Санкт-Петербургский государственный университет

Математическое обеспечение и администрирование информационных систем
Системное программирование

Кузьмина Илия Викторовна

Анализ программного кода методами машинного обучения

Выпуская квалификационная работа

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

Рецензент:
ст. преп. Шпильман Алексей Александрович

Санкт-Петербург
2017

SAINT-PETERSBURG STATE UNIVERSITY

Software and Administration of Information Systems
Software Engineering

Iliia Kuzmina

Machine Learning Based Analysis of Object Oriented Code

Graduation Project

Scientific supervisor:
assoc. prof. Bryksin T.A.

Reviewer:
lecturer Shpilman A.A.

Saint-Petersburg
2017

Оглавление

Введение

Методы машинного обучения активно совершенствуются в течение последних десятилетий и применяются в разных областях, например, в медицине, экономике, компьютерном зрении, биоинформатике, информационном поиске и т.д. Однако, на сегодняшний день всё ещё существуют такие задачи, с которыми человек справляется намного лучше обученного алгоритма. Одно из таких направлений — анализ программного кода.

Можно выделить несколько конкретных задач, связанных с анализом кода, в которых применимо машинное обучение:

- поиск одинаковых участков кода;
- поиск паттернов и антипаттернов проектирования;
- поиск и исправление ошибок в коде;
- поиск и исправление ошибок архитектуры.

Объектно-ориентированный код наиболее интересен для исследований, так как этот подход к разработке сейчас является наиболее массово используемым. Кроме того, объектно-ориентированный код имеет относительно несложную структуру, и его компоненты легко поддаются анализу.

Ошибки проектирования при разработке программных продуктов усложняют переиспользование и дальнейшее сопровождение. На исправление таких ошибок может уходить много времени и сил, и в этом могли бы помочь средства автоматической реструктуризации программ.

Постановка задачи

Целью данной работы является применение методов машинного обучения в области автоматического рефакторинга объектно-ориентированного кода и проведение соответствующих исследований. Для достижения этой цели были поставлены следующие задачи.

1. Изучить, как в настоящий момент используется машинное обучение при анализе кода.
2. Выбрать алгоритмы и метрики, на основе которых будет осуществляться реструктуризация.
3. Реализовать алгоритм автоматической реструктуризации.
4. Провести тестирование алгоритма на учебных примерах и реальных программных продуктах.

1. Обзор существующих решений

1.1. Поиск паттернов проектирования в объектно-ориентированном коде

С помощью алгоритмов работы с нейронными сетями [2] в программах, написанных на объектно-ориентированном языках, могут быть найдены распространённые шаблоны проектирования [21, 23].

Для реализации конкретного паттерна проектирования обычно требуется взаимодействие нескольких классов, поэтому процесс распознавания паттернов делится на два этапа:

1. для каждого класса в программном коде определить, какую функцию в каждом из рассматриваемых паттернов этот класс может играть;
2. фиксируя наиболее вероятные комбинации ролей классов, определить, образуют ли они в совокупности один из рассматриваемых паттернов.

В качестве признаков, на основе которых осуществляется обучение нейронной сети, используются следующие характеристики:

- NOF (Number of Fields) — количество полей в классе;
- NSF (Number of Static Fields) — количество статических полей;
- NOM (Number of Methods) — количество методов;
- NOAM (Number of Abstract Methods) — количество абстрактных методов;
- NOPC (Number of Private Constructors) — количество приватных конструкторов;
- и т.д.

Также могут быть использованы более сложные метрики, например RFC (Response for Class) [22] или Ca (Afferent Couplings) [13].

1.2. Поиск ошибок в объектно-ориентированном коде

Ещё одна важная задача в области анализа программного кода объектно-ориентированных приложений — предсказание ошибок без тестирования. Здесь могут быть применены различные алгоритмы машинного обучения, в частности, Alternating Decision Tree [14] и LogitBoost [11], который показал наилучшие результаты [20] по сравнению с Artificial Neural Networks [2], Random Forest [3], Naive Bayes [6] и K-Star [12].

1.3. Поиск и исправление ошибок архитектуры

Автоматическая реструктуризация программ может осуществляться на уровне методов, классов или пакетов. В первом случае изменения происходят в структуре операторов метода, во втором — атрибуты и методы классов должны быть реорганизованы между классами. В случае реструктуризации на уровне пакетов некоторые классы могут быть перемещены в другие пакеты.

1.3.1. Реструктуризация на уровне методов

Один из механизмов автоматического рефакторинга методов в объектно-ориентированных приложениях использует взвешенный граф зависимостей, основанный на истории модификаций методов [15].

Также были опубликованы работы [17], посвящённые выявлению рефакторингов "Извлечение метода" (Extract Method Refactoring [5]). Одна из них основана на анализе графа зависимостей между операторами исходного кода, при построении которого учитывается плотность связи между ними [24].

1.3.2. Реструктуризация на уровне классов

К данному типу относятся алгоритмы Automatic Refactorings Identification (ARI) [26] и Hierarchical Agglomerative Clustering (HAC) [16], в основе

которого лежит векторизация сущностей (классов и методов). В вектор входят 4 метрики программного обеспечения:

1. глубина класса в дереве наследования (Depth in Inheritance Tree, DIT);
2. количество прямых классов-наследников (Number of Children, NOC);
3. метрика Fan-In (для классов: количество классов, ссылающихся на данный, для методов: количество методов, вызывающих данный);
4. метрика Fan-Out (для классов: количество классов, на которые ссылается данный, для методов: количество методов, которые вызываются из данного).

Также для классов и векторов строятся множества связанных сущностей (Relevant properties, RP). Далее вводится функция расстояний между векторизованными сущностями.

$$d(X,Y) = \begin{cases} \sqrt{\frac{1}{m} * ((1 - \frac{|RP_X \cap RP_Y|}{|RP_X \cup RP_Y|}) + \sum_{i=1}^{4}(v_{X_i} - v_{Y_i})^2)}, RP_X \cap RP_Y \neq \emptyset \\ \infty, otherwise \end{cases} \tag{1}$$

Далее на основе этой метрики были выполнены различные алгоритмы кластеризации, в том числе, алгоритм иерархической кластеризации [16].

1.3.3. Реструктуризация на уровне пакетов

Для количественной оценки структуры пакетов в приложении была введена специальная метрика (Package Structure Analysis metric [8]), а также был реализован инструмент, предназначенный для разработчиков, для анализа и визуализации структуры пакетов.

Также был предложен адаптированный алгоритм выделения сообществ в графе зависимостей между классами исходного кода (Constrained

Community Detection Algorithm, CCDA [25]). Взвешенный граф зависимостей между классами устроен следующим образом: каждой вершине соответствует класс, а вес ребра определяется как количество связанных методов или атрибутов двух классов. Дальнейший алгоритм основан на показателе качества (Quality Index) [18] Qw, адаптированном для использования во взвешенных графах.

2. Подход к реструктуризации кода на уровне классов

2.1. Построение моделей

В одном из рассматриваемых подходов для проведения рефакторинга типов "Перемещение метода" (Move Method) и "Перемещение поля" (Move field) [5] были построены векторные модели методов и полей, а также классов исходного кода. Для методов и классов использовались те же 4 метрики, что и в ARI [26]: DIT, NOC, Fan-In и Fan-Out, для полей метрика Fan-Out не определена. Множества Relevant Properties определяются следующим образом.

1. Для классов:
    - данный класс;
    - все его атрибуты и методы;
    - все интерфейсы, которые он реализует;
    - все классы-наследники.
2. Для методов:
    - данный метод и класс, который его содержит;
    - все атрибуты, к которым запрашивается доступ;
    - все методы, которые вызываются из данного;
    - все переопределения данного метода в классах-наследниках.
3. Для полей:
    - данное поле и класс, в котором оно содержится;
    - все методы, в которых запрашивается доступ к полю.

Расстояние между сущностями вычисляется по следующей формуле, которая является модифицированной версией функции расстояния, используемой в ARI (1).

$$d(X,Y) = \begin{cases} \sqrt{\frac{1}{m+1} * (2 * (1 - \frac{|RP_X \cap RP_Y|}{|RP_X \cup RP_Y|}) + \sum_{i=1}^{4} (v_{X_i} - v_{Y_i})^2)}, \\ RP_X \cap RP_Y \neq \emptyset \\ \infty, otherwise \end{cases} \tag{2}$$

Во втором подходе строится граф зависимостей между методами и атрибутами исходного кода. Каждому полю и методу ставится в соответствие вершина. Рёбра проводятся в двух случаях:

- между вызываемым и вызывающим методом;
- между используемым атрибутом и методом.

2.2. Алгоритм на основе поиска сообществ

В отличие от алгоритма, приведённого в статье, посвящённой инструменту для автоматической реструктуризации на уровне пакетов [25], в данном случае рассматриваемый граф является невзвешенным, поэтому используется классическое определение показателя качества Q [18]:

$$Q = \sum_{i=1}^{n} (e_{ii} - a_i^2), \tag{3}$$

где n — количество сообществ, e_{ii} — доля рёбер, соединяющих вершины сообщества i между собой, a_i — доля рёбер, у которых хотя бы один конец лежит в сообществе i.

В условиях данной задачи изначальное разбиение на сообщества уже дано, оно соответствует имеющемуся распределению атрибутов и методов между классами. Далее, на каждом этапе выбирается такое перемещение элемента между сообществами, которое ведёт к максимальному увеличению показателя качества Q. Процесс выполняется до

тех пор, пока это увеличение положительно; также на практике можно ограничивать его снизу заранее заданной константой.

Таким образом, некоторые из сообществ могут стать пустыми. В результате оставшиеся сообщества и входящие в них элементы образуют новое распределение полей и методов по классам.

Стоит отметить, что не требуется на каждой итерации вычислять значение Q заново. Его достаточно посчитать один раз на начальном этапе, а затем поддерживать актуальное значение после каждого изменения в структуре сообществ. Для этого нужно поддерживать текущие значения a_i для каждого из них, а значения e_{ii} хранить необязательно; при перемещении сущности s из сообщества i в сообщество j достаточно учитывать разницу в изменении e_{ii} и e_{jj}. Пусть $di_i(s)$ — количество соседей сущности s внутри сообщества i, $do_i(s)$ — количество соседей s вне i. Тогда:

- e_{ii} уменьшится на $\frac{di_i(s)}{w}$;
- e_{jj} увеличится на $\frac{di_j(s)}{w}$;
- a_i уменьшится на $\frac{do_i(s)}{w}$;
- a_j увеличится на $\frac{do_j(s)}{w}$.

То есть, изменение будет равно $\frac{di_j(s)-di_i(s)}{w} + (a_i^{old})^2 + (a_j^{old})^2 - (a_i^{new})^2 - (a_j^{new})^2$.

2.3. Алгоритмы на основе кластеризации

В рамках данной работы было реализовано три алгоритма кластеризации, основывающихся на введённой функции расстояния (2). В отличие от предложенных ранее алгоритмов автоматической реструктуризации программ, в них выделен отдельный тип сущности, соответствующий атрибутам классов. Это позволило расширить пространство предлагаемых исправлений к программе рефакторингом "Перемещение поля".

2.3.1. Модификация алгоритма ARI

Для каждой сущности исходного кода, соответствующей атрибуту или методу, считается расстояние до всех сущностей-классов. Среди них выбирается тот класс, расстояние до которого минимально, и сущность помещается в него.

В этом алгоритме не учитываются расстояния между методами и атрибутами. В результате могут быть предложены исправления "Перемещение метода" и "Перемещение поля".

2.3.2. Модификация алгоритма HAC

В вариации алгоритма иерархической кластеризации, применённой к векторной модели исходного кода, все три типа сущностей (классы, поля и методы) считаются однородными и обрабатываются единообразно.

На этапе инициализации каждая сущность помещается в отдельный кластер. Далее, на каждой итерации алгоритма находятся два ближайших кластера. Для вычисления расстояний между кластерами используется метод полной связи (complete-linkage [4]): это означает, что выбирается максимум по всем попарным расстояниям между элементами рассматриваемых кластеров. Найденные два кластера с минимальным расстоянием объединяются в один. Это продолжается до тех пор, пока существуют два кластера, расстояние между которыми не больше единицы.

Так как, согласно формуле (2), расстояние между сущностями может быть равно бесконечности, по завершению алгоритма все кластеры не будут объединены в один. Полученное разбиение на кластеры соответствует новому распределению сущностей по классам.

Таким образом, этот алгоритм идентифицирует возможные исправления четырёх типов: "Перемещение поля", "Перемещение метода", а также "Извлечение класса" и "Встраивание класса".

2.3.3. Алгоритм FMC

В ходе данной работы был предложен алгоритм кластеризации полей и методов (Fields and Methods Clustering, FMC), совмещающий в себе подходы иерархической кластеризации [9] и метода K-средних [27]. В нём рассматриваются только сущности, соответствующие полям и методам. Алгоритму требуется заранее знать количество кластеров, которые должны быть сформированы по окончанию, оно выбирается равным исходному количеству классов.

На начальном этапе создаётся указанное количество кластеров, центры которых инициализируются случайным образом из множества всех рассматриваемых сущностей, центры помещаются в кластеры. Далее, для каждой сущности вычисляются расстояния до всех кластеров по методу полной связи, среди них выбирается ближайший, и сущность помещается в него.

Также алгоритму необходимо указать количество описанных выше итераций. Для апробации на больших примерах эмпирически было выбрано сто итераций.

Этот алгоритм, как и ARI, предлагает возможные исправления по перемещению метода или поля между классами.

Стоит также отметить, что алгоритм FMC, несмотря на то, что определяет новую структуру атрибутов и методов и в ходе работы не оперирует с классами исходного кода, при выдаче результатов каждому построенному кластеру ставит в соответствие один из существующих классов.

3. Особенности реализации

3.1. Реализация плагина для среды разработки IntelliJ IDEA

Оба алгоритма автоматического рефакторинга реализованы в виде плагина для IntelliJ IDEA [10]. За основу был взят плагин MetricsReloaded [1], который предзназначен для вычисления основных метрик программного обеспечения. Однако, его функциональность не предоставляет возможности вычислить конкретную метрику для конкретного класса или метода. Поэтому в ходе работы был также выполнен анализ и дора архитектуры плагина.

Ниже представлена диаграмма классов, используемых в MetricsReloaded (Рис. 1).

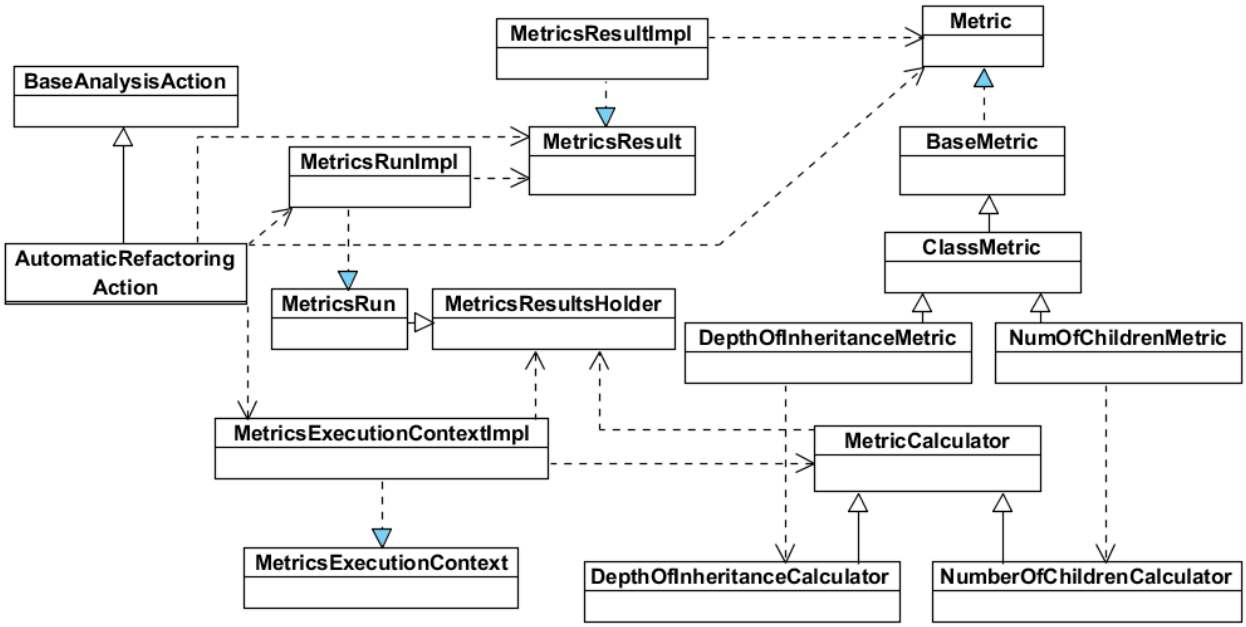

Рис. 1: Архитектура плагина MetricsReloaded

Интерфейс Metric хранит в себе общую информацию о метрике, MetricCalculator выполняет рекурсивный обход заданного файла и элементов программного кода, который в нём содержится, и для сохранения вычисленных значений использует MetricsResultHolder. Интерфейс MetricRun его расширяет, добавляя возможность получить результаты вычисления метрик для элементов разных типов (классов, методов, ин-

терфейсов, пакетов и т.д.), которые сохраняются в реализации интерфейса MetricsResult.

Интерфейсы Metric и MetricCalculator и их наследники не могут быть использованы обособленно от остальных элементов.

На диаграмме также присутствует класс AutomaticRefactoringAction, который является связующей компонентой между частью MetricsReloaded и алгоритмами автоматической реструктуризации.

3.2. Архитектура

На диаграмме представлена архитектура созданного инструмента автоматической реструктуризации программ (Рис. 2).

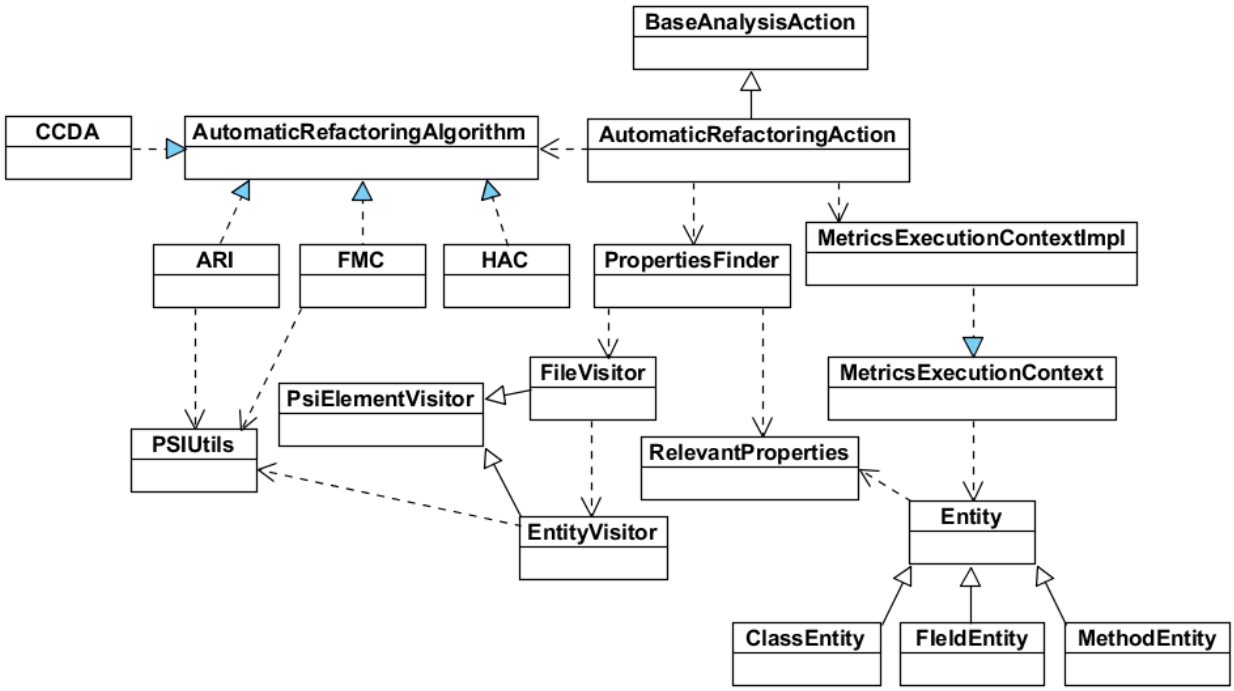

Рис. 2: Архитектура

Из компоненты AutomaticRefactoringAction сначала вызывается анализатор исходного кода PropertiesFinder, который реализован с помощью средсв PSI. Он строит граф зависимостей между полями и методами и множества связанных сущностей RelevantProperties.

Для сущностей каждого типа используются классы ClassEntity, MethodEnti и FieldEntity, являющиеся наследниками абстрактного класса Сущность (Entity).

После анализа кода выполняются четыре алгоритма автоматической реструктуризации (CCDA, ARI, HAC и FMC), объединённые общим интерфейсом AutomaticRefactoringAlgorithm.

3.3. Рекурсивный обход элементов кода с помощью PSI

Для построения графа зависимостей между полями и методами, а также для построения множества Relevant Properties для полей, методов и классов, были использованы средства Project Structure Interface (PSI) [19], предоставляемые IntelliJ Platform SDK. Для анализа кода классов и методов (PsiClass и PsiMethod соответственно) создаются наследники класса PsiElementVisitor. Некоторые из интересующих элементов, входящих в состав множества Relevant Properties (например, класс, к которому относится данный метод), уже хранятся в соответствующих классах. Однако, вычислить используемые атрибутов и вызываемые методов внутри класса PsiMethod не представляется возможным. Для этого в классе EntityVisitor нужно было переопределить методы обхода PsiReferenceExpression и PsiMethodCallExpression. Первый обрабатывает все ссылки на используемые переменные, а второй — все вызовы методов. Чтобы не терять информацию о том, внутри какого метода осуществляется доступ к переменной или вызов метода, при обходе методов все элементы PsiMethod кладутся на стек.

4. Апробация

4.1. Первый пример

Алгоритмы были протестированы на небольшом примере, которому соответствует исходный код, представленный ниже.

```
class class_A
{
    static void methodA1()
    {
        attributeA1=0;
        methodA2();
    }
    static void methodA2()
    {
        attributeA2=0;
        attributeA1=0;
    }
    static void methodA3()
    {
        attributeA1=0;
        attributeA2=0;
        methodA1();
        methodA2();
    }
    static int attributeA1;
    static int attributeA2;
}
```

```
class class_B
{
    static void methodB1()
    {
        class_A.attributeA1=0;
        class_A.attributeA2=0;
        class_A.methodA1();
    }
    static void methodB2()
    {
        attributeB1=0;
        attributeB2=0;
    }
    static void methodB3()
    {
        attributeB1=0;
        methodB1();
        methodB2();
    }
    static int attributeB1;
    static int attributeB2;
}
```

Рис. 3: Первый пример

Пример представляет собой реализацию двух классов, у каждого из которых по два атрибута и три метода, при этом, один из методов класса B использует атрибут класса A и вызывает его методы. Для человека перемещение метода methodB1() в класс A будет интуитивно понятным.

4.1.1. Результат работы алгоритма CCDA

В ходе работы алгоритма CCDA для данного примера исходного кода был построен следующий граф зависимостей между атрибутами и методами.

Изначально метод methodB1() относится к классу B, и показатель

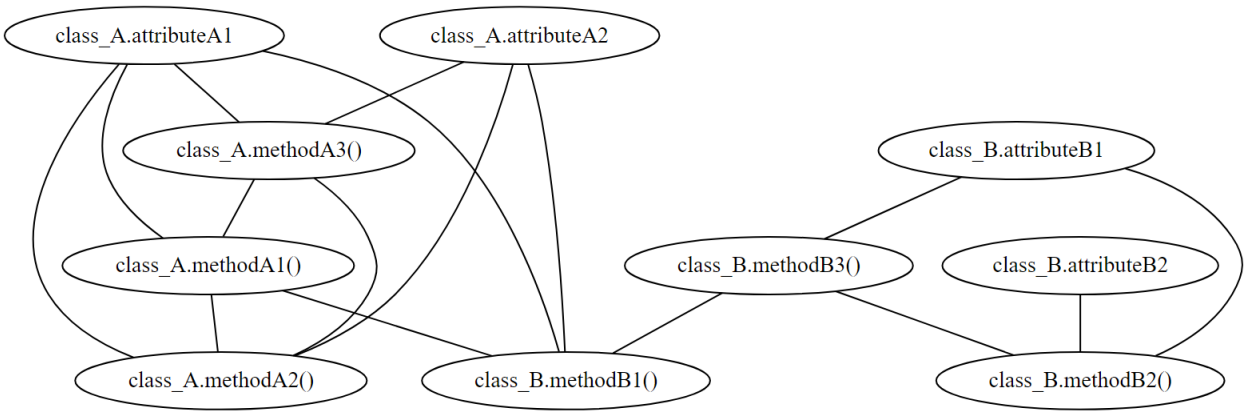

Рис. 4: Граф зависимостей, соответствующий первому примеру

качества Q приблизительно равен 0.08984375. На первом шаге алгоритма было идентифицировано перемещение сущности в другое сообщество, дающее максимальное увеличение Q: это перемещение метода methodB1() в сообщество, соответствующее классу A. После этого показатель качества увеличивается до 0.27734375 и далее не может быть улучшен.

4.1.2. Результат работы алгоритма ARI

Ниже представлена таблица расстояний между классами и атрибутами с методами (Рис. 5).

	aA1	aA2	mA1()	mA2()	mA3()	aB1	aB2	mB1()	mB2()	mB3()
class_A	0,4347	0,4295	0,4040	0,3486	0,4208	inf	inf	0,5	inf	inf
class_B	0,7150	0,6599	inf	inf	inf	0,4409	0,4677	0,5193	0,4040	0,3118

Рис. 5: Таблица расстояний между сущностями

В данном случае расстояние между методом methodB1() и классом A оказалось меньше, чем расстояние от него до класса B. Для остальных же атрибутов и методов их изначальные классы оказались ближе. Поэтому, единственное предложенное исправление для данного примера — это перемещение метода methodB1() в класс A.

4.1.3. Результат работы алгоритма HAC

На изображении представлена дендрограмма, полученная в ходе выполнения алгоритма иерархической кластеризации классов, методов и атрибутов (Рис. 6).

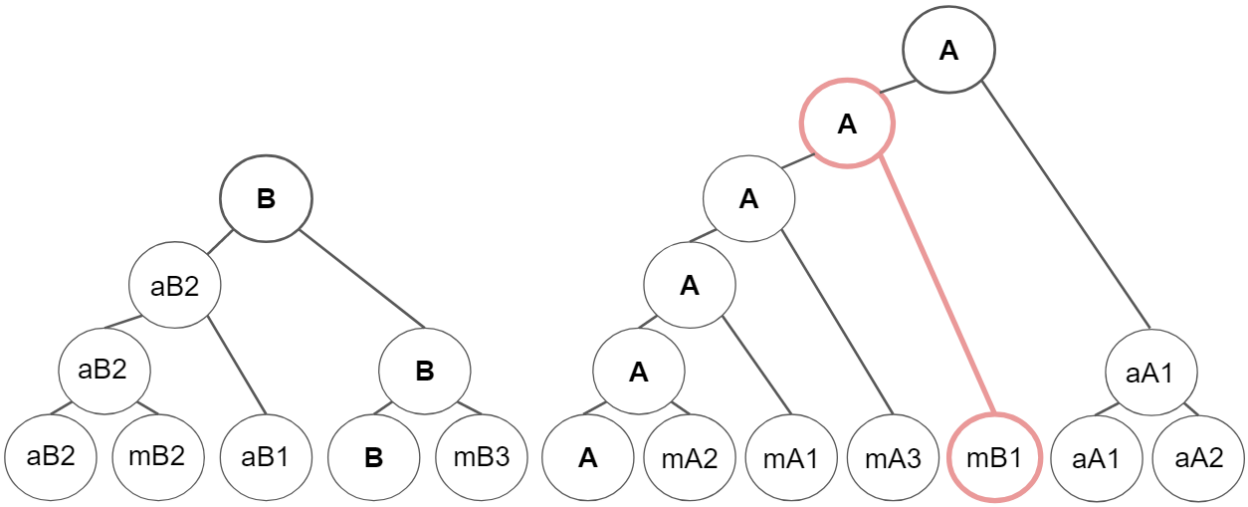

Рис. 6: Дендрограмма сущностей

На одном из шагов алгоритма метод methodB1() объединился с кластером, относящимся к классу A. Остальные сущности из класса B при этом объединились в другой кластер. Так как между некоторыми сущностями классов A и B расстояние равно бесконечности, итоговые два кластера не объединяются в один, что соответствует верному разбиению на классы.

4.1.4. Результат работы алгоритма FMC

После завершения первой итерации алгоритма FMC распределение сущностей по кластерам выглядит следующим образом (Рис. 7).

В данном случае центры кластеров были инициализированы методом methodA1() и атрибутом attributeB1. Затем, для метода methodB1() ближайшим кластером оказался кластер с центром methodA1(), для всех остальных сущностей ближайшими кластерами оказались те, чьи центры соответствуют тем же классам. Дальнейшие итерации не приводят к изменению структуры кластеров, поэтому алгоритм завершается, предложив только одно исправление, которое является верным.

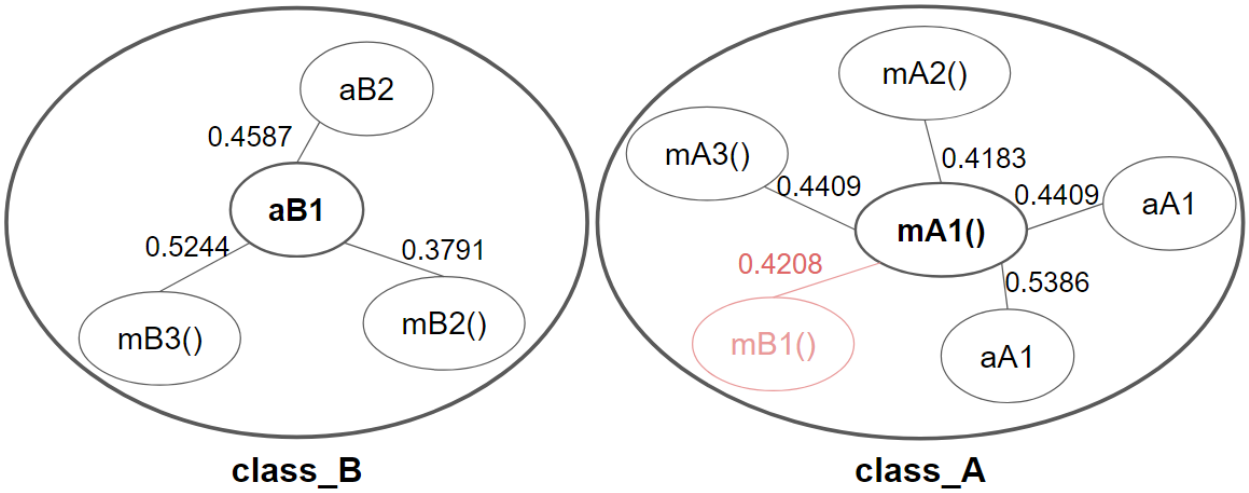

Рис. 7: Распределение сущностей по кластерам

4.2. Второй пример

Следующий пример иллюстрирует реализацию отношений Покупатель (класс Customer) и Услуга (класс Rental), в которой допущена ошибка: атрибут, хранящий в себе имя покупателя (cutomerName) относится к классу Rental, а не Customer.

Из четырёх рассмотренных алгоритмов только два смогли идентифицировать эту ошибку и корректно её исправить: CCDA и FMC.

Существенное отличие ARI и HAC от CCDA и FMC заключается в том, что они рассматривают классы как отдельные сущности, наравне с методами и атрибутами. Из этого можно сделать вывод, что в некоторых случаях взаимосвязи между сущностями, находящимися на одном уровне в дереве исходного кода (т.е. методами и атрибутами), должны быть учтены в первую очередь, а связи между атрибутами и классами, в которых они располагаются, должны быть ослаблены.

4.3. Третий пример

В качестве примера исходного кода реального приложения был использован проект JHotDraw [7]. Он был выбран по причине того, что является известным примером правильного использования шаблонов проектирования и имеет хорошую структуру классов. Проект состоит из 298 классов, внутри которых определены 3242 метода и 790 полей.

Первый алгоритм, CCDA, выявил 122 возможных исправлений по перемещению метода. ARI идентифицировал 91 рефакторинг по перемещению метода и 12 — по перемещению поля. В пересечении они генерируют всего 4 рефакторинга, 2 из которых являются корректными.

Алгоритмом FMC было предложено 323 исправления, при этом 59 из них также были найдены алгоритмом ARI. Среди семи случайно выбранных рефакторинга из их пересечения три оказались корректными.

Алгоритм HAC предложил 85 исправлений по перемещению метода и 4 по перемещению поля. Все рефакторинги последнего типа также были предложены ARI и не нарушают архитектуру программы.

В результатах каждого из перечисленных алгоритмов большой процент исправлений относится к перемещению метода или поля из класса в один из его абстрактных родительских классов или интерфейсов.

Заключение

В ходе выполнения данной работы были достигнуты следующие результаты.

1. Произведён анализ предметной области применения машинного обучения для анализа объектно-ориентированного кода.

2. Для реализации выбраны алгоритмы кластеризации и поиска сообществ в сложных сетях.

3. Реализован инструмент автоматической реструктуризации программ в виде плагина к среде разработки IntelliJ IDEA.

    - Модифицированы и реализованы два алгоритма кластеризации сущностей исходного кода и алгоритм выделения сообществ в сложных сетях.
    - Разработан и реализован алгоритм кластеризации методов и полей (FMC).

4. Произведена апробация алгоритмов на трёх примерах, в том числе на исходном коде проекта JHotDraw.

Результаты данной работы позволили расширить возможности алгоритмов автоматической реструктуризации посредством добавления нового типа рефакторинга "Перемещение поля". Кроме того, благодаря введению новой сущности, соответствующей атрибутам классов, векторная модель сущностей исходного кода стала полнее отражать существующие в нём зависимости.

Список литературы

[1] Automated code metrics plugin for IntelliJ IDEA. — 2011. — URL: https://github.com/BasLeijdekkers/MetricsReloaded (online; accessed: 19.04.2017).

[2] Bishop C. M. Neural Networks for Pattern Recognition. — Oxford University Press, 1995.

[3] Breiman L. Random Forests // Machine Learning. — Vol. 45. — P. 5–32.

[4] Complete-linkage clustering. — 2016. — URL: https://en.wikipedia.org/wiki/Complete-linkage_clustering (online; accessed: 20.05.2017).

[5] Fowler Martin. Refactoring: Improving the Design of Existing Code. — Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999, 1999.

[6] G. John P. Langley. Estimating Continuous Distributions in Bayesian Classifiers // Eleventh Conference on Uncertainty in Artificial Intelligence. — P. 338–345.

[7] Gamma Erich. JHotDraw. — 2017. — URL: https://sourceforge.net/projects/jhotdraw/ (online; accessed: 07.04.2017).

[8] Hautus E. Improving Java software through package structure analysis // Proceedings of the 6th IASTED International Conference on Software Engineering and Applications, IEEE. — 2002.

[9] Hierarchical clustering. — 2017. — URL: https://en.wikipedia.org/wiki/Hierarchical_clustering (online; accessed: 20.05.2017).

[10] IntelliJ IDEA. — 2000. — URL: https://www.jetbrains.com/idea/ (online; accessed: 19.04.2017).

[11] J. Friedman T. Hastie R. Tibshiran. Additive Logistic Regression: a Statistical View of Boosting. — Stanford University, 1998.

[12] John G. Cleary Leonard E. Trigg. K*: An Instance-based Learner Using an Entropic Distance Measure // 12h International Conference on Machine Learning. — P. 108–114.

[13] Jureczko M., Spinellis D. Using object-oriented design metrics to predict software defects // Proceedings of the Fifth InternationalConference on Dependability of Computer Systems, Monographs of System Dependability. — 2010. — P. 69–81.

[14] Kannadhasan N., Maheswari B. Uma. Machine Learning based Methodology for Testing Object Oriented Applications // Journal of Engineering and Applied Sciences. — 2015. — Vol. 10, no. 17. — P. 7400–7405.

[15] Katsuhisa Maruyama Ken-ichi Shima. Automatic Method Refactoring Using Weighted Dependence Graphs // Proceedings of the 1999 International Conference on Software Engineering, 1999. — 1999.

[16] Marian Zsuzsanna. A STUDY ON HIERARCHICAL CLUSTERING BASED SOFTWARE RESTRUCTURING // STUDIA UNIV. BABES-BOLYAI, INFORMATICA. — 2012. — Vol. LVII, no. 2.

[17] N. Tsantalis A. Chatzigeorgiou. Identification of extract method refactoring opportunities for the decomposition of methods // Journal of Systems and Software. — Vol. 84, no. 10. — P. 1757–1782.

[18] Newman M. E. J. Fast algorithm for detecting community structure in networks // Physical Review E. — 2004. — Vol. 69, no. 6.

[19] Project Structure. — 2015. — URL: `http://www.jetbrains.org/intellij/sdk/docs/basics/project_structure.html` (online; accessed: 19.04.2017).

[20] Ruchika Malhotra Yogesh Singh. On the Applicability of Machine Learning Techniques for Object Oriented Software Fault Prediction // Software Engineering: An International Journal. — Vol. 1, no. 1. — P. 24–37.

[21] S. Alhusain S. Coupland R. John, Kavanagh M. Detecting Design Patterns in Object-Oriented Program Source Code by Using Metrics and Machine Learning // 13th UK Workshop on Computational Intelligence (UKCI). — 2013. — P. 244–251.

[22] S. K. Dubey A. Sharma. Comparison study and review on object-oriented metrics // Global Journal of Computer Science and Technology. — 2012. — Vol. 12, no. 7.

[23] Satoru Uchiyama Atsuto Kubo Hironori Washizaki Yoshiaki Fukazawa. Detecting Design Patterns in Object-Oriented Program Source Code by Using Metrics and Machine Learning // Journal of Software Engineering and Applications. — 2014. — P. 983–998.

[24] T. Kanemitsu Y. Higo S. Kusumoto. A visualization method of program dependency graph for identifying extract method opportunity // Proceedings of the 4th Workshop on Refactoring Tools. — P. 8–14.

[25] Wei-Feng Pan Bo Jiang Bing Li. Refactoring Software Packages via Community Detection in Complex Software Networks // International Journal of Automation and Computing. — 2013. — P. 157–166.

[26] Zsuzsanna Marian Gabriela Czibula Istvan Gergely Czibula. Using Software Metrics for Automatic Software Design Improvement // Stud Inf Control. — Vol. 21. — P. 249–258.

[27] k-means clustering. — 2017. — URL: `https://en.wikipedia.org/wiki/K-means_clustering` (online; accessed: 20.05.2017).