ИЗВЕСТИЯ РАН. ТЕОРИЯ И СИСТЕМЫ УПРАВЛЕНИЯ, 2023, № 4, с. 98–104

ИСКУССТВЕННЫЙ ИНТЕЛЛЕКТ

УДК 004.8

МАШИННОЕ ОБУЧЕНИЕ В ПРОБЛЕМЕ ТЕХНИЧЕСКОГО ДОЛГА ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

© 2023 г. В. В. Качанов[a,b,*], С. И. Марков[a,], В. И. Цурков[c,***]**

[a] *ИСП РАН, Москва, Россия*

[b] *МФТИ, Долгопрудный, МО, Россия*

[c] *ФИЦ ИУ РАН, Москва, Россия*

**e-mail: vkachanov@ispras.ru*

***e-mail: markov@ispras.ru*

****e-mail: tsur@ccas.ru*

Поступила в редакцию 21.02.2023 г.
После доработки 27.02.2023 г.
Принята к публикации 03.04.2023 г.

При модернизации программного обеспечения возникает проблема технического долга, когда часть исходного кода напрямую не участвует в обновлении, а исправляется во вторую очередь как устаревшее. Представлены три соответствующие модели. Для поиска и исправления дефектов используется машинное обучение. Устанавливается эффективность подхода для конкретных данных и намечается перспектива расширения на большее количество различных случаев.

DOI: 10.31857/S0002338823040078, **EDN:** OCRJMR

Введение. Одна из основных задач разработчика программного обеспечения (ПО) заключается в развитии продукта. Процесс постоянного внесения новой функциональности не может продолжаться вечно, так как из-за нарастающей сложности программного продукта и сжатых сроков качество написанного исходного кода (текста программы) ухудшается. Иногда настолько, что дальнейшее расширение функциональности становится невозможным. В литературе принято такое явление называть “техническим долгом” [1]. Можно провести параллель с обычным. Технический долг – это деньги, которые вы берете на разработку новой функциональности (расширение своего производства), а процесс рефакторинга (приведение вашего исходного кода ПО в должный вид, который не приносит для пользователей ничего нового, но занимает какое-то время) – выплаты по этому долгу. Соответственно чем чаще вы гонитесь за быстрым развитием функциональности, не погашая долг, тем больше проценты по нему придется платить, а значит, более долгий и значительный рефакторинг придется совершать. Стоит также заметить, что технический долг не всегда оказывается чем-то безусловно плохим, иногда приоритет быстрого решения проблемы над надежным необходим для продвижения проекта. Одни из ярких представителей симптомов технического долга называются “запахи кода” [1, 2] (англ. code smells) – антипаттерны программирования. Такие участки текста программы затрудняют понимание, отладку, тестирование, расширение программы в целом. Они не являются технически некорректными и не препятствуют функционированию программы [3]. Вместо этого они указывают на уязвимые места в архитектуре, которые могут замедлить разработку или увеличить риск ошибок и сбоев в будущем. В данный термин включены архитектурные неточности, недостатки дизайна ПО и проблемы реализации. Одна из особенностей запахов кода – отсутствие строгого определения и четких характеристик, по которым можно было бы сказать, считается ли исходный код плохим. Однако можно выделить список из 5-6 основных наиболее изученных типов антипаттернов [1, 2].

Запахи кода могут охватывать различные уровни абстракции разработки ПО. Так Божественный Класс (англ. God Class) – это некоторый объект в исходном коде программы, который управляет слишком многими другими объектами в системе, превратившись в класс, который делает все. Например, класс Customer, который содержит в себе имя и фамилию человека, его адрес проживания, страну, город и почтовый код. Кроме простого хранения информации, необходи-

мы ещё методы взаимодействия с данными полями и, возможно, некоторые служебные функции. С одной стороны, в этом нет ничего критического, однако намного удобнее иметь отдельный класс Address, который хранит внутри себя всю информацию об адресе и методы работы с ней, а в Customer держать только ссылку на объект класса Address. Более масштабный и неопределенный пример антипаттерна программирования – это Стрельба дробью (англ. Shotgun Surgery), который описывает ситуацию, когда при изменении незначительного поведения вашей программы в одном месте приходится делать дополнительные изменения в большом количестве других классов. Этот запах кода не ограничивается сущностью одного класса, для его обнаружения необходимо следить за историей изменения проекта и искать ситуации подобного множественного исправления текста программы. Кажется, что одними из наиболее простых для обнаружения и исправления запахами кода являются Длинный Метод (англ. Long Method) и Сложный Метод (англ. Complex Method). Названия данных антипаттернов программирования говорят сами за себя, это слишком разросшиеся в “длину” или “ширину” функции. Сложность их обнаружения заключается в том, что иногда такая реализация необходима и иным способом ее не сделать. Например методы, содержащие большое количество switch/case конструкций либо большое количество схожих операций преобразования чисел/строк, которые проблематично вынести в подфункции. Также понятие “длинности” или “сложности” зависит от конкретного проекта, языка программирования, команды разработчиков. Исследования в рамках данной области ставят перед собой цель наиболее точно и без пропусков определять наличие запахов кода.

Инструменты поиска элементов технического долга являются актуальным предметом исследования уже более 20 лет [4]. В последнее время для улучшения качества определения таких дефектов используются методы машинного обучения [5–7]. В основе подобных методов стоит задача обучения с учителем на наборе размеченных данных. В области создания таких наборов данных есть два направления: ручная [8–10] и автоматическая [7, 11] разметки. Наиболее актуальный представитель первой группы работ [8] содержит набор данных, где собрано большое количество ручных разметок примеров. Большинство из них оказались размечены как отрицательные примеры (без какого-либо дефекта). Представители второй группы используют готовый инструмент поиска дефектов исходного кода для обнаружения запахов кода на большом наборе, анализируя текущие версии проектов, а не историю изменений.

Основная проблема такого метода сбора примеров заключается в том, что исследователи/детекторы/оракулы, которые размечают положительные случаи, не являются разработчиками данного ПО, соответственно совершенно не знакомы с его строением и идеями архитектора ПО. Длинные и Сложные Методы вне зависимости от замыслов разработчиков – плохая практика. Другое дело Завистливые Методы (англ. Feature Envy) и Божественные Классы: такие случаи технического долга сложно определить без детального анализа структуры проекта. Есть инструменты, которые принимают на вход полноценный проект, анализируют связи между классами/методами/переменными и меняют структуру, решая задачу оптимизации зависимостей между сущностями [12], предлагая свой вид проекта. У такого подхода основным недостатком является сложность проведения генерального рефакторинга, так как в автоматическом режиме проводить столь масштабные изменения рискованно.

Еще один подход к созданию набора данных с исправлениями исходного кода представлен в ManySStuBs4J Dataset [13]. В этой публикации авторы собирают исправления реальных разработчиков, которые те посчитали необходимым сделать. Однако исправления, которые ищут, являются однострочными: замена переменной, численной константы, имени метода. Такой подход мы хотим применить к поиску примеров запахов кода. А именно, будем искать в истории изменения проектов с открытым исходным кодов те исправления, в которых разработчики исправляли имеющиеся дефекты своего проекта.

Основным результатом данной работы являются:

- предложенный новый подход к созданию набора данных в области запахов кода, который легко масштабируется;
- открытый набор примеров исправлений, содержащий около 6 тыс. пар (ДО и ПОСЛЕ), со ссылкой на коммиты в которых они проводились.

1. Существующие подходы. Авторы статьи [14] одними из первых опубликовали открытые наборы данных с примерами запахов кода, представили 243 примера дефектов пяти типов. Однако на данный момент (февраль 2023 г.) результаты недоступны. В работах [9, 15] говорится о 17 350 и 40 888 примерах 13 различных типов проявлений технического долга. Однако сами наборы данных закрыты, и получить к ним доступ не удалось. Открытый набор данных приведен в [10], здесь рассмотрен набор из 1986 примеров для четырех типов запахов кода. Недостатком можно

считать то, что примеры взяты из набора проектов Qualitas Corpus [16], последняя дата обновления которого была в 2012 г., что означает меньшую актуальность кодовой базы. Более новой публикацией подобного плана является [8], в которой авторы попросили 20 специалистов, имеющих различный опыт в разработке ПО, оценить некоторое количество примеров исходного кода на наличие в них симптомов технического долга. Итоговый набор данных содержит 14739 пометок для 4770 фрагментов кода. Безусловным плюсом данной работы можно отметить открытый доступ к самой разметке и опросам, на которые отвечали специалисты. К недостаткам стоит отнести довольно небольшое количество примеров, которые специалисты разметили как фрагменты, содержащие тот или иной дефект (1504 пометки “critical” или “major” для 795 различных примеров на четырех типах дефектов).

Описанные выше наборы данных объединяет ручная разметка примеров некоторым количеством проверяющих, что само по себе очень хорошо влияет на качество набора данных. Недостаток такого подхода следующий: этот процесс занимает значительное количество ресурсов опытных разработчиков, соответственно не может быть масштабируем на большие наборы данных в десятки тысяч примеров.

Другой подход используется в [7, 11]. Для получения наборов данных в этих работах применяются существующие инструменты поиска проявлений технического долга на большом количестве проектов с открытым исходным кодом. В [11] проводился анализ 33 проектов при помощи инструмента SonarQube[1], в [7] было проанализировано 1,072 C# и 100 Java проектов инструментами Designite[2] и DesigniteJava[3] соответственно. Эти публикации содержат большое количество размеченных фрагментов кода, однако они получены некоторым инструментом, а не разработчиками. Разметку, составленную при помощи DesigniteJava, будем использовать в дальнейшем, так как сам инструмент имеет открытый исходный код и мы имеем доступ непосредственно к алгоритму детекторов запахов кода. Основными ограничениями предложенных выше наборов данных являются две крайности. Во-первых, их трудозатратность для создания и масштабирования при высокой точности разметки. Во-вторых, вызывающее сомнение качество данных при практически неограниченном количестве примеров.

2. Построение набора данных. Рассмотрим схему получения набора данных, который доступен публично на Zenodo[4].

2.1. Выбор проектов с историей изменений. Для дальнейшего анализа было загружено топ-100 проектов с Github по количеству положительных отметок, основным языком которых является Java. Полный список и версии проектов можно найти в отдельном файле *projects.csv* вместе с набором данных. Большинство исследуемых проектов находятся в стадии активной разработки, 30% проанализированных изменений исходного кода приходятся на последние три года.

2.2. Инструмент сбора примеров. Инструмент анализирует каждый проект по очереди, проходя всю историю его изменений. На каждом шаге при помощи созданных детекторов запахов кода происходит анализ изменившихся в коммите файлов. Если анализ показал, что в коммите есть случаи исправления определенного дефекта, то каждый такой пример попадает в итоговый список своего детектора.

Определение исправлений кода. Будем считать парой ДО и ПОСЛЕ фрагменты кода перед исправлением, вносимым разработчиком, и после него соответственно. Детекторы ищут дефекты в версии файлов ДО и в версии файлов ПОСЛЕ. Затем отчеты этих двух запусков сравниваются. Если в файле пропал дефект в каком-то методе или классе, то этот случай фиксируется и происходит дополнительная проверка двух фрагментов.

Детекторы запахов кода. В текущем эксперименте использовалось три детектора: Божественный Класс (God Class), Сложный Метод (Complex Method), Длинный Метод (Long Method).

Для обучения базовых моделей применялись наборы данных MLCQ [8] и DLS [7], численные характеристики которых указаны в табл. 1.

В силу ограниченности количества примеров для Длинных Методов и Божественных Классов был использован подход, основанный на подсчете метрик по коду и модель машинного обучения Случайный Лес (Random Forest) для бинарной классификации фрагментов кода. Для

[1] https://www.sonarsource.com/.
[2] https://www.designite-tools.com/.
[3] https://github.com/tushartushar/DesigniteJava/.
[4] https://doi.org/10.5281/zenodo.7612725/.

Таблица 1. Описание обучающей выборки базовых моделей

Тип дефекта	Набор данных	Положительных	Отрицательных
Сложный Метод	DLS	22000	22000
Длинный Метод	MLCQ	140	2119
Божественный Класс	MLCQ	246	1719

Таблица 2. Аннотация полученных примеров

Тип дефекта	Всего	Положительных	Отрцательных	Положительных, %
Сложный Метод	164	106	58	0.65
Длинный Метод	132	97	35	0.73
Божественный Класс	32	24	8	0.75

Длинных Методов применялись LOC, NCLOC, MCC, TokenCount; для Божественных Классов — LCOM5, NAD, NMD [2]. Для работы со Сложными Методами была реализована сверточная нейронная сеть (CNN-2D), описанная в [7].

Постфильтры. В ходе валидации результатов сбора примеров стало ясно, что некоторые из исправлений, которые мы находили, не были нацелены на исправление дефектов, связанных с запахами кода. В "положительные" примеры исправления Длинных Методов попадали ситуации, в которых разработчик вносил незначительные стилистические исправления в свой код. В противоположной ситуации весь Длинный Метод переносился, оставляя в старом только вызов нового метода.

Для фильтрации таких примеров были написаны проверки на основе количественных метрик вносимых изменений. Итоговые фильтры получились следующими (срабатывание хотя бы одного из условий исключает данное исправление).

Длинный Метод:

- тело метода ПОСЛЕ меньше пяти строк;
- разница длин методов ДО и ПОСЛЕ меньше 12;
- относительное уменьшение длины метода меньше 0.2;
- нет достаточно длинного сплошного участка удаления кода из фрагмента ДО с заменой на небольшой участок в ПОСЛЕ.

Сложный Метод:

- тело метода ПОСЛЕ меньше пяти строк;
- относительное уменьшение сложности (по McCabe's Cyclomatic Complexity[5]) фрагмента меньше 0.25.

Божественный Класс:

- разница между количеством удаленных и добавленных методов меньше трех.

Все пороговые значения были получены экспериментально исходя из тестовой выборки объемом ~5%. Полученные детекторы и дополнительные фильтрации были использованы в анализе истории изменений 100 проектов.

2.3. Валидация полученных примеров. Для более удобного просмотра и валидации примеры отправлялись в LabelStudio[6] – инструмент для совместной аннотации примеров. Вручную было размечено 132 примера Длинных Методов, 164 примера Сложных Методов и 32 примера Божественных Классов. Аннотацию полученных примеров (табл. 2) проводили два опытных разработчика. Коэффициент согласия составил 92%.

Не очень высокий уровень положительных разметок, однако, не означает, что остальные пары не имеют дефекта в коде ДО, разметка проводилась с целью оценки пар с исправлениями дефектов. В остальных ~25–35% пар изменения не были явно направлены на рефакторинг или исправления искомого дефекта.

[5] https://radon.readthedocs.io/en/latest/intro.html#cyclomatic-complexity/.
[6] https://github.com/heartexlabs/label-studio/.

Таблица 3. Общие характеристики полученных данных

Характеристика	Значение
Общее число примеров	5912
Сложных Методов	2967
Длинных Методов	2517
Божественных Классов	428
Проанализировано проектов	100
Проанализировано коммитов	788485

3. Характеристики данных. Ниже представлены численные характеристики собранных данных и описание предоставляемого архива.

3.1. Запахи кода. Основные численные характеристики приведены в табл. 3. При анализе 100 проектов было получено около 6000 пар фрагментов исходного кода (с дефектом и без него).

3.2. Структура данных. Основная директория архива dataset содержит три поддиректории по каждому типу запахов кода: ComplexMethod, GodClass, LongMethod, в каждой из которых хранится список директорий с примерами before.java и after.java. Также на верхнем уровне находятся файлы с описанием примеров: ComplexMethod.csv, GodClass.csv, LongMethod.csv. Каждая запись о примере содержит следующую информацию:

- `project` – имя проекта, из которого взят пример,
- `path` – относительный путь к файлу в директории репозитория,
- `entity` – идентификатор интересующей нас сущности в формате ИмяКласса:ИмяМетода(типы аргументов),
- `url` – гиперссылка на коммит,
- `before file` – путь к файлу, содержащему метод/класс до изменения, относительно директории dataset,
- `after file` – путь к файлу, содержащему метод/класс после изменения, относительно директории dataset.

Также в корневой директории dataset содержится файл projects.csv, описывающий использованные для анализа проекты.

4. Апробация данных. Полученный набор данных можно использовать для обучения новых моделей машинного обучения для создания более точных детекторов технического долга. Из этих пар примеры до исправлений брались как код с дефектом, а примеры после исправлений – как код без дефекта. Количество примеров Длинных Методов стало значительно больше, чем в наборе данных MLCQ, соответственно, появилась возможность обучить более сложную модель, а именно LSTM, описанную в [7]. В данном эксперименте сравниваем модели, обученные на новых данных с предыдущими результатами. В качестве валидационных данных применялись размеченные вручную примеры из проектов: ant[7], drill[8], error-prone[9], giraph[10], hive[11], truth[12]. При разметке этого валидационного набора данных использовались три пометки: истинные срабатывания (TP), ложные срабатывания (FP) и истинные срабатывания, не требующие исправления (WF). Применение пометки “не требующие исправления” вызвано тем, что в данном случае “дефект” мог вноситься осознанно и специально, либо исправление такого участка кода потребует значительных затрат ресурсов. Оценивались точность (PR), полнота (RE) и F1-мера (F1).

Как видно из табл. 4, результаты моделей по всем типам дефектов улучшились. Для Длинных Методов за счет перехода на более сложную модель удалось получить значительное улучшение точности при незначительной потере полноты. Таким образом показано, что данный набор данных можно использовать для обучения новых более точных детекторов дефектов исходного кода.

[7] https://github.com/apache/ant.git, f61ac1296.
[8] https://github.com/apache/drill.git, 85d270f3.
[9] https://github.com/google/error-prone.git, 54fa3f38.
[10] https://github.com/apache/giraph.git, 2c63aa23.
[11] https://github.com/apache/hive.git, de0a7ecb.
[12] https://github.com/google/truth.git, eaddc49f.

Таблица 4. Результаты валидации обученных моделей

Модель	F1	PR	RE	WF	TP	FP
Длинный Метод						
Базовая	0.277	0.163	0.928	116	39	84
Новая	0.493	0.346	0.857	47	36	21
Сложный Метод						
Базовая	0.518	0.35	1.0	36	70	94
Новая	0.757	0.626	0.957	23	67	17
Божественный Класс						
Базовая	0.092	0.05	0.625	68	5	27
Новая	0.141	0.079	0.625	37	5	21

5. Текущие ограничения подхода. Очевидно, что мы нашли не все участки исправлений данных типов запахов кода на рассматриваемых проектах. С другой стороны, применение моделей, основанных на методах машинного обучения, и ручные эвристики могут вносить и ложноположительные срабатывания, однако такие примеры, скорее, не являются исправлениями дефекта. Метод/класс так или иначе стал меньше и проще для понимания, однако, возможно, не настолько, насколько это можно было сделать, если бы рефакторинг проводился целенаправленно.

Еще встречается рефакторинг вынесения значительного участка кода из метода/класса, который нельзя считать хорошим, если он был выделен целиком и стал новым примером запаха кода. Также в работе не были рассмотрены ситуации с переименованием файлов в коммитах, учитывались только изменения в рамках одного файла. Данный набор был собран исключительно на проектах с Java. Применять такой подход возможно и для других языков программирования, однако это требует наличия относительно неплохих базовых детекторов.

Заключение. Представлен набор данных существующих исправлений проявлений технического долга исходного кода, запахов кода. Отличительной чертой данного набора является предложенный способ сбора подобных примеров в автоматическом режиме, используя опыт непосредственных разработчиков ПО. Новые детекторы, обученные на собранном наборе данных, превосходят по точности классификации первоначальные.

СПИСОК ЛИТЕРАТУРЫ

1. *Fowler M.* Refactoring: Improving the Design of Existing Code. Boston, MA, USA: Addison-Wesley, 1999.
2. *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 // Тр. Института системного программирования РАН. 2021. Т. 33. № 6. С. 95–110.
3. *Tufano M., Palomba F., Bavota G. et al.* When and Why Your Code Starts to Smell Bad // IEEE/ACM 37th IEEE Intern. Conf. on Software Engineering. Florence, Italy, 2015. P. 403–414.
4. *Kokol P., Kokol M., Zagoranski S.* Code Smells: A Synthetic Narrative Review // Available at: https://arxiv.org/abs/2103.01088 (дата обращения 2023-01-25).
5. *Fontana F. A., Zanoni M.* Code Smell Severity Classification Using Machine Learning Techniques // Knowledge-Based Systems. 2017. V. 128. C. 43–58.
6. *Barbez A., Khomh F., Guéhéneuc Y. G.* A Machine-learning Based Ensemble Method For Anti-patterns Detection // J. Systems and Software. 2020. V. 161. P. 110486.
7. *Sharma T., Efstathiou V., Louridas P. et al.* On the Feasibility of Transfer-learning Code Smells Using Deep Learning // Available at: https://arxiv.org/abs/1904.03031 (дата обращения 2023-01-25).
8. *Madeyski L., Lewowski T.* MLCQ: Industry-relevant Code Smell Data Set // Proc. Evaluation and Assessment in Software Engineering. 2020. P. 342–347.
9. *Palomba F., Bavota G., Di Pentaet M. et al.* A Large-scale Empirical Study on the Lifecycle of Code Smell Co-occurrences // Information and Software Technology. 2018. V. 99. P. 1–10.

10. *Arcelli Fontana F. Mantyla M., Zanoniet M. et al.* Comparing and Experimenting Machine Learning Techniques for Code Smell Detection // Empirical Software Engineering. 2016. V. 21 C. 1143–1191.
11. *Lenarduzzi V., SaarimГ¤ki N., Taibi D.* The Technical Debt Dataset // Proc. 15th Intern. Conf. on Predictive Models and Data Analytics in Software Engineering. Recife, Brazil, 2019. P. 2–11.
12. *Wang Y. Yu H., Zhu Zh. et al.* Automatic Software Refactoring Via Weighted Clustering in Method-level Networks // IEEE Transactions on Software Engineering. 2017. V. 44. № 3. P. 202–236.
13. *Karampatsis R. M., Sutton C.* How Often do Single-statement Bugs Occur? The ManySStuBs4J Dataset // Proc. 17th Intern. Conf. on Mining Software Repositories. Online, 2020. P. 573–577.
14. *Palomba F., Di Nucci D., Tufano M. et al.* Landfill: An Open Dataset of Code Smells With Public Evaluation // IEEE/ACM 12th Working Conf. on Mining Software Repositories. IEEE. Florence, Italy, 2015. P. 482–485.
15. *Palomba F., Bavota G., Di Pentaet M. et al.* On The Diffuseness and The Impact on Maintainability of Code Smells: a Large Scale Empirical Investigation // Proc. 40th Intern. Conf. on Software Engineering. Gothenburg, Sweden, 2018. P. 482–482.
16. *Qualitas Corpus*. Available at: http://qualitascorpus.com/docs/history/20120401.html, (дата обращения 2023-01-25).