Министерство образования и науки Российской Федерации

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

“САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ”

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА

«РАЗРАБОТКА СИСТЕМЫ АВТОМАТИЗИРОВАННОГО АНАЛИЗА РЕПОЗИТОРИЕВ С ОТКРЫТЫМ ИСХОДНЫМ КОДОМ ПРИ ПОМОЩИ СТАТИЧЕСКИХ АНАЛИЗАТОРОВ КОДА»

Автор Припадчев Артём Александрович _____
(Фамилия, Имя, Отчество) (Подпись)

Направление подготовки (специальность)
09.04.04 Программная инженерия

Квалификация магистр
(бакалавр, магистр)

Руководитель Радченко И.А., к.т.н., доцент _____
(Фамилия, И., О., ученое звание, степень) (Подпись)

К защите допустить

Зав. кафедрой Муромцев Д.И., к.т.н., доцент _____
(Фамилия, И., О., ученое звание, степень) (Подпись)

“_____” _____ 20 ____г.

Санкт-Петербург, 2018 г.

Студент Припадчев А.А. Группа Р4217 Кафедра ИПМ Факультет ПИиКТ
(Фамилия, И.О.)

Направленность (профиль), специализация
«Разработка программно-информационных систем»

Консультант (ы):
а) Чистяков Александр Анатольевич _____
(Фамилия, И., О., ученое звание, степень) (Подпись)

б) _____
(Фамилия, И., О., ученое звание, степень) (Подпись)

ВКР принята “____” _____20 ____г.

Оригинальность ВКР _____%

ВКР выполнена с оценкой _____

Дата защиты “____” _____20 ____г.

Секретарь ГЭК _____ _____
(ФИО) (подпись)

Листов хранения _____

Демонстрационных материалов/Чертежей хранения _____

Министерство образования и науки Российской Федерации

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

“САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ”

АННОТАЦИЯ

ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЫ

Студент Припадчев Артём Александрович
(ФИО)

Наименование темы ВКР: Разработка системы автоматизированного анализа репозиториев с открытым исходным кодом при помощи статических анализаторов кода

Наименование организации, где выполнена ВКР: Университет ИТМО

ХАРАКТЕРИСТИКА ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЫ

1 Цель исследования: Разработать систему автоматизированного анализа программного кода при помощи статических анализаторов, результатом работы которой является получение и обработка различных метрик программного кода

2 Задачи, решаемые в ВКР:

- Рассмотреть существующие статические анализаторы программного кода для языка программирования Python.
- Определить формат данных результатов анализа программного кода.
- Разработать подход для получения метрик программного кода при помощи статических анализаторов.
- Оценить качество кода наиболее популярных GitHub репозиториев с помощью разработанной системы.

3 Число источников, использованных при составлении обзора: 14

4 Полное число источников, использованных в работе: 37

5 В том числе источников по годам

Отечественных			**Иностранных**		
Последние 5 лет	От 5 до 10 лет	Более 10 лет	Последние 5 лет	От 5 до 10 лет	Более 10 лет
1	3	1	-	5	12

6 Использование информационных ресурсов Internet: Да, 15
(Да, нет, число ссылок в списке литературы)

7 Использование современных пакетов компьютерных программ и технологий (Указать, какие именно, и в каком разделе работы)

Пакеты компьютерных программ и технологий	**Раздел работы**
Microsoft Word	1-4
Internet	1-4
http://draw.io	2-4
Microsoft Visual Studio	3
Python, C#	3

8 Краткая характеристика полученных результатов: В рамках данной работы была разработана система автоматизированного анализа программного кода при помощи статических анализаторов, результатом работы которой является получение и обработка различных метрик программного кода.

9 Полученные гранты, при выполнении работы _____
(Название гранта)

10 Наличие публикаций и выступлений на конференциях по теме выпускной работы: Да
(Да, нет)

а) 1 Припадчев А.А. Сравнение систем статического анализа Python-кода // Альманах научных работ молодых ученых Университета ИТМО -2017. - Т. 5. - С. 258-261
2 Припадчев А.А., Радченко И.А., Чурсин Н.В. Об особенностях реализации автоматизированной системы анализа программного кода при помощи статических анализаторов // Сборник тезисов докладов конгресса молодых ученых. Электронное издание – 2017
3 Chistiakov A.A., Pripadchev A., Radchenko I. On development of a framework for massive source code analysis using static code analyzers//ACM International Conference Proceeding Series, IET - 2017, Vol. Part F133327, pp. 3166114

б) 1 Припадчев А.А. GitHub и статический анализ кода. Слет сообществ Санкт-Петербурга и Ленинградской области IT. . – Global Meetup #9, 2016
2 Припадчев А.А. GitHub и статический анализ кода. VIII Научно-практическая конференция молодых ученых «Вычислительные системы и сети (Майоровские чтения)». – Университет ИТМО, 2016
3 Припадчев А.А. Сравнение систем статического анализа Python-кода. XLVI Научная и учебно-методическая конференция. – Университет ИТМО, 2017
4 Припадчев А.А., Радченко И.А., Чурсин Н.В., Об особенностях реализации автоматизированной системы анализа программного кода при помощи статических анализаторов. VI Всероссийский конгресс молодых ученых. – Университет ИТМО, 2017
5 Припадчев А.А, Разработка системы автоматизированного анализа программного кода при помощи статических анализаторов. VII Конгресс молодых ученых (КМУ). – Университет ИТМО, 2018

Студент: Припадчев Артём Александрович _____
(ФИО) (подпись)

Руководитель: Радченко Ирина Алексеевна _____
(ФИО) (подпись)

“_____”_____20___г.

Министерство образования и науки Российской Федерации

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

“САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ”

УТВЕРЖДАЮ

Зав. кафедрой_____

_____
(ФИО) (подпись)

«____» «_____» 20___ г.

З А Д А Н И Е
НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ

Студенту Припадчеву А.А. **Группа** P4217 **Кафедра** ИПМ **Факультет** ПИиКТ

Руководитель Радченко И.А., к.т.н., доцент
(ФИО, ученое звание, степень, место работы, должность)

1 Наименование темы: Разработка системы автоматизированного анализа репозиториев с открытым исходным кодом при помощи статических анализаторов кода

Направление подготовки (специальность) 09.04.04 Программная инженерия

Направленность (профиль) Разработка программно-информационных систем

Квалификация магистр

2 Срок сдачи студентом законченной работы «_____» «_____» 20____г.

3 Техническое задание и исходные данные к работе

3.1 Необходимо разработать систему автоматизированного анализа программного кода при помощи статических анализаторов, результатом работы которой является получение и обработка различных метрик программного кода.

3.2 Исходными данными являются файлы с программным кодом на языке программирования python.

4 Содержание выпускной квалификационной работы (перечень подлежащих разработке вопросов)

4.1 Обзор предметной области.

4.2 Сравнение существующих статических анализаторов программного кода python.

4.3 Описание подхода системы автоматизированного анализа программного кода.

4.4 Анализ результатов.

5 Перечень графического материала (с указанием обязательного материала)

6 Исходные материалы и пособия

7 Дата выдачи задания «____» «_____» 20____г.

Руководитель ВКР_____
(подпись)

Задание принял к исполнению_____ «____» «_____» 20____г.
(подпись)

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Информационные технологии стали неотъемлемой частью современной жизни и продолжают интенсивно развиваться. Нет сомнений, что использование систем, позволяющих автоматизировать повторяющиеся операции, очень удобно для человека. Такие системы увеличивают продуктивность работы и снижают количество ошибок при ее выполнении. Но ввиду того, что программный код пишется также человеком, которому свойственно ошибаться, неизбежно появляется риск возникновения ошибки в программе [1].

Мировое сообщество всерьез занимается решением проблемы появления различных дефектов в программном коде. Это подтверждается созданием различных международных стандартов (CMM/CMMI, ISO/IEC [2]), новых методологий и подходов к разработке программных систем. Целью методологий является получение качественного и стабильного продукта в процессе его создания. Часть методологий рассматривает процессы разработки программного обеспечения в виде четкой последовательности, другие, наоборот, стремятся к упрощению и созданию более гибкого процесса. При этом общая цель всех таких методологий – создание программной системы, которая будет отвечать всем требованиям его пользователей.

Для того, чтобы какой-то процесс контролировать, нужно иметь возможность оценивать его состояние. Это же относится и к качеству программного кода. Без каких-либо четких количественных характеристик он является просто «черным ящиком» для человека-исследователя или разработчика. Для решения проблемы измерения качества кода предполагается разработка системы автоматизированного анализа.

Одной из методологий оценки качества программного кода является статический анализ. Он вместе с другими метриками кода позволяет

оценивать состояние кодовой базы проекта, его развитие, а также возможные риски реализации. Такой подход помогает ответить на вопросы:

- Много или мало ошибок в проекте?
- Улучшается или ухудшается качество кода?
- Как связаны динамика количества сообщений об ошибках и рост кодовой базы?

Контролировать метрики программного кода следует потому, что трудно улучшить то, что не измеряется. Если этого не делать, то со временем поддержка и добавление новой функциональности будут стоить дороже и дороже.

Использование полученных метрик, в свою очередь, позволит ответить на вопросы:

- Сколько времени потребуется, чтобы устранить ошибки?
- Какие части проекта необходимо улучшить?
- Сколько дефектов предполагается исправить?
- По каким критериям будет осуществлена оценка, чтобы качество программного кода достигло требуемого уровня?

Также метрики кода можно использовать: для быстрого получения представления об программном коде другого разработчика при проведении интервью; для исследования исходного кода на интересующие гипотезы; для получения общей информации о безопасности используемых сторонних решений.

Далее приведены объект, предмет, цели и задачи исследования по разработке системы автоматизированного анализа репозиториев с открытым исходным кодом при помощи статических анализаторов кода.

Объект исследования: файлы с программным кодом на языке программирования python.

Предмет исследования: получение качественных показателей программного кода (метрик программного кода).

Цель исследования: разработать систему автоматизированного анализа программного кода при помощи статических анализаторов, результатом работы которой является получение и обработка различных метрик программного кода.

Задачи исследования:

1. Рассмотреть существующие статические анализаторы программного кода для языка программирования Python.
2. Определить формат данных результатов анализа программного кода.
3. Разработать подход для получения метрик программного кода при помощи статических анализаторов.
4. Оценить качество кода наиболее популярных GitHub [3] репозиториев с помощью разработанной системы.

Глава 1. ОБЗОР ПРЕДМЕТНОЙ ОБЛАСТИ

1.1 Качество программного кода

При создании любой программы главной целью разработчиков прежде всего является ее работоспособность. Даже при идеальном проектировании архитектуры и внимательному следованию различным руководствам по написанию кода, программа, неотвечающая поставленным функциональным требованиям, бесполезна [4].

Однако редки случаи, когда жизненный цикл ПО ограничивается одной версией. В любом приложении в процессе использования находятся дефекты, требующие исправления. Также, функциональность большинства разрабатываемого ПО наращивается постепенно, от версии к версии (рис. 1).

Рисунок 1 – Жизненный цикл разработки ПО

Здесь и возникает проблема изменения уже существующего программного кода, который не редактировался месяцы или больше. При непрерывной разработке команде программистов очень легко вносить правки, т.к. все хорошо помнят принципы работы и архитектуру приложения. В

противном случае требуется дополнительное время на обзор кода, часто запутанного, для понимания его функционирования. Из-за разнообразных задач даже собственноручно написанный разработчиком программный код после продолжительного перерыва может показаться трудным для изменения. На этом фоне ситуации, когда приходится править чужие программы, выглядят еще сложнее.

Качество кода – сложное и многогранное понятие. Обычно под ним понимаются свойства программного кода, определяющие уровень его читаемости, структуры, безопасности, а также сложности редактирования. Качество кода можно считать высоким, если он легко поддерживается. Не возникает трудностей при добавлении в него новой функциональности, не тратиться время на построение сложных и хрупких блоков нового кода для сохранения работы старого.

Далее рассмотрены основные моменты и рекомендации к программному коду, применяемые в различных проектах при разработке [5].

Любой программный код является текстом, требующим прочтения. Поэтому его визуальное представление очень важно для аспекта качества. Легко воспринимаемый код позволяет быстрее вносить в него изменения и разбираться в условиях функционирования.

Код должен быть написан максимально просто, лаконично и единообразно. Сюда относятся правила именования функций, классов, переменных и пр. В процессе разработки стоит избегать прямого использования константных значений. Лучше для них объявить именованную константу, название которой пояснит ее смысл. При невозможности средств языка программирования создания констант можно использовать обычные переменные с определенными правилами форматирования, например, их написание прописными буквами.

Использование константных значений непосредственно чаще всего вызывает вопросы, почему именно это число или строка участвуют в том или ином блоке кода. Конечно, можно добавить дополнительный комментарий, но

если таких блоков будет множество, то как их понимание другим человеком, так и самостоятельное поддержание будет затруднительно и не разумно.

Помимо констант, при оформлении кода, также важны принципы обособления блоков программы. Используются ли в качестве отступов пробелы или табуляция, их количество. Требуется ли перенос скобок на новую строку или же они остаются на том же уровне, что и оператор их определяющий. Нужно ли соблюдать определенное ограничение длины, а сложные конструкции разбивать на несколько строк. Обязательно ли описание документации в тексте программы для добавляемых функций, переменных и классов.

Кажется очевидным, что в рамках одного проекта стоит придерживаться единого стиля. Каждый добавляемый файл или блок кода должен восприниматься в общей структуре. К сожалению, т.к. программный код пишется разными людьми, его стиль перемешивается, отчего усложняется общее восприятие. Но не только форматированием оно определяется.

Практика разбиения программы на небольшие блоки применяется уже давно. Это и классы, методы, свойства в объектно-ориентированном программировании, и функции в процедурной парадигме. Данный подход дополнительно хорошо сказывается на переиспользовании кода, что снижает риск ошибки.

Следующим фактором является документированность кода. Идеальный код в комментировании не нуждается, однако такое возможно лишь в простейших программах. Часто можно встретить неочевидную реализацию, которую невозможно было избежать при разработке. Конечно, к снижению количества поясняющих комментариев стоит стремиться, но иногда дополнительное описание может сэкономить другому программисту огромное количество времени.

Современные текстовые редакторы и IDE позволяют значительно продвинуться в повышении качества кода. Их предварительная настройка позволит применять множество правил форматирования автоматически. Это

относится и к написанию нового кода, так и возможности привести в порядок уже существующий программный код средствами редактора.

Запуск различных анализаторов – полезная практика, особенно если их работа автоматизирована. При этом не стоит ожидать того, что все ошибки будут обнаружены, как и того, что для всех полученных сообщений потребуются действия по исправлению. Иногда анализаторы могут вызывать ложные срабатывания, и это нормально.

Таким образом, читаемость кода и его оформление очень важны при разработке и поддержке ПО для программистов. Эта профессия очень мобильна в современных реалиях, нередко приходится переключаться на разные проекты и участвовать в разных командах как внутри одной компании, так и в различных. За время работы одни и те же строки кода обозреваются и изменяются множеством людей. Использование единого стиля и общепринятых правил написания кода помогают сократить время на его изучение и сосредоточиться на непосредственной реализации новой функциональности и исправлении найденных дефектов. Также при этом не будет возникать постоянной необходимости в переопределении стилевых правил для работы с различными участками кода.

1.2 Динамический и статический анализ кода

Анализ программного кода выполняемый при реальном исполнении программ называется процессом динамического анализа кода.

Он состоит из трех основных частей:

- подготовки данных для тестирования;
- запуска программы в режиме, позволяющего собирать и оценивать изменение различных динамических параметров;
- оформление отчета о полученных результатах анализа.

Посредством динамического анализа можно проверить любые современные программы, реализованные с помощью высокоуровневых языков программирования. Это, например, Python, C++, Perl, C#, Java. Данный анализ поможет представить, как именно будет функционировать приложение в условиях приближенным к реальному использованию.

Динамический анализ обычно выполняются с помощью особого программного обеспечения, которое позволяет запустить исполнение кода и провести исследование его работы с сохранением промежуточных результатов диагностической информации. Например, IDE Microsoft Visual Studio содержит в себе подобные утилиты для динамического анализа и тестирования кода (рис.2).

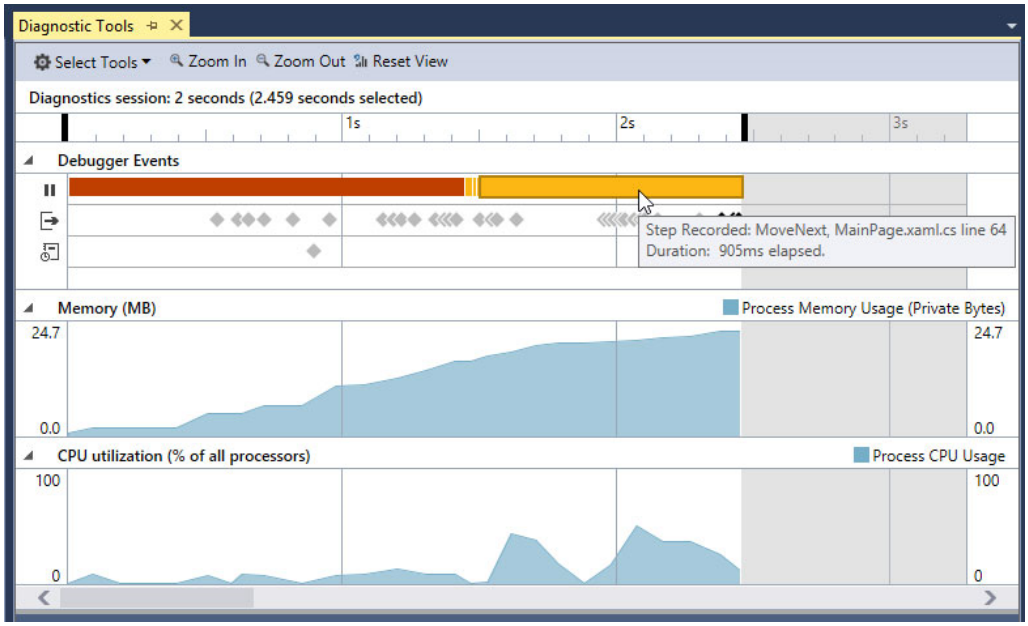


Рисунок 2 – Динамический анализ производительности ПО в Visual Studio

В ходе проведения динамического анализа программного обеспечения одним из методов взаимодействия с анализируемым ПО является инструментирование кода. Это его проверка, при которой функциональность приложения остается неизменной, но добавляются специальные механизмы. Они позволяют отслеживать состояние исполняемого кода, включая просмотр значений отдельных переменных и результатов функций в течение всего

процесса работы. При этом, конечно, стоит учитывать, что из-за инструментирования общее время, требуемое для выполнения необходимой операции, будет отличаться в большую сторону по сравнению с обычным функционированием.

Инструментирование кода подразделяется на его добавление при компиляции или же исполнении программы. В первом случае механизмы проверки будут применяться к исходному коду, а во втором – к его модифицированному представлению после обработки компилятором или интерпретатором.

Для динамического анализа программного обеспечения требуется подготовить тестовую информацию, которая будет использована при запуске и работе приложения. Поэтому качество и, соответственно, результативность анализа, будут напрямую зависеть от качества этих данных, их структуры, свойств, количества. Мера энтропии такой информации также будет оказывать влияние на метрику оценки анализа, называемой величиной покрытия кода.

Другие примеры метрик динамического анализа:

- ошибки выполнения программы: ошибки построения архитектуры приложения и неправильной последовательности исполнения различной логики; утечка памяти при определенных входных данных, что влечет за собой аварийное завершение программы операционной системой; ошибки по использованию объектов, значения которых не определены; нарушение арифметических правил;
- наличие возможных уязвимостей (sql-инъекции, удаленное исполнение кода, обход прав доступа и др.)
- сложность покрытия программного кода тестами.

При разработке программного обеспечения динамический анализ нужно применять обязательно, особенно там, где даже от незначительного сбоя в работе программы будет нанесен значительный ущерб хранимой и обрабатываемой информации и качеству функционирования системы в целом.

Стоит следить за откликом программы и используемыми ресурсами. Примером подобных систем может являться ПО применяемое при производстве, при обеспечении безопасности и контроля, при проведении работ с возможными последствиями для жизнедеятельности человека.

Существуют различные методы проведения динамического анализа:

- метод «белого ящика» при котором тестировщик имеет знания об исходном коде и может использовать эту информацию при подготовке исходных данных;
- метод «черного ящика» при котором, напротив, у тестировщика нет информации о реализации программы и данные для анализа готовятся на основе общего описания логики работы;
- метод «серого ящика», который является чем-то средним между двумя предыдущими. В нем тестировщик имеет доступ к коду и его архитектуре, но по большей части не использует эту информацию.

Также при оценке результатов динамического анализа стоит обращать большое внимание на потребляемые ресурсы (оперативную память, загруженность процессора, дисковую память, количество сетевых соединений и др.), ошибки в системном и программном логах, временные характеристики выполнения операций.

Динамический анализ хорош тем, что если в процессе его выполнения возникают ошибки, то это явное сообщение о том, что в логике, а значит и в коде, есть проблема. Ложные срабатывания при таком методе сведены к минимуму. Все обнаруженные проблемы требуют обязательного рассмотрения, исправления или доработки. Однако, если по результатам анализа сообщений об ошибках не было получено, это еще не значит, что программный код полностью правильно функционирует. Данный подход не позволяет осуществить стопроцентное покрытие всех возможных ситуаций, покрыть все логические аспекты. Стоит также учитывать случаи, когда приложение работает правильно, а полученные ошибки являются ошибками в программном обеспечении, осуществляющем динамический анализ.

Данные для анализа можно подготавливать вручную, но разумнее использовать для этого специальные утилиты. Они помогают на основе некоторых правил готовить тестовый набор данных многократно при повторяемом исполнении приложения. Часть данных выбирается подстановкой из некоторого общего хранилища, часть генерируется автоматически, что позволяет обнаружить не совсем ожидаемые при разработке входные параметры. При этом эти утилиты отлично подвергаются конфигурированию, что позволяет проводить различные сценарии анализа: обычный, нагрузочный, стрессовый.

К достоинствам динамического анализа можно отнести возможность анализа программного обеспечения, исходный код которого не доступен. Также, при ненарушенной целостности функционирования используемых инструментов, любая ошибка – это призыв к действию, сообщение о непосредственном дефекте. При этом для разработчика, занимающимся его исправлением, будет доступна полная информация об окружении и данных в момент нарушения работы. Это позволит быстро идентифицировать проблему и избавиться от нее в будущем.

К недостаткам стоит отнести невозможность выявления ошибок, связанных с неполнотой покрытия всевозможных входных данных. А даже имея большое их количество, время и затрачиваемые вычислительные ресурсы на полную проверку динамическим анализом могут быть нецелесообразно велики, а результативность невысокой. По этой причине количество таких тестов придется сократить.

Статический анализ кода – анализ программного обеспечения, производимый без реального выполнения программ [6]. Именно он используется в данной работе, и его результатом является получение определенной аналитики, с помощью которой можно получить представление о качестве программного кода. Также статический анализ применяется и для других целей. Так, в [7] с помощью машинного обучения классифицируют код android-приложений на два типа: утилиты и игры. Много работ посвящено

детектированию возможных уязвимостей в программах на стадии их разработки [8-10]. В [11] осуществляют извлечение характеристик кода для последующего поиска дефектов. Еще одной областью применения статического анализа кода является автоматизированное обнаружение вредоносного кода [12].

Обзор кода (или code-review) – это отличный способ для нахождения дефектов как в оформлении кода, так и в его логике. При этом методе изменения в тексте программы каждый раз просматриваются другими разработчиками. Они могут как одобрить правки, так и дать комментарии по улучшению, после которых им снова придется выполнить обзор.

На практике программисты с опытом способны легко найти недостатки и ошибки в коде других разработчиков. Однако это занятие достаточно рутинное, отчего относиться к нему могут поверхностно.

Помимо этого, себестоимость такого ручного обзора невероятно высока. При каждом изменении в кодовой базе проекта требуется тратить время других разработчиков на внимательное изучение новых или модифицированных строк. Этот процесс требует переключения контекста, вспоминания основных идей затронутой логики приложения, а также требует периодических перерывов. Если их не делать, то выполняющий обзор человек устает и вероятность пропустить важное изменение, которое станет причиной ошибки, значительно возрастает.

Возникает вопрос, возможно ли как-то облегчить процесс обзора кода, как минимум для выявления простых и крайне популярных проблем, выполняя его при этом регулярно? Помочь с решением данного вопроса могут специальные автоматизированные анализаторы кода. Они способны запускаться автоматически при каждом изменении или же по определенному расписанию. При обнаружении проблем будет отправлено соответствующее сообщение. Работа таких анализаторов практически бесплатна по сравнению с выполнением обзора кода человеком. Но несмотря на всю простоту решения, данные системы все же не смогут полностью заменить человека. Очень часто

появляются нестандартные ситуации, обнаружить которые способен только человек. Но если комбинировать оба подхода, то можно ожидать высокого качества для разрабатываемого программного продукта.

Среди выполняемых статическим анализом задач можно выделить следующие:

- Проверка оформления кода. Анализаторы в соответствии с определенным стандартом к программному коду выполняют его проверку. Под стандартом имеется в виду использование стилевых правил. Это, например, параметры отступов, именования, ограничения длин строк, блоков кода и др.;
- Исследование кода на наличие ошибок;
- Сбор метрик (см. раздел 1.3)

Также статический анализ можно применять при начальном знакомстве и изучении правил форматирования кода для новых лиц, участвующих в разработке проекта.

К достоинствам статического анализа программного кода можно отнести низкую себестоимость обнаружения ошибок. Чем раньше дефект будет найден, тем быстрее и проще его можно исправить, и тем меньше будут его последствия.

Также статический анализ предоставляет:

- Полное покрытие кода. Это означает, что анализу подвергается весь программный код, даже почти неиспользуемая функциональность. Ни динамический анализ, ни ручной обзор кода такой возможности не дают. Статический анализ обеспечивает обнаружение и предотвращение ошибок еще до выполнения, что снижает требуемое количество вычислительных ресурсов, а также предотвращает трату времени на воспроизведение дефекта.
- Избавление от рисков ошибок при компиляции. Статический анализ не зависит от средств разработки и может функционировать автономно, используя лишь текст программы. Этот метод позволяет выявить

ошибки сразу же, а не спустя время в релизной версии на реальном окружении. Эти дефекты проявляются в виде непредсказуемого поведения приложения в случайные моменты времени.

- Обнаружение опечаток и изменений в программном коде вследствие переиспользования кода путем его копирования.

К отрицательным сторонам статического анализа можно отнести:

1. Статический анализ позволяет диагностировать и исправить лишь простые и популярные ошибки.
2. Ложные срабатывания. Автоматизированный статический анализ устроен так, что может выдавать сообщения об ошибках в тех местах программы, которые на самом деле внимания к себе не требуют. Данный процесс называется ложно-позитивным срабатыванием. Несмотря на то, что ложное срабатывание потребует выделения дополнительных ресурсов разработчика, в ряде случаев это оказывается полезным.

Статический анализ является активно используемым методом контроля и повышения качества кода. Средства для его выполнения постоянно развиваются параллельно с разработкой новых плавил и стандартов к программному коду.

1.3 Метрики программного кода

Метрика программного кода – это количественная мера, которая представляет свойства ПО. Некоторые из метрик уже упоминались в работе ранее, а также представлены в работе [13].

Для того, чтобы получать качественные показатели программного кода, необходимо иметь численные характеристики этих показателей. В качестве таких характеристик могут выступать метрики программного кода. Зачастую их вычисление осуществляется путем анализа графа исполнения программы, а

также непосредственной архитектуры кода [14]. Самые популярные метрики – мера сложности блоков кода, количество строк, количество ошибок предупреждений и т.д. При этом вычисление метрик осуществляется компьютером, поэтому гарантируется их повторяемость при многократном исследовании одного и того же программного кода. Также метрики можно оценивать во временном диапазоне и формировать, используя их, отчеты.

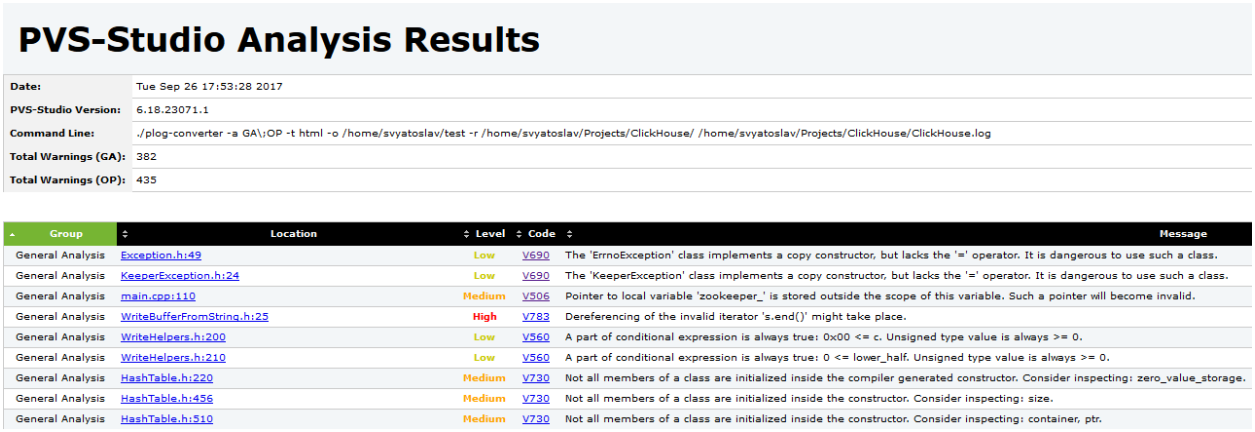
Метрика – величина статистическая. Поэтому в зависимости от решаемых задач и требований к программному обеспечению для этих показателей стоит устанавливать различный приоритет, а общий результат анализа строить комбинируя и агрегируя их.

Глава 2. СТАТИЧЕСКИЕ АНАЛИЗАТОРЫ ПРОГРАММНОГО КОДА

2.1 Инструменты анализа программного кода

Для языков программирования со статической типизацией (например, C и C++) существует ряд статических анализаторов: Klocword Insigth, PVS-Studio, Svace и другие. Данные продукты содержат в себе набор плагинов, которые позволяют автоматически обнаруживать дефекты. Это достигается путем обхода синтаксического дерева (AST) и анализа движения данных.

Общим для всех существующих статических анализаторов является то, что в большей степени они представляют собой инструменты контроля. Их часто встраивают в системы непрерывной интеграции (CI), где они выполняют автоматический анализ. Если в процессе будут выявлены нарушения, то изменения к программному коду не будут применены. При этом часть анализаторов могут генерировать отчеты с различными метриками, позволяющие получить полную аналитику результатов анализа. В отчете может быть реализована возможность поиска и фильтрации сообщений по типу, коду, уровню и тексту предупреждения (рис. 3).



PVS-Studio Analysis Results

Date:	Tue Sep 26 17:53:28 2017
PVS-Studio Version:	6.18.23071.1
Command Line:	./plog-converter -a GA\:OP -t html -o /home/svyatoslav/test -r /home/svyatoslav/Projects/ClickHouse/ /home/svyatoslav/Projects/ClickHouse/ClickHouse.log
Total Warnings (GA):	382
Total Warnings (OP):	435

Group	Location	Level	Code	Message
General Analysis	Exception.h:49	Low	V690	The 'ErrnoException' class implements a copy constructor, but lacks the '=' operator. It is dangerous to use such a class.
General Analysis	KeeperException.h:24	Low	V690	The 'KeeperException' class implements a copy constructor, but lacks the '=' operator. It is dangerous to use such a class.
General Analysis	main.cpp:110	Medium	V506	Pointer to local variable 'zookeeper_' is stored outside the scope of this variable. Such a pointer will become invalid.
General Analysis	WriteBufferFromString.h:25	High	V783	Dereferencing of the invalid iterator 's.end()' might take place.
General Analysis	WriteHelpers.h:200	Low	V560	A part of conditional expression is always true: 0x00 <= c. Unsigned type value is always >= 0.
General Analysis	WriteHelpers.h:210	Low	V560	A part of conditional expression is always true: 0 <= lower_half. Unsigned type value is always >= 0.
General Analysis	HashTable.h:220	Medium	V730	Not all members of a class are initialized inside the compiler generated constructor. Consider inspecting: zero_value_storage.
General Analysis	HashTable.h:456	Medium	V730	Not all members of a class are initialized inside the constructor. Consider inspecting: size.
General Analysis	HashTable.h:510	Medium	V730	Not all members of a class are initialized inside the constructor. Consider inspecting: container, ptr.

Рисунок 3 – Пример главной страницы Html отчета PVS-Studio

Основной идеей данной работы является выделение логики получения информации об анализе программного кода в отдельный инструмент

аналитики. С помощью него можно будет выполнять анализ программного кода при помощи различных статических анализаторов и строить отчеты по полученным результатам. Данный инструмент позволит получить все сообщения с возможностями фильтрации, поиска и сортировки, а также будет иметь графическое представление статистики метрик программного кода (например, количества ошибок, обнаруженных анализатором).

2.2 Статические анализаторы кода Python

Во всём множестве языков программирования можно выделить большую группу динамических языков. Основным фактором для определения языка в эту группу является типизация данных. Динамические языки программирования очень популярны, о чем говорит индекс TIOBE [15], в вершине которого можно встретить: Python, PHP, JavaScript, Ruby.

В работе в качестве языка программирования для статического анализа кода выбран язык python, который в последние годы все чаще используют для создания крупных программных продуктов, т.к. данный язык позволяет сократить время на разработку. Программы на нем выглядят очень лаконично, компактно и хорошо читаются. Также python содержит развитую инфраструктуру для решения различных исследовательских задач (статистика, визуализация и др.).

Инструментов с открытым исходным кодом для аудита программного кода на разных языках программирования не так много. Статистика по самым популярным из них, применимых к Python, полученная с GitHub, представлена в таблице 2.1.

Таблица 2.1 – Статистика GitHub по статическим анализаторам кода Python (по состоянию на апрель 2018 года)

Название	**Кол-во наблюдателей**	**Кол-во отметок «избранное»**	**Кол-во форков**	**Кол-во вопросов (открытые / закрытые)**	**Кол-во коммитов**
pylint	58	1126	266	321/1295	3563
pycodestyle	134	3230	520	89/363	1064
pyflakes	24	524	74	50/161	418
coala	94	2157	997	580/2081	4164
pylama	16	411	45	32/34	452
flake8	20	330	50	72/337	1403

Инструменты pylint [16], pycodestyle [17] и pyflakes [18] являются просто статическими анализаторами кода с небольшим количеством настроек, а coala [19], pylama [20] и flake8 [21] уже включают в себя ряд анализаторов с возможностью самостоятельного расширения, гибкой системой настроек анализа и форматированным выводом результатов. Последние также позволяют:

- Запускать анализ всего проекта или его части;
- Исключать определенные сообщения;
- Выбирать анализаторы и указывать их настройки;
- Выборочно игнорировать строки кода;
- Сохранять конфигурацию в файл.

Поскольку flake8 специализирован на стилистической проверке и не имеет возможности отключения стандартных плагинов, из дальнейшего рассмотрения он исключен.

Все статические анализаторы кода Python построены по общей многоуровневой модели:

- Прикладные алгоритмы:
    - инспекция кода;
    - рефакторинг;
- Движок понимания кода:
    - определение атрибутов;
    - определение типов;
    - определение имен;
    - парсер;
    - лексер.

Лексический и синтаксический анализы являются самыми низкими и основными уровнями в этой модели. Первый преобразует строку кода в список более крупных единиц – токенов, а второй – преобразует полученный поток лексических единиц в синтаксическое дерево.

Пример лексического анализа кода:

```
def f(x):                   NAME    'def'
    if x > 0:               NAME    'f'
        return x            OP      '('
    return 0                NAME    'x'
                            OP      ')'
                            OP      ':'
                            NEWLINE '\n'
                            INDENT  '    '
                            NAME    'if'
                            ...
```

Слева находится код, подвергаемый лексическому анализу, а справа – список токенов. В python подобный анализ осуществляет модуль *tokenize*. Уже на этом уровне выполняются некоторые проверки соответствия кода стандарту PEP8.

Во время синтаксического анализа полученный список токенов преобразуется в древовидное представление, т.е. в набор структур данных в памяти анализатора, которые соответствуют программному коду:

```
def f(x):           NAME  'def'  Module
    if x > 0:       NAME  'f'      FunctionDef[f]
        return x    OP    '('        args=Name[x]
    return 0        NAME  'x'        If
                    OP    ')'          left=Name[x]
                    OP    ':'          op=GreaterThan
                    ...                right=Num[0]
                                       Return
                                         Name[x]
                                     Return
                                       Num[0]
```

Соответственно, парсер позволяет анализатору представить иерархическую структуру программы. В стандартной библиотеке python есть модуль *ast,* который позволяет работать с синтаксическим деревом и встроенная функция *compile,* которая выполняет его построение.

На основе этих стандартных модулей осуществлен синтаксический анализ в таких инструментах, как pyflakes и pylint.

2.3 Сравнение coala и pylama

Coala представляет универсальный интерфейс командной строки для контроля стиля и исправления кода. У coala есть набор плагинов (bears) для различных языков программирования, а также есть возможность пополнять стандартный набор своими собственными. Pylama – инструмент для аудита кода на языках Python и JavaScript, включающий в себя следующий утилиты: pycodestyle, pydocstyle, pyFlakes, mccabe, pylint, radon, gjslint. Имеет более скромную функциональность, что связано с гораздо меньшей популярностью среди сообщества GitHub.

После запуска coala (рис. 4) осуществляет поочередное чтение секций с параметрами для анализа в конфигурационном файле или в заданных в командной строке в качестве аргументов. Далее, в зависимости от настроек, запускается один или ряд процессов, а также еще один, который управляет

информацией о них. Каждый процесс записывает получаемые сообщения в общий буфер, которые впоследствии извлекаются управляющим процессом и отправляются пользователю.

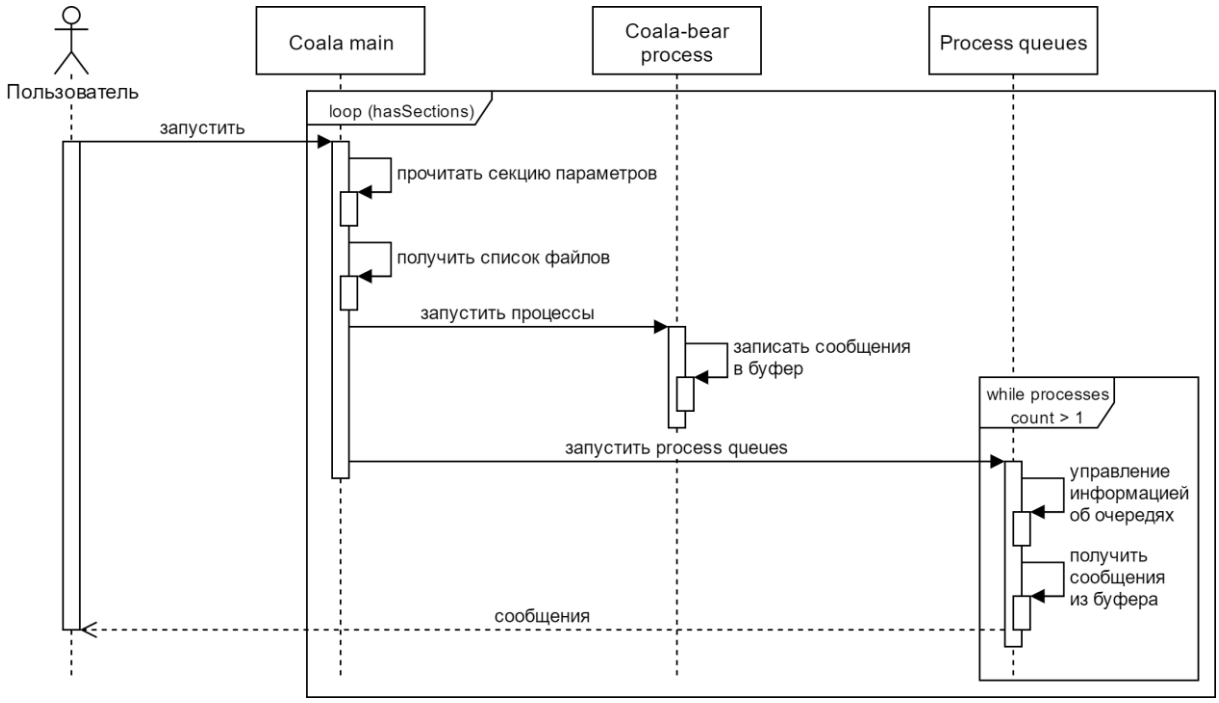


Рисунок 4 – Общая схема работы coala

В pylama процесс работы проще (рис. 5). Он включает:

- Получение списка анализируемых файлов;
- Поочерёдный запуск необходимых анализаторов;
- Аккумулирование сообщений анализаторов в общем списке для всего анализа;
- Отправление результата пользователю, инициировавшего анализ, по завершению работы.

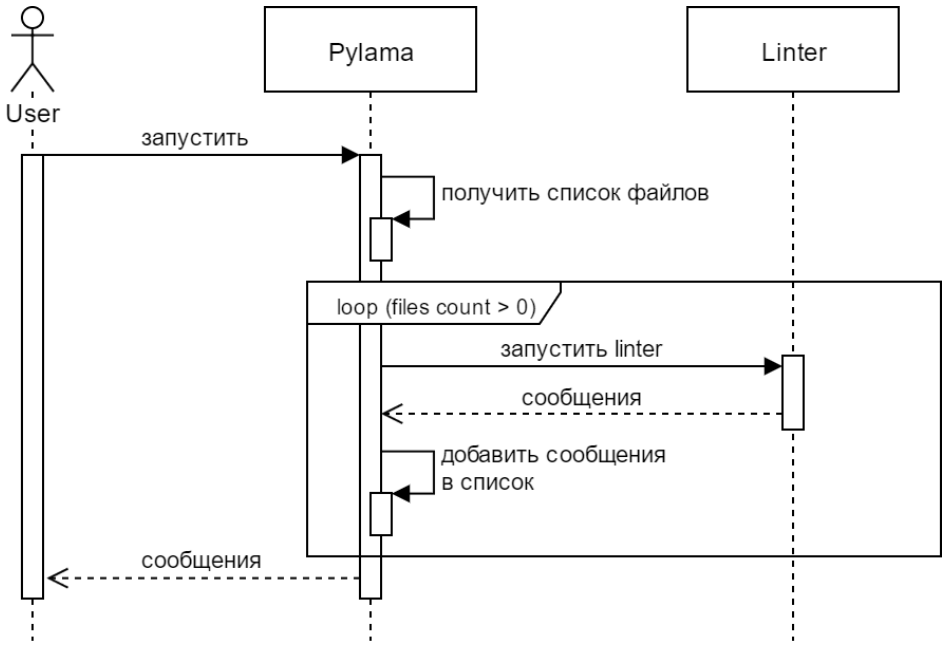


Рисунок 5 – Общая схема работы pylama

Оба инструмента задействуют процессор в полной мере, однако среднее время анализа исходного кода одного и того же небольшого проекта (около 130 файлов) различается. Используя два анализатора, для coala оно составило 112 секунд, а для pylama – 183 секунды (использовалась виртуальная машина с Ubuntu 14.04 Server, 3Gb RAM, 3 CPUs).

Ключевой метрикой оказалось сравнение используемой утилитами оперативной памяти. Ввиду того, что pylama постоянно накапливает информацию об ошибках, потребность в памяти возрастает линейно (рис. 6). Это влечет за собой критический дефект, заключающийся в том, что на больших проектах объем используемой памяти достигает максимальных значений и операционная система аварийно завершает выполнение программы. Coala осуществляет периодическую отправку сообщений пользователю, поэтому использование памяти осуществляется равномерно.

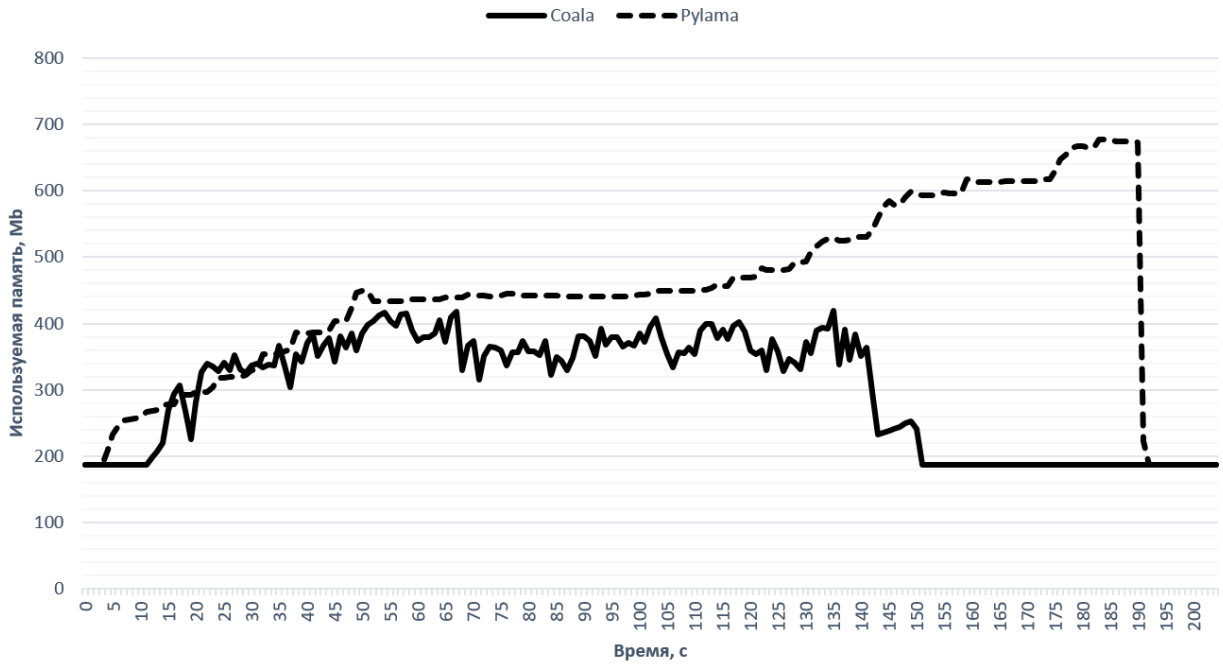


Рисунок 6 – Использование памяти утилитами coala и pylama

Общие результаты сравнения отображены в таблице 2.2. Таким образом, сравнивая coala и pylama можно указать на явное преимущество первой.

Таблица 2.2 – Сравнение coala и pylama

	coala	pylama
Популярность среди пользователей GitHub	+	–
Задействование процессора в полной мере	+	+
Отсутствие «утечек» оперативной памяти	+	–
Возможность параллельной обработки	+	–
Время анализа, с	112	183

Глава 3. СИСТЕМА АВТОМАТИЗИРОВАННОГО АНАЛИЗА ПРОГРАММНОГО КОДА

3.1 Описание подхода

В данной работе для вычисления и исследования метрик, позволяющих оценить качество программного кода и его версионное развитие, разработана следующая структурная схема (рис. 7).

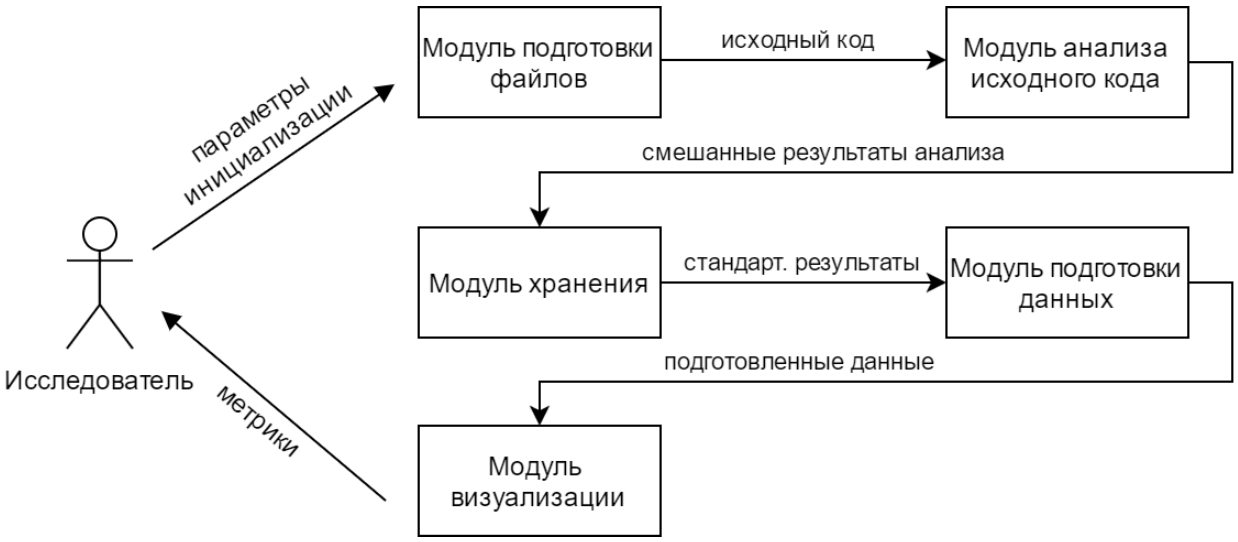


Рисунок 7 – Структурная схема системы автоматизированного анализа программного кода при помощи статических анализаторов

Модуль подготовки файлов является входной точкой для пользователя, где он задает всю необходимую конфигурацию последующего анализа. Это указание проверяемой директории с исходным кодом проекта, необходимый набор анализаторов, исключение определенных типов сообщений и др.

После указания конфигурации и запуска проверки система переходит к непосредственному анализу программного кода с помощью различных статических анализаторов, которые, в свою очередь, не имеют общей одной модели данных для получаемых результатов.

Следующим этапом является обработка результатов анализа модулем хранения, который осуществит приведение данных к общему формату и их сохранение в базу данных.

Результаты измерения различных метрик на данном этапе будут представлены в виде таблиц, содержащих их числовые значения, а также сопровождающей метаинформацией об исследуемом проекте. Далее с помощью модуля обработки данных осуществляется их интерпретация и обработка (например, агрегация или фильтрация) при необходимости. Следующим и последним этапом является визуализация результатов, получателем которых является пользователь инициировавший анализ исходного кода. При этом на этапе визуализации могут быть осуществлены различные виды аспектов сравнения [22]. Это, например, временной аспект, который показывает развитие проекта в определенных хронологических границах, или покомпонентный аспект, позволяющий сравнить каждый компонент в процентах от некоего общего целого. Результирующая инфографика может быть представлена в различных формах. Это таблицы, карты, графики и диаграммы.

Ряд подходов для визуализации результатов измерения качества программного кода описаны в работах [23-25].

3.2 Формат данных результатов анализа программного кода

Изучив предоставляемые статическими анализаторами результаты, а также дополнив их метаинформацией об исследуемом проекте был определен следующий унифицированный формат данных результатов анализа программного кода:

- Анализатор – плагин, содержащий подпрограмму, которая отвечает за проверку исходного кода согласно определенной спецификации;
- Язык программирования;

- Версия языка программирования;
- Имя проекта;
- Версия проекта;
- Сообщение – текстовая информация, описывающая результат работы анализатора;
- Путь к файлу с исходным кодом;
- Номер строки кода;
- Уровень сообщения – уровень «тяжести» сообщения (информационный, средний, высокий);
- Вероятность сообщения – число от 0 до 100, характеризующее величину вероятности результата, что он является реальной проблемой;
- Предлагаемые изменения – предлагаемые изменения для оптимизации исходного кода;
- Расширенные параметры – информация о значимых настройках используемого анализатора.

Для хранения результатов анализа используется бесплатно распространяемая с открытым исходным кодом база данных – SQLite [27]. Это легковесная встраиваемая реляционная база данных. Слово «встраиваемый» означает, что SQLite не осуществляет общения по типу «клиент-сервер», т.е. SQLite не работает в виде отдельного процесса, к которому нужно подключаться из своей программы, а является просто библиотекой, которая импортируется в программу, и далее можно осуществлять взаимодействие с базой данных посредством обращений к предоставленным интерфейсам.

Благодаря тому, что не требуется осуществлять обмен информацией со сторонним процессом, а все взаимодействие осуществляется через обращение к функциям (API) библиотеки SQLite, уменьшаются накладные расходы при выполнении программы. Это, например, время отклика и скорость обработки запросов к базе данных. SQLite позволяет хранить все данные, а также их представления и индексы, лишь в одном файле на исполняемом программу

устройстве. Это реализовано с помощью функции блокировки файла базы данных при осуществлении транзакций на изменение его содержимого.

При этом блокировки при операциях чтения данных не осуществляются, поэтому их можно выполнять как в несколько потоков, так и из различных процессов. При попытке записи в SQLite, файл базы данных которой занят обработкой другого запроса, будет получено сообщение об ошибке и операцию необходимо будет повторить через некоторое время. Из этого следует, что все операции к базе данных нужно разрабатывать как можно более быстрыми и с меньшим количеством инструкций, чтобы не снижать скорость работы приложения.

SQLite поддерживает типы данных Text, Integer, Real, Blob и Null. Для прочих типов данных, используемых в программе, перед сохранением должна быть осуществлена их конвертация или приведение к существующим. Методы этой базы данных обычно не проверяют передаваемые типы из указанной строки или столбца, если эти типы данных уже были описаны выше в листинге, т.е. можно записать целое число в столбец строки и наоборот.

3.3 Подсистема анализа и веб-клиент для просмотра результатов

Разработанную систему анализа программного кода глобально можно разделить на две части:

1. Подсистема анализа, разработанная с помощью языка программирования python, которая реализует первые три модуля системы (подготовки файлов, анализа исходного кода и хранения). Помимо стандартных, использованы следующие сторонние библиотеки: coala, sqlite3. Общая схема работы описана в диаграмме последовательности (рис. 8);

2. Веб-клиент для просмотра результатов, написанный на языке программирования C#, реализующий модули подготовки данных и визуализации. Данное решение включает в себя три .NET сборки:
    - Statan.Core (рис. 10) – содержит основные классы, интерфейсы и расширения;
    - Statan.Database (рис. 11) – слой доступа к данным, реализованный с использованием шаблона проектирования «Репозиторий» [26] и дополнительного использования библиотек: SQLite, Unity;
    - Statan.Web (рис. 12) – веб-приложение на основе фреймворка ASP.NET MVC5, реализующего шаблон Model-View-Controller и дополнительными сторонними решениями: Newtonsoft.Json, Unity, MVC5.Grid [28], jQuery, Chart.js [29].

Подсистема анализа и веб-клиент архитектурно изолированы друг от друга и пересекаются лишь использованием SQLite базы данных. Это позволяет при необходимости подменять их реализацию без обновления всей системы. Также, при соблюдении модели хранения данных веб-клиент можно использовать отдельно для просмотра результатов анализа, полученных любым образом и различными подсистемами, в том числе и при их одновременной работе.

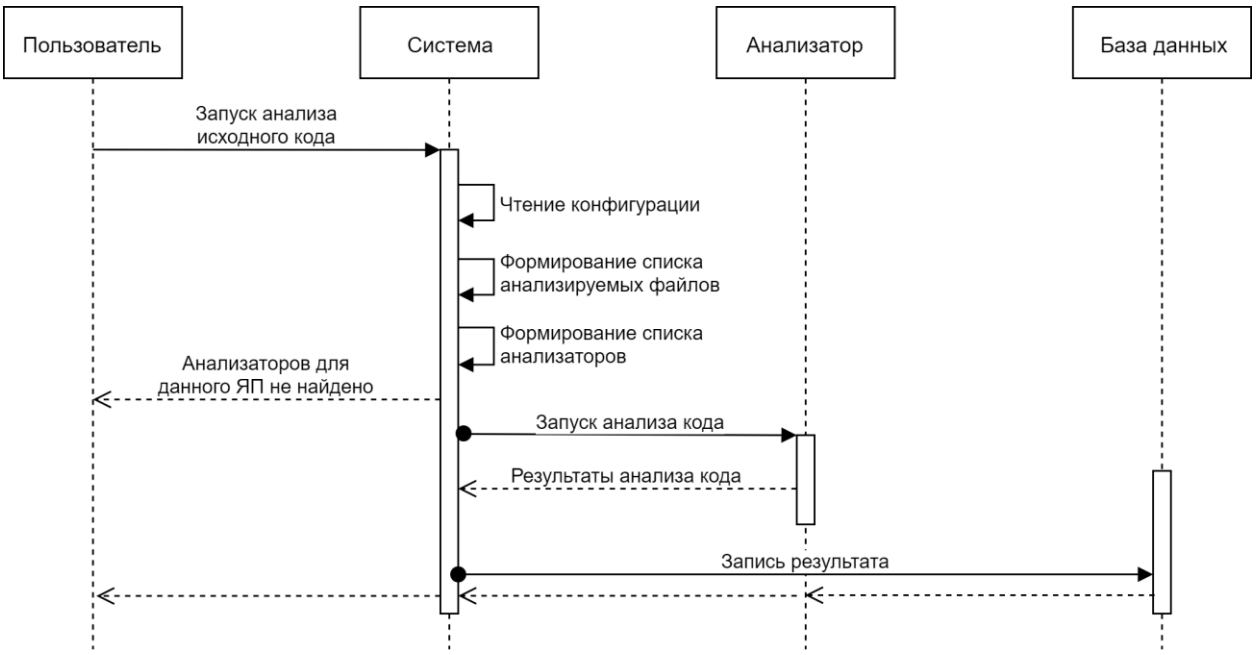


Рисунок 8 – Диаграмма последовательности подсистемы анализа программного кода

Сборки .NET в веб-клиенте связываются между собой с помощью внедрения зависимостей (Dependency Injection, DI). Это технология создания приложений со слабой связанностью, которая позволяет упростить код, обеспечить построение зависимостей между компонентами приложения специально предназначенному для этого единому механизму (рис. 9).

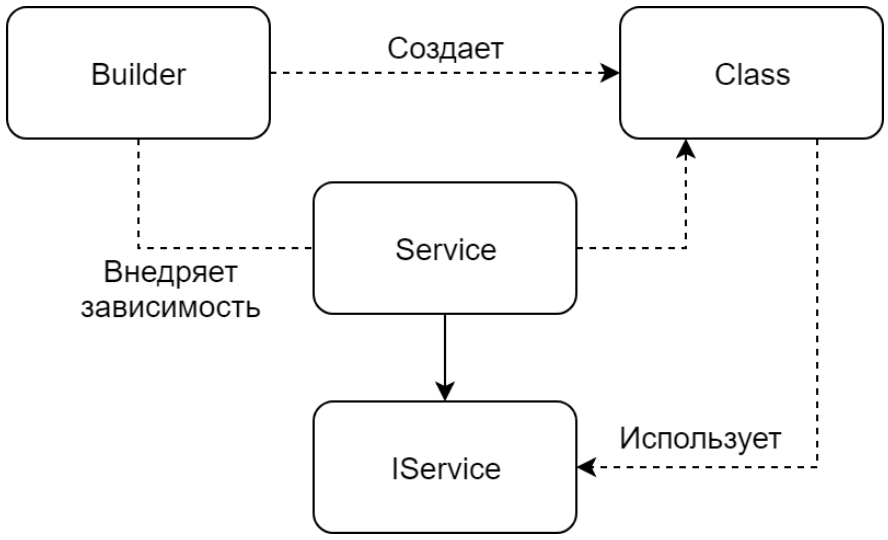


Рисунок 9 – Механизм работы внедрения зависимостей

Процесс внедрения зависимостей позволяет исключить явное описание зависимостей в приложении при разработке и создание объектов в коде во время выполнения, но предоставляет реализовать устанавливающие методы в объекте и/или такой конструктор с параметрами, с помощью которых будут внедряться зависимости.

Достоинства процедуры внедрения зависимостей:

- ослабление связей между классами. IoC-контейнер, который предоставлен механизмом внедрения зависимостей, на основе полученной им конфигурации создает связи между объектами, используя их свойства (поля), хранящие ссылку на необходимый тип сервиса. И при создании объекта реализация требуемого сервиса будет подставлена в соответствующее поле;
- упрощение проверки создаваемого кода. По типу конструкторов, свойствам и методам создаваемых классов просто определяются используемые ими объекты и существующие зависимости. Разрешение зависимостей в классе производится при помощи указания требуемых типов или интерфейсов;
- улучшение тестирования. Положительным моментом внедрения зависимостей является создание различных вариантов реализации используемого сервиса, один из которых указывается в конфигурационном файле, не внося изменений в использующий его объект, что выгодно при модульном тестировании классов этих объектов.

Внедрение зависимостей может осуществляться различными видами IoC-контейнеров таких, как CastleWindsor, Spring.Net, Autofac, Ninject, Unity.

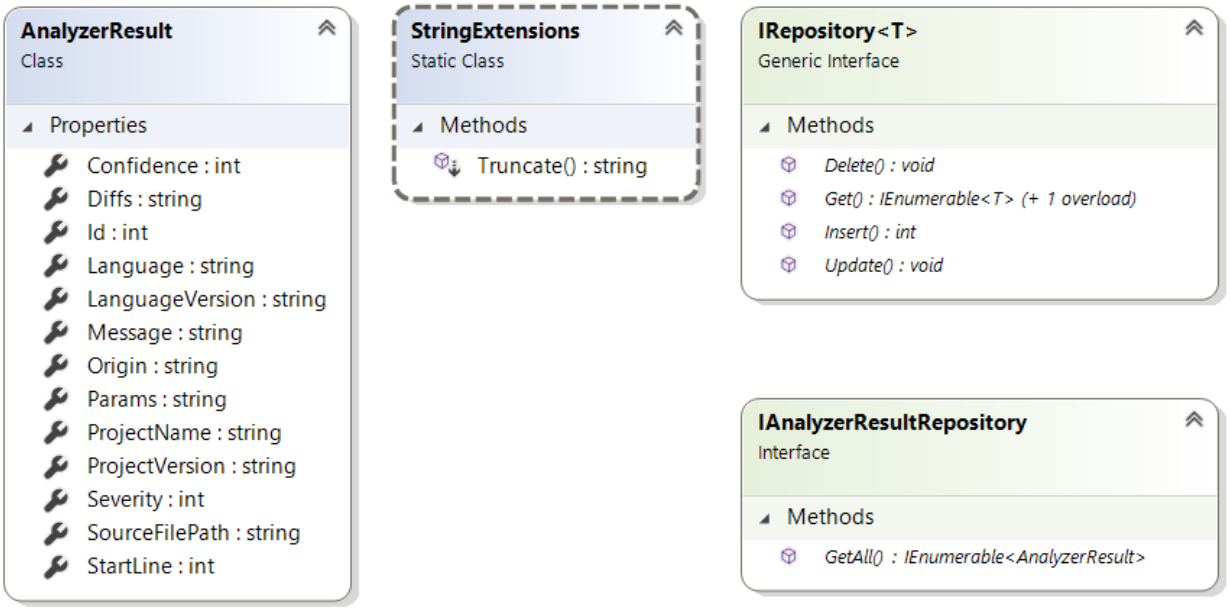


Рисунок 10 – Диаграмма классов сборки Statan.Core

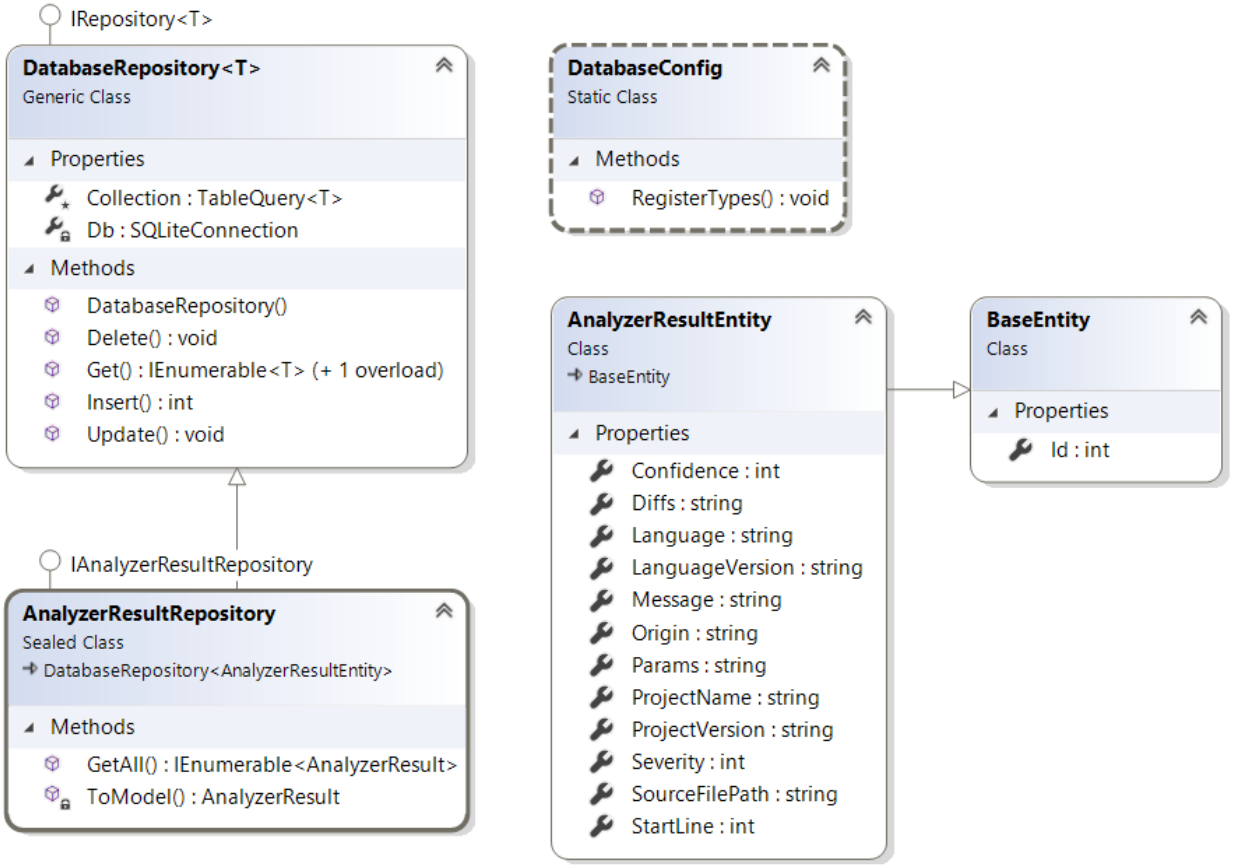


Рисунок 11 – Диаграмма классов сборки Statan.Database

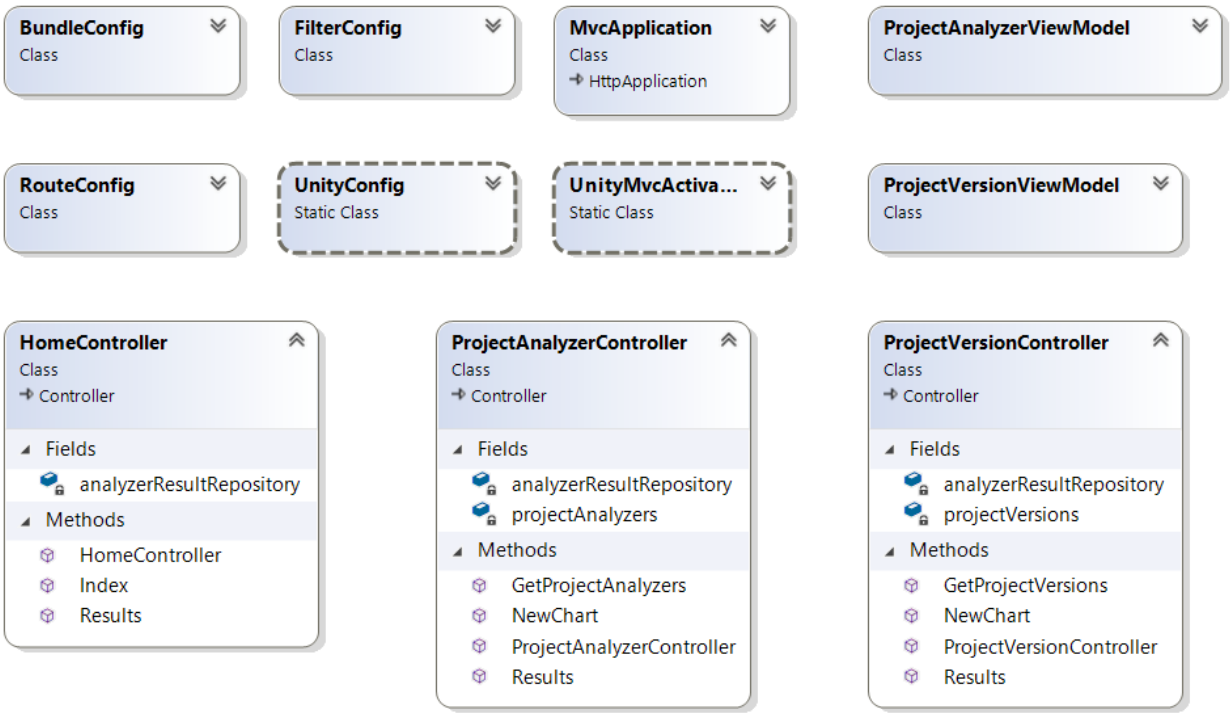


Рисунок 12 – Диаграмма классов сборки Statan.Web

Суть шаблона проектирования «Репозиторий» (рис. 13), использованного в сборке Statan.Database, в том, что он унифицирует доступ к сущностям. Его необходимо реализовать в связи с необходимостью унификации доступа к данным из контроллеров. Это позволит избежать дублирования кода. Данный шаблон создан средствами обобщенных классов.

«Репозиторий» позволяет не использовать конкретные подключения к базам данных, необходимых для работы программы, а также представляет собой прослойку доступа к данным сервисам, которые непосредственно взаимодействуют с ними.

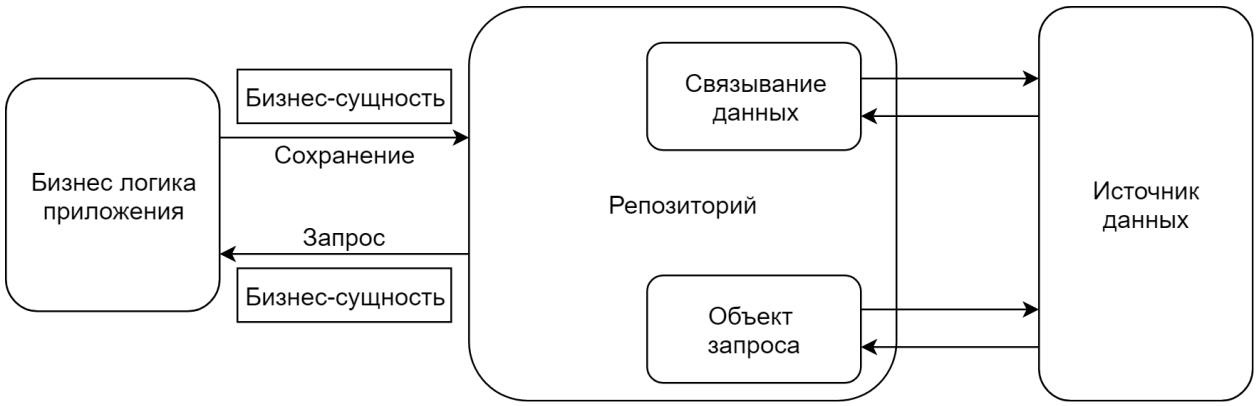


Рисунок 13 – Схема работы шаблона проектирования «Репозиторий»

Используемый шаблон MVC (Модель-Представление-Контроллер) позволяет разбить доступ к данным, их обработку и представление на три отдельные части. Модель содержит данные, представление обеспечивает их отображение, а контроллер обрабатывает события, влияющие на модель или представление (рис. 14).

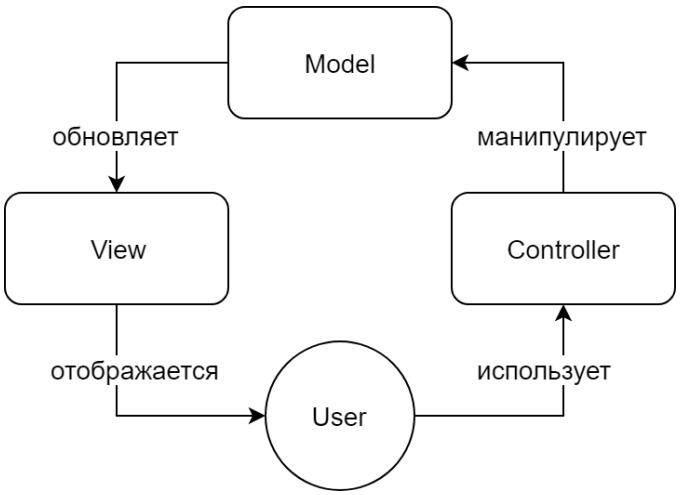


Рисунок 14 – Архитектура MVC-шаблона

Контроллер оперирует входными данными, после чего сообщает модели и представлению о необходимости соответствующего действия. Модель обычно является описанием объекта бизнес-логики приложения, предоставляет данные для View, и реагирует на события изменения своего состояния. Представление отображает на устройстве ввода конечный результат обработки.

После того, как контроллер получает данные, он оповещает об этом модель. Данные обрабатываются моделью. Далее модель сообщает об обновлении представлению. В отдельных случаях цепочка дополняется взаимодействием с контроллером, чтобы, например, отобразить ошибки валидации данных. Для одной модели может существовать несколько представлений. Стоит отметить, что в данной архитектуре модель не зависит от двух других модулей. Однако представление и контроллер зависимость имеют. Это позволяет отделить модель от ее визуального представления. В веб-приложениях представление и контроллер отличаются кардинально.

Модель и представление являются отдельными ключевыми модулями информационной системы. Для представления важное значение имеет реализация интерфейса. Ключевое для модели – бизнес-логика и связь с хранилищем данных. Отдельная реализация представления и модели дает возможность иметь для одной и той же модели множество интерфейсов, будь то отображение на веб-странице или результат удаленного вызова API метода или вывод командной строки. Также это позволяет удобнее тестировать логику домена, т.к. можно исключить работу с визуальным представлением.

Выделение контроллера позволяет создать несколько вариантов использования для одного представления. Однако зачастую, представление и контроллер соотносятся один к одному.

Наряду с MVC существуют и другие шаблоны проектирования, наиболее популярные из них – MVVM и MVP.

Model-View-Presenter (MVP)

Шаблон MVP имеет много общего с MVC, но в большей степени предназначен для разработки пользовательского интерфейса.

В данном шаблоне проектирования Presenter (рис. 15) реализует функции посредника (как контроллер в MVC), отвечая за обработку событий инициированных пользователем в UI, что в других шаблонах может выполнять представление.

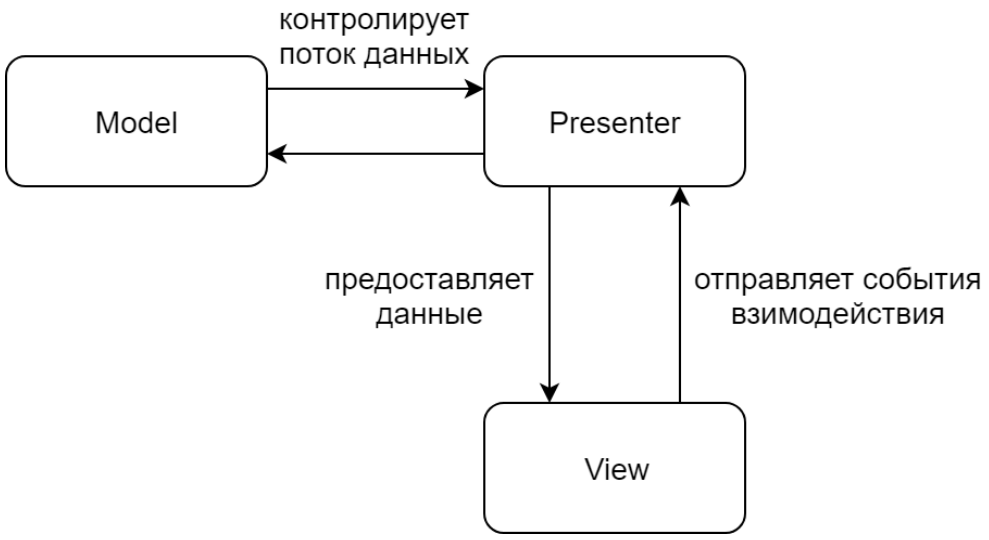


Рисунок 15 – Архитектура MVP-шаблона

Model-View-ViewModel (MVVM)

Данный шаблон часто используется при разработке мобильных приложений в случаях, когда средства используемого стека технологий позволяют осуществлять автоматическое связывание данных (рис.16).

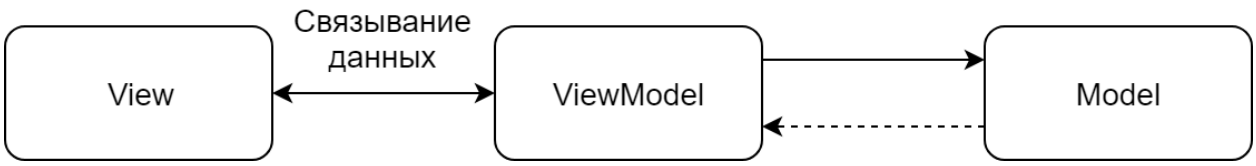


Рисунок 16 – Архитектура MVVM-шаблона

В рассмотренных MVC и MVP шаблонах существуют посредники, отвечающие за обработку событий в UI, следовательно, непосредственного влияния интерфейса на модель нет. Технологии WPF и Silverlight позволяют осуществлять связывание данных в обе стороны, в связи с чем создания отдельных обработчиков не требуется.

В шаблоне MVVM выделяют три модуля:

- Модель;
- Представление;

- Модель представления, которая является оберткой для Модели с которой будет осуществлено связывание. Модель представления содержит описание Модели, адаптированное для пользовательского интерфейса, а также имеет функциональность для обновления Модели, используемую представлением.

Глава 4. АНАЛИЗ РЕЗУЛЬТАТОВ

4.1 Набор данных

Для выбора источника репозиториев с программным кодом были рассмотрены ряд хостингов (табл. 4.1) [30]. Среди них был выбран GitHub, т.к. он является самым большим хранилищем кода в мире и содержит более чем 38 миллионов проектов.

Таблица 4.1 – Сравнение хостингов для проектов с открытым исходным кодом (по состоянию на апрель 2018 г.)

Имя	**Кол-во пользователей**	**Кол-во проектов**	**Рейтинг Alexa (меньше = лучше)**
Alioth	22731	1106	–
Assembla	–	526581	26366
Bitbucket	5000000	–	696
GitHub	24000000	69000000	60
GitLab	100000	546000	2422
GNU Savannah	77137	3825	50323
Launchpad	3965288	40881	6480
OSDN	54826	6294	8592
Ourproject.org	6353	1846	1278995
SourceForge	3700000	500000	347
CodePlex	–	107712	2689
Google Code	–	250000	–
Gna!	8710	1450	147570
Tigris.org	–	–	96848

В качестве исследуемого кода были выбраны наиболее популярные на GitHub репозитории из категории языка программирования python:

- requests 2.18.4 – предоставляет удобные способы взаимодействия с HTTP запросами;
- awesome-machine-learning – ссылки на фреймворки, библиотеки и программы, применимые к машинному обучению;
- awesome-python – ссылки на фреймворки, библиотеки, программы и ресурсы, применимые к языку программирования python;
- django 2.0.2 – высокоуровневый мощный python фреймворк, упрощающий разработку веб-приложений;
- flask 0.12.2 – легковесный фреймворк, упрощающий создание несложных веб-приложений;
- httpie 0.9.8 – инструмент командной строки, предоставляющий HTTP-клиент;
- public-apis – список общедоступных JSON API-интерфейсов для использования в веб-разработке;
- thefuck 3.25 – приложение, которое автоматически исправляет некорректную предыдущую консольную команду;
- youtube-dl 2018.02.22 – инструмент командной строки, позволяющий загружать видео с youtube и других видеосайтов;
- ansible 2.5.0 – платформа для автоматизации развертывания приложений и систем.

Репозитории awesome-machine-learning, awesome-python, public-apis из дальнейшего рассмотрения были исключены, т.к. содержат в себе лишь наборы полезных ссылок на документацию и не содержат программного кода. Остальные репозитории активно поддерживаются сообществом

разработчиков, имеют большое количество коммитов и релизных версий (табл. 4.2).

Таблица 4.2 – Исследуемые репозитории python-кода (по состоянию на апрель 2018 г.)

Репозиторий	**Количество коммитов**	**Количество релизных версий**
requests	5416	131
django	25626	184
flask	3223	21
httpie	966	29
thefuck	1463	76
youtube-dl	16076	980
ansible	36744	204

4.2 Исследование максимальных длин строк и использования пробелов

Чтение программного кода происходит чаще, чем его добавление или изменении. Соответственно, и рекомендации о стиле, как писать код, созданы для того, чтобы повысить его качество и обеспечить единство его восприятия для всего проекта. Если будет использован единый стиль, то любой сможет легко прочесть этот код.

Для языка python существует руководство по написанию кода – PEP8 [31]. Одной из рекомендаций является ограничение максимальной длины строки 79 символами. Это удобно при чтении программного кода на мониторе с малой диагональю, вертикальной ориентацией или с несколькими открытыми параллельно файлами.

Результаты исследования максимальных длин строк (табл. 4.3) показывают, что описанному в руководстве ограничению придерживаются не все. Причиной этому может быть популярность широкоформатных мониторов, а также настройки по умолчанию в популярных средах разработки программ. Ограничение длины строки в 120 символов не описано в стандарте, но является популярным, поэтому также было рассмотрено. С ним в исследуемых репозиториях результат был хорошим: от общего количества не более 1% строк нарушали ограничение.

Таблица 4.3 – Исследование максимальных длин строк

Репозиторий	**Кол-во строк (комментарии + код)**	**Кол-во строк с длиной > 79**	**Кол-во строк с длиной > 120**
requests	1943 + 5741	269	11
django	48709 + 221294	20522	20
flask	3534 + 8879	84	0
httpie	964 + 4112	4	0
thefuck	1791 + 8716	308	38
youtube-dl	6715 + 109565	8739	1250
ansible	259900 + 342607	42838	4618

Вопрос использования пробелов или табуляции для форматирования кода всё еще часто обсуждается разработчиками и исследователями [32-33]. То же руководство PEP8 рекомендует использование пробелов в качестве отступов. Среди рассмотренных репозиториев эта рекомендация полностью соблюдена (табл. 4.4), что по большей части является заслугой сред по разработке программного обеспечения, которые автоматически заменяют табуляцию на пробелы. Ошибки, полученные при данном анализе – это сообщения об отсутствии пустой строки в конце файла, наличие которой

является требованием POSIX [34] стандарта. Также, среди ошибок встречаются сообщения об имеющихся бесполезных пробелах в конце строк.

Таблица 4.4 – Исследование использования пробелов

Репозиторий	**Кол-во строк (комментарии + код)**	**Кол-во ошибок использования пробелов**
requests	1943 + 5741	0
django	48709 + 221294	0
flask	3534 + 8879	1
httpie	964 + 4112	0
thefuck	1791 + 8716	17
youtube-dl	6715 + 109565	1
ansible	259900 + 342607	25

Для репозитория requests было произведено исследование количества нарушений максимальных длин строк во временном аспекте. В табл.4.5 представлены результаты анализа программного кода для нескольких отобранных релизных версий.

Таблица 4.5 – Исследование максимальных длин строк репозитория requests

Версия	**Кол-во строк (комментарии + код)**	**Кол-во строк с длиной > 79**
0.10.0	1706 + 2805	106
0.13.0	1573 + 3469	212
1.0.0	2819 + 8850	2981
1.2.3	3115 + 9680	3001
2.0	3490 + 10344	3022
2.5.0	4015 + 11629	3084
2.10.0	4425 + 13300	3094
2.15.0	5799 + 25238	1954
2.18.4	1899 + 5490	240

Для версий 0.10.0 и 0.13.0 количество сообщений о нарушении ограничения длины строки в 79 символов было мало. На протяжении версий 1.0.0 – 2.10.0 кодовая база репозитория увеличивалась (рис. 17). Значительному увеличению количества строк способствовало добавление в проект файлов с исходным кодом используемых сторонних решений. При этом многократно возросло и количество нарушений ограничения длины строки.

Интересный результат наблюдается для версии 2.15.0. Количество нарушений сократилось, несмотря на стремительное увеличение общего количества строк. Это свидетельствует о проведенном рефакторинге существующего кода.

В самой последней на текущий момент релизной версии 2.18.4 весь программный код сторонних решений был удален, а информация о них перенесена в файл с зависимостями для разрабатываемой библиотеки.

Благодаря этому количество строк, длина которых превышает ограничение в 79 символов вернулось к состоянию начальных версий репозитория и, возможно, это те же самые строки.

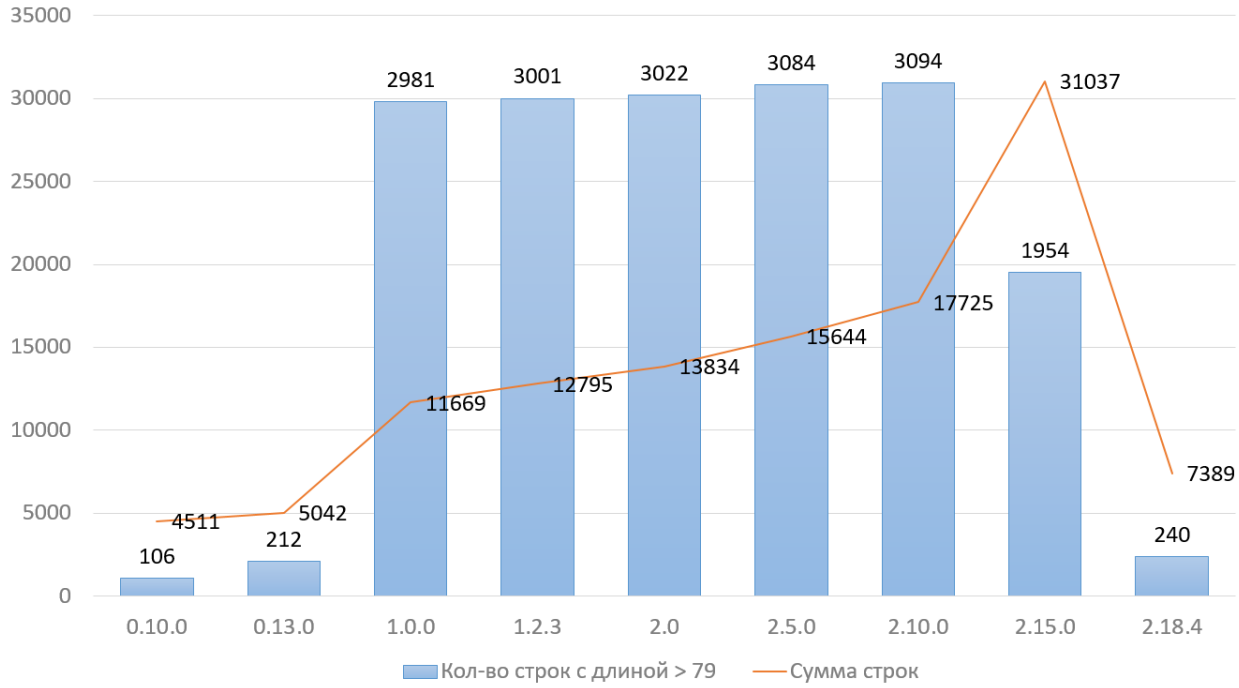


Рисунок 17 – Исследование максимальных длин строк репозитория requests

4.3 Исследование кода на наличие возможных проблем безопасности и стабильности работы

Статический анализ кода позволяет обнаружить ошибки не только в стиле написания кода. Он привлекает внимание к участкам кода, которые могут вызывать различные проблемы безопасности. Причем это не только проблемы, связанные с возможностями обхода проверок и взлома программы, но и ситуации, когда разработанное приложение ведет себя непредсказуемо. Например, возникают странные ошибки, которые очень тяжело поддаются

диагностированию, или приложение в случайные моменты времени непредсказуемо завершает работу.

В языке python существует конструкция *assert*, которая используется для проверки истинности указанного утверждения. Ее использование в отладочных целях позволяет быстро обнаружить ситуации, которые не являются ожидаемыми во время работы программы. Проверяемые данной конструкцией утверждения не являются исключительными ситуациями, которые следует дополнительно обрабатывать, а являются указанием, что код содержит ошибку.

В исследуемых репозиториях кода данная конструкция встречается. В табл. 4.6 приведено количество ее использований в программном коде, исключая описание автоматизированных тестов.

Таблица 4.6 – Кол-во использований конструкции assert

requests	**django**	**flask**	**httpie**	**thefuck**	**youtube-dl**	**ansible**
8	93	10	13	0	93	0

Несмотря на то, что использование *assert* видится хорошим способом для обнаружения ситуаций, которые никогда не должны происходить, к этой конструкции нужно относиться с осторожностью. В python нет строгого различия между режимами debug и release. Поэтому для автоматического игнорирования конструкций *assert* при сборке релизных версий необходимо указывать флаг *-O*, о чем можно забыть при компиляции своего программного кода и кода сторонних используемых библиотек. Следствием этого будет возникновение необработанных исключений при работе приложения у пользователей, что является недопустимым поведением.

Еще одной возможностью динамического языка программирования python является возможность компиляции и выполнения кода из строковой переменной во время работы программы с помощью конструкции *exec*. Количество использований в исследуемых репозиториях приведено в таблице 4.7.

Таблица 4.7 – Кол-во использований конструкции exec

requests	**django**	**flask**	**httpie**	**thefuck**	**youtube-dl**	**ansible**
1	6	2	0	0	2	3

Первым аргументом против использования данной конструкции является низкая производительность. Блок кода, передаваемый в *exec* будет компилироваться каждый раз во время выполнения. Следующим является опасность выполнения передаваемых инструкций, т.к. они могут быть получены не из доверенного источника и содержать в себе блоки кода, которые позволят осуществить выполнение кода преследуемого для взлома приложения и/или кражи данных.

Обработка исключений является мощным механизмом в любом языке программирования, и в языке python она также присутствует. В некоторых случаях при возникновении исключения его обработка может содержать лишь конструкцию *pass* или *continue*, что является простым игнорированием возникшей проблемы. Иногда это оправдано, например, когда попытка выполнения будет повторена или использован какой-то сценарий по умолчанию.

Однако такое игнорирование является плохой практикой написания кода и его лучше избегать. В случаях, когда вы точно уверены, что исключение можно полностью игнорировать обработчик всё равно следует

описать как минимум добавлением соответствующей информации в лог. Это упростит отладку программы и избавит от непредсказуемых ошибок в других участках программного кода.

Для исследуемого кода все сообщения об игнорируемых исключениях (табл. 4.8) требуют внимания разработчика, выполняющего обзор кода, для полной уверенности в избежании проблем, связанных с его использованием.

Таблица 4.8 – Кол-во сообщений об игнорировании исключений

requests	**django**	**flask**	**httpie**	**thefuck**	**youtube-dl**	**ansible**
0	7	6	1	1	3	149

Статический анализ позволяет обнаружить и множество других возможных проблем безопасности, которые также важны при разработке. Например, ошибки связанные с использованием библиотек кода, которые уличены в нестабильности работы и нарушении мер сохранности данных. Или ошибки при работе с SSL сертификатами из-за чего может быть нарушена коммуникация. Также это могут быть вероятные возможности исполнения вредоносного кода. Для исследуемых репозиториев с исходным кодом их количество приведено в табл. 4.9. Все подобные сообщения требуют внимания разработчика или исследователей безопасности продукта.

Таблица 4.9 – Кол-во сообщений о возможных проблемах безопасности

requests	**django**	**flask**	**httpie**	**thefuck**	**youtube-dl**	**ansible**
9	97	0	0	9	4	17

4.4 Исследование программного кода на наличие «мертвого кода» и неработающих ссылок

«Мертвый код» - это программный код, который уже никогда не выполняется в программе. Несмотря на это, он является проблемой при дальнейшей разработке и поддержке.

При написании программного кода, часто возникает желание переписать какую-то его часть для избавления от известных дефектов или просто для его улучшения. Старый код при этом зачастую комментируется вместо удаления и остается в проекте после окончания работы. Это плохой подход к разработке, т.к. со временем количество участков закомментированного кода будет увеличиваться, и они только будут отвлекать внимание.

Еще одной причиной комментирования кода вместо его удаления является ожидание, что он может потребоваться в будущем. Это также неправильный подход, т.к. все современные проекты разрабатываются с использованием систем контроля версий, поэтому удаленный код не будет потерян навсегда.

Закомментированный код всегда остается «зависшим» в проекте. Другие разработчики также не удаляют его, т.к. считают, что это временная мера, и работа над этим кодом будет продолжена другим программистом.

В исследуемых репозиториях количество сообщений о наличии закомментированного кода приведено в табл. 4.10.

Таблица 4.10 – Кол-во сообщений о наличии закомментированного кода

requests	**django**	**flask**	**httpie**	**thefuck**	**youtube-dl**	**ansible**
58	158	37	4	0	606	546

Некоторые используемые блоки кода могут быть неочевидны другим разработчикам. В таких случаях к ним добавляются комментарии, в которых,

для получения дополнительной информации, могут использоваться ссылки на внешние ресурсы. Со временем некоторые из таких ссылок могут стать неработоспособными, что повлечет за собой снижение качества комментария. По возможности, даже при использовании дополнительных источников, стоит полностью описывать основную идею в тексте комментария.

В исследуемом программном коде неработоспособные ссылки также встречаются, однако часть из них являются примерами url-адресов (напр., some.host.com).

Таблица 4.11 – Кол-во сообщений об использовании неработающих ссылок

requests	**django**	**flask**	**httpie**	**thefuck**	**youtube-dl**	**ansible**
71	173	15	5	2	757	678

ЗАКЛЮЧЕНИЕ

Контроль качества программного кода – актуальная задача для разработчиков, людей, участвующих в развитии проекта, а также тех, кто желает получить информацию о качестве программного кода и его временное изменение для иных целей. Для ее решения требуется инструмент, позволяющий уровень этого качества оценить, некоторый инструмент аналитики.

В рамках данной работы была разработана система автоматизированного анализа программного кода при помощи статических анализаторов, результатом работы которой является получение и обработка различных метрик программного кода.

В ходе разработки были решены все поставленные задачи:

- Рассмотрены существующие статические анализаторы программного кода для языка программирования python.
- Определен формат данных результатов анализа программного кода.
- Разработан подход для получения метрик программного кода при помощи статических анализаторов.
- Выполнено ряд исследований качества кода наиболее популярных GitHub репозиториев в категории языка программирования python с помощью разработанной системы.

Файлы исследований и программный код доступны в онлайн-хранилище GitHub [35-36]. Основные положения разработанной системы представлены в статье [37], индексируемой реферативной базой данных Scopus.

В дальнейшем компоненты разработанной системы можно дорабатывать в следующих направлениях:

- расширение набора отчетов анализа программного кода в веб-клиенте;

- добавление других статических анализаторов в модуль анализа программного кода;
- добавление модуля, позволяющего осуществлять автоматизированное скачивание репозиториев из онлайн-хранилищ и запуск указанных статических анализаторов на загруженном программном коде.

СЛОВАРЬ ТЕРМИНОВ

.NET: Платформа Microsoft для разработки и выполнения приложений.

.NET сборка: Базовая структурная единица в .NET.

CI (англ. Continuous Integration): Подход к разработке программного обеспечения, при котором выполняются частые слияния и автоматизированные сборки проекта для быстрого выявления и разрешения дефектов.

continue (python): Оператор, позволяющий пропустить часть логики цикла и перейти к выполнению следующей итерации.

IDE (англ. Integrated Development Environment): Интегрированная среда разработки программного обеспечения.

IoC-контейнер: Компоновщик, выполняющий подстановку объектов ООП (экземпляров класса) на место вызываемых интерфейсов во время исполнения программы.

pass (python): Оператор-заглушка, который равносилен отсутствию операции.

SSL (англ. Secure Sockets Layer – уровень защищённых сокетов): Криптографический протокол, подразумевающий более безопасную коммуникацию.

UI (англ. «User Interface»): Пользовательский интерфейс.

коммит (англ. «commit» – зафиксировать): Зафиксированное пользователем состояние репозитория.

лог (лог-файл): Файл, который содержит системную информацию о работе программы, в который пишется информация об определенных событиях и действиях пользователя.

плагин: Дополнение, позволяющее расширить функциональные возможности.

релиз: Версия продукта, содержащая дополнительную функциональность, полученная в результате одной или нескольких итераций разработки.

репозиторий: Хранилище данных.

рефакторинг (англ. refactoring – реорганизация): Изменение кодовой базы при неизменной функциональности программы. Выполняется для упрощения кода и повышения его читаемости.

форк (англ. «fork» – ответвление): Копирование программного кода из стороннего репозитория в свой для последующей модифицированной реализации продукта.

фреймворк (англ. framework – каркас, структура): Программное обеспечение, которое позволяет упростить и ускорить разработку приложения за счет использования сторонних компонентов.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] Звездин С. В. Проблемы измерения качества программного кода //Вестник Южно-Уральского государственного университета. Серия: Компьютерные технологии, управление, радиоэлектроника. – 2010. – №. 2 (178).

[2] Applying ISO 9001: 2000, MPS. BR and CMMI to achieve software process maturity: BL informatica's pathway //Proceedings of the 29th international conference on Software Engineering. – IEEE Computer Society, 2007. – С. 642-651.

[3] GitHub [Электронный ресурс]. Режим доступа: https://github.com, свободный.

[4] Кайгородцев Г. И. Введение в курс метрической теории и метрологии программ //Новосибирск: Изд-во НГТУ. – 2011.

[5] Fernandez E. B. et al. A methodology to develop secure systems using patterns //Chapter. – 2006. – Т. 5. – С. 107-126.

[6] Louridas P. Static code analysis //IEEE Software. – 2006. – Т. 23. – №. 4. – С. 58-61.

[7] Shabtai A., Fledel Y., Elovici Y. Automated static code analysis for classifying android applications using machine learning //Computational Intelligence and Security (CIS), 2010 International Conference on. – IEEE, 2010. – С. 329-333.

[8] Antunes N., Vieira M. Comparing the effectiveness of penetration testing and static code analysis on the detection of sql injection vulnerabilities in web services //Dependable Computing, 2009. PRDC'09. 15th IEEE Pacific Rim International Symposium on. – IEEE, 2009. – С. 301-306.

[9] Baca D. et al. Static code analysis to detect software security vulnerabilities-does experience matter? //Availability, Reliability and Security, 2009. ARES'09. International Conference on. – IEEE, 2009. – С. 804-810.

[10] Zitser M., Lippmann R., Leek T. Testing static analysis tools using exploitable buffer overflows from open source code //ACM SIGSOFT Software Engineering Notes. – ACM, 2004. – Т. 29. – №. 6. – С. 97-106.

[11] Menzies T., Greenwald J., Frank A. Data mining static code attributes to learn defect predictors //IEEE transactions on software engineering. – 2007. – Т. 33. – №. 1.

[12] Moser A., Kruegel C., Kirda E. Limits of static analysis for malware detection //Computer security applications conference, 2007. ACSAC 2007. Twenty-third annual. – IEEE, 2007. – С. 421-430.

[13] Звездин Сергей Владимирович Формирование метрик кода программной системы //Научно-технические ведомости СПбГПУ. Информатика. Телекоммуникации. Управление. 2008. №6 (69).

[14] Черноножкин С. К. Меры сложности программ (Обзор) //Системная информатика. – 1997. – №. 5. – С. 188-227.

[15] TIOBE Index | TIOBE - The Software Quality Company [Электронный ресурс]. Режим доступа: https://www.tiobe.com/tiobe-index/, свободный.

[16] Pylint – code analysis for Python [Электронный ресурс]. Режим доступа: https://www.pylint.org, свободный.

[17] pycodestyle's documentation – pycodestyle 2.4.0 documentation [Электронный ресурс]. Режим доступа: http://pycodestyle.pycqa.org, сободный.

[18] pyflakes – PyPI [Электронный ресурс]. Режим доступа: https://pypi.org/project/pyflakes, свободный.

[19] coala – linting and fixing code for all languages [Электронный ресурс]. Режим доступа: https://coala.io, свободный.

[20] pylama: Code audit tool for python [Электронный ресурс]. Режим доступа: https://github.com/klen/pylama, свободный.

[21] Flake8: Your Tool for Style Guide Enforcement [Электронный ресурс]. Режим доступа: http://flake8.pycqa.org, свободный.

[22] Нефедьева К. В. Инфографика визуализация данных в аналитической деятельности //Труды Санкт-Петербургского государственного университета культуры и искусств. – 2013. – Т. 197.

[23] Lanza M., Marinescu R. Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems. – Springer Science & Business Media, 2007.

[24] Lanza M., Ducasse S. Polymetric views-a lightweight visual approach to reverse engineering //IEEE Transactions on Software Engineering. – 2003. – Т. 29. – №. 9. – С. 782-795.

[25] Lanza M., Ducasse S. Understanding software evolution using a combination of software visualization and software metrics //In Proceedings of LMO 2002 (Langages et Modèles à Objets. – 2002.)

[26] Weiss M., Birukou A. Building a pattern repository: Benefitting from the open, lightweight, and participative nature of wikis //International Symposium on Wikis (WikiSym), ACM. – 2007. – С. 21-23.

[27] Встраиваемая система управления базами данных SQLite [Электронный ресурс]. Режим доступа: http://sqlite.org, свободный.

[28] Grid controls for ASP.NET MVC 5 projects [Электронный ресурс]. Режим доступа: https://github.com/NonFactors/MVC5.Grid, свободный.

[29] Simple yet flexible JavaScript charting for designers & developers [Электронный ресурс]. Режим доступа: https://www.chartjs.org, свободный.

[30] Comparison of source code hosting facilities [Электронный ресурс]. Режим доступа:
https://en.wikipedia.org/wiki/Comparison_of_source_code_hosting_facilities, свободный

[31] van Rossum G., Warsaw B., Coghlan N. PEP 8: style guide for Python code //Python. org. – 2001.

[32] Dawson B. Game scripting in Python //Proc. GDC. – 2002.

[33] O'Sullivan B., Goerzen J., Stewart D. B. Real world haskell: Code you can believe in. – " O'Reilly Media, Inc.", 2008.

[34] ISO/IEC/IEEE 9945 Information technology – Portable Operating System Interface (POSIX®) Base Specifications, Issue 7, Reference number ISO/IEC/IEEE 9945:2009(E)

[35] System designed for processing the results of open source repositories' discovery [Электронный ресурс]. Режим доступа: https://github.com/IPMITMO/statan-research, свободный.

[36] Sub-module of statan-research project, containing the source code of the developed system [Электронный ресурс]. Режим доступа: https://github.com/IPMITMO/statan, свободный.

[37] Chistiakov A.A., Pripadchev A., Radchenko I. On development of a framework for massive source code analysis using static code analyzers//ACM International Conference Proceeding Series, IET - 2017, Vol. Part F133327, pp. 3166114