DOI: 10.15514/ISPRAS-2021-33(6)-7

Технический долг в жизненном цикле разработки ПО: запахи кода

[1,2] *В.В. Качанов, ORCID: 0000-0002-9371-6483 <vkachanov@ispras.ru>*
[1] *М.К. Ермаков, ORCID: 0000-0002-0483-6097 <mermakov@ispras.ru>*
[1] *Г.А. Панкратенко, ORCID: 0000-0003-3996-8281 <gpankratenko@ispras.ru>*
[1] *А.В. Спиридонов, ORCID: 0000-0001-8374-2453 <aspiridonov@ispras.ru>*
[1] *А.С. Волков, ORCID: 0000-0003-2492-6003 <asvolkov@ispras.ru>*
[1] *С.И. Марков, ORCID: 0000-0002-6687-4937 <markov@ispras.ru>*

[1] *Институт системного программирования им. В.П. Иванникова РАН, 109004, Россия, г. Москва, ул. А. Солженицына, д. 25*
[2] *Московский физико-технический институт, 141701, Россия, Долгопрудный, Институтский пер., д. 9*

Аннотация. Данная статья посвящена обзору наиболее популярных запахов кода, одного из компонентов технического долга, а также методов и инструментов их поиска. В статье проводится сравнительный анализ результатов работы таких инструментов как *DesigniteJava*, *PMD*, *SonarQube*. Инструменты были применены к набору проектов с открытым исходным кодом для вычисления точности обнаружения и согласованности выбранных инструментов. Показаны сильные и слабые стороны подхода, основанного на подсчете метрик кода и отсечения по пороговым значениям, который используется в инструментах. Ручная разметка результатов работы показала низкий процент истинных срабатываний (10% для божественного класса и 20% для сложного метода). Проведён обзор работ, предлагающих усовершенствование стандартного подхода и альтернативные, не использующие метрики. Для оценки потенциала альтернативных подходов разработан прототип обнаружения длинных методов с системой фильтрации ложноположительных срабатываний, использующие методы машинного обучения.

Ключевые слова: запахи кода; технический долг; машинное обучение; метрики кода

Для цитирования: Качанов В.В., Ермаков М.К., Панкратенко Г.А., Спиридонов А.В., Волков А.С., Марков С.И. Технический долг в жизненном цикле разработки ПО: запахи кода. Труды ИСП РАН, том 33, вып. 6, 2021 г., стр. 95-110. DOI: 10.15514/ISPRAS–2021–33(6)–7

Technical debt in the software development lifecycle: code smells

[1,2] *V.V. Kachanov, ORCID: 0000-0002-9371-6483 <vkachanov@ispras.ru>*
[1] *M.K. Ermakov, ORCID: 0000-0002-0483-6097 <mermakov@ispras.ru>*
[1] *G.A. Pankratenko, ORCID: 0000-0003-3996-8281 <gpankratenko@ispras.ru>*
[1] *A.V. Spiridonov, ORCID: 0000-0001-8374-2453 <aspiridonov@ispras.ru>*
[1] *A.S. Volkov, ORCID: 0000-0003-2492-6003 <asvolkov@ispras.ru>*
[1] *S.I. Markov, ORCID: 0000-0002-6687-4937 <markov@ispras.ru>*

[1] *Ivannikov Institute for System Programming of the Russian Academy of Sciences, 25, Alexander Solzhenitsyn st., Moscow, 109004, Russia*
[2] *Moscow Institute of Physics and Technology, 9, Insitutsky per., Dolgoprudny, 141701, Russia*

Abstract. This paper is dedicated to the review of the most popular code smells, which is one of the technical debt components, in addition to methods and instruments for their detection. We conduct a comparative analysis of multiple instruments such as DesigniteJava, PMD, SonarQube. We apply existing tools to set of open-source projects to deduce detection precision and coherence of the selected instruments. We highlight strengths and weaknesses of the approach based on metrics computation and threshold filtering that is used in instruments. Citing of code smells detected by the instruments shows low percentage of true positives (10% for god class and 20% for complex method). We perform literature review of papers suggesting enhancements of the standard approach and alternative approaches that doesn't involve metrics. To assess the potential of alternative methods we introduce our long method detection prototype with a false positive filtering system based on machine learning methods.

Keywords: code smells; technical debt; machine learning; source code metrics.

For citation: Kachanov V.V., Ermakov M.K., Pankratenko G.A., Spiridonov A.V., Volkov A.S., Markov S.I. Technical debt in the software development lifecycle: code smells. Trudy ISP RAN/Proc. ISP RAS, vol. 33, issue 6, 2021, pp. 95-110 (in Russian). DOI: 10.15514/ISPRAS–2021–33(6)–7

1. Введение

Разработка и поддержка кода программного обеспечения включает в себя значительное количество процессов и активностей, направленных в значительной степени на достижение единственной цели – получить продукт, реализующий требуемую функциональность с определенным уровнем надёжности [1-2]. Сложность ПО, человеческий фактор, а также издержки производства (конкуренция, фиксированные сроки и т.д.) обычно приводят к тому, что в тех или иных аспектах страдает качество разрабатываемого кода [3]. Подобную потерю качества и связанные с ней проблемы описывают термином «технический долг» [1]. Ключевой особенностью наличия технического долга является то, что дальнейшая разработка, связанная с добавлением новой функциональности или исправлением дефектов, требует бóльших усилий, чем это могло было быть. Ошибки проектирования или реализация архитектуры ПО с известными ограничениями (в угоду желанию получить готовый продукт как можно раньше) являются причиной увеличенного расхода времени и трудовых ресурсов на дальнейшую поддержку [3-4].

Разумеется, на практике крайне редко имеется возможность запланировать все возможные требования потенциальных пользователей ПО и предусмотреть все технические проблемы, проявляющиеся при эксплуатации продукта. Более того, реализация сверхгибкой архитектуры, требующая значительного объема ресурсов, является излишней, если подобная гибкость никогда не будет востребована, а самая базовая требуемая функциональность не будет поддержана в разумные сроки. Тем не менее, возможность обнаруживать потенциальные ошибки проектирования непосредственно на этапе разработки является крайне востребованной, особенно если данный процесс будет производиться автоматически,

а решение о том, что делать с обнаруженной проблемой, будет оставаться на усмотрение программиста или системного архитектора [5].

Одним из компонентов технического долга является наличие так называемого «запаха» в программном коде (*code smell*) [2]. Данный термин описывает ряд архитектурных недостатков, обычно затрудняющих понимание, изменение и отладку конкретных фрагментов кода. Стоит отметить, что запахи кода обычно не имеют строгого определения или трактовки, более того, в некоторых случаях их наличие может быть оправдано [6] (например, техническими ограничениями или необходимостью использовать унаследованный код). Тем не менее, существует ряд запахов кода, которые можно считать основополагающими – они являются объектом изучения многочисленных работ по рефакторингу и анализу кода. Для многих из них были предложены правила обнаружения, часть из которых непосредственно используется в инструментах автоматического анализа кода [2].

Фундаментальные и практические исследования в рамках данной области в первую очередь преследуют цель научиться максимально точно и полно определять наличие запахов кода [7-8]. В данной статье будет показано, что несмотря на значительные успехи, доступные на данный момент популярные инструменты обнаружения запахов кода нельзя использовать в качестве универсального оракула, оценивающего является ли определенный фрагмент или компонент кода корректно спроектированным.

Основной причиной такой ситуации является отсутствие фиксированных формальных определений запахов кода, отсюда возникает необходимость использования косвенных признаков в виде определенных атрибутов кода, которые не всегда достаточно полно отражают действительность.

Применение методов машинного обучения для уточнения правил за счёт обработки наборов фрагментов кода, вручную проанализированных экспертами на предмет наличия запахов, является перспективным направлением для преодоления данной проблемы.

В рамках данной статьи проведен обзор наиболее популярных запахов кода, инструментов их обнаружения и исследований, посвящённых задаче повышения точности обнаружения. Разд. 2 содержит некоторое словесное описание общепризнанных запахов кода и стандартные методологии их обнаружения по определенным атрибутам кода. В разд. 3 рассмотрены инструменты обнаружения и представлены результаты анализа набора проектов с открытым исходным кодом; на основе результатов сделаны выводы о соответствии конкретных результатов различных инструментов на одних и тех же целевых проектах и оценка практической пользы от их применения. Разд. 4 посвящён обзору исследований в области анализа кода, нацеленных на уточнение правил обнаружения запахов кода. В разд. 5 представлен прототип инструмента обнаружения длинных методов с системой фильтрации ложноположительных срабатываний, использующей методы машинного обучения. Финальный раздел содержит общие выводы по текущему состоянию области.

2. Запахи кода

Термин запахи кода был предложен Кентом Бэком (Kent Beck) и популяризован Мартином Фаулером (Martin Fowler) [1] в контексте проведения рефакторинга кода – модификации с целью устранения архитектурных недостатков. Каждая разновидность запахов кода описывает определенный набор особенностей, которые в той или иной степени затрудняют работу с кодом.

В качестве иллюстрации того, на каком уровне абстракции задаются запахи кода, рассмотрим следующие примеры [1].

- «Божественный класс» (*God Class*) – компонент, который реализует большое количество разноплановой функциональности. Исправление кода для устранения запаха предполагает разбиение функциональности по принципу «разделяй и властвуй».

- «Завистливый метод» (*Feature Envy*) – метод, который обращается к полям других классов больше, чем к полям собственного класса. Такие ситуации могут свидетельствовать о том, что инкапсуляция данных применяется некорректно, что затрудняет написание тестов и требует создания лишних объектов. Исправление кода в данной ситуации предполагает перенос функции в более подходящий класс.
- «Стрельба дробью» (*Shotgun Surgery*) описывает ситуацию, в которой изменение класса приводит к необходимости делать незначительные изменения в большом количестве других классов. Причинами подобной архитектуры может быть выделение слишком узкоспециализированных сущностей, с которыми производятся одни и те же действия. Для исправления подобной ситуации стоит перенести в один класс все совместно изменяемые методы и поля.
- «Сложный метод» (*Complex Method*) – метод с излишне сложной реализацией (с точки зрения графов потока управления и данных). Понимание и изменение подобных методов потребует больших ресурсов, а реализация тестов может не покрывать все граничные случаи или иметь свою сложную структуру, которая потребует собственного тестирования. Исправление подобного метода предполагает вынесение некоторой части функциональности в отдельный метод.
- «Длинный метод» (*Long Method*) – метод, реализация которого включает слишком большой объем кода (в пересчёте на строки, инструкции и т.д.). Исправление аналогично сложному методу.
- «Длинный список параметров» (*Long Parameter List*) описывает проблему методов и функций, имеющих слишком много формальных параметров. Подобная ситуация может возникать, например, если в метод передают много полей некоторого класса, тогда лучшим решением было бы передать объект класса целиком и получать нужные поля уже внутри метода.

Данный список сформирован на основе следующих принципов – представленные в нём запахи кода фигурируют в большинстве научных работ по данной области и/или могут быть автоматически обнаружены популярными инструментами анализа кода [7-9].

Как видно, описания запахов кода опираются на крайне общие понятия структуры и семантики исходного кода. Обнаружение требует ручного анализа, проводимого экспертом, или использования косвенных характеристик, которые так или иначе отражают «суть проблемы» [6]. В качестве примера рассмотрим божественный класс и попытаемся предположить, по каким характеристикам можно было бы его идентифицировать. По неформальному определению божественный класс реализует большой объем независимой функциональности. С практической точки зрения это как минимум должно означать, что класс определяет большое количество публичных методов. Классы, количество методов которых ниже заданного порога, маловероятно можно будет назвать божественными. Независимость реализуемой функциональности можно определить следующим образом: если класс определяет некоторое количество внутренних полей, а два его метода осуществляют доступ к непересекающимся подмножествам этих полей, то эти методы никак не связаны. Чем меньше степень связности методов класса при большом их количестве, тем более несогласованным является класс. Процесс подсчёта подобных характеристик может быть довольно сложным на практике. Так, например, «геттеры» и «сеттеры» – методы доступа к переменным – вряд ли имеет смысл учитывать при подсчёте связности, потому что обычно они не должны оперировать больше, чем одним полем класса.

В отличие от неформальных определений, подобные характеристики поддаются численному описанию. В контексте области анализа кода подобные числовые характеристики называются термином метрики [10]. Подсчёт метрик может быть легко автоматизирован на основе инструмента, который осуществляет разбор исходного кода. Так, количество методов

класса (NMD – Number of Methods Declared) и степень несогласованности методов (LCOM – Lack of Cohesion of Methods) – метрики, которые часто входят в набор поддерживаемых инструментами анализа кода.

Подход на основе метрик на данный момент является основным среди инструментов, предоставляющих автоматический поиск запахов кода. Конкретный набор метрик, выбор граничных значений и формулы, составленные на основе данных метрик, могут варьироваться и настраиваться на конкретные языки программирования и/или проекты – это и есть основополагающие проблемы данного подхода.

В следующем разделе приведены примеры таких инструментов и анализ результатов их работы.

3. Поиск запахов кода

Определение запахов кода на основе метрик позволяет автоматизировать анализ кода проекта. Данный анализ может быть встроен в процесс непрерывной разработки с целью поддержания качества ПО, однако это оправдано только в том случае, если его применение будет эффективнее ручного труда (с учётом доступных вычислительных и временных ресурсов). Рассмотрим возможную последовательность этапов применения инструмента:

- настройка инструмента для целевой кодовой базы (опционально);
- запуск инструмента и сбор предупреждений о запахах кода;
- ручной анализ каждого предупреждения с целью выбора одной из трёх доступных меток:
  - истинное срабатывание, требует исправления;
  - истинное срабатывание, не требует исправления;
  - ложное срабатывание.
- Дополнительная настройка инструмента с целью уточнения правил (опционально).

Пометка предупреждений «не требует исправления» может быть аргументирована тем, что в данном случае использование «проблемной» архитектуры могло быть сделано осознанно, или же исправление потребует слишком большого количества ресурсов (что противоречит изначальной цели избавиться от технического долга, однако может быть оправдано практическими соображениями). Если инструмент предоставляет некоторые способы настройки на конкретную кодовую базу, то эти настройки должны служить для снижения количества ложных срабатываний. Также неплохой особенностью была бы возможность обучения инструмента игнорировать предупреждения, похожие на те, которые были ранее помечены как «не требующие исправления».

В общем случае, ключевыми метриками практичности инструмента являются количество обнаруживаемых предупреждений и соотношение истинных и ложных срабатываний. Если инструмент не находит никаких предупреждений или находит слишком много ложных предупреждений, то его применение будет негативно сказываться на процессе разработки.

В цели исследования входила оценка следующих аспектов:

- какой объем предупреждений заданных типов выдаёт инструмент?
- каков процент истинных срабатываний среди выданных предупреждений?
- как соотносятся между собой результаты различных инструментов при обработке одной кодовой базы?

3.1 Выбор инструментов для исследования

В рамках данной статьи было проведено исследование набора свободно распространяемых инструментов автоматического поиска запахов кода. Для предварительной оценки были рассмотрены следующие: DesigniteJava, PMD, SonarQube, JDeodorant, SpotBugs, SAD, iPlasma, inFusion, JSpIRIT, DECOR.

DECOR, JSpIRIT, inFusion, iPlasma были отброшены, так как они не были найдены в свободном доступе как полноценные инструменты или дополнения к IDE, либо они больше не поддерживаются. SAD и JDeodorant являются дополнениями к IDE Eclipse, однако в рамках исследования их не удалось настроить для выбранных проектов. Список дефектов, поддерживаемых в SpotBugs, не включает в себя запахи кода, обнаруживаемые другими инструментами. В итоге, для дальнейшего исследования остались 3 инструмента: DesigniteJava, PMD, SonarQube.

PMD [11] – статический анализатор кода, который ищет стандартные ошибки программирования. Этот инструмент поддерживает 8 языков программирования, в том числе Java. В PMD есть некоторый список встроенных проверок кода (около 400 для Java), но также есть возможность их изменения и добавления собственных. Среди встроенных есть детекторы CognitiveComplexity (сложный метод), LongVariable (длинный идентификатор), GodClass (божественный класс), ExcessiveMethodLength (длинный метод).

DesigniteJava [12] – автономный инструмент оценки качества кода, написанного на Java. Версия, распространяемая в свободном доступе, ищет 17 антипаттернов проектирования и 10 запахов реализации кода, в том числе LongMethod (длинный метод), ComplexMethod (сложный метод) и LongIdentifier (длинный идентификатор).

SonarQube [13] – инструмент автоматической проверки кода для обнаружения ошибок, уязвимостей и запахов кода. Инструмент поддерживает 27 языков программирования. Для Java написано 621 правило, среди которых 153 проверки ошибок и 396 проверок запахов кода, в том числе CognitiveComplexity (сложный метод).

Чтобы найти запахи кода, все три инструмента вычисляют метрики и сравнивают эти значения с некоторыми пороговыми. Если порог пересечён, отмечается нарушение.

3.2 Сравнение результатов

Все выбранные инструменты поиска запахов кода рассматривают Java как целевой язык программирования, в связи с чем было принято решение использовать в рамках исследования набор из 9 популярных проектов с открытым исходным кодом на этом языке, а именно: ant[1], drill[2], error-prone[3], ExoPlayer[4], giraph[5], gson[6], guava[7], hive[8], truth[9]. В табл. 1 численные характеристики этих проектов, показывающие объём исходного кода, количество объектов для анализа на наличие антипаттернов. Наличие запахов кода присуще, также, и другим языкам программирования (C++, C#, Python). Однако для этих языков не было найдено достаточное количество инструментов для их рассмотрения.

[1] https://github.com/apache/ant.git, f61ac1296
[2] https://github.com/apache/drill.git, 85d270f3
[3] https://github.com/google/error-prone.git, 54fa3f38
[4] https://github.com/google/ExoPlayer.git, 47b82cdc
[5] https://github.com/apache/giraph.git, 2c63aa23
[6] https://github.com/google/gson.git, f319c1b8
[7] https://github.com/google/guava.git, 3dfd7076
[8] https://github.com/apache/hive.git, de0a7ecb
[9] https://github.com/google/truth.git, eaddc49f

Табл. 1. Характеристики рассматриваемых проектов
Table 1. Projects specifications

Проект	Количество строк кода	Количество пакетов	Количество классов	Количество методов
ant	201k	93	1803	16844
drill	675k	389	6998	79465
error-prone	113k	61	3342	16618
ExoPlayer	238k	88	2038	21592
giraph	112k	134	1750	11125
gson	29k	23	489	3005
guava	548k	31	6840	75049
hive	1412k	590	12661	129015
truth	32k	6	305	4873

Каждый из выбранных инструментов был запущен на всём наборе проектов.

В табл. 2 отображено количество нарушений, которые были найдены инструментами на каждом из проектов. Для проектов ant, ExoPlayer, giraph, guava и hive не удалось настроить SonarQube, поэтому отчеты по этим проектам отсутствуют.

Из всех полученных данных были отфильтрованы только те записи, что относятся к 3 видам запаха кода: длинный метод, божественный класс, сложный метод. Однако даже такой базовый список запахов кода ищут не все инструменты.

Табл. 2. Общее количество срабатываний инструментов на проектах
Table 2. Total number of tool warnings on projects

Проект	**PMD**	**Designite**	**SonarQube**
ant	37261	5954	-
drill	201483	49644	15773
error-prone	54890	11390	2489
ExoPlayer	81847	24769	-
giraph	25436	6698	-
gson	9347	1928	385
guava	223384	52193	-
hive	666423	108106	-
truth	14890	5601	642

Далее на основе отчётов была проведена оценка корреляции результатов работы инструментов. Отсутствие общего стандарта описания ошибок и формирования отчетов усложняет процесс совмещения выявленных нарушений на объектах проекта по файлу/классу/методу.

На рис. 1 и рис. 2 изображены диаграммы Венна, показывающие все пересечения отчётов запахов кода от различных инструментов. Сложные методы ищут все выбранные инструменты, поэтому приведена статистика только для запусков на общих проектах (на которых удалось запустить SonarQube). На рис. 1 видно, что значительная часть записей общая для всех инструментов (374 метода из 1132-х отмеченных хотя бы одним инструментом). DesigniteJava определяет как сложные ещё 422 метода, на которые другие инструменты не отреагировали. Длинные методы SonarQube не определяет, поэтому статистка бралась по всем проектам. Отчет PMD практически включает в себя отчет

DesigniteJava, но также содержит ещё значительное количество записей о других методах. Божественный класс определяет только PMD, поэтому диаграмма Венна для этого запаха кода не приведена.

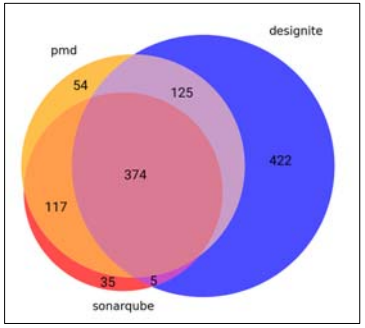

Рис. 1. Диаграмма Венна для «сложного метода»
Fig. 1. Venn diagram for «complex method»

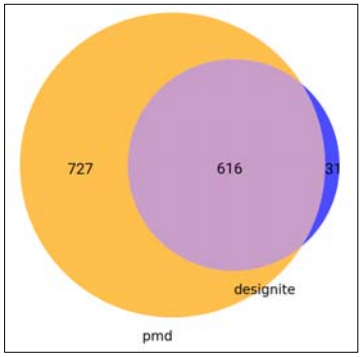

Рис. 2. Диаграмма Венна для «длинного метода»
Fig. 2. Venn diagram for «long method»

3.3 Статистика ложных срабатываний

Для оценки точности обнаружения запахов кода часть предупреждений была исследована вручную. Для каждого типа предупреждений была сформирована случайная выборка из 100 элементов (исключая автоматически сгенерированный и тестовый код), общих для всех инструментов. В рамках ручного анализа каждому предупреждению была назначена одна из трёх оценок – истинное срабатывание, ложное срабатывание и истинное срабатывание, не требующее исправления (англ. won't fix). В табл. 3 представлены результаты анализа. Длинный метод как наиболее «понятный» запах кода, обнаружение которого зачастую связано с количеством строк, имеет наивысший уровень истинных срабатываний, близкий к 100%. С другой стороны, при беглом осмотре части файлов проекта truth было обнаружено много методов, которые эксперты определили как «длинный», но инструменты их не обнаружили. Божественный класс и сложный метод показывают крайне низкий уровень истинных срабатываний (10% и 20% соответственно). Так как процесс оценки дефектов занимает определенное время, подобную статистику можно считать неприемлемой на практике.

Табл. 3. Статистика ложных срабатываний
Table 3. False positive statistics

Оценка эксперта	Божественный класс	Сложный метод	Длинный метод
Истинное срабатывание	10%	20%	90%
Истинное срабатывание, не требует исправления	21%	25%	5%
Ложное срабатывание	69%	55%	5%

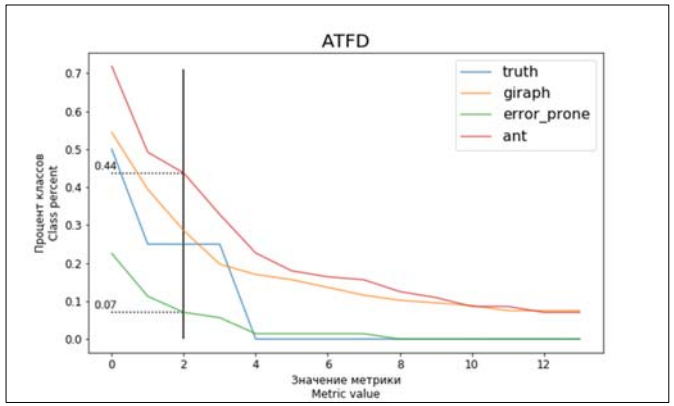

Рис. 3. График распределения классов по метрике ATFD
Fig. 3. The graph of the distribution of classes according to the ATFD metric

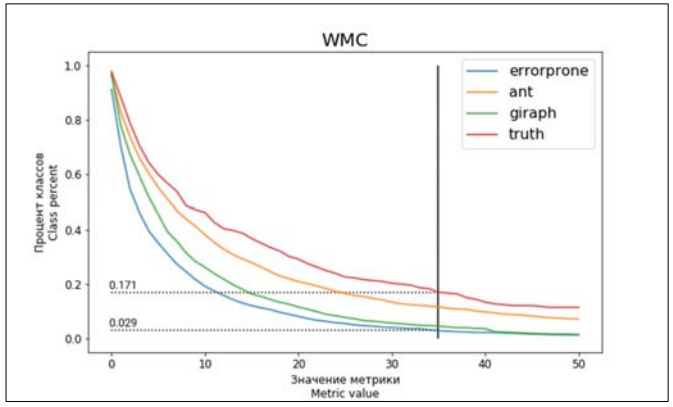

Рис. 4. График распределения классов по метрике WMC
Fig. 4. The graph of the distribution of classes according to the WMC metric

Для демонстрации проблем, связанных с необходимостью выбора пороговых значений, рассмотрим следующую зависимость, которая отображает процент объектов, со значением метрики больше некоторого заданного. На рис. 3, рис. 4 изображены такие графики для метрик ATFD и WMC [14] - базовых метрик для определения божественного класса. ATFD (доступ к сторонним данным) – это количество атрибутов из несвязанных классов, доступ к которым осуществляется напрямую или путем вызова методов доступа. WMC (взвешенные методы класса) – сумма всех сложностей методов в классе. Вертикальной линией отмечены

пороговые значения, взятые из документации PMD[10]. Как видно, при одинаковом пороговом значении процент объектов, превышающих этот порог, а значит определяемых как божественный класс – сильно отличаются, по метрике ATFD 43.75% классов ant, тогда как в error-prone - 7%. Для метрики WMC при установленном пороговом значении error-prone имеет около 3% классов-нарушителей, тогда как truth - чуть больше 17%. Подобное поведение показывает, что пороговые значения необходимо подбирать индивидуально под проект. Рассмотрим также аналогичный график для метрики Cognitive Complexity (некоторая вариация цикломатической сложности, разработанная в SonarQube) рис. 5 – по нему видно, что в проекте error-prone в целом значение этой метрики ниже, чем у drill на 5%, что свидетельствует о различных стилях оформления кода в этой паре проектов.

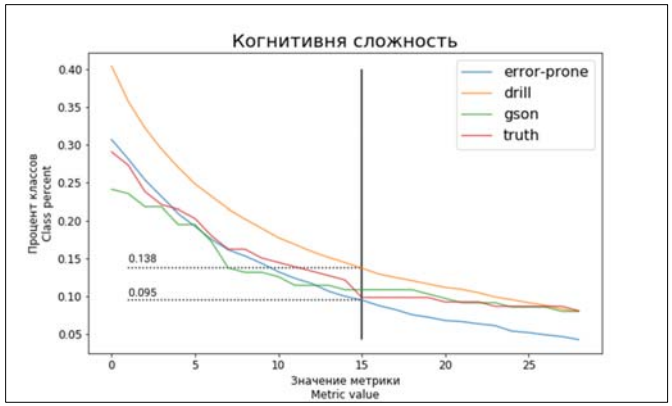

Рис. 5. График распределения методов по метрике когнитивной сложности
Fig. 5. The graph of the distribution of methods according to the cognitive complexity metric

3.4 Ответы на исследовательские вопросы

ИВ1: Какой объем предупреждений заданных типов выдаёт инструмент?

Существующие инструменты при стандартной настройке довольно шумные: PMD выдает предупреждения на 18 - 48% (ant и error-prone) строчек кода проекта (отношение числа репортов к количеству строк кода), Designite чуть сдержаннее – 3 - 17.5% (ant и truth).

ИВ2: Каков процент истинных срабатываний среди выданных предупреждений?

Для божественных классов и сложных методов довольно низкий (10% и 20% соответственно), для длинных методов – 92%, но это не делает данный детектор хорошим, так как полнота его отчёта низкая.

ИВ3: Как соотносятся между собой результаты различных инструментов при обработке одной кодовой базы?

Инструменты имеют значительное пересечение по отчётам на одной кодовой базе, например, 374 из 1132-х общих сложных методов и 616 из 1374 общих длинных методов. Но также имеется и большое количество замечаний, которые пишут одни инструменты, но не другие (рис. 1, рис. 2).

[10] https://pmd.github.io/latest/pmd_rules_java_design.html#godclass

4. Развитие стандартных методов и альтернативные подходы

Как было сказано выше, запахи кода не имеют чёткого определения. Также понятно, что запахи кода имеют сложную структуру и сложные зависимости с другими компонентами кода, что усложняет их идентификацию. Чтобы повысить точность работы рассмотренных инструментов на конкретном проекте необходимо очень точно настроить граничные значения метрик, к тому же, полученные эвристики не могут быть применены к произвольному проекту. Этот процесс может быть очень трудоёмким и неэффективным. Довольно тяжело вручную выбрать более важные метрики и построить правильные эвристики. Использование методов машинного обучения может облегчить задачу и автоматизировать выбор более важных метрик (признаки), а также может находить сложную взаимосвязь между метриками и наличием того или иного антипаттерна в коде.

В [15-16] рассматривается подход использования различных моделей машинного обучения (логистической регрессии, решающих деревьев, метода опорных векторов), основанных на значениях метрик, для определения наличия в данном фрагменте одного из четырёх антипаттернов (божественный класс, класс данных, длинный метод и завистливые функции). В качестве описания одного экземпляра класса используются 63 метрики, для метода – 84. Для обучения моделей был использован набор проектов Qualitas Corpus: 20120401r [17]. Для разметки этих проектов были запущены несколько инструментов (PMD, iPlasma, Fluid Tool) и использованы их отчеты. Оценка качества моделей производилась на наборе примеров, размеченных вручную. Объём размеченного набора: по 420 примеров для каждого типа запаха кода. В результате исследований были определены лучшие модели (J48, Random forest, JRip). Точность определения антипаттернов на предложенном оценочном наборе данных варьируется от 76 до 93 процентов.

В [18] предложено развитие [15-16], а именно применение глубоких нейронных сетей. С помощью автоматического кодировщика [19] вычисленные по коду метрики преобразовываются в новый список атрибутов, далее происходит сокращение размерности полученного вектора, который, в свою очередь, поступает на вход классификаторам, обученным на поиск конкретного запаха кода. Для обучения и тестирования использовался описанный выше [15-16] набор данных. В результате такой обработки метрик авторы статьи смогли определять антипаттерны с точностью от 97 до 99 процентов.

К сожалению, ни тестовый набор данных из [15-16], ни разработанную в [18] модель найти в свободном доступе не удалось.

В [20] предлагают объединить результаты работы нескольких инструментов для более точного поиска антипаттернов. Собранные метрики подаются на вход полносвязной нейронной сети [21]. Наборы данных для обучения и оценки качества были собраны из самих инструментов (DECOR, HIST, JDeodorant, InCode). В результате применения модели к полученным метрикам удалось добиться лучших предсказаний запахов кода. Метрикой качества был выбран коэффициента корреляции Мэтьюса между предсказанными и действительными экземплярами. Для завистливых функций у существующих инструментов лучшие показатели были у InCode – 49%, разработанная модель достигла значения в 56%. Для божественных классов – среди готовых инструментов лучшим был DECOR с показателем в 35%, а авторская модель улучшила этот результат до 49%.

Авторы [22] полагают, что для определения экземпляров запахов кода необходимы не только некоторые абстрактные метрики, но и реальный контекст. Например, методы с конструкцией switch-case будут считаться сложными по версии большинства инструментов, так как цикломатическая сложность превышает некоторый порог. Однако для разработчика, читающего этот код, подобный метод не покажется сложным, потому что все case случаи имеют схожую структуру. В статье проводится исследование о возможности использования токенизированного исходного кода в связке с глубокими нейронными сетями для определения антипаттернов. В качестве первого приближения авторами было предложено

использовать набор данных, размеченный существующим инструментом, работающим на метриках. Хотя, как указывается в статье, лучшим решением было бы использование размеченного вручную набора данных значительных размеров. Для оценки качества полученных моделей был создан небольшой набор примеров (166 классов и 280 методов), размеченных вручную. В результате исследования лучшая из разработанных моделей показала следующие значения F1: 0.64, 0.21 для сложного метода и завистливых функций соответственно.

В ходе обзора существующих решений, основанных на применении машинного обучения, было выдвинуто предположение, что основной проблемой такого подхода является отсутствие большого открытого набора данных, размеченного специалистами, на котором можно проводить общую оценку качества разработанных методов детектирования запахов кода. Эта же мысль озвучена также в [22].

В исследовании [23] авторы постарались решить существующую проблему и создали набор данных из 14739 примеров. Для разметки были выбраны файлы из активных проектов, разрабатываемых на Java, с открытым исходным кодом. Этот набор данных был проанализирован различными специалистами с некоторым процентом перекрестных оценок для повышения качества. Для каждого примера указан тип антипаттерна, его серьёзность и ссылка на сам исходный код. Также указаны некоторые характеристики проверяющих специалистов, такие как уровень образования, длительность работы в области разработки ПО, опыт разработки используя ООП, функциональное программирование, опыт программирования на Java и т.д.

5. Разработанный альтернативный подход

В рамках работы над данной статьёй нами был разработан прототип инструмента обнаружения длинных методов. В данной секции мы рассмотрим его устройство, а также как к нему может быть применена система фильтрации срабатываний, использующая методы машинного обучения.

5.1 Классификатор длинных методов

В качестве алгоритма классификации нами была взята модель произвольного леса деревьев (англ. Random Forest) из библиотеки sklearn[11]. В качестве входа для модели предоставлялись метрики исходного кода, подсчитанные на методах.

Данные для обучения были получены из набора MLCQ, содержащий 2430 уникальных примеров методов. Изначально авторы разделили методы на 4 категории по степени уверенности наличия в примере длинного метода. Далее эта разметка была переведена в бинарный вид наличия или отсутствия запаха кода. А также нами был реализован прототип для сбора метрик исходного кода: LOC (количество строк кода), NCLOC (количество строк кода без комментариев и пустых строк), MCC (цикломатическая сложность Маккаби), TokenCount (количество токенов).

5.2 Фильтрация результатов

После обнаружения моделью длинного метода, при помощи анализа графа потока данных, для каждого блока кода (в случае Java - фрагмент кода, ограниченный фигурными скобками) определяются зависимости переменных внутри этого блока от переменных, находящихся во вне. Далее выбирается блок, в котором зависимых переменных меньше, чем в остальных. Такой блок мы считаем кандидатом для вынесения в отдельную функцию и производим

[11] https://scikit-learn.org

соответствующую трансформацию, а именно создается отдельный файл, класс и метод для этого блока, а в исходном коде на его место вставляется вызов созданного метода.

Далее проводится валидация описанной выше модификации, состоящая из двух шагов.

На первом шаге уже упомянутая модель запускается на обоих участках кода и проверяет, что среди них нет длинных методов. Если данное условие не выполняется, операция выделения откатывается, а предупреждения о том, что исходные метод является длинным, отфильтровывается.

На втором шаге те же участки кода, что и в первом шаге, обрабатываются через модель code2vec[12]. Данная модель по участку кода формирует внутреннее представление, описывающее его семантику, и далее по этому представлению присваивает несколько имён каждому методу. Если для двух участков кода присвоенные их методам имена не пересекаются, мы считаем такие участки семантически различимыми. В случае если выделенный блок текста, помещенный в новый метод нового класса, семантически различим с исходным длинным методом, из которых он был изъят, мы считаем такую трансформацию успешной и, тем самым, подтверждаем, что исходный метод был длинным. В отличном случае, как и в шаге 1, данная трансформация откатывается, а предупреждение о том, что метод является длинным, отфильтровывается.

Детектор длинных методов, реализованный подобным образом, имеет несколько особенностей.

Во-первых, применение методов машинного обучения должно помочь использованию подобного инструмента на различных кодовых базах без дополнительной настройки.

Во-вторых, так как разработчик, оценивая предупреждения рассматривает возможные варианты их исправления, шаги 1 и 2 фильтрации отбрасывают как раз то, что было бы оценено человеком как ложные срабатывания или истинные срабатывания, не требующие исправления.

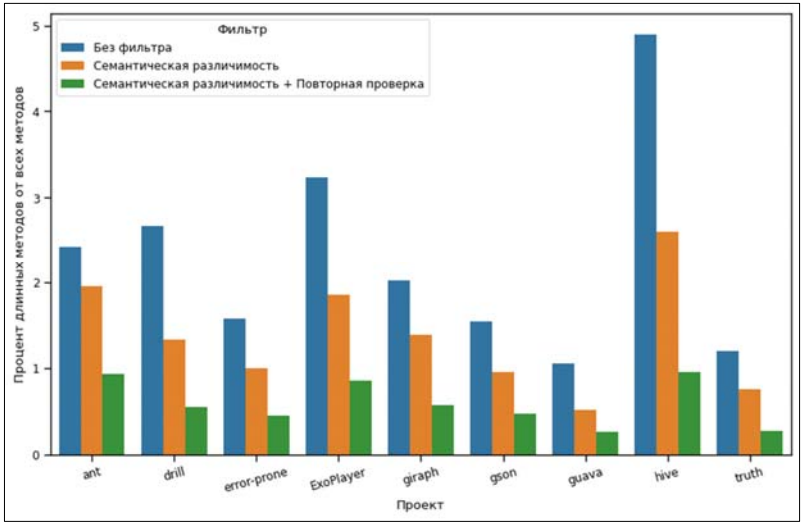

Рис. 6. Эффективность фильтрации длинных методов в зависимости от алгоритма
Fig. 6. Filtering efficiency of long methods depending on the algorithm

[12] https://github.com/tech-srl/code2vec

Для оценки эффективности фильтрации предупреждений о длинных методах мы запустили прототип классификатора на проектах из части 3.2. На результатах классификации была запущена 2-х шаговая фильтрация, описанная выше.

На рис. 6 приведены результаты этой фильтрации. При использовании алгоритма, основанного на семантической различимости выносимого блока от его контекста, прототип отфильтровал 19-50% (проекты ant и guava) предупреждений о длинных методах. Если же помимо указанного алгоритма дополнительно фильтровать те методы, которые после предложенной прототипом трансформации остаются длинными методами, то удастся снизить общее количество предупреждений на 60-80% (проекты ant и hive) от изначального числа.

5.2.1 Оценка качества разработанных методов фильтрации

Из всех полученных базовым классификатором предупреждений были выбраны случайным образом по 30 примеров из каждого проекта. Далее три эксперта разметили эти примеры на три вышеупомянутые категории: истинное срабатывание, ложное срабатывание и срабатывание, не требующее исправления. Для оценки согласованности разметки была использована каппа-статистика Коэна[13]. Коэффициенты парной согласованности: 0.888, 0.826, 0.848. Общий коэффициент согласованности определен как среднее значение парных статистик – 0.85. К разметке были применены описанные выше фильтры: семантическая различимость и семантическая различимость с повторной проверкой классификатором.

В табл. 4 приведены результаты применения фильтров к размеченным данным. В первом столбце указаны типы меток. Во втором и третьем столбцах представлена статистика распределения классов для исходного классификатора. Видно, что к истинным срабатываниям отнесли только 56 (21,2% от всех) предупреждений. В четвертом и пятом столбце показаны данные по работе фильтра, основанного на выделении возможного кандидата и сравнении его семантики с базовым методом. Видно, что он отсеял 60% ложных срабатываний, 38,5% примеров, не требующих исправления, и менее 9% истинных срабатываний. Дополнительная проверка того, остается ли метод длинным, устраняет более 75% ложных срабатываний и тех, что не требуют исправления, но также и более половины (53%) истинных предупреждений. Такое поведение можно объяснить тем, что текущий алгоритм пытается извлечь только один блок операций. В случае очень крупного метода такого исправления может быть недостаточно, чтобы полученный метод больше не считался длинным. Исследование других предикатов фильтрации и извлечения нескольких участков кода выходят за рамки данной работы.

Табл. 4. Статистика качества фильтрации
Table 4. Filtration quality statistics

Метка	До фильтрации, количество	До фильтрации, %	Отсечено фильтром, количество	Отсечено фильтром, % от начального количества
Ложное срабатывание	125	47,4%	75	60%
Не требующее исправления	83	31,4%	32	38,5%
Истинное срабатывание	56	21,2%	5	8,9%
Всего	264		112	

[13] https://en.wikipedia.org/wiki/Cohen%27s_kappa

Таким образом, общее количество длинных методов на рассмотренных проектах оказалось в диапазоне от 0.26% до 0.95% (guava и hive) от общего числа методов на этих проектах. По полученным результатам, мы считаем, что описанный алгоритм фильтрации показал свою эффективность, а итоговое количество длинных методов является достаточно небольшим (в сравнении с количеством всех методов проекта), чтобы охарактеризовать данный подход как «нешумный».

6. Заключение

Запахи кода имеют лишь неформальное определение и могут подвергаться субъективной трактовке авторами различных статей и инструментов. В данной работе проведено сравнение 3 инструментов (DesigniteJava, PMD, SonarQube) обнаружения запахов кода на наборе из 9 проектов (ant, drill, error-prone, ExoPlayer, giraph, gson, guava, hive, truth) с открытым исходным кодом. В силу выбранного авторами подхода с использованием метрик исходного кода и пороговых значений возникает необходимость настройки этих самых пороговых значений под конкретный проект, что показано на рис. 3, 4, 5. Запуск инструментов на реальных проектах показал довольно низкий процент истинных срабатываний (10% и 20% для божественного класса и сложного метода, соответственно). На рис. 1, 2 приведен анализ согласованности отчетов выбранных инструментов.

Также в данной статье проведен обзор существующих предложений по развитию стандартных методов обнаружения с применением методов машинного обучения. По заверениям их авторов, предложенные подходы помогают значительно повысить качество обнаружения запахов кода.

Для проверки гипотезы о потенциале альтернативных подходов в работе разработан прототип инструмента обнаружения длинных методов с системой фильтрации ложноположительных срабатываний, использующие методы машинного обучения. Предложенный алгоритм фильтрации снижает общее количество срабатываний на 60-80%.

Список литературы / References

[1]. Fowler M. Refactoring: Improving the Design of Existing Code. Addison-Wesley, 1999, 431 p.
[2]. Lanza M., Marinescu R. Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems. Springer, 2010, 231 p.
[3]. van Emden E., Moonen L. Java quality assurance by detecting code smells. In Proc. of the Ninth Working Conference on Reverse Engineering, 2002. pp. 97-106.
[4]. Khomh F., Di Penta M., Gueheneuc Y.G. An exploratory study of the impact of code smells on software change-proneness. In Proc. of the 16th Working Conference on Reverse Engineering, 2009, pp. 75-84.
[5]. Langelier G., Sahraoui H., Poulin P. Visualization-based analysis of quality for large-scale software systems. In Proc. of the 20th IEEE/ACM International Conference on Automated Software Engineering, 2005, pp. 214-223.
[6]. Olbrich S., Cruzes D., Sjberg D. Are all code smells harmful? A study of God Classes and Brain Classes in the evolution of three open source systems. In Proc. of the IEEE International Conference on Software Maintenance, 2010, pp. 1-10
[7]. Paiva T., Damasceno A. et al. On the evaluation of code smells and detection tools. Journal of Software Engineering Research and Development, vol. 5, no. 7, 2017, article no. 7.
[8]. Palomba F., Bavota G. et al. Do they really smell bad? a study on developers' perception of bad code smells. In Proc. of the IEEE International Conference on Software Maintenance and Evolution, 2014, pp. 101-110.
[9]. Tufano M., Palomba F. et al. When and why your code starts to smell bad. In Proc. of the IEEE/ACM 37th IEEE International Conference on Software Engineering, 2015, pp. 403-414
[10]. Fenton, N.E., Neil M. Software metrics: successes, failures and new directions. Journal of Systems and Software, vol. 47, no. 2, 1999, pp.149-157.
[11]. PMD. URL: https://pmd.github.io/latest/index.html, accessed: 25/10/2021.
[12]. DesigniteJava. URL: https://github.com/tushartushar/DesigniteJava, accessed: 25/10/2021.
[13]. SonarQube. URL: https://www.sonarqube.org/, accessed: 25/10/2021.

[14]. Ferme V. JCodeOdor: A Software Quality Advisor Through Design Flaws Detection. Master's thesis, Università degli Studi di Milano-Bicocca, Italy, 2013.
[15]. Arcelli Fontana F., Mantyla M., Zanoni M., Marino A. Comparing and experimenting machine learning techniques for code smell detection. Empirical Software Engineering, vol. 21, issue 3, 2016, pp. 1143-1191.
[16]. Arcelli Fontana F., Zanoni M. Code smell severity classification using machine learning techniques. Knowledge-Based Systems, vol. 128, 2017, pp. 43-58.
[17]. Qualitas Corpus. URL: http://qualitascorpus.com/docs/history/20120401.html, accessed: 25/10/2021
[18]. Hamdy A., Tazy M. Deep hybrid features for code smells detection. Journal of Theoretical and Applied Information Technology, vol. 98, 2020, pp. 2684-2696
[19]. Bank D., Koenigstein N., Giryes R. Autoencoders. arXiv:2003.05991, 2021.
[20]. Barbez A., Khomh F., Gu´eh´eneuc Y.G. A machine-learning based ensemble method for anti-patterns detection, arXiv:1903.01899, 2019.
[21]. Zhang P. Neural networks for classification: A survey. IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews), vol. 30, no. 4, pp. 451-462.
[22]. Sharma T., Efstathiou V., Louridas P., Spinellis D. On the feasibility of transfer-learning code smells using deep learning, arXiv:1904.03031, 2019.
[23]. Madeyski L., Lewowski T. Mlcq: Industry-relevant code smell data set, In Proc. of the International Conference on Evaluation and Assessment in Software Engineering, 2020, pp. 342-347.

Информация об авторах / Information about authors

Владимир Владимирович КАЧАНОВ – аспирант МФТИ, программист в ИСП РАН. Сфера научных интересов: машинное обучение, программная инженерия.

Vladimir Vladimirovich KACHANOV – PhD Student at MIPT, Programmer at ISP RAS. Research interests: machine learning, software engineering.

Михаил Кириллович ЕРМАКОВ – кандидат технических наук, младший научный сотрудник. Сфера научных интересов: динамический анализ кода, фаззинг, символьное исполнение, статический анализ кода.

Mikhail Kirillovich ERMAKOV – Candidate of Technical Sciences, Researcher. Research interests: dynamic program analysis, fuzzing, symbolic execution, static program analysis.

Георгий Александрович ПАНКРАТЕНКО – стажер-исследователь. Сфера научных интересов: программная инженерия, компиляторы.

Georgiy Alexandrovich PANKRATENKO – Intern Researcher. Research interests: program engineering, compilers.

Александр Вячеславович СПИРИДОНОВ – программист. Сфера научных интересов: программная инженерия.

Alexander Vyacheslavovich SPIRIDONOV – Programmer. Research interests: program engineering.

Александр Сергеевич ВОЛКОВ – младший научный сотрудник. Сфера научных интересов: программная инженерия, компиляторы.

Alexander Sergeevich VOLKOV –Researcher. Research interests: program engineering, compilers.

Сергей Игоревич МАРКОВ – научный сотрудник. Сфера научных интересов: статический анализ кода, динамический анализ кода, программная инженерия.

Sergei Igorevich MARKOV – Researcher. Research interests: static program analysis, dynamic program analysis, program engineering.