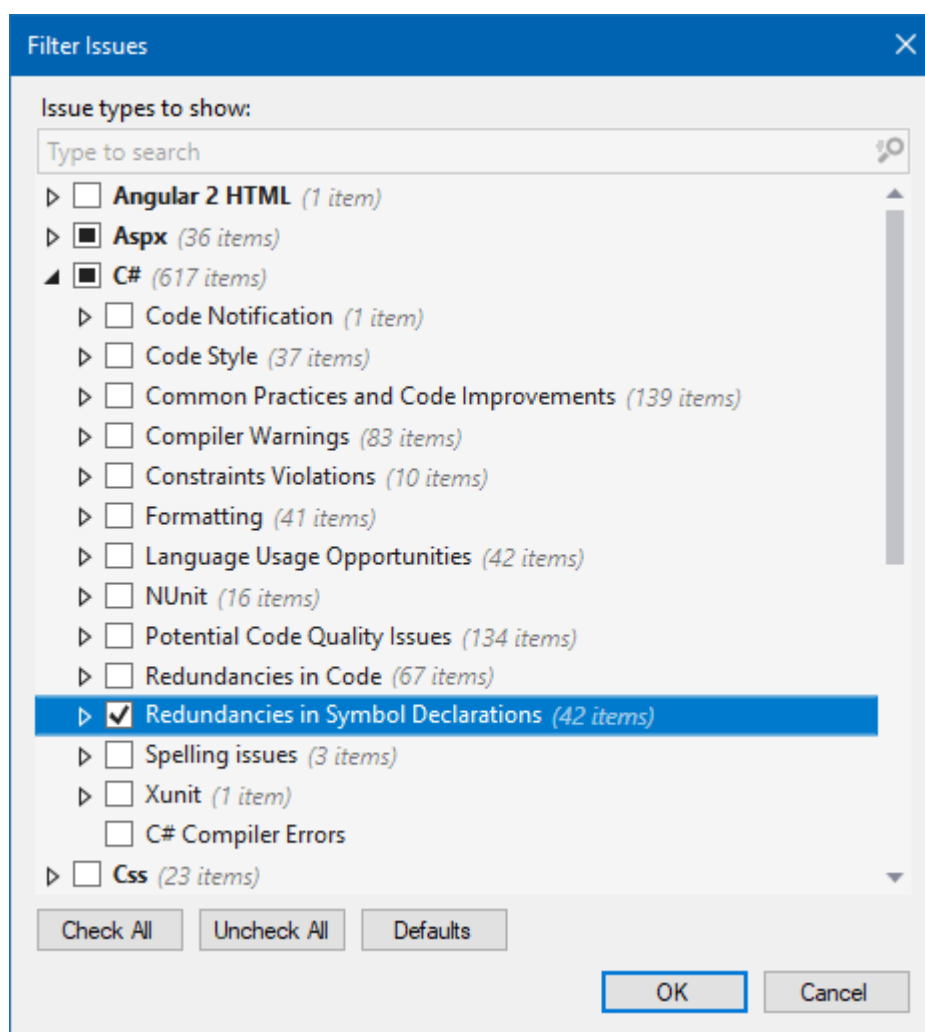


Рисунок 8 — Выбор правил для поиска избыточности в программе ReSharper

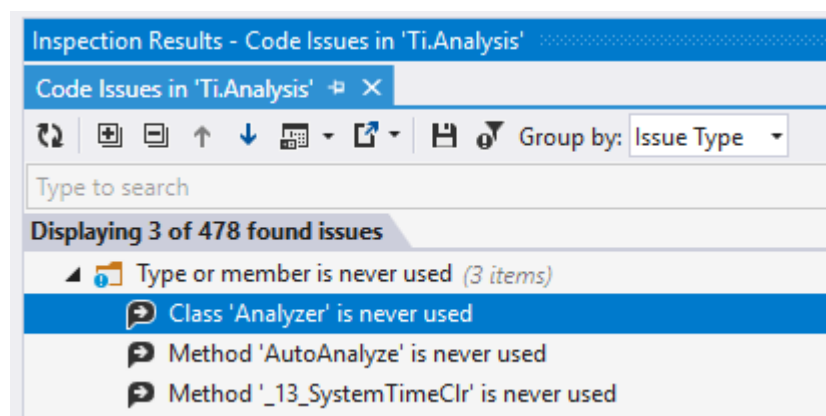


Рисунок 7— Результат поиска избыточных классов и методов

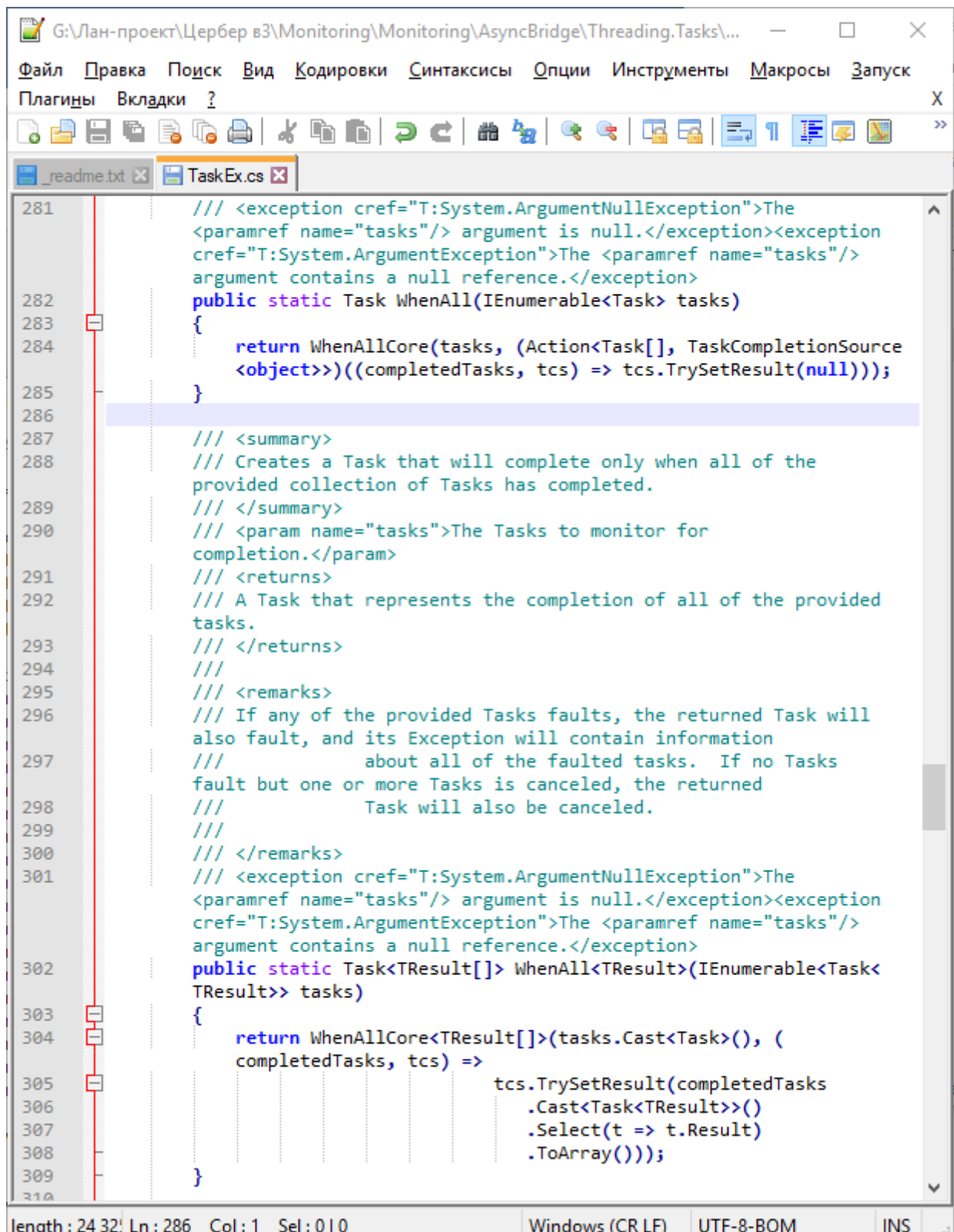
Следует отметить, что ReSharper предоставляет возможность анализа качества кода только на поддерживаемых языках программирования. Кроме того, анализ проводится только в рамках одного решения (VisualStudioSolution), что не всегда возможно организовать на практике.

Ещё одним инструментом для всестороннего анализа исходных текстов является программа *SciToolsUnderstand*. Среди возможностей программы присутствует генерация отчётов с перечнем неиспользуемых типов и функций. Пример такого отчёта приведён на рисунке **Ошибка! Не**

Unused Program Units Report	
=====	
AsyncBridge\AsyncCompatLibExtensions.cs	
AsyncCompatLibExtensions.ConfigureAwait	64
AsyncCompatLibExtensions.ConfigureAwait	81
AsyncCompatLibExtensions.GetAwaiter	31
AsyncCompatLibExtensions.GetAwaiter	47
AsyncBridge\Threading.Tasks\TaskEx.cs	
System.Threading.Tasks.TaskEx.(anon_method_1)	23
System.Threading.Tasks.TaskEx.Delay	149
System.Threading.Tasks.TaskEx.Delay	163
System.Threading.Tasks.TaskEx.Delay.(anon_method_1)	207
System.Threading.Tasks.TaskEx.Delay.(anon_method_2)	216
System.Threading.Tasks.TaskEx.Run	44
System.Threading.Tasks.TaskEx.Run	70
System.Threading.Tasks.TaskEx.Run	96
System.Threading.Tasks.TaskEx.Run	122
System.Threading.Tasks.TaskEx.WhenAll	242
System.Threading.Tasks.TaskEx.WhenAll	262
System.Threading.Tasks.TaskEx.WhenAll.(anon_method_1)	284
System.Threading.Tasks.TaskEx.WhenAll.(anon_method_1)	304
System.Threading.Tasks.TaskEx.WhenAll.(anon_method_1).(anon_method_2)	307
System.Threading.Tasks.TaskEx.WhenAllCore.(anon_method_1)	330
System.Threading.Tasks.TaskEx.WhenAny	364
System.Threading.Tasks.TaskEx.WhenAny	404
System.Threading.Tasks.TaskEx.Yield	455
BaseLibrary.UI\Converters\BooleanNegationConverter.cs	
BaseLibrary.UI.Converters.BooleanNegationConverter.Convert	19
BaseLibrary.UI\Converters\BooleanToVisibilityConverter.cs	
BaseLibrary.UI.Converters.BooleanToVisibilityConverter.Convert	20

Рисунок 9 — Отчёт с перечнем неиспользуемых функций, сформированный программой Understand

указано имя закладки.



```
281      /// <exception cref="T:System.ArgumentNullException">The
282      <paramref name="tasks"/> argument is null.</exception><exception
283      cref="T:System.ArgumentException">The <paramref name="tasks"/>
284      argument contains a null reference.</exception>
285      public static Task WhenAll(IEnumerable<Task> tasks)
286      {
287          return WhenAllCore(tasks, (Action<Task[], TaskCompletionSource
288          <object>>)((completedTasks, tcs) => tcs.TrySetResult(null)));
289      }
290
291      /// <summary>
292      /// Creates a Task that will complete only when all of the
293      /// provided collection of Tasks has completed.
294      /// </summary>
295      /// <param name="tasks">The Tasks to monitor for
296      /// completion.</param>
297      /// <returns>
298      /// A Task that represents the completion of all of the provided
299      /// tasks.
300      /// </returns>
301      /// <remarks>
302      /// If any of the provided Tasks faults, the returned Task will
303      /// also fault, and its Exception will contain information
304      /// about all of the faulted tasks. If no Tasks
305      /// fault but one or more Tasks is canceled, the returned
306      /// Task will also be canceled.
307      /// </remarks>
308      /// <exception cref="T:System.ArgumentNullException">The
309      <paramref name="tasks"/> argument is null.</exception><exception
310      cref="T:System.ArgumentException">The <paramref name="tasks"/>
311      argument contains a null reference.</exception>
312      public static Task<TResult> WhenAll<TResult>(IEnumerable<Task<
313      TResult>> tasks)
314      {
315          return WhenAllCore<TResult>(tasks.Cast<Task>(), (
316          completedTasks, tcs) =>
317              tcs.TrySetResult(completedTasks
318                  .Cast<Task<TResult>>()
319                  .Select(t => t.Result)
320                  .ToArray()));
321      }
```

Рисунок 10 — Анонимные методы, ошибочно показанные неиспользуемыми

Программа Understand поддерживает возможность автоматизации в формате написания скриптов на языках Perl и Python. Существенным недостатком программы является достаточно большое количество ошибок.

Так, например, анонимные функции внутри метода `WhenAll` (строки 284 и 304) показаны как неиспользуемые, в то время как из исходного кода видно, что они передаются в качестве параметров в другие методы (см. рисунок **Ошибка! Не указано имя закладки.**).

Среди декомпиляторов и анализаторов исполнимого кода были рассмотрены *Reflector.NET* и *dnSpy*. Обе программы предоставляют одинаковые возможности по поиску обращений к заданному классу либо методу. При этом обе программы предполагают интерактивное применение. Примеры использования программ приведены на рисунках **Ошибка! Не указано имя закладки.** и **Ошибка! Не указано имя закладки.**.

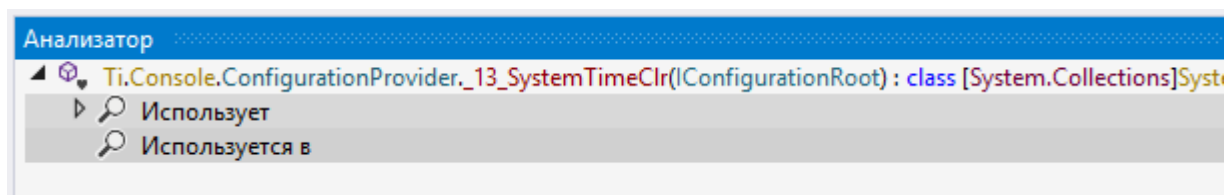


Рисунок 11 — Поиск обращений к методу с помощью программы dnSpy

1.5.1. Вывод

Таким образом, все существующие в настоящее время программные средства, позволяющие выявлять избыточность приложений для платформы Microsoft .NET на уровне функциональных объектов, либо являются

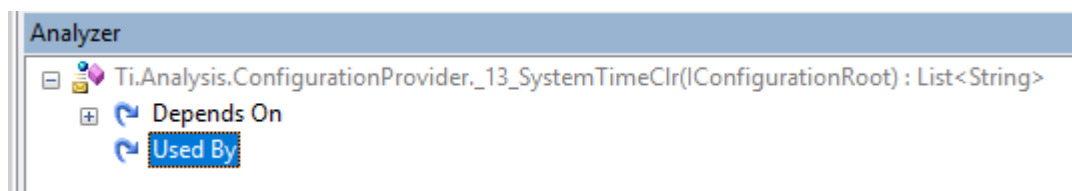


Рисунок 12 — Поиск обращений к методу с помощью программы Reflector.NET

интерактивными по своей сути, либо требуют дополнительного интерактивного анализа результатов их работы вследствие недостаточного качества анализа. Автоматизированные решения отсутствуют.

1.6. Выводы по главе 1

Был проведён анализ задачи выявления избыточного кода в приложениях, предназначенных для выполнения под управлением платформы Microsoft .NET, дано определение и приведена классификация избыточного кода. Были рассмотрены подходы к выявлению избыточного кода.

Также были изучены особенности структурирования кода приложений, предназначенных для выполнения в среде Microsoft .NET, рассмотрены различные виды функциональных объектов и варианты их использования.

Был проведён анализ имеющихся в настоящее время средств, которые могут быть использованы при выявлении избыточных функциональных объектов в приложениях для платформы Microsoft .NET и сделан вывод об отсутствии автоматизированного средства, применимого при решении данной задачи.

2. РАЗРАБОТКА АНАЛИТИЧЕСКОЙ СИСТЕМЫ ВЫЯВЛЕНИЯ ИЗБЫТОЧНОСТИ В УПРАВЛЯЕМЫХ ПРИЛОЖЕНИЯХ

2.1. Постановка задачи на разработку автоматизированного средства выявления избыточности

Под неиспользуемыми функциональными объектами будем понимать методы управляемых классов, вызовов которых (прямых, виртуальных, косвенных, посредством отражения) не содержится в некоторой совокупности исполнимых файлов, которую назовём исследуемым программным обеспечением.

Функциональный объект-метод класса является используемым, если в коде присутствуют обращения к этому методу одного или нескольких следующих видов:

- прямой вызов статического либо экземплярного метода, в т. ч. виртуального;
- вызов виртуального метода базового класса (интерфейса), если в коде создаётся объект рассматриваемого класса и этот объект может появляться в месте вызова;
- получение указателя на метод (косвенный вызов метода).

Функциональный объект-класс является используемым, если в коде присутствует:

- создание объекта этого класса;
- вызов статического метода этого класса;
- некоторый метод, принимающий либо возвращающий объект этого класса;
- некоторый тип либо метод, обобщённым параметром (generic) которого является данный класс;
- унаследованный от него другой класс.

Функциональные объекты, не подходящие ни под один из этих критериев, являются неиспользуемыми. Также неиспользуемыми считаются

функциональные объекты, используемые только в таких функциональных объектах, которые в свою очередь являются неиспользуемыми. Таким образом, процесс выявления неиспользуемых — избыточных — функциональных объектов является итеративным.

Специальным видом использования является создание объектов и вызов методов посредством механизма отражения (Reflection). При этом, если используемые функциональные объекты идентифицируются именами, факт такого использования может быть детектирован автоматически. Если же функциональные объекты выбираются из некоторого множества в соответствии с определёнными критериями, связь их с объявлением в коде не может быть установлена автоматически. В этом случае для обозначения используемости функционального объекта требуется использование дополнительных маркеров. В сборках для платформы .NET в качестве таких маркеров могут выступать настраиваемые атрибуты (CustomAttributes) специального типа.

2.2. Выбор подхода к решению задачи

По результатам анализа подходов к анализу программного кода на предмет выявления избыточности было принято решение о применении в аналитической системе подхода, основанного на анализе исполнимых файлов.

Решение обусловлено следующими соображениями.

- 1) Анализ может проводиться для исполнимых файлов, скомпилированных из исходного кода на любом языке программирования.
- 2) Анализ могут быть подвергнуты исполнимые файлы стороннего производства, не имеющие исходных текстов.
- 3) Исполнимые файлы имеют точно определённую структура как на уровне метаданных, так и на уровне тел методов, что существенно упрощает разбор данных при подготовке к анализу.

4) Наличие свободно распространяемых инструментов для разбора метаданных и тел методов и простота их применения.

2.3. Алгоритмы решения задачи

Ниже приведены блок-схемы алгоритмов работы аналитической системы.

На блок-схемана рисунке **Ошибка! Не указано имя закладки.**представлен общий алгоритм работы системы. Он выполняется для всех сборок, входящих в исследуемое программное обеспечение. Анализируется состав сборки и определённые в ней типы.

Если тип является классом, выполняется алгоритм анализа класса, представленный на блок-схеме на рисунке **Ошибка! Не указано имя закладки.** Алгоритм отмечает как используемый базовый класс, от которого унаследован заданный, если базовый класс входит в состав исследуемого программного обеспечения.

Для всех методов, определённых в классе, выполняется алгоритм анализа метода, представленный на блок-схеме на рисунке **Ошибка! Не указано имя закладки.** Ко всем типам, содержащимся в сигнатуре (типы параметров и возвращаемого значения), и конкретным типам обобщённых параметров, применяется алгоритм анализа типа, представленный на блок-схеме на рисунке **Ошибка! Не указано имя закладки.**, который отмечает тип как используемый.

Если метод имеет реализацию, ко всем его инструкциям применяется алгоритм анализа инструкции, представленный на блок-схеме на рисунке **Ошибка! Не указано имя закладки.** Для инструкций, операндом которых является токен метода (инструкция вызова метода, инструкция создания объекта и вызова конструктора, инструкция получения указателя на метод) выполняется анализ метода, к которому выполняется обращение. Этот метод отмечается как используемый; кроме того, выполняется анализ конкретных

типов обобщённых параметров (см. блок-схему на рисунке **Ошибка! Не
указано имя закладки.**).

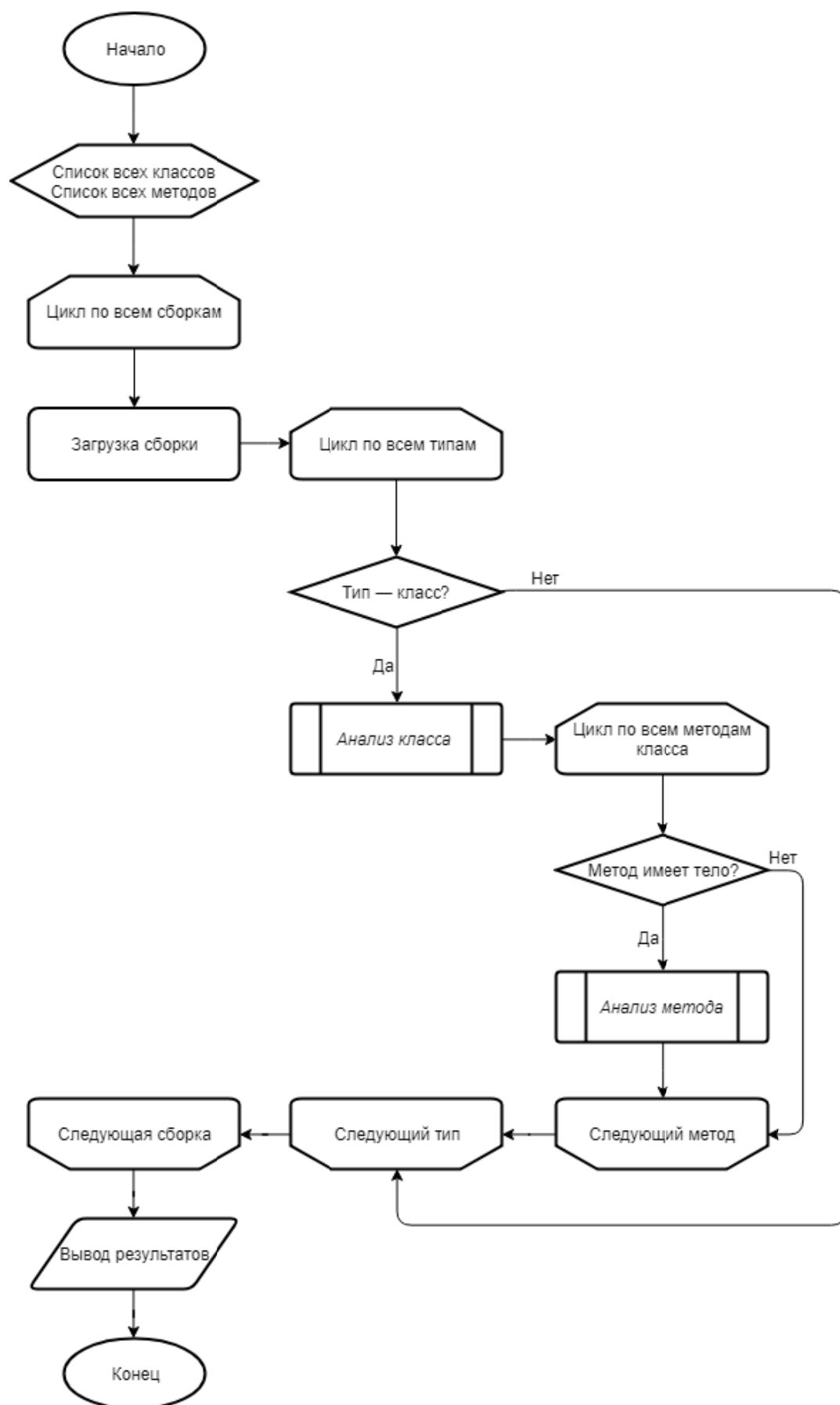


Рисунок13 — Основной алгоритм работы аналитической системы

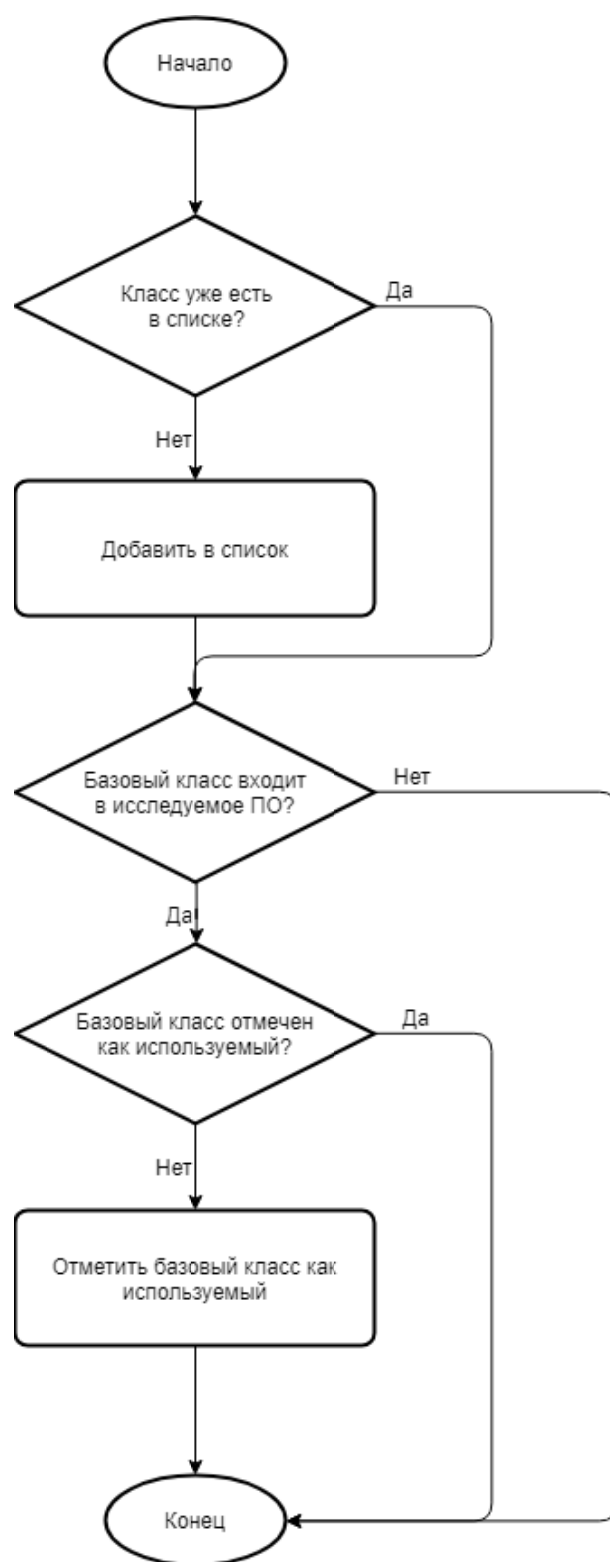


Рисунок14 — Алгоритм анализа класса, содержащегося в сборке

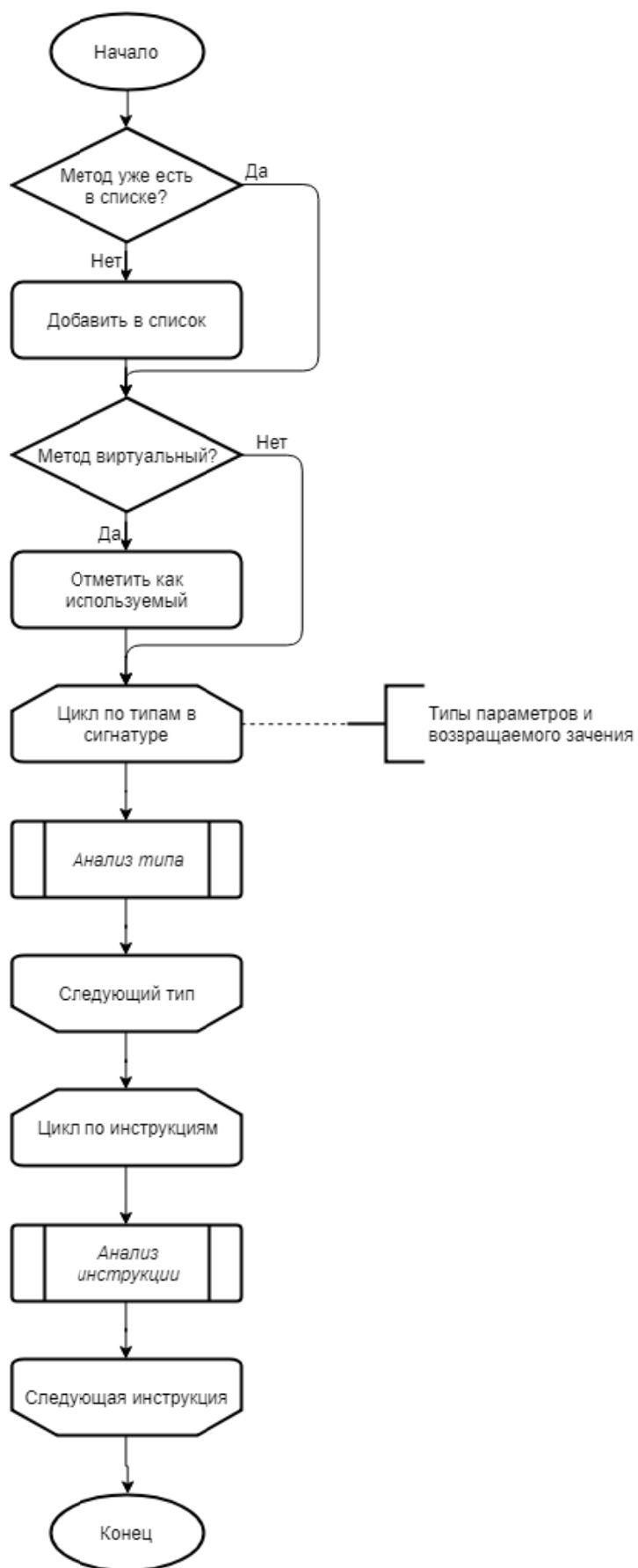


Рисунок15 — Алгоритм анализа метода класса, содержащегося в сборке

Рисунок16 — Алгоритм анализа типа аргумента, возвращаемого значения
либо обобщённого параметра



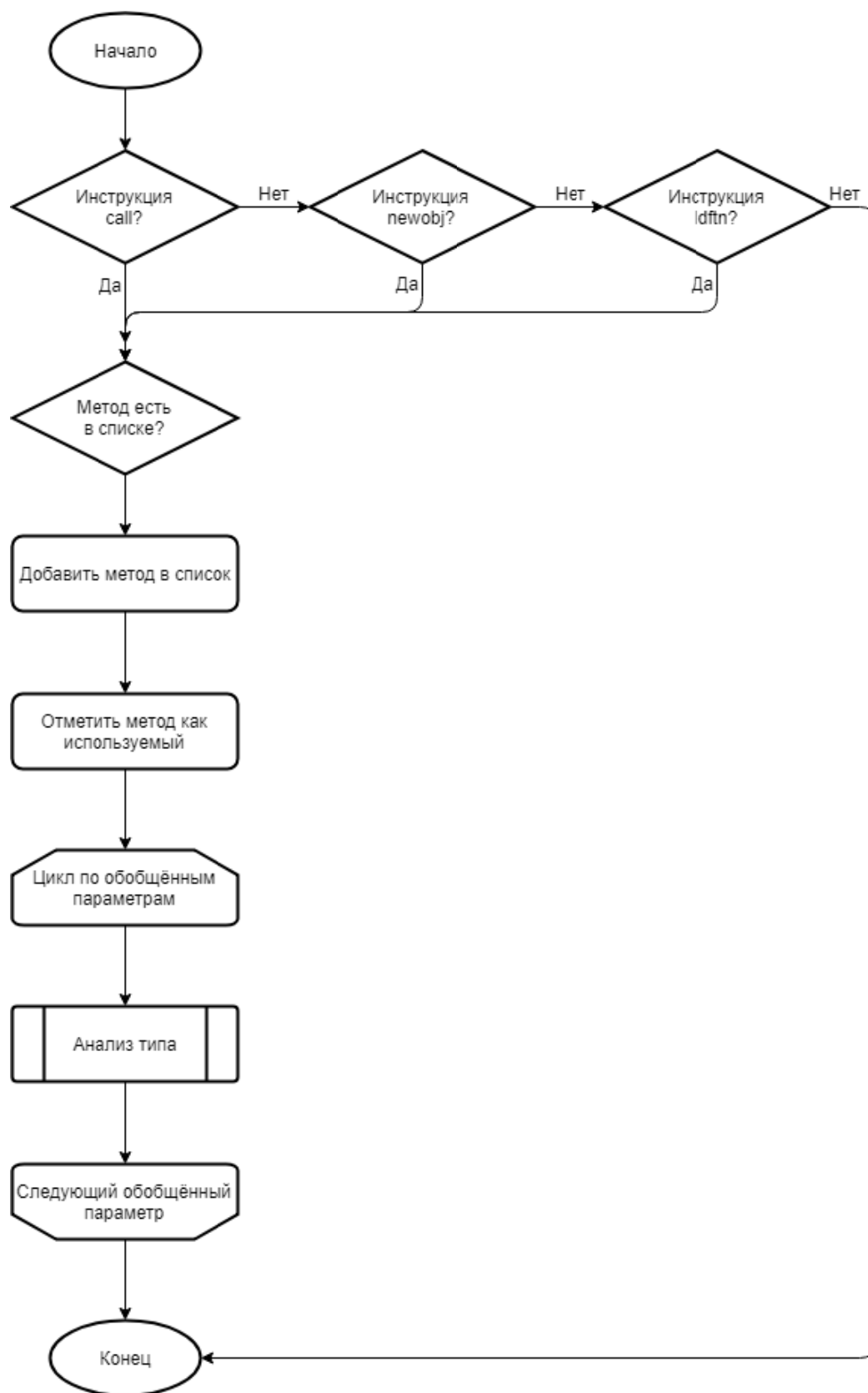


Рисунок17 — Алгоритм анализа инструкции

2.4. Реализация аналитической системы

Приведённые выше алгоритмы работы аналитической системы реализованы в виде приложения, предназначенного для выполнения под управлением платформы Microsoft .NET версии не ниже 4.0. Приложение написано на языке C# в среде Microsoft Visual Studio 2019 Community.

В качестве программного интерфейса для доступа к метаданным сборок используется свободно распространяемая библиотека с открытым кодом dnlib (<https://github.com/0xd4d/dnlib>). Библиотека предлагает возможность загрузки и разбора метаданных в сборках, предназначенных для различных версий среды выполнения.

Аналитическая система поставляется в виде установочного дистрибутива для операционной системы Microsoft Windows 7 и более новых.

2.5. Выводы по главе 2

Была сформулирована задача выявления избыточных функциональных объектов в приложениях, предназначенных для выполнения под управлением платформы Microsoft .NET, представлены критерии отнесения функционального объекта к категории избыточного. Был обоснован выбор подхода к выявлению избыточных функциональных объектов.

Была разработана аналитическая система поиска избыточных функциональных объектов в исполняемых файлах, разработанных для платформы Microsoft .NET, способная в автоматическом режиме выявлять неиспользуемые функциональные объекты в некоторой совокупности сборок, составляющих исследуемое программное обеспечение.

3. МЕТОДИКА ВЫЯВЛЕНИЯ ИЗБЫТОЧНОСТИ С ПРИМЕНЕНИЕМ АНАЛИТИЧЕСКОЙ СИСТЕМЫ

3.1. Подготовка входных данных. Проведение анализа

В качестве входных данных для проведения анализа указывается набор сборок, составляющих исследуемое программное обеспечение, объединённых в некоторую папку.

Для ускорения анализа предварительно рекомендуется провести анализ связей между сборками на предмет выявления избыточности на уровне сборок. Такая избыточность может возникнуть в процессе разработки программного обеспечения в результате использования специальных сборок, содержащих тесты, эволюции кода, а также ошибочно определения набора связанных сборок. Выявленные неиспользуемые сборки рекомендуется исключить из состава исследуемого программного обеспечения.

Для запуска аналитической системы следует активировать соответствующий значок на рабочем столе. В появившемся окне следует нажать кнопку «Папка» и в появившемся диалоговом окне выбрать папку, содержащую сборки исследуемого программного обеспечения. После этого следует нажать кнопку «Далее».

Ход анализа отображается в главном окне аналитической системы в разделе «Ход анализа».

По окончании анализа в главном окне аналитической системы отображаются его результаты. Результаты можно сохранить на диск нажав кнопку «Сохранить».

3.2. Результаты работы аналитической системы. Устранение избыточности и проведение повторного анализа

По результатам работы аналитической системы может быть выявлена избыточность на уровне функциональных объектов.

Для повышения качества программного обеспечения выявленную избыточность рекомендуется устранить.

В случае наличия исходных текстов сборки, содержащей избыточные функциональные объекты, рекомендуется исключить избыточные функциональные объекты из исходных текстов, после чего провести повторную компиляцию программного обеспечения.

В случае, если избыточные функциональные объекты выявлены в сборке стороннего производителя, для которой отсутствуют исходные тексты, удаление можно произвести на уровне исполнимого файла с помощью программы dnSpy (<https://github.com/0xd4d/dnSpy>). Для этого следует загрузить сборку в программу, найти класс или метод, указанный аналитической системой как избыточный, нажать правую клавишу мыши и в появившемся контекстном меню выбрать пункт «Удалить». После этого следует сохранить модифицированную сборку выбрав пункт меню «Файл» → «Сохранить».

После удаления из состава исследуемого программного обеспечения всех функциональных объектов, указанных аналитической системой как избыточные, могут появиться другие избыточные функциональные объекты. Такая ситуация может возникнуть в том случае, если на некоторый функциональный объект ранее ссылались только функциональные объекты, признанные избыточными и удалённые по результатам анализа.

В такой ситуации рекомендуется провести повторный анализ исследуемого программного обеспечения с применением аналитической системы, а также повторное устранение избыточных функциональных объектов.

3.3. Ограничения на применение аналитической системы.

Направления дальнейшего развития

Очевидным ограничением аналитической системы является невозможность обнаружения связей между функциональными объектами, реализованных с применением механизма рефлексии.

Для исключения такого ограничения планируется разработка механизма маркеров, которыми в исходных текстах должны отмечаться функциональные объекты, используемые исключительно посредством механизма рефлексии. В качестве таких маркеров планируется использование настраиваемых атрибутов (CustomAttributes) специального типа. Планируется доработка аналитической системы с целью поддержки таких маркеров.

Отмеченная в предыдущем параграфе необходимость повторного проведения анализа после устранения избыточных функциональных объектов является значительным неудобством при применении аналитической системы. Планируется углубление и усовершенствование алгоритмов анализа для автоматического выявления ситуаций, когда некоторый функциональный объект используется только в других — избыточных — функциональных объектах.

Углубление алгоритмов анализа планируется также применительно к виртуальным методам. Планируется разработка алгоритмов анализа потока данных на предмет установления связей между созданием объекта, содержащего виртуальные методы, и вызовом этих методов. Такие алгоритмы позволят в точке виртуального вызова иметь информацию о всех возможных типах объекта, к которому применяется вызов. Как следствие, в данной точке используемыми могут быть отмечены только методы таких типов, а не всех типов, реализующих данный виртуальный метод.

3.4. Выводы по главе 3

Была разработана методика применения аналитической системы для проведения поиска избыточных функциональных объектов в исполняемых файлах, предназначенных для выполнения под управлением платформы Microsoft .NET, включающая подготовку входных данных, проведение анализа и оценку его результатов.

Были рассмотрены ограничения на применение аналитической системы и представлены возможные направления её дальнейшего развития.

ЗАКЛЮЧЕНИЕ

В ходе работы был проведён анализ задачи выявления избыточного кода в приложениях, предназначенных для выполнения под управлением платформы Microsoft .NET, дано определение и приведена классификация избыточного кода. Были рассмотрены подходы к выявлению избыточного кода.

В рамках работы были изучены особенности структурирования кода приложений, предназначенных для выполнения в среде Microsoft .NET, рассмотрены различные виды функциональных объектов и варианты их использования.

Была разработана аналитическая система поиска избыточных функциональных объектов в исполняемых файлах, разработанных для платформы Microsoft .NET, способная в автоматическом режиме выявлять неиспользуемые функциональные объекты в некоторой совокупности сборок, составляющих исследуемое программное обеспечение.

Также была разработана методика применения аналитической системы для проведения поиска избыточных функциональных объектов в исполняемых файлах, предназначенных для выполнения под управлением платформы Microsoft .NET.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Автоматизация поиска избыточных функциональных объектов в исполнимых файлах для платформы Microsoft .NET. Д.В. Сафронов, А.В. Дубравин — Пензенский государственный университет, 2017 г.
2. Разработка методов исключения избыточных функциональных объектов в исполняемых файлах, разработанных для платформы Microsoft .NET. Д.В. Сафронов, Д.В. Ломизов, А.В. Дубравин — Пензенский государственный университет, 2019 г.
3. Standard ECMA-334 (4th Edition / June 2006). C# Language Specification
4. Standard ECMA-335 (4th Edition / June 2006). CommonLanguage Infrastructure (CLI)
5. CLR via C#, Jeffrey Richter, Microsoft Press
6. Microsoft Developer Network Library, msdn.microsoft.com
7. CERT. www.cert.org
8. Зегжда Д.П., Ивашко А.М. Как построить защищенную информационную систему. Технология создания безопасных систем/ Под ред. П.Д. Зегжды, В.В. Платонова - СПб.: НПО Мир и Семья-95, 1997.
9. Зегжда Д.П., Ивашко А.М. Основы безопасности информационных систем. - М.: Горячая линия - Телеком, 2000. 452 с., ил
10. Гайдамакин Н.А. Разграничение доступа к информации в компьютерных системах. - Екатеринбург: Изд-во Урал. гос. ун-та, 2003.
11. Гайдамакин Н.А. Автоматизированные информационные системы, базы и банки данных. - М.: Гелиос АРВ, 2002.
12. Щеглов А.Ю. Защита компьютерной информации от несанкционированного доступа. - СПб: Наука и Техника, 2004. - 384 с.: ил.
13. Бабенко Л.К., Басан А.С., Макаревич О.Б. Разработка мандатной модели доступа и анализ существующих моделей в современных

СУБД // Материалы международной конференции. Ноябрь 2006, Минск (Л.К. - Нальчик).

14. M. Harrison, W.Ruzzo, J. Uhlman "Protection in operation systems", Communications of the ACM, 1976

15. M. Harrison, W.Ruzzo "Monotonic protection systems", Foundation of secure computation, 1978

16. Joshi J. B.D., Aref W.G., Ghafoor A. and Spafford E.H. Security models for web- based applications. Communications of the ACM, 44(2).Pp. 38-44, February, 2001

17. Брагг Р. Система безопасности Windows 2000.: Пер. с англ. - М.:Издательский дом «Вильямс», 2001. - 592 с.

18. Девянин П.Н. Анализ безопасности управления доступом информационными потоками в компьютерных системах. Учеб. пособие для вузов. -М.: Радио и связь, 2006.-176 с.

19. Гостехкомиссия России. Руководящий документ. Средства вычислительной техники. Защита от несанкционированного доступа к информации. Показатели защищенности от несанкционированного доступа к информации. - М., 1992.

20. Гостехкомиссия России. Руководящий документ. Автоматизированные системы. Защита от несанкционированного доступа к информации. Классификация автоматизированных систем и требования по защите информации. - М., 1992.

21. D. Elliot Bell, Leonard J. LaPadula // "Secure Computer Systems: Mathematical Foundations". MITRE Technical Report 2547, 1973. Vol. II.

22. Описание СУБД Линтер. Официальный сайт компании РЕЛЭКС. - http://www.relex.ru/linter_rus.php

23. Theriault M., Newman A. Oracle Security Handbook. - Osborn/McGraw, Oracle Press Inc., 2001.

24. Theriault M., Heney W. Oracle Security. - O'Reilly & Associates, Inc., 1998

25. Oracle Label Security. Официальный сайт компании Oracle, <http://www.oracle.com/database/label-security.html>

26. Пржиялковский В. Изучаем метки доступа к строкам: задание свойств столбца доступа к таблице.-<http://www.citforum.ru/database/oracle/LearnOLS1>

27. Ferraiolo D. F., Kuhn D. R. (October 1992). "Role Based Access Control". 15th National Computer Security Conference: 554—563.

28. Sandhu R., Coyne E. J., Feinstein H. L., Youman C. E. (August 1996). «Role-Based Access Control Models». IEEE Computer (IEEE Press) 29(2): 38—47.

29. Silvia Castano, Mariagrazia Fugini, Giancarlo Martella, Pierangela Samarati. "Database security". Addison-Wesley. 1994.

Приложение. Презентация

Тема работы: Аналитическая система поиска избыточных функциональных объектов в исполняемых файлах, разработанных для платформы Microsoft .NET

Автор работы: магистрант группы 17ВВм1
Сафронов Д.В.

Научный руководитель: к.т.н., доцент Дубравин
А.В.

Цель работы

Разработка методов автоматизации поиска избыточных функциональных объектов в исполняемых файлах, разработанных для платформы Microsoft .NET

Основные задачи

- определение и классификация избыточного кода, рассмотрение подходов к выявлению избыточного кода
- изучение особенностей структурирования кода приложений, предназначенных для выполнения в среде Microsoft .NET, рассмотрение различных видов функциональных объектов и вариантов их использования
- разработка аналитической системы поиска избыточных функциональных объектов в исполняемых файлах
- разработка методики применения аналитической системы для проведения поиска избыточных функциональных объектов

Актуальность работы

Высокие темпы разработки современного программного обеспечения требуют автоматизации проверки качества кода.

Возрастающие объёмы программных продуктов приводят к снижению качества, в частности, в появлению фрагментов кода, которые не используются в остальных частях ПО.

Это отрицательно сказывается:

- на производительности
- на безопасности
- на возможности поддержки

Актуальность работы (продолжение)

- Microsoft .NET — актуальная платформа для разработки
- Языки программирования в основном объектно-ориентированные и функциональные, следовательно, высокоуровневые
- Применяются шаблоны проектирования
- Редко применяется ручная оптимизация кода

Функциональный объект

- Функция, метод класса
- Класс (т.е. тип, имеющий реализацию)

Варианты использования функций и методов

- *Прямой вызов (direct call)*. Непосредственный вызов конкретного метода у заданного объекта
- *Косвенный вызов (indirect call)*. Вызов метода по идентификатору, предварительно помещённому в некоторую переменную
- *Виртуальный вызов (virtual call)*. Вызов метода, реализация которого выбирается во время выполнения на основе фактического типа объекта
- *Вызов посредством механизма рефлексии*
- *Неявные вызовы*. Осуществляются средой выполнения

Варианты использования классов

- *Наследование.* Не является непосредственно обращением к коду класса; обозначает, что класс не может быть целиком признан избыточным
- *Создание объектов.* Инициализация экземпляра посредством вызова конструктора
- *Вызов статических методов и экземплярных методов*
- *Использование класса в качестве обобщённого (generic) параметра для обобщённого класса или метода*
- *Обращения посредством механизма рефлексии*

Использование функционального объекта

- В текущем модуле
- В других модулях исследуемого ПО
- В стороннем ПО — требуется дополнительная маркировка открытых функциональных объектов

Обзор имеющихся средств

- Visual Studio 2019 — «Найти все ссылки»
- Visual Studio 2019, Visual Studio Code — CodeLens
- JetBrains ReSharper
- SciTools Understand
- Декомпиляторы Reflector.NET и dnSpy

Результаты

```
<Unused>
<Classes>
  <Class Name="CirconRegistry.Program" File="Program.cs" Line="8" />
  <Class Name="Circon.Administration.AuthDataExtensions"
File="AuthDataExtensions.cs" Line="7" />
  ...
</Classes>
<Methods>
  <Method
Name="Circon.Registry.FormsImportWizard.FormsImportWizardViewModel.AreParametersValid"
File="FormsImportWizardViewModel.cs" Line="143" />
  <Method
Name="Circon.Registry.FormsImportWizard.FormsImportWizardViewModel.Browse"
File="FormsImportWizardViewModel.cs" Line="99" />
  ...
  <Method
```

Методика применения аналитической системы

- Исключение избыточных сборок
- Проведение анализа
- Устранение избыточности
- Повторный анализ

Полученные результаты

- Разработана аналитическая система поиска избыточных функциональных объектов в исполняемых файлах для платформы Microsoft .NET, способная в автоматическом режиме выявлять неиспользуемые функциональные объекты исследуемом программном обеспечении
- Разработана методика применения аналитической системы для проведения поиска избыточных функциональных объектов в исполняемых файлах, предназначенных для выполнения под управлением платформы Microsoft .NET

Ограничения и направления дальнейшего развития

- Невозможность выявления использования посредством механизма рефлексии — разработка механизма маркеров
- Усовершенствование алгоритмов анализа для автоматического выявления ситуаций, когда функциональный объект используется только в других — избыточных — функциональных объектах
- Разработка алгоритмов анализа потока данных на предмет установления связей между созданием объекта, содержащего виртуальные методы, и вызовом этих методов

Спасибо за внимание!

УТВЕРЖДАЮ

зам. директора

ООО НТП «Криптософт»

С.Н. Миронов

2019 г.



АКТ

о внедрении результатов выпускной магистерской диссертации Сафронова Д.В.

Комиссия в составе: председатель — заместитель директора ООО НТП «Криптософт» Миронов Сергей Николаевич; члены комиссии — начальник отдела Хоренко Денис Анатольевич, ведущий специалист Хоменко Алексей Владимирович, настоящим актом подтверждают, что результаты магистерской диссертации Сафронова Д.В. были внедрены в составе работ ООО НТП «Криптософт».

Результаты работы были использованы на этапе разработки и изготовления опытного образца и получили положительную оценку.

Подписи членов комиссии:

С.Н. Миронов

Д.А. Хоренко

А.В. Хоменко