Санкт-Петербургский государственный университет

Кафедра системного программирования

Группа 19Б.10-мм

Власов Илья Максимович

Разработка алгоритмов по оценке качества кода для онлайн-платформ Stepik и Hyperskill

Отчёт по учебной практике

Научный руководитель:
доц. кафедры СП, к. т. н. Т. А. Брыксин

Консультант:
программист-исследователь JetBrains Research А. И. Бирилло

Санкт-Петербург
2021

Оглавление

Введение **3**

1. Цель и задачи **5**

1.1. Цель 5

1.2. Задачи 5

2. Обзор **6**

2.1. Системы для оценки качества кода 6

2.2. Оценка качества кода в обучении 7

2.3. Система Hyperstyle 8

3. Реализация **11**

3.1. Улучшение процесса непрерывной интеграции 11

3.2. Анализаторы кода для языка Python 11

3.3. Алгоритм штрафов 16

4. Тестирование **24**

4.1. Тестовые данные 24

4.2. Тестирование новых анализаторов 24

4.3. Тестирование алгоритма штрафов 29

Заключение **32**

Список литературы **34**

Введение

Образование является неотъемлемой частью общества [3]. При этом в настоящий момент наблюдается процесс активной цифровизации образования [5, 11]. Из-за этого все больше людей получают доступ к образовательным онлайн-курсам, в частности, по программированию.

Однако некоторые работы [7, 8] указывают на то, что студенты во время обучения программированию очень редко обращают внимание на качество кода, фокусируясь на прохождении тестов. С другой стороны, умение писать качественный код является важным качеством профессионального программиста, так как это напрямую влияет на читаемость, сложность и поддержку такого кода.

Существует два основных способа оценки качества кода:

1. **Ручная оценка**. Такой вариант даст самые точные результаты, так как человек способен учитывать контекст, что не способны делать автоматические инструменты [8]. Однако, в рамках образовательных платформ, где необходимо проверять тысячи решений в минуту, такой вариант не подходит из-за высокой стоимости человеко-часов [10].

2. **Использование инструментов для автоматического анализа кода**. Данный подход не так эффективен как ручная оценка кода, но такие инструменты работают достаточно быстро, что позволяет использовать их в образовательных платформах. Однако, анализаторы кода позволяют находить лишь ограниченное множество ошибок, что не подходит для полноценной проверки кода [10, 12].

Как указано в работе Sakerman [10], помочь исправить минусы автоматических анализаторов смог бы инструмент, объединяющий результаты работы нескольких из них.

На образовательных онлайн-платформах Stepik [22] и Hyperskill [21] был создан подобный инструмент Hyperstyle [16], предназначенный для проверки качества кода на решениях студентов на Python, Java, Kotlin

и JavaScript. Стоит отметить, что на платформах Stepik и Hyperskill наиболее популярными являются языки программирования Python и Java, поэтому в первую очередь инструмент Hyperstyle направлен на поддержку именно их.

Одним из значительных недостатков данного инструмента можно отметить поиск в большей степени только стилистических ошибок. Однако студенты могут совершать и другие типы ошибок, касающихся сложности кода, использования языковых конструкций и пр. [7, 13]. Кроме того, для оценки качества кода существует множество метрик [9], используемых в настоящее время при разработке программного обеспечения.

Кроме того, другой из недостатков инструмента связан с невозможностью анализа предыдущих ошибок. Во время обучения программированию студенты, как показывают исследования, часто совершают одни и те же ошибки и не стремятся их исправлять [7]. Одним из способов контролирования данного процесса является использование алгоритмов, учитывающих историю прошлых ошибок при оценке качества кода студента. Таким образом, итоговая оценка качества будет зависеть не только от текущего кода студента, но и от его прошлых решений, указывая на слабые стороны, на которые следует обратить внимание при дальнейшей работе с задачами из курса. К сожалению, такого рода механизмы не поддерживаются в настоящее время инструментом Hyperstyle.

1. Цель и задачи

1.1. Цель

Целью работы является улучшение системы оценки качества кода Hyperstyle [16] для онлайн-платформ Stepik [22] и Hyperskill [21].

1.2. Задачи

Для выполнения цели были поставлены следующие задачи:

1. сделать обзор исследуемой области;
2. усовершенствовать процесс непрерывной интеграции (Continuous Integration, CI) системы оценки качества кода;
3. добавить поддержку различных анализаторов кода для языка Python;
4. разработать и внедрить алгоритм штрафов за повторные ошибки, совершенные студентом;
5. оценить качество решения на реальных данных.

2. Обзор

2.1. Системы для оценки качества кода

В настоящее время существует множество систем для анализа и оценки качества кода. Обычно такие инструменты включают в себя набор инспекций для анализа кода и ориентированы на промышленные продукты с большой кодовой базой.

Одной из таких систем является **Qodana**[1] — в основе которой лежит платформа Intellij[2]. Данная платформа является основой многих интегрированных сред разработки (Integrated Development Environment, IDE), разработанных в компании JetBrains[3]. Qodana для анализа использует специальную структуру данных, расширяющую абстрактное синтаксическое дерево (Abstract Syntax Tree, AST), и набор инспекций, оперирующих с ней.

Существуют и другие инструменты, позволяющие проводить более комплексный анализ за счет запуска общедоступных или проприетарных анализаторов кода. Примерами таких инструментов могут послужить **Codacy**[4] и **SonarQube**[5].

Такого рода инструменты, как правило, поддерживают популярные современные языки программирования и фреймворки, однако их целью не является обучение разработчика писать качественный код, а скорее, предотвратить появление неэффективного и трудно читаемого кода внутри продукта, автоматизировав часть процесса проверки. Кроме того, построение AST или же запуск инспекций для анализа кода является дорогостоящей по времени и памяти операцией, что делает невозможным использование таких инструментов, если необходимо производить их запуск на тысячах решений в минуту.

[1]Qodana: https://www.jetbrains.com/help/qodana/
[2]IntelliJ: https://www.jetbrains.com/idea/
[3]JetBrains: https://www.jetbrains.com/
[4]Codacy: https://www.codacy.com/
[5]SonarQube: https://www.sonarqube.org/

2.2. Оценка качества кода в обучении

Как свидетельствуют многочисленные научные работы [6, 7, 8], оценивать качество кода важно не только в промышленных продуктах, но и на этапе обучения программированию.

В работе [6] группа исследователей опросила 34 человека: 12 студентов, 11 преподавателей и 11 профессиональных разработчиков, с целью определения метрик, которые наиболее важны при оценке качества кода. По результатам исследования можно сделать вывод, что среди рассмотренных групп нет единого определения качества кода, однако все три группы выделяют некоторые признаки как наиболее важные. Так, например, *читаемость* и хорошая *структуризация* кода лидируют во всех группах. Однако, такие показатели как *документирование* кода или удовлетворение кода распространенным *метрикам оценки качества* значительно отставали в популярности у профессиональных разработчиков, в то время как лидировали у преподавателей и студентов. Кроме того, показатель *понятность* кода, который может показаться достаточно важным на первый взгляд, преподаватели отметили как наименее используемый признак при оценке качества, в отличие от остальных двух групп.

Похожий опрос провели в работе [8], где было задействовано 30 опытных учителей. В ходе работы было выявлено, что учителя имеют разные взгляды на то, как следует повышать качество кода, в отличие от анализаторов, подчиняющихся единому алгоритму работы. Кроме того, в большинстве случаев преподаватели обращают внимание на ошибки, связанные с алгоритмической сложностью или дублированием кода, однако, большинство анализаторов кода не способны находить подобные ошибки. Это связано не только с самой природой ошибок, но и с выставленными границами в конфигурационных файлах таких инструментов. Зачастую данные значения установлены исходя из общих практик промышленной разработки и не являются подходящими под специфику обучающих задач.

В работе [7] был проведен анализ набора данных, состоящих из ре-

шений задач на языке Java, собранных с помощью BlueJ[6] IDE. Целью исследования являлось определить набор наиболее часто встречающихся ошибок и изучить, как часто студенты исправляют такие ошибки. По итогам анализа, авторы показывают, что студенты делают больше всего ошибок в двух категориях: *выражения* (слишком сложные выражения) и *декомпозиция* (длинные функции и дублирование кода). При этом студенты реже всего исправляют ошибки, относящиеся к *модуляризации* (классы с низкой связностью и высокой связанностью, с большим количеством методов и атрибутов), что может быть связано с непониманием того, как исправлять данные ошибки.

Все вышеперечисленные работы показывают, что, с одной стороны, студенты часто совершают ошибки, приводящие к снижению качества кода, с другой стороны, существующие инструменты для анализа кода разрабатывались для применения в промышленных проектах и зачастую не предназначены для задач, используемых в курсах по обучению программирования.

2.3. Система Hyperstyle

Hyperstyle [16] — инструмент для запуска набора предварительно настроенных анализаторов кода и оценки качества кода. Он используется на платформах для онлайн-обучения программированию Stepik и Hyperskill для проверки качества кода учащихся.

В качестве основного языка разработки в Hyperstyle используется Python. К моменту начала работы над проектом основная архитектура уже была реализована и инструмент использовался на платформах. Для анализа кода были использованы следующие инструменты:

- **Python**: flake8[7], Pylint[8], PythonAST[9].

[6] BlueJ: https://bluej.org/
[7] flake8: https://flake8.pycqa.org/en/latest/
[8] Pylint: https://www.pylint.org/
[9] PythonAST: https://docs.python.org/3/library/ast.html

- **Java**: PMD[10], Checkstyle[11]
- **Kotlin**: Detekt[12]
- **JavaScript**: ESlint[13]

Общая схема работы инструмента выглядит следующий образом: на заданном фрагменте кода студента производится запуск заранее настроенных анализаторов кода, после чего результаты агрегируется и формируется итоговый отчет с оценкой качества кода и найденными ошибками.

Каждый анализатор представлен инспектором (*Inspector*), а также конфигурационным файлом, который позволяет настраивать набор запускаемых инспекций. *Inspector* запускает анализатор, после чего формируется список найденных ошибок.

Для каждой ошибки (*issue*) определяется ее тип с помощью заранее определенного конфигурационного файла. Существует несколько основных типов ошибок:

1. **Code style issue** означает, что ошибка нарушила одно из правил оформления кода.
2. **Best practice issue** означает, что код не соответствует общепринятым рекомендациям. Например, некоторые возможности языка могут быть использованы неэффективно.
3. **Error-prone issue** означает, что в коде содержится потенциальная ошибка, например в ряде случаев код может отрабатывать некорректно.
4. **Code complexity issue** означает, что код плохо спроектирован или имеет слишком сложную структуру.

[10]PMD: https://pmd.github.io/
[11]Checkstyle: https://checkstyle.sourceforge.io/
[12]Detekt: https://detekt.github.io/detekt/
[13]ESlint: https://eslint.org/

Кроме типа, ошибка может содержать некоторое значение, которое характеризует эту ошибку. Например, для ошибки «Связность» таким значением будет являться число, характеризующее недостаток связности (lack of cohesion).

Во избежание дублирования, ошибки, найденные всеми анализаторами, фильтруются от дубликатов, после чего группируются по анализируемым файлам.

Для каждого файла собирается статистика, в которую включены количество ошибок каждого типа и максимальное значение, которое характеризует ошибку, если оно имеется.

Для каждого типа ошибки существует правило (*Rule*), которое позволяет по статистике с помощью эмпирически определённых правил оценить качество кода по 4-бальной шкале.

Итогом работы инструмента является суммарная оценка, полученная путем агрегации полученных ранее оценок:

- **EXCELLENT** означает, что код соответствует всем требованиям.
- **GOOD** означает, что код читабельный и относительно легко редактируется, но в нём есть некоторые стилистические проблемы.
- **MODERATE** означает, что код только частично соответствует требованиям, и иногда его сложно понять.
- **BAD** означает, что код полностью не соответствует требованиям. Такой код трудно читать, понимать и редактировать.

3. Реализация

3.1. Улучшение процесса непрерывной интеграции

К моменту начала работы над проектом в нем были настроены GitHub Actions[1], позволяющие запускать тесты, а также анализировать код самого инструмента Hyperstyle при каждом его изменении (совершении коммита или открытие пулл-реквеста на GitHub). Такой подход позволяет контролировать процесс разработки и заблаговременно выявлять возникающие ошибки. Стоит отметить, что здесь рассматривается оценка *исходного кода* Hyperstyle сторонними инструментами для контроля процесса его разработки.

Для проверки качества кода используется статический анализатор `flake8`. Однако процесс непрерывной интеграции в проекте был выстроен не сразу, поэтому многие инспекции были отключены из проверки. В ходе работы над проектом одной из задач было снизить количество игнорируемых инспекций путем проведения рефакторинга кода. Результаты данной работы представлены в Таблице 1. В результате выполнения данной задачи количество игнорируемых ошибок снизилось с 10 до 4.

3.2. Анализаторы кода для языка Python

3.2.1. Схема работы анализаторов кода

Общая схема работы инструмента Hyperstyle представлена на рисунке 1.

Каждый анализатор кода представлен инспектором (*Inspector*), содержащий метод *inspect*, который запускает анализатор и возвращает список ошибок. Кроме того, если анализатор кода необходимо запустить не в стандартной конфигурации, например, отключить нахождение каких-либо ошибок, он должен содержать специальный конфигурационный файл, специфичный для конкретного анализатора.

[1]GitHub Actions: https://github.com/features/actions

Ошибка	Описание	Поддержка
I100	Неверный порядок импортов	✓
I101	Неверный порядок имён внутри импорта	✓
I202	Неверный перенос строки между импортами	✓
I201	Лишняя строка внутри группы импортов	✓
R504	Присвоение переменной перед `return`	✗
A003	Затенение встроенных переменных	✗
E800	Закомментированный код	✗
SC100	Орфографическая ошибка в комментариях	✓
SC200	Орфографическая ошибка в имени	✓
E402	Импорт не в начале файла	✗

Таблица 1: Поддержка игнорируемых инспекций

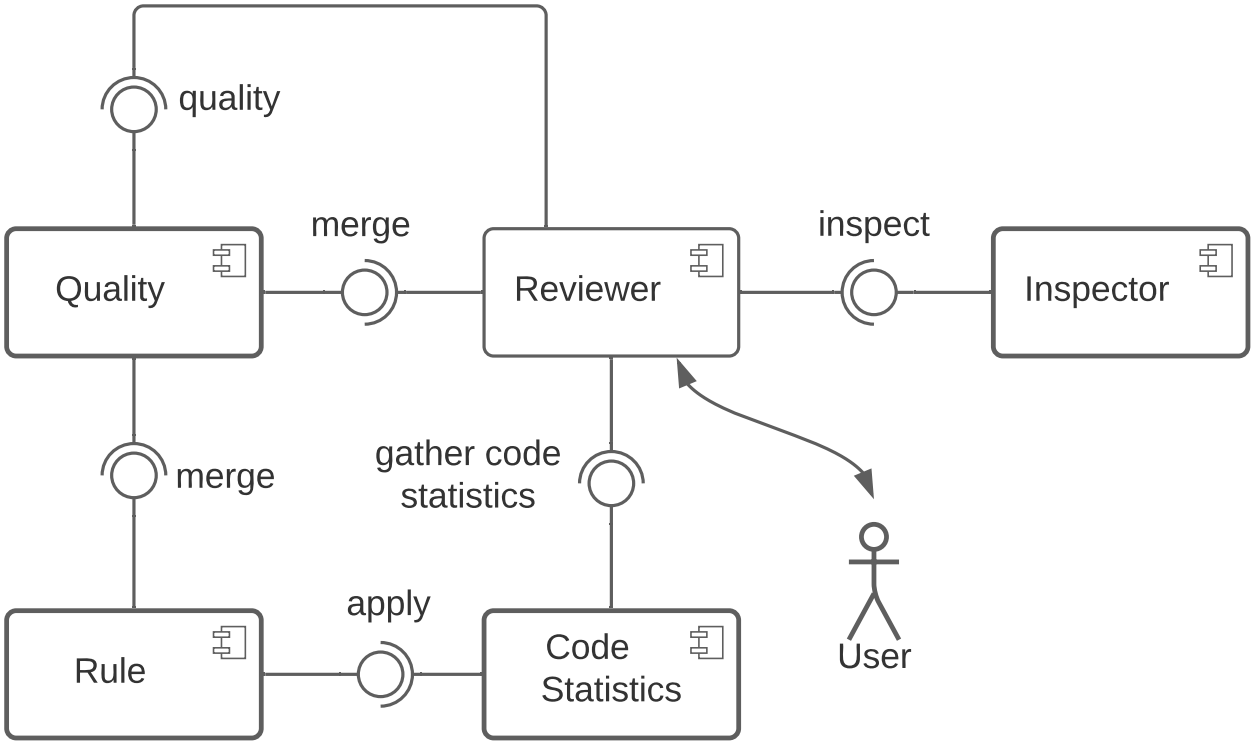

Рис. 1: Схема работы Hyperstyle

Каждая найденная ошибка содержит необходимую информацию, такую как путь к файлу, в котором была найдена ошибка, номер строки и столбца, описание, класс и тип. В случае, если ошибка является измеримой (например, для связности существует метрика «недостаток связности»), она хранит в себе значение для необходимой метрики.

Как было сказано в секции 2.3, инструмент Hyperstyle может находить ошибки нескольких типов: *code style issue*, *best practice issue*, *error-prone issue* и *code complexity issue*. Однако промышленные анализаторы зачастую не делят найденные ошибки по данным категориям, поэтому для корректной классификации ошибок, полученных от анализаторов кода, необходимо сопоставить каждой из ошибок необходимую категорию. В настоящее время это происходит с помощью конфигурационного файла, который создается вручную.

После того, как получены все найденные ошибки с сопоставленными ими категориями, необходимо оценить качество кода. Для каждого типа ошибки в каждой из категорий производится подсчет статистики. Статистика может содержать количество найденных ошибок данного типа или же максимальное значение, если ошибка является измеримой. Границы, по которым формируется финальная оценка (EXCELLENT, GOOD, MODERATE или BAD) для каждой из ошибок подобраны эмпирическим образом.

Рассмотрим пример вычисления оценки качества кода. Предположим, что в файле встретились ошибки, представленные в Таблице 2. Статистика по данным ошибкам представлена в Таблице 3. Столбец *Оценка* содержит итоговую оценку для каждой из категорий. Она получена путем сравнения значений из столбца *Статистика* с соответствующими порогами из конфигурационных файлов. Взяв минимум из найденных оценок, получаем итоговую оценку качества кода в файле — MODERATE.

Таким образом, для внедрения новых анализаторов кода необходимо сделать следующее:

1. добавить новый или изменить существующий *Inspector* с реализованной функцией *inspect* и необходимым конфигурационным фай-

лом;

2. добавить конфигурационный файл для категоризации ошибок анализатора по существующим категориям;

3. включить анализ новых ошибок в подсчет финальной оценки качества.

Ошибка	Категория	Частота	Метрика
SC200	BEST_PRACTICES	1	-
W0212	BEST_PRACTICES	3	-
WPS519	BEST_PRACTICES	1	-
WPS336	BEST_PRACTICES	1	-
C815	CODE_STYLE	2	-
H601	COHESION	-	20

Таблица 2: Список ошибок для примера расчета оценки качества кода инструментом Hyperstyle

Категория	Статистика	Оценка
BEST_PRACTICES	6 ошибок	MODERATE
CODE_STYLE	2 ошибки	GOOD
COHESION	20% не хватает до полной связности	GOOD

Таблица 3: Статистика для примера расчета оценки качества кода инструментом Hyperstyle

3.2.2. Плагины для flake8

В ходе работы над проектом была добавлена поддержка пяти плагинов для анализатора `flake8`:

- WPS (wemake-python-styleguide) [15]
- flake8-broken-line [18]

- flake8-string-format [20]
- flake8-commas [19]
- cohesion [17]

Так как плагины являются частью инспектора `flake8`, то их внедрение подразумевало классификацию ошибок, которые способен найти плагин, изменение конфигурационного файла для запуска инспектора, а также учет найденных ошибок в подсчете финальной оценки качества.

WPS (wemake-python-styleguide) [15] — плагин для `flake8`, который позиционируется как самый точный анализатор кода для Python. Данный плагин способен поддерживать более 200 инспекций[2]. Так как не все ошибки были подходящими для работы с образовательными задачами, часть из них (около 20) была исключена из анализа, а остальным присвоена необходимая категория. После проведенного самостоятельного анализа всех инспекций была получена экспертная консультация от руководителей проекта для финального разделения инспекций по категориям. Результаты данного анализа представлены в таблице[3].

flake8-broken-line [18], **flake8-string-format** [20], **flake8-commas** [19] — плагины для `flake8`, проверяющие правильность переноса строк, форматирование строк и правильность расстановки запятых соответственно. При их добавлении потребовалось изменить конфигурационный файл `flake8`, чтоб он корректно определял категорию ошибок.

cohesion [17] — плагин для `flake8`, который позволяет вычислять связность внутри класса. В связи с тем, что ранее инструмент не умел находить недостаток связности в классах, при добавлении плагина, помимо изменения конфигурационного файла `flake8`, потребовалось добавить новый тип ошибки (*Cohesion*), определить новое правило (*CohesionRule*) для её оценивания, указав допустимые значения для каждой оценки, и включить данный тип в подсчёт финальной оценки.

[2]Список всех инспекций WPS: https://wemake-python-stylegui.de/en/latest/pages/usage/violations/index.html

[3]Таблица с распределением: https://bit.ly/3yms06n

3.2.3. Анализатор кода Radon

Radon [14] — статический анализатор кода, который позволяет вычислять несколько метрик кода: *цикломатическую сложность*, *метрики Холстеда* [4], *индекс ремонтопригодности* и ряд других «сырых» метрик (количество строк кода, количество пустых строк и так далее). Так как поддержки данного анализатора ранее не было, необходимо было добавить его с нуля. Среди вычисляемых анализатором метрик была выбрана метрика *индекс ремонтопригодности*[4]. Это связано с тем, что некоторые метрики вычисляются с помощью других анализаторов, например *цикломатическая сложность*, а некоторые, такие как *метрики Холстеда* напрямую не подходят для оценки качества, так как используются при вычислении других метрик (например, *индекса ремонтопригодности*).

3.3. Алгоритм штрафов

Так как во время обучения студенты часто совершают одни и те же ошибки и не стремятся их исправлять, то возникла необходимость в создании алгоритма, который по истории ошибок смог бы штрафовать студента за их повторение.

3.3.1. Взаимодействие с платформой

Инструмент Hyperstyle не является частью платформ Stepik и Hyperskill. Он вызывается как внешний инструмент, принимая данные через аргументы командой строки и возвращая результат в виде JSON файла, в котором указана оценка за каждый файл и проект целиком, а также список найденных ошибок.

Листинг 1 демонстрирует пример JSON файла с результатом работы инструмента. Из него видно, что общее качество кода в проекте оценено как *GOOD*, а также представлен список ошибок для файла *main.py*, из

[4]Формула для вычисления индекса ремонтопригодности: https://radon.readthedocs.io/en/latest/intro.html#maintainability-index

которого для каждой ошибки можно определить не только саму ошибку и ее категорию, но и позицию, которая используется при подсветке ошибки в онлайн-редакторе на платформах Stepik и Hyperskill.

Листинг 1: Пример результата работы инструмента Hyperstyle

```
{
   ''quality'':{
      ''code'':''GOOD'',
      ''text'':''Code quality (beta): GOOD''
   },
   ''file_review_results'':[
      {
         ''file_name'':''main.py'',
         ''quality'':{
            ''code'':''GOOD'',
            ''text'':''Code quality (beta): GOOD''
         },
         ''issues'':[
            {
               ''code'':''WPS331'',
               ''text'':''Found variables that are only used for
                  ↪ return'',
               ''line'':''return result'',
               ''line_number'':3,
               ''column_number'':5,
               ''category'':''BEST_PRACTICES''
            }
         ]
      }
   ]
}
```

Однако для успешной реализации алгоритма штрафов инструмент Hyperstyle должен получать из платформ информацию об истории предыдущих ошибок. Для этого был предложен аналогичный формат взаимодействия путем передачи специального JSON файла. Так как инструмент поддерживает работу с несколькими языками, ошибки в таком JSON файле распределены по языкам, кроме того, каждая ошибка

содержит информацию о том, сколько раз она встречалась ранее. Листинг 2 демонстрирует пример JSON файла с историей ошибок пользователя.

Листинг 2: Пример истории предыдущих ошибок

```
{
   ''python'':[
      {
         ''origin_class'':''SC200'',
         ''number'':20
      },
      {
         ''origin_class'':''WPS314'',
         ''number'':3
      }
   ]
}
```

История формируется по каждому отдельному пользователю за предыдущие 30 дней от текущего момента взаимодействия с платформой. Это связано с тем, что не совсем корректно штрафовать пользователя за давние ошибки. Также цифра 30 выбрана по той причине, что из-за особенностей текущего взаимодействия с платформой, файл передаётся непосредственно как аргумент командной строки, размер которого ограничен.

3.3.2. Алгоритм

Для того, чтобы понять, на сколько следует снижать итоговую оценку студента, следует вычислить *штрафной коэффициент*. Псевдокод алгоритма его нахождения представлен в Листинге 3.

В строчке 2 получаем список *штрафующих ошибок*. Он формируется путем пересечения списка текущих и предыдущих ошибок. Это означает, что список *штрафующих ошибок* содержит только те ошибки, которые пользователь уже совершал за последние 30 дней в других задачах, но при этом допустил такие же ошибки и в текущем решении.

Листинг 3: Алгоритм нахождения штрафного коэффициента

```
1  def get_penalty_coefficient(current, previous):
2      penalizing_issues = get_penalizing_issues(current, previous)
3
4      nonnormalized_penalty_coefficient = 0
5      for issue in penalizing_issues:
6          nonnormalized_penalty_coefficient += issue.number *
           ↪ get_category_coefficient(issue.category)
7
8      penalty_coefficient = nonnormalized_penalty_coefficient / (
       ↪ nonnormalized_penalty_coefficient + len(previous))
9
10     return penalty_coefficient
```

В строчках 4–6 вычисляется *ненормированный штрафной коэффициент*, зависящий от частоты текущей ошибки в прошлом и *коэффициента категории ошибки. Коэффициент категории ошибки* находится в диапазоне от 0 до 1 и показывает то, насколько следует учитывать данную категорию при вычислении штрафа. *Ненормированный штрафной коэффициент* далее будет использован при вычислении штрафного коэффициента.

В строчке 8 нормируем коэффициент по следующей формуле:

$$k = \frac{k'}{k' + n}$$

где k' — ненормированный штрафной коэффициент, n — количество текущих ошибок.

Таким образом, в результате работы алгоритма получаем штрафной коэффициент, находящийся в диапазоне от 0 до 1. В зависимости от полученного значения, исходная оценка снижается на 1, 2 или 3 уровня. Возможные итоговые оценки с учетом работы алгоритма штрафов представлены в Таблице 4.

Пороги, позволяющие определить, на сколько уровней снижается оценка для различных значений штрафного коэффициента, были определены эмпирически.

Итоговая оценка \ Исходная оценка	EXCELLENT	GOOD	MODERATE	BAD
EXCELLENT	$[0, 0.5)$	—	—	—
GOOD	$[0.5, 0.7)$	$[0, 0.5)$	—	—
MODERATE	$[0.7, 0.9)$	$[0.5, 0.7)$	$[0, 0.5)$	—
BAD	$[0.9, 1]$	$[0.7, 1]$	$[0.5, 1]$	$[0, 1]$

Таблица 4: Влияние штрафного коэффициента на итоговую оценку

Алгоритм был реализован в отдельном модуле (*Punisher*) и интегрирован в существующую архитектуру как показано на рисунке 2.

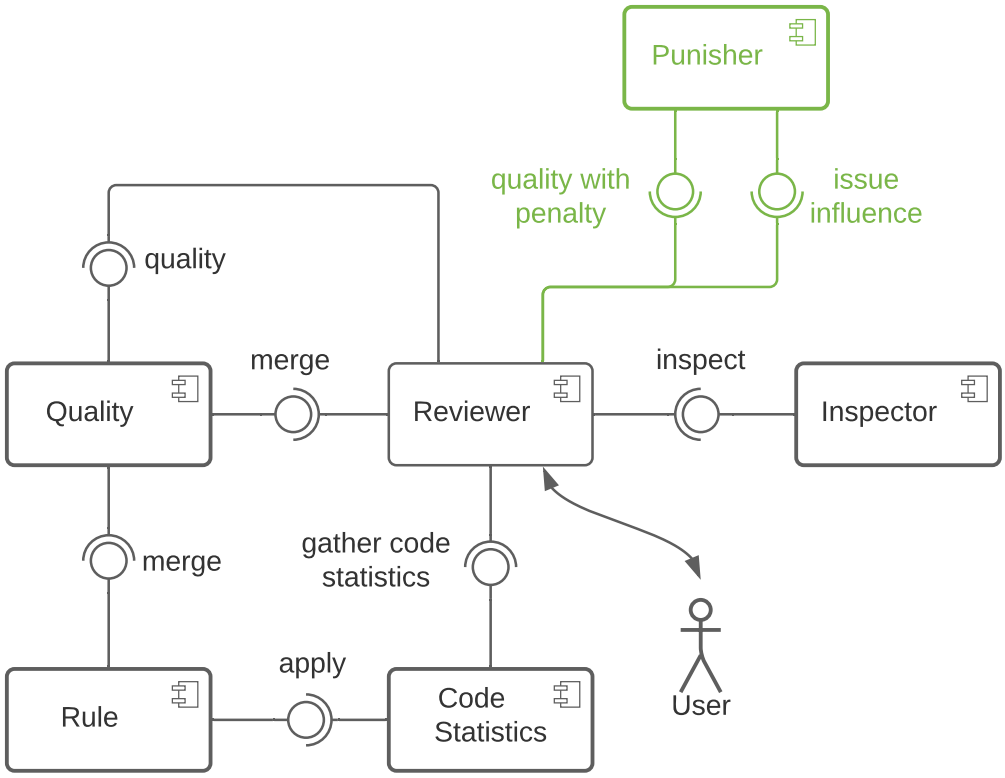

Рис. 2: Интеграция Punisher в существующую архитекуру

Для выделения наиболее влияющих на качество кода ошибок на стороне платформ Stepik и Hyperskill, для каждой ошибки в финальном JSON файле, представленного на Листинге 1, указывается дополнительное поле «influence_on_penalty», значение которого можно интерпретировать как процент влияния данной ошибки на штрафной коэф-

фициент.

3.3.3. Настройка коэффициентов

Для определения коэффициентов для каждой категории было изучено несколько научных работ [6, 7, 13], анализирующих качество кода в учебных курсах.

В результате проведенного анализа было принято решение оценить каждую категорию по 3 критериям: распространённость (prevalence, PR), трудность исправления (difficulty, DIF) и важность (importance, IMP). Оценка происходила с использованием 5-бальной шкалы по следующему правилу для каждого критерия:

- **Распространённость (PR)**: 5 – самый распространённый, 1 – самый нераспространенный.
- **Трудность исправления (DIF)**: 5 – легко исправить, 1 – трудно исправить.
- **Важность (IMP)**: 5 – очень важен, 1 – совсем не важен.

Итоговый коэффициент вычисляется по формуле:

$$k = \frac{PR + DIF + IMP}{max}$$

где max – максимальное значение суммы по всем категориям.

В результате для каждой категории получен штрафной коэффициент, лежащий в диапазоне от 0 до 1. Таблица 5 демонстрирует описание используемых в проекте категорий, а Таблица 6 — результаты проведенного анализа.

Категория	Описание
CODE_STYLE	Следование общепринятым стандартам оформления кода
LINE_LEN	Длина строки
BEST_PRACTICES	Использование общепринятых рекомендаций по использованию языка
CYCLOMATIC_COMPLEXITY	Цикломатическая сложность
ERROR_PRONE	Наличие потенциальных ошибок
COUPLING	Связанность между классами
COHESION	Связность внутри класса
FUNC_LEN	Длина функции
BOOL_EXPR_LEN	Длина логического выражения
COMPLEXITY	Код излишне сложен
MAINTAINABILITY	Индекс ремонтопригодности
METHOD_NUMBER	Количество методов в классе
CHILDREN_NUMBER	Количество дочерних классов
WEIGHTED_METHOD	Сумма цикломатических сложностей по всем методам
CLASS_RESPONSE	Количество вызовов уникальных методов в классе
INHERITANCE_DEPTH	Количество классов между текущим и суперклассом

Таблица 5: Описание используемых категорий

Категория	PR	DIF	IMP	Коэффициент
CODE_STYLE	5	4	5	1.0
LINE_LEN	4	5	5	1.0
BEST_PRACTICES	5	3	5	0.9
CYCLOMATIC_COMPLEXITY	4	3	3	0.7
ERROR_PRONE	3	4	2	0.6
COUPLING	4	1	4	0.6
COHESION	4	1	4	0.6
FUNC_LEN	3	3	2	0.6
BOOL_EXPR_LEN	3	2	3	0.6
COMPLEXITY	2	2	3	0.5
MAINTAINABILITY	1	1	2	0.3
METHOD_NUMBER	1	1	1	0.2
CHILDREN_NUMBER	1	1	1	0.2
WEIGHTED_METHOD	1	1	1	0.2
CLASS_RESPONSE	1	1	1	0.2
INHERITANCE_DEPTH	1	1	1	0.2

Таблица 6: Оценка используемых категорий по 3 критериям: распространённость (PR), трудность исправления (DIF) и важность (IMP)

4. Тестирование

4.1. Тестовые данные

В качестве тестовых данных были использованы решения пользователей с платформы Stepik в период с 3 апреля по 3 мая 2021 года (30 дней).

Данные представляли собой таблицу, состоящую из нескольких столбцов: идентификатор (ID) решения, время отправки, фрагмент кода, язык, идентификатор (ID) пользователя и флаг публичности (согласен ли пользователь на публикацию его решения).

Тестовые данные представляют собой решения пользователей, которые не могут быть выложены в открытый доступ для повсеместного использования.

Всего было получено 163 457 решений на языке Python, из которых 68 859 решений содержат уникальные фрагменты кода.

4.2. Тестирование новых анализаторов

После добавления новых анализаторов инструмент научился находить более 230 новых инспекций.

Для проверки корректности вычисления оценки качества кода после внесенных в инструмент изменений было проведено сравнение оценок, найденных старой версией инструмента, и оценок, найденных новой версией инструмента на всех уникальных решениях. В новой версии инструмента оценка должна остаться прежней или стать хуже из-за найденных новых инспекций. В результате тестирования было выяснено, что для всех фрагментов кода обновленный инструмент работает корректно.

Далее, для каждого фрагмента были определены инспекции, которых не было в прошлой версии. На рисунке 3 показано, что в 39 830 уникальных фрагментах из 68 859 были обнаружены новые инспекции, что составляет 58% от общего числа фрагментов.

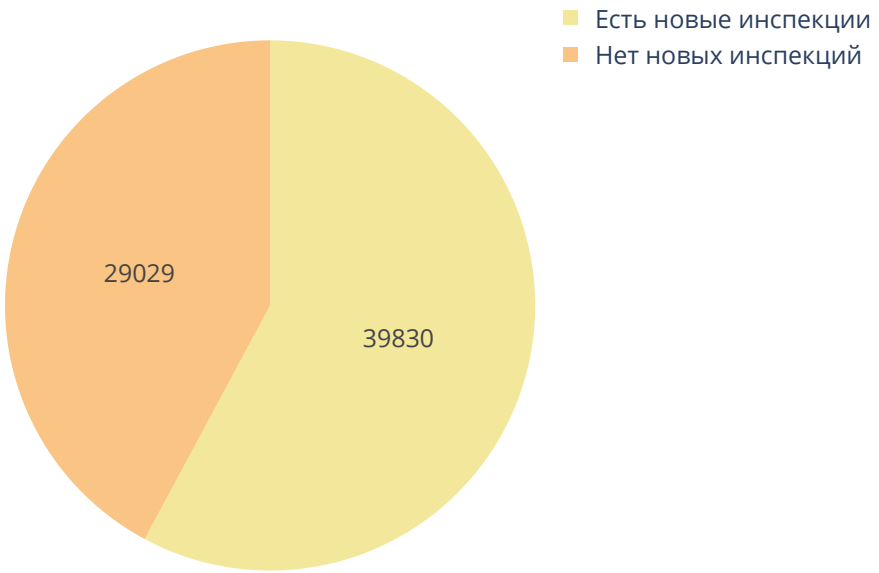

Рис. 3: Соотношение уникальных фрагментов, содержащих новые инспекции, и уникальных фрагментов, в которых они не присутствовали.

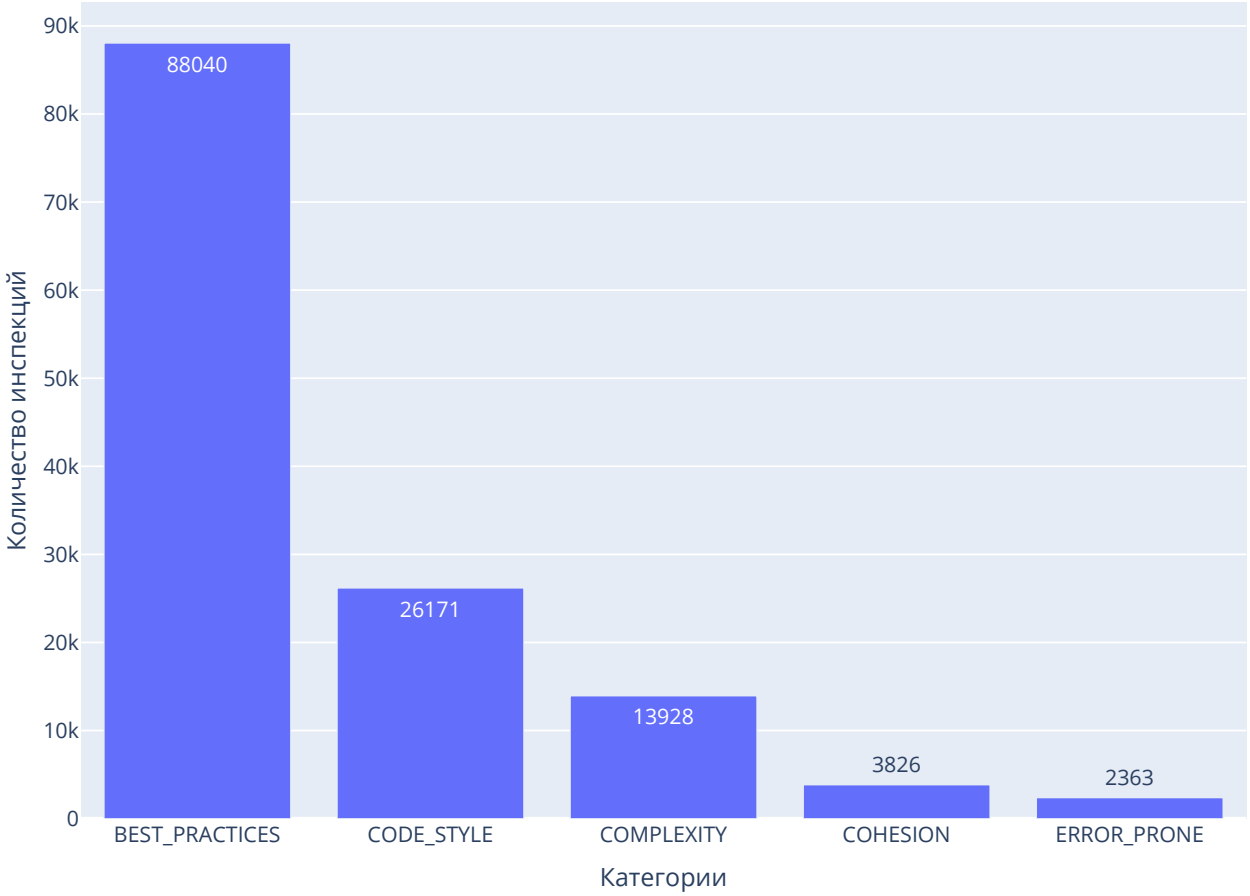

Рис. 4: Количество найденных новых инспекций, сгруппированных по категориям

До обновления инструмента, он позволял находить в основном ошибки, относящиеся к категории CODE_STYLE. Новые инспекторы позволили выявлять ошибки из таких категорий, как BEST_PRACTICES, COHESION и ERROR_PRONE.

На рисунке 4 видно, что большинство найденных новых инспекций относится к категории BEST_PRACTICES (около 88 тысяч). Более чем в 3 раза реже встречались ошибки из категории CODE_STYLE (около 26 тысяч). Менее всего встречались ошибки из категории COHESION и ERROR_PRONE.

В таблице 7 показаны 10 наиболее популярных ошибок, найденных после добавления новых анализаторов. Из таблицы видно, что ошибка `SC200` встретилась почти в 4 раза чаще, чем следующая за ней ошибка `WPS432`. Данный выброс нуждается в дальнейшем изучении, и может быть связан с ложными срабатываниями плагина `flake8-spellcheck`, отвечающий за нахождение ошибок в именах функций, переменных и так далее. Кроме того, видно, что ошибки, не принадлежащие категории CODE_STYLE, встречаются довольно часто среди студентов, что подтверждает необходимость использования новых инспекций.

В таблице 8 показан результат работы инструмента для 3 фрагментов. Данные фрагменты помечены флагом публичности, то есть их публикация разрешена. Из таблицы видно, что при работе с фрагментом 1 [23] старая версия инструмента не находила ошибок и выставляла максимальную оценку. При этом новая версия инструмента обнаружила 9 ошибок. Кроме того, для оставшихся фрагментов большинство найденных ошибок обнаружены за счет добавления новых инспекций в инструмент.

Подводя итог всего вышесказанного, добавление новых анализаторов кода позволяет находить ошибки в большинстве решений и ожидаемо влиять на итоговую оценку качества кода. Однако, следует уделить особое внимание некоторым инспекциям, которые встречаются неоправданно часто, например, `WPS432`. Такие инспекции нуждаются в дополнительном изучении на предмет того, действительно ли их стоит включать в итоговый инструмент.

Ошибка	Описание	Категория	Количество
SC200	Орфографическая ошибка в имени	BEST_PRACTICES	64 435
WPS432	Магическое число	CODE_STYLE	17 477
WPS221	Излишне сложная строчка	COMPLEXITY	10 618
WPS336	Явная конкатенация строк вместо форматирования	BEST_PRACTICES	4965
H601	Низкая связность	COHESION	3826
SC100	Орфографическая ошибка в комментарии	BEST_PRACTICES	2719
WPS319	Неправильно расставленные скобки	CODE_STYLE	2655
WPS317	Неправильные отступы	CODE_STYLE	2575
WPS515	`open()` без менеджера контекста	BEST_PRACTICES	1783
WPS503	Ненужная `else`-ветка	BEST_PRACTICES	1611

Таблица 7: 10 наиболее популярных ошибок, найденных новыми анализаторами кода

Фрагмент	Ошибка	Описание	Всего	Новых
№1 [23]	SC100 (2)	Орфографическая ошибка в имени	9	9
	WPS221	Излишне сложная строчка		
	WPS237	Излишне сложная f-строчка		
	WPS432 (2)	Магическое число		
	WPS326	Неявная конкатенация		
	WPS319	Неправильно расставленные скобки		
	H601	Низкая связность		
№2 [24]	WPS221	Излишне сложная строчка	11	6
	E231	Отсутствующий пробел		
	C819	Замыкающая запятая в неверном месте		
	SC200	Орфографическая ошибка в имени		
	WPS442	Лямбда-выражение внутри цикла		
	C0200 (4)	Неявный вызов `enumerate()`		
	WPS519	Неявный вызов `sum()`		
	WPS336	Явная конкатенация вместо форматирования		
№3 [25]	N400	Использован \ для переноса строки	7	6
	E124	Неправильный отступ		
	C812	Замыкающая запятая отсутствует		
	WPS432	Магическое число		
	WPS221	Излишне сложная строчка		
	WPS317 (2)	Неправильный отступ		

Таблица 8: Найденные ошибки на нескольких публичных фрагментах кода. В скобках указано количество ошибок данного типа. Новые ошибки подчёркнуты.

4.3. Тестирование алгоритма штрафов

Для тестирования алгоритма штрафов был использован набор данных, который был описан в секции 4.1. Из 163 457 фрагментов, распределённых по 4671 пользователям, только в 43 512 на оценку качества кода повлияла история. Это означает, что на остальных фрагментах либо история отсутствовала, либо ошибки у пользователя не повторялись.

Для проверки корректности работы инструмента после внедрения алгоритма штрафов необходимо было проверить, что оценка, найденная с применением алгоритма будет *не больше*, чем оценка, найденная без него. В результате тестирования на всех фрагментах отрицательных результатов не было получено.

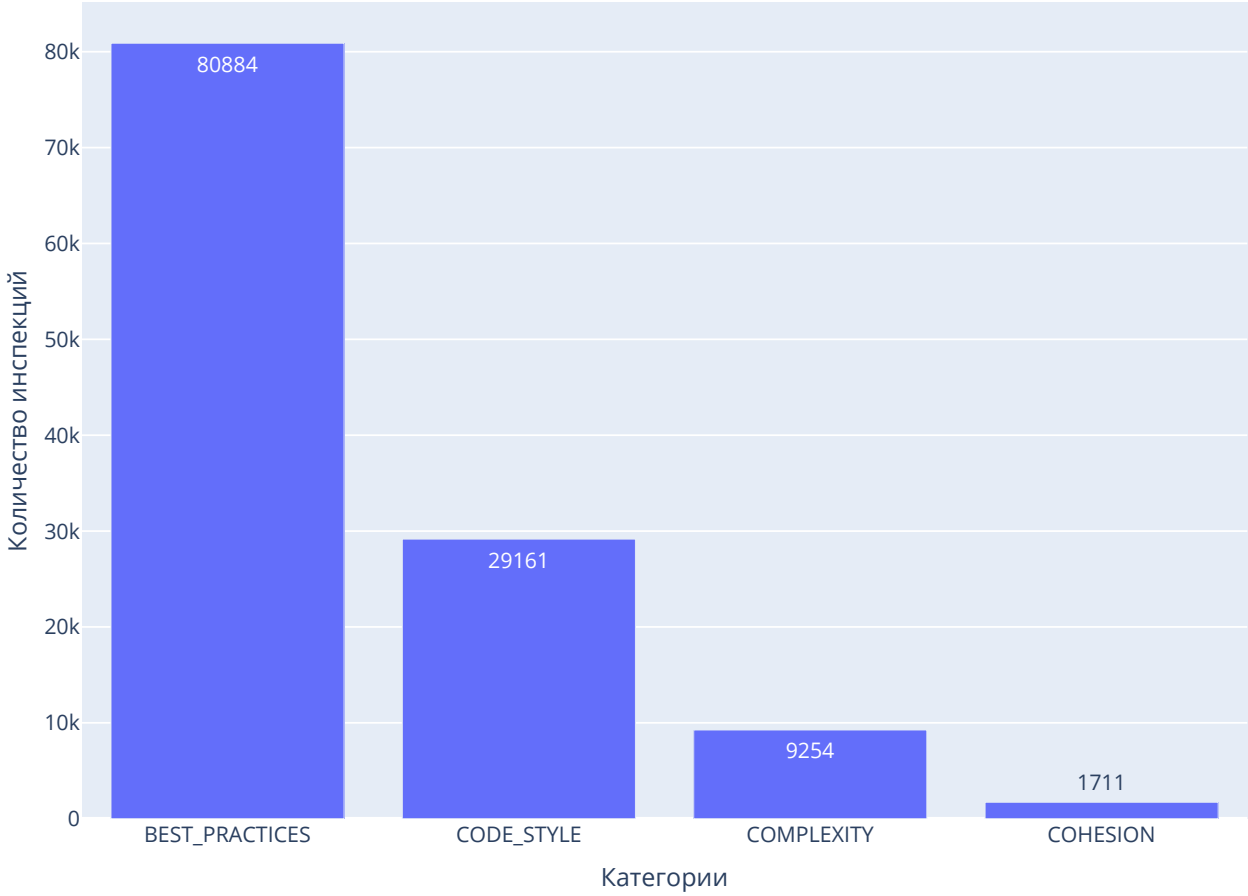

Рис. 5: Количество найденных новых инспекций, повлиявших на штраф и сгруппированных по категориям

Как показано на рисунке 5, больше всего инспекций, которые повлияли на штраф, пришлось на категорию BEST_PRACTICES. Примерно в 2.5 раза реже, встречаются подобные инспекции из категории

CODE_STYLE. Менее всего встречались ошибки из категории COMPLEXITY и COHESION. Данные результаты согласуются с общим распределением новых инспекций, представленные на рисунке 4. Стоит отметить, что ошибки из остальных категорий никак не влияли на итоговую оценку. Это может быть связано с тем, что в выборке присутствовало большое количество несложных задач, в которых не всегда могут быть обнаружены ошибки из остальных категорий.

Как показывает рисунок 6, наиболее влиятельными на итоговую оценку оказались ошибки из таких категорий, как BEST_PRACTICES и CODE_STYLE. Ошибки из категорий COMPLEXITY и COHESION влияли в меньшей степени. Данные результаты согласуются с полученными коэффициентами в таблице 6.

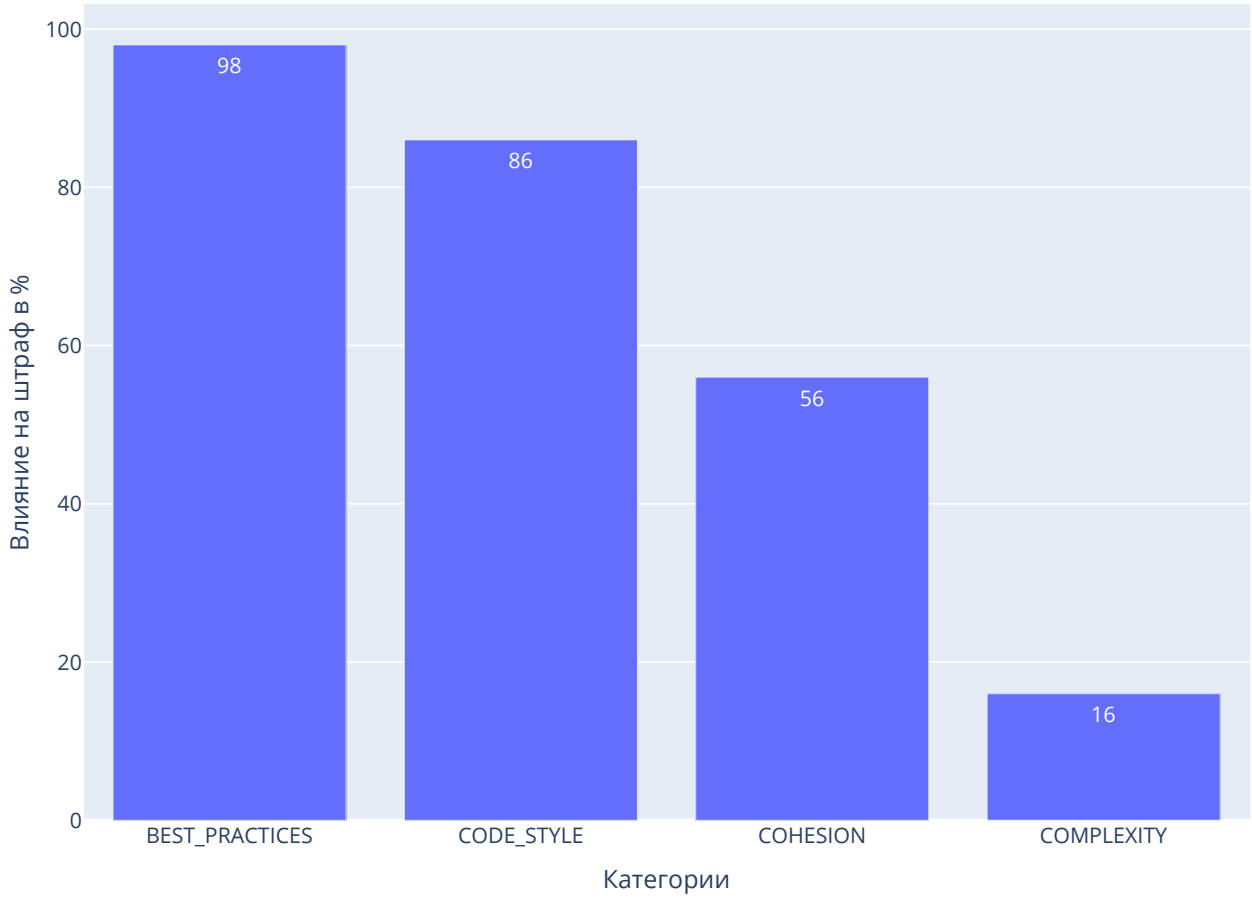

Рис. 6: Влияние алгоритма штрафов на итоговую оценку по каждой категории. Представлены медианные значения

В таблице 9 показаны 10 наиболее популярных ошибок, повлиявших на штраф. Можно заметить, что большинство ошибок из данной таб-

лицы есть в таблице 7, которая описывает самые популярные ошибки, найденные после добавления новых анализаторов. Данная закономерность не удивительна, так как алгоритм штрафов учитывает количество ранее повторённых ошибок.

Ошибка	Описание	Категория	Количество
SC200	Орфографическая ошибка в имени	BEST_PRACTICES	59 467
WPS432	Магическое число	CODE_STYLE	13 169
WPS221	Излишне сложная строчка	COMPLEXITY	8212
SC100	Орфографическая ошибка в комментарии	BEST_PRACTICES	5288
E226	Отсутствует пробел около оператора	CODE_STYLE	3537
WPS336	Явная конкатенация строк вместо форматирования	BEST_PRACTICES	3034
WPS515	`open()` без менеджера контекста	BEST_PRACTICES	2468
E231	Отсутствующий пробел после ‘`,`’, ‘`;`’, ‘`:`’	CODE_STYLE	2183
H601	Низкая связность	COHESION	1711
E225	Отсутствует пробел около оператора	CODE_STYLE	1532

Таблица 9: 10 наиболее популярных ошибок, повлиявших на штраф

Таким образом, данный алгоритм действительно позволил в разной степени учитывать предыдущие ошибки пользователя при вычислении оценки качества кода. Однако, действительно ли алгоритм помогает пользователям учиться исправлять прошлые ошибки, стоит проверить позднее, после внедрения алгоритма в платформу.

Заключение

В рамках данной работы была улучшена система оценки качество кода Hyperstyle для онлайн-платформ Stepik и Hyperskill.

При работе были выполнены все поставленные задачи:

- сделан обзор исследуемой области;
- усовершенствован процесс непрерывной интеграции системы оценки качества кода;
- добавлена поддержка различных анализаторов кода для языка Python;
- разработан и внедрён алгоритм штрафов за повторные ошибки, совершенные студентом;
- оценено качество решения на реальных данных.

С исходным кодом инструмента можно ознакомиться в официальном репозитории [16].

Новая версия инструмента с добавленными анализаторами уже в ближайшее время будет использоваться на платформах, а поддержка алгоритма штрафов будет интегрирована в платформу в течении месяца.

Работа над данным проектом будет продолжена в рамках летней стажировки. Во время стажировки планируется выполнить следующие задачи:

- Добавить новые анализаторы кода для Java, JavaScript и Kotlin;
- Добавить персонализацию:
  - разработать модель машинного обучения для выделения концепций по описанию и решениям задачи;
  - внедрить алгоритм по выделению концепций в алгоритм подсчета оценки качества кода;

- Добавить алгоритм выявления наиболее сложных для исправления инспекций, основанный на теории тестовых заданий (Item Response Theory, IRT) [2];
- Разработать модель машинного обучения для имитации работы статических анализаторов, таких как Qodana.

Работа над последней задачей ведется в данный момент. В рамках данной задачи был размечен датасет с помощью специальных скриптов по запуску Qodana и перевода данных в необходимый для модели формат.

Список литературы

[1] 22nd Annual Conference on Innovation and Technology in Computer Science Education. — Bologna, Italy, 2017. — July.

[2] Berges Marc, Hubwieser Peter. Evaluation of Source Code with Item Response Theory. — 2015. — July.

[3] Global citizenship and the importance of education in a globally integrated world / William C Smith, Pablo Fraser, Volha Chykina et al. // Globalisation, Societies and Education. — 2017. — Vol. 15, no. 5. — P. 648–665.

[4] Halstead Maurice H. Elements of Software Science (Operating and Programming Systems Series). — USA : Elsevier Science Inc., 1977. — ISBN: 0444002057.

[5] He Wu, Xu Guandong, Kruck SE. Online IS education for the 21st century // Journal of Information Systems Education. — 2019. — Vol. 25, no. 2. — P. 101–105.

[6] "I know it when I see it" – Perceptions of Code Quality / Jürgen Börstler, Harald Störrle, Daniel Toll et al. // Proceedings of ITiCSE'17. — Bologna, Italy, 2017. — July.

[7] Keuning Hieke, Heeren Bastiaan, Jeuring Johan. Code Quality Issues in Student Programs // Proceedings of ITiCSE'17. — Bologna, Italy, 2017. — July. — P. 110–115.

[8] Keuning Hieke, Heeren Bastiaan, Jeuring Johan. How Teachers Would Help Students to Improve Their Code // Proceedings of ITiCSE'19. — Aberdeen, Scotland UK, 2019. — July.

[9] Maria Caulo, Giuseppe Scanniello. A Taxonomy of Metrics for Software Fault Prediction // Proceedings of 2020 Euromicro Conference on Software Engineering and Advanced Applications (SEAA). — Portoroz, Slovenia, 2020. — August.

[10] Sakerman Liisa. Overview of the advantages and disadvantages of static code analysis tools. — 2021.

[11] Schmidt Joel T, Tang Min. Digitalization in Education: Challenges, Trends and Transformative Potential // Führen und Managen in der digitalen Transformation. — Springer, 2020. — P. 287–312.

[12] Static Versus Dynamic Source Code Analysis / Korhan Akcura, Reza Shalchian, Abhijit Patil et al. — 2015. — November.

[13] Understanding semantic style by analysing student code / Ruvo Giuseppe, Tempero Ewan, Luxton-Reilly Andrew et al. — 2018. — January. — P. 73–82.

[14] Документация Radon. — URL: https://radon.readthedocs.io/en/latest/ (online; accessed: 2021-05-20).

[15] Документация WPS. — URL: https://wemake-python-stylegui.de/en/latest/ (online; accessed: 2021-05-20).

[16] Исходный код Hyperstyle. — URL: https://github.com/hyperskill/hyperstyle (online; accessed: 2021-05-05).

[17] Репозиторий cohesion. — URL: https://github.com/mschwager/cohesion/ (online; accessed: 2021-05-20).

[18] Репозиторий flake8-broken-line. — URL: https://github.com/wemake-services/flake8-broken-line/ (online; accessed: 2021-05-20).

[19] Репозиторий flake8-commas. — URL: https://github.com/PyCQA/flake8-commas/ (online; accessed: 2021-05-20).

[20] Репозиторий flake8-string-format. — URL: https://github.com/xZise/flake8-string-format/ (online; accessed: 2021-05-20).

[21] Сайт Hyperstyle. — URL: https://hyperskill.org/ (online; accessed: 2021-05-09).

[22] Сайт Stepik. — URL: https://stepik.org/ (online; accessed: 2021-05-09).

[23] Фргамент №1. — URL: https://gist.github.com/GirZ0n/c065c063dc09c548dbee76b2c7f9431a/ (online; accessed: 2021-05-23).

[24] Фргамент №2. — URL: https://gist.github.com/GirZ0n/46fae8250a4d75983e74c6c17d0ed6c2/ (online; accessed: 2021-05-23).

[25] Фргамент №3. — URL: https://gist.github.com/GirZ0n/a31e4bd9e476e3de02d5c5955211c11b/ (online; accessed: 2021-05-23).