Санкт-Петербургский государственный университет

Программная инженерия
Кафедра системного программирования

Кутленков Дмитрий Александрович

Рекомендация улучшений кода на Java в IntelliJ IDEA

Бакалаврская работа

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

Рецензент:
Программист ООО “ИнтеллиДжей Лабс”
Красильщиков Д. В.

Санкт-Петербург
2021

SAINT-PETERSBURG STATE UNIVERSITY

Software Engineering

Dmitrii Kutlenkov

Recommendation of Intention Actions in IntelliJ IDEA

Graduation Thesis

Scientific supervisor:
PhD Timofey Bryksin

Reviewer:
Software Developer
IntelliJ Labs
Dmitrii Krasilshchikov

Saint-Petersburg
2021

Оглавление

Введение

Интегрированные среды разработки (Integrated Development Environments, IDEs) на сегодняшний день являются одним из важнейших инструментов для программистов [1]. С течением времени падает число пользователей, использующих простые текстовые редакторы, и растет число пользователей, использующих среды разработки [2]. В частности, программисты активно используют средства автоматического рефакторинга кода, предоставляемые ими [3, 4].

В среде разработки IntelliJ IDEA [5] существует механизм под названием “Intention Actions”, название которого можно перевести как “намерения”. Намерения позволяют программисту применять некоторые часто выполняемые преобразования кода в автоматическом режиме. Например, такие действия могут оптимизировать код или делать его более читаемым, что экономит время и усилия программиста, позволяя не совершать монотонную работу вручную.

Данный механизм имеет несколько особенностей. Первая из них заключается в том, что намерения не всегда улучшают код: например, некоторые из них созданы для того, чтобы подготовить код к дальнейшим преобразованиям. Вторая особенность заключается в том, что иногда для получения наилучшего варианта кода приходится применять несколько намерений одно за другим. В результате программист может остановиться, применив одно намерение, но не получить при этом оптимальный вариант кода. Кроме того, само понятие качества кода субъективно: разные люди могут по-разному относиться к примененному к коду преобразованию.

Чтобы учесть данные особенности, предлагается научиться распознавать целые цепочки намерений, которые делают код лучше. Так как количество комбинаций цепочек, а также участков кода, к которым они могут применяться, очень велико, предлагается использовать для решения этой задачи методы машинного обучения.

Постановка задачи

Целью данной работы является разработка плагина для IntelliJ IDEA, способного классифицировать намерения и цепочки намерений, выдавая пользователю те из них, которые могут быть полезны в выбранной им позиции. Для достижения этой цели необходимо выполнить следующие задачи:

- Разработать инструмент, способный извлекать необходимые данные из кода, а именно, все возможные варианты кода, которые можно получить в выбранной позиции применением любого числа намерений. Применить данный инструмент к некоторой кодовой базе и привести полученные данные к подходящему для разметки виду.
- Обучить на полученных данных несколько моделей машинного обучения с различной архитектурой и выбрать из них наиболее точную. Модели должны определять, какие из вариантов кода, полученных с помощью механизма намерений, улучшают код.
- Разработать плагин для среды разработки IntelliJ IDEA, в который будет встроена обученная модель.
- Провести его апробацию на пользователях.

1. Обзор

Опишем подробнее задачу, которую мы хотим решить. На данный момент при применении механизма намерений среды IntelliJ IDEA на определенной позиции кода пользователь может выбрать какое-либо намерение, предлагаемое ему средой разработки (Рис. 1). Так как каждое из этих намерений меняет код по-своему, пользователь по сути выбирает, какой вариант кода он хочет получить в данной позиции (Рис. 2).

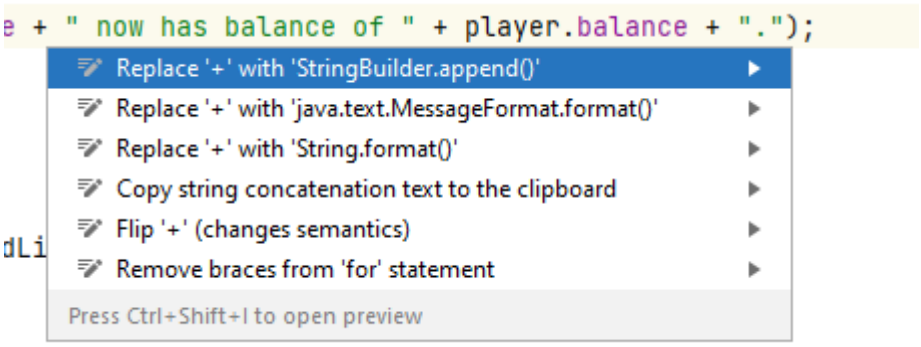

Рис. 1: Текущий вид механизма намерений

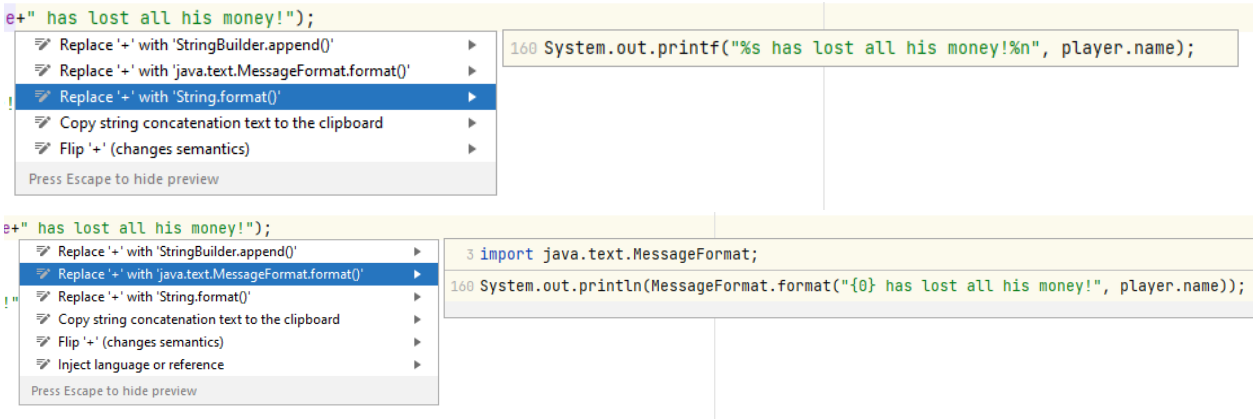

Рис. 2: Два варианта кода, получаемые с помощью разных намерений

Обычно число предлагаемых намерений невелик, в основном, около 5 штук. Однако же, после применения намерения операцию можно повторить, что может дать пользователю выбор из новых вариантов кода, недоступных после одного применения намерения. Таким образом, множество всех возможных вариантов кода (получаемых любым числом преобразований) можно представить в виде графа (Рис. 3).

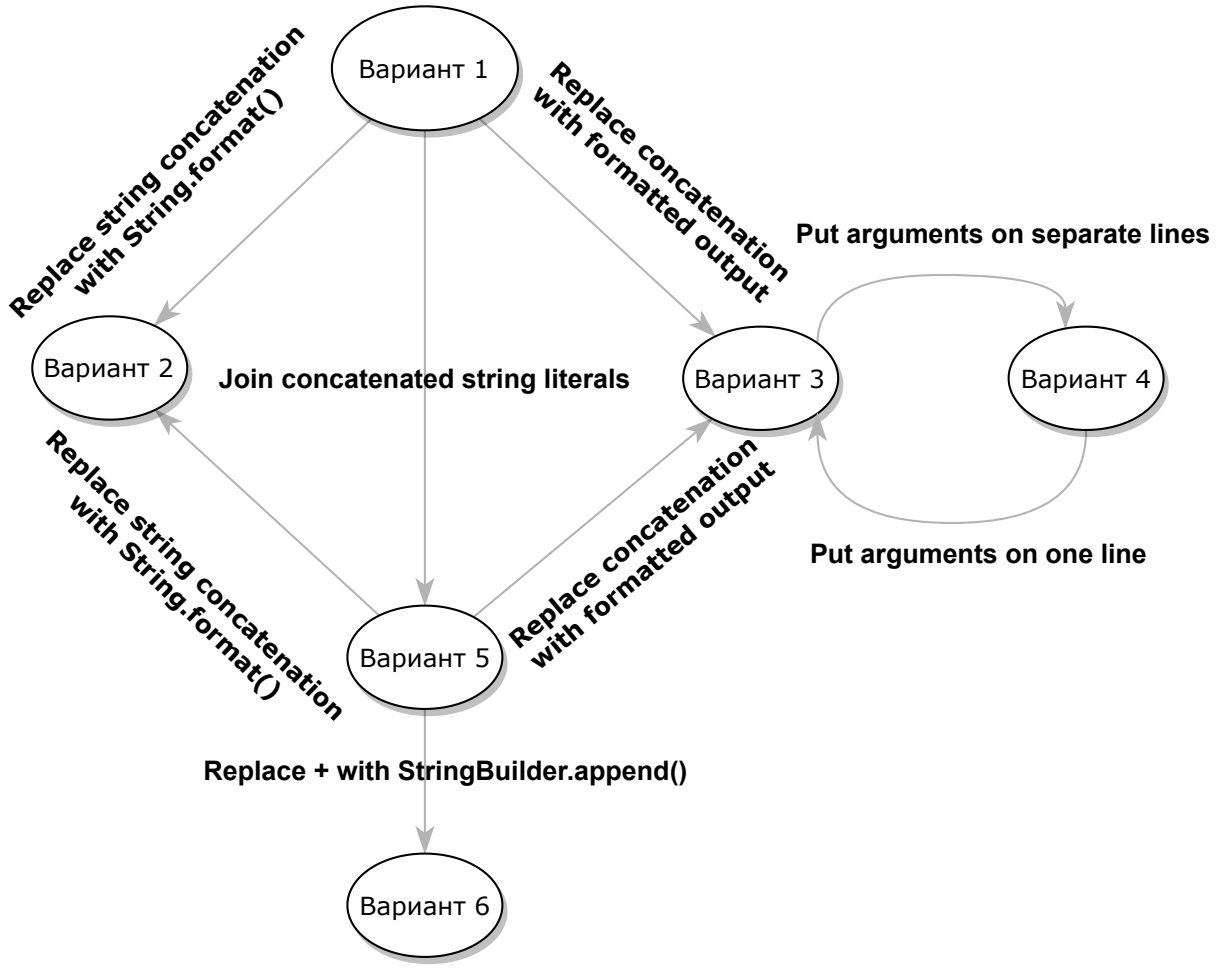

Рис. 3: Пример графа намерений. В узлах находится хэш получаемого кода, на ребрах — применяемое намерение.

Целью данной работы является создание плагина, который сможет выделять из этих вариантов те, что улучшают код. Другими словами, он должен предлагать пользователю те варианты преобразований, которые наиболее вероятно будут применены пользователем. Заметим, что такой плагин должен работать быстро, не дольше нескольких сотен миллисекунд, чтобы пользователь не замечал процесса работы инструмента (длительные задержки будут нивелировать полезность плагина, ценность которого заключается в том числе в ускорении процесса написания кода).

Для решения этой задачи необходимо уметь сравнивать варианты кода между собой. В частности, в рамках данной задачи мы должны будем сравнивать между собой довольно похожие варианты кода: так как намерения применяются в одном месте файла и обычно за-

трагивают лишь несколько строк кода, бóльшая часть файла остается неизменной. Таким образом, нам нужно будет сравнивать небольшие участки кода вокруг места, в котором произошло изменение. Из этого следует, что метод сравнения должен быть достаточно чувствителен к небольшим изменениям.

Учитывая все перечисленные ограничения, накладываемые задачей, рассмотрим существующие решения задач, похожих на те, которые необходимо решить в рамках данной работы.

1.1. Рекомендация преобразований кода

Согласно Мартину Фаулеру, рефакторинг представляет собой процесс такого изменения программной системы, при котором не меняется внешнее поведение кода, но улучшается его внутренняя структура. [6]. Чаще всего рефакторинг производится с целью улучшения читаемости кода или его подготовки к дальнейшим преобразованиям.

В целом, подходы к рекомендации рефакторингов можно разделить на две основные категории — подходы, основанные на эвристиках, и машинное обучение. Мы рассмотрим несколько статей из каждой категории, чтобы решить, какой подход будет перспективнее всего в нашей задаче.

1.1.1. Эвристические методы

В статье Haas et al. [7] авторы использовали подход, основанный на функции оценки. Идея авторов заключалась в том, чтобы обнаружить в коде участки, являющиеся кандидатами на рефакторинг “Извлечение метода”, а затем применить к каждому из них оценочную функцию. Оценочная функция была придумана авторами на основе некоторых признаков кода, таких как его сложность, глубина и число параметров, необходимых потенциальному методу. Опрос, в котором приняли участие 10 опытных программистов, показал, что 74% из них применили бы рекомендованные рефакторинги. Также подход, основанный на функции оценки, использует плагин JExtract [8] для Eclipse IDE.

В статье Palomba et al. [9] был использован метод анализа текста для определения запахов кода. Авторы извлекали текстовую информацию из файлов, приводили к единому виду, а затем пытались подсчитать вероятность того, что тот или иной запах кода присутствует в обработанном фрагменте.

Таким образом, главным недостатком эвристических методов является необходимость создания отдельной эвристики для нахождения каждого запаха кода. Это делает их применение в нашей задаче невозможным, так как наше решение должно быть универсальным и рекомендовать все возможные рефакторинги.

1.1.2. Методы машинного обучения

В статье Yue et al. [10] был использован подход, основанный на машинном обучении. Авторы искали в коде дублирования, а затем исследовали историю изменений кода, чтобы разбить их на группы, в которых содержится дублированные участки. Далее, для каждого участка кода извлекались 34 признака, которые были использованы для обучения модели, которая должна была предсказывать, пригоден ли код для рефакторинга. В качестве классификатора использовалась модель AdaBoost [11], основанная на деревьях решений. В результате авторы получили точность и полноту, равные 77% при тестировании на шести крупных проектах с открытым исходным кодом. Эта работа показывает, что извлечение признаков кода с последующим применением к ним деревьев решений показывает хорошие результаты в задаче распознавания мест в коде, в которых рекомендуется применить рефакторинг.

Схожие идеи использовались и в статье Fontana et al. [12], авторы которой распознавали различные запахи кода (что является по сути поиском кандидатов на рефакторинг). Авторы использовали 76 проектов, извлеченных из набора данных Qualitas Corpus [13]. Для классов из них они подсчитывали набор метрик, как глобальных (относящихся к проекту в целом), так и локальных (относящихся к определенному участку кода). Авторы тестировали множество моделей, как основанных на деревьях решений, так и использующие другие подходы, напри-

мер, байесовский классификатор. Одной из метрик, использованных авторами в данной работе, является F1-score — гармоническое среднее между точностью и полнотой, значение равное 1 означает идеальное предсказание. В исследовании рассматривались несколько запахов кода, но для каждого из них модели, основанные на деревьях решений, показали значение этой метрики не менее 0,94.

Глубокое машинное обучение для обнаружения нескольких запахов кода использовалось в статье Liu et al [14]. Авторы извлекали текстовую информацию из кода, а затем превращали ее в векторы с помощью технологии *word2vec* [15]. Полученные вектора отправлялись в сверточную нейронную сеть для определения наличия запаха кода.

В данной работе мы хотим научиться анализировать весь диапазон рефакторингов, предоставляемый средой IntelliJ IDEA. Таким образом, нам не подойдет подход, основанный на эвристиках, так как он требует индивидуальной оценки для каждого преобразования. Вместо этого будем использовать подход, основанный на машинном обучении.

1.2. Сравнение вариантов кода

Рассмотрим другие существующие подходы к сравнению вариантов кода, чтобы понять, не можем ли мы применить какой-то из них в нашей задаче. Наиболее часто задача такого сравнения возникает при поиске дубликатов в коде. Самым простым можно назвать подход, рассматривающий код как текст [16, 17]: он хорошо подходит для поиска точных копий кода, однако плохо справляется с измененным кодом. Несмотря на наличие модификаций, позволяющих учитывать незначительные изменения, такие как переименование переменных [18], область применения таких методов сильно ограничена их неспособностью учитывать сложные изменения.

Развитием предыдущего подхода можно назвать метод, рассматривающий код как набор токенов [19, 20]. Наиболее популярным способом работы с ними является метод bag-of-tokens, в котором текст представляется как множество пар из токена и частоты его встречаемости. Бла-

годаря высокой скорости работы данный метод применим на больших объемах данных, из-за чего он распространен в инструментах для поиска дубликатов в коде, например, он используется в SourcererCC [21] — инструменте для поиска дубликатов в большом количестве файлов. Однако в нашей задаче сравнения вариантов кода с небольшими изменениями данный метод не подходит в силу того, что он разрушает стуктуру кода.

Кроме того, код можно представлять в виде абстрактного синтаксического дерева (Abstract Syntax Tree, AST) [22]. Данный подход используется в известном инструменте для поиска клонов DECKARD [23]. В нашей задаче его применение не представляется возможным, так как рассматриваемые нами изменения могут и не менять AST, однако же сильно влиять, например, на читаемость кода.

1.2.1. Подход на основании метрик

Также для сравнения вариантов кода используют подход на основании метрик. Данный подход является наиболее универсальным среди перечисленных, так как позволяет исследователю выбирать именно те метрики, которые подходят к его задаче. Еще одним его плюсом является возможность описать большое количество преобразований с помощью небольшого числа метрик, так как для разных преобразований решающее значение будут иметь разные комбинации метрик. Мы рассмотрим несколько статей и используемые в них метрики.

В работах Milutin [24], Varela et al. [25] представлен обширный обзор метрик кода. Бóльшая их часть описывает метрики сложности больших программ. В нашей же задаче мы будем сравнивать два участка кода, имеющие небольшие изменения в пределах нескольких строк, из чего следует, что нам не нужно обрабатывать целый файл с кодом, так как бóльшая его часть будет совпадать у обоих примеров. Поэтому нам не подойдут метрики для работы с большими участками кода (такие как, например, метрики Маккейба [26], которые оценивают сложность программы через граф потока управления).

Статья Cotroneo et al. [27] фокусируется на предсказании ошибок

программы с помощью метрик кода. Из метрик авторы используют метрики Холстеда, Маккейба, а также метрики, связанные со структурой программы, такие как число выражений, число пустых строк и другие. Учитывая ограничения на размер участков кода, описанные выше, перспективными выглядят именно последние.

Статья Muthukumaran et al. [28] также имела целью предсказание ошибок с помощью метрик кода и моделей машинного обучения. Авторы использовали 4 группы метрик — метрики Холстеда, Маккейба, количество строк кода определенного вида и другие количественные метрики. Первые три группы проанализированы выше, а среди последних можно выделить такие метрики, как число параметров, число условных выражений, которые мы можем попробовать использовать в нашей работе.

В силу своей универсальности подход на основании метрик выглядит наиболее перспективным. Однако, его использование связано с необходимостью разработки собственного набора метрик, подходящего для решения нашей задачи. В процессе разметки данных можно будет выделить некоторые признаки, которые влияют на восприятие кода, которые затем можно будет реализовать в виде метрик.

1.3. Модели машинного обучения

Как было сказано ранее, для решения задачи выбора лучшего из двух вариантов кода необходимо научиться сравнивать варианты между собой. Предлагается сделать это с помощью классификатора, который будет определять, какой вариант кода лучше — изначальный или преобразованный. Для решения задачи классификации на табличных данных, которые мы получим после извлечения признаков из кода, подходит большое количество моделей. Однако, при выборе моделей, с которыми мы будем проводить эксперименты, необходимо учитывать, что в итоге модель будет встраиваться в плагин к IDE. Это, например, не позволяет нам использовать графические ускорители, которые используют многие модели (например, нейронные сети), потому что не

все пользователи их имеют на своих компьютерах, где запускают IDE. Таким образом, нам необходимы легковесные модели, способные быстро работать без графического ускорителя.

В литературе среди таких моделей лучше всего себя показывают деревья решений. В частности, в статье Fontana et al. [29], в которой исследовались запахи кода, авторы использовали подход, основанный на выделении признаков, и провели сравнение 25 моделей. Лучшими на основании метрики F1-score оказались модели, основанные на деревьях решений, — J48 и Random Forest [30] из инструмента Weka [31]. Деревья решений хорошо себя показывали и в предыдущей работе [12] этих авторов. Позднее, другие исследователи подтвердили [32] полученные ими результаты в части превосходства моделей, основанных на деревьях решений. Также стоит заметить, что деревья решений хороши своей интерпретируемостью и скоростью работы, что стоит принимать во внимание при подборе модели для решения нашей задачи.

В рамках данной работы будут протестированы вышеперечисленные и другие модели, а показавшая наилучшие результаты будет использована в плагине.

1.4. Компоненты IntelliJ IDEA

Важным компонентом среды разработки IntelliJ IDEA является Program Structure Interface (PSI). Этот интерфейс отвечает за парсинг файлов и создание синтаксической и семантической моделей кода. С его помощью создается PSI-дерево — представление кода в виде иерархии его элементов, которое по сути является конкретным синтаксическим деревом (Concrete Syntax Tree, CST), по которому можно перемещаться с помощью указателя. С помощью этого представления можно получать информацию о коде, а также изменять его. Все механизмы IntelliJ IDEA, работающие с кодом, основаны на этом механизме.

Все элементы дерева реализуют интерфейс *PSIElement*, с помощью которого происходит навигация по дереву. Например, интерфейс содержит методы для получения списка детей, ссылки на родителя, следу-

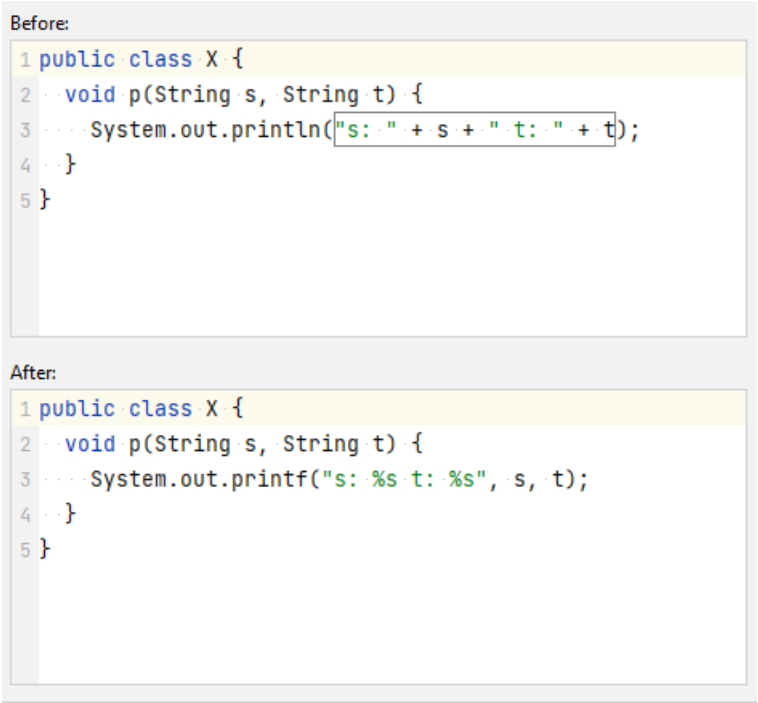

Рис. 4: Пример применения намерения “Replace string concatenation with String.format()”.

ющего и предыдущего соседей. По каждому элементу можно получить его позицию в тексте.

Для обхода дерева существует абстрактный класс *PsiRecursiveElementWalkingVisitor*, для которого необходимо определить метод *visitElement()*, который будет вызываться для каждого элемента в дереве. С его помощью можно перебрать все элементы и для каждого построить граф намерений, что будет нам полезно при сборе данных.

Механизм “Intention Actions” (Рис. 4) активно использует PSI-дерево для проверки и применения намерений. В частности, каждое намерение реализует интерфейс *IntentionAction*, в котором метод *isAvailable()* отвечает за проверку применимости намерения в данный момент, а метод *invoke()* применяет его путем изменения PSI-дерева.

Помимо PSI-дерева, в IntelliJ IDEA существует множество классов, позволяющих программно изменять состояние редактора. Для доступа к различным частям среды разработки существует набор классов-менеджеров. Например, класс *PsiDocumentManager* позволяет получить объект *Document*, который отвечает за состояние открытого докумен-

та. С его помощью можно записывать и отменять произведенные изменения. Работа с данным классом необходима, так как IntelliJ IDEA использует блокировку чтения-записи при работе с кодом. В результате этого каждая операция по изменению кода должна быть вызвана с помощью класса *WriteAction*. Каждое изменение, произведенное таким образом, должно быть сохранено с помощью класса *Document*, прежде чем оно станет видимо для остальных частей программы.

Класс *Editor* позволяет имитировать поведение пользователя, например, перемещать курсор. Это полезно, так как применение намерения подразумевает, что оно вызывается пользователем и использует положение курсора для определения места применения. Так как мы применяем намерения программно, нам необходимо программно передвигать курсор.

Для вызова плагинов IntelliJ IDEA использует следующий механизм. В системе регистрируется действие (например, нажатие на кнопку), при выполнении которого запускается код из класса, созданного разработчиком. Такой класс должен наследоваться от класса *AnAction()* и реализовывать метод *actionPerformed()*.

1.5. Выводы

Как было описано выше, ключевым требованием к выбранному способу решения задачи является его быстродействие, так как конечный инструмент должен работать мгновенно, незаметно для пользователя. Этому требованию удовлетворяет подход, основанный на выделении признаков из кода, с последующим обучением некоторой быстродействующей модели, способной работать без графического ускорителя. Перспективными кажутся модели, основанные на деревьях решений, однако модель, которая будет использоваться в конечном результате, будет определяться на основании экспериментов.

2. Подготовка модели

2.1. Инструмент для извлечения намерений

Для обучения классифицирующих моделей первым делом необходимо собрать данные для их обучения и провести их разметку. В нашем случае данные можно извлечь из графа намерений. Для построения такого графа и приведения информации из него к удобному для разметки виду был создан вспомогательный инструмент в виде плагина для среды разработки IntelliJ IDEA. Рассмотрим далее основные составляющие разработанного инструмента.

На Рис. 5 изображена диаграмма потока данных — последовательность преобразований, которые происходят с файлом, прежде чем по нему будут сгенерированы данные для разметки. После запуска плагина пользователю демонстрируется окно, где он может указать путь к файлу, который он хочет обработать. Если данный файл имеет расширение *.sample*, то программа рассматривает его как набор путей к файлам, которые необходимо обработать, и передает его в технический класс *PathApplier*, который передает каждый элемент файла в *FileApplier*. Если же пользователь указал только один файл с расширением *.java*, он сразу отправляется в *FileApplier*.

Класс *FileApplier* отвечает за извлечение позиций, в которых целесообразно построить граф намерений. Он совершает обход по PSI-дереву и отвечает за то, чтобы для каждого PSI-элемента и позиции в файле запуск проводился не более одного раза. Таким образом происходит применение инструмента к каждому синтаксическому элементу вместо применения к каждой позиции в файле, что сокращает число вызовов инструмента. При этом мы не теряем информацию о каких-либо позициях, так как механизм намерений также работает с синтаксическими элементами.

Для каждой позиции вызывается класс *SequentialApplier*. В нем заключена основная логика применения всех возможных намерений к данной позиции. Происходит построение графа намерений (Рис. 3), а

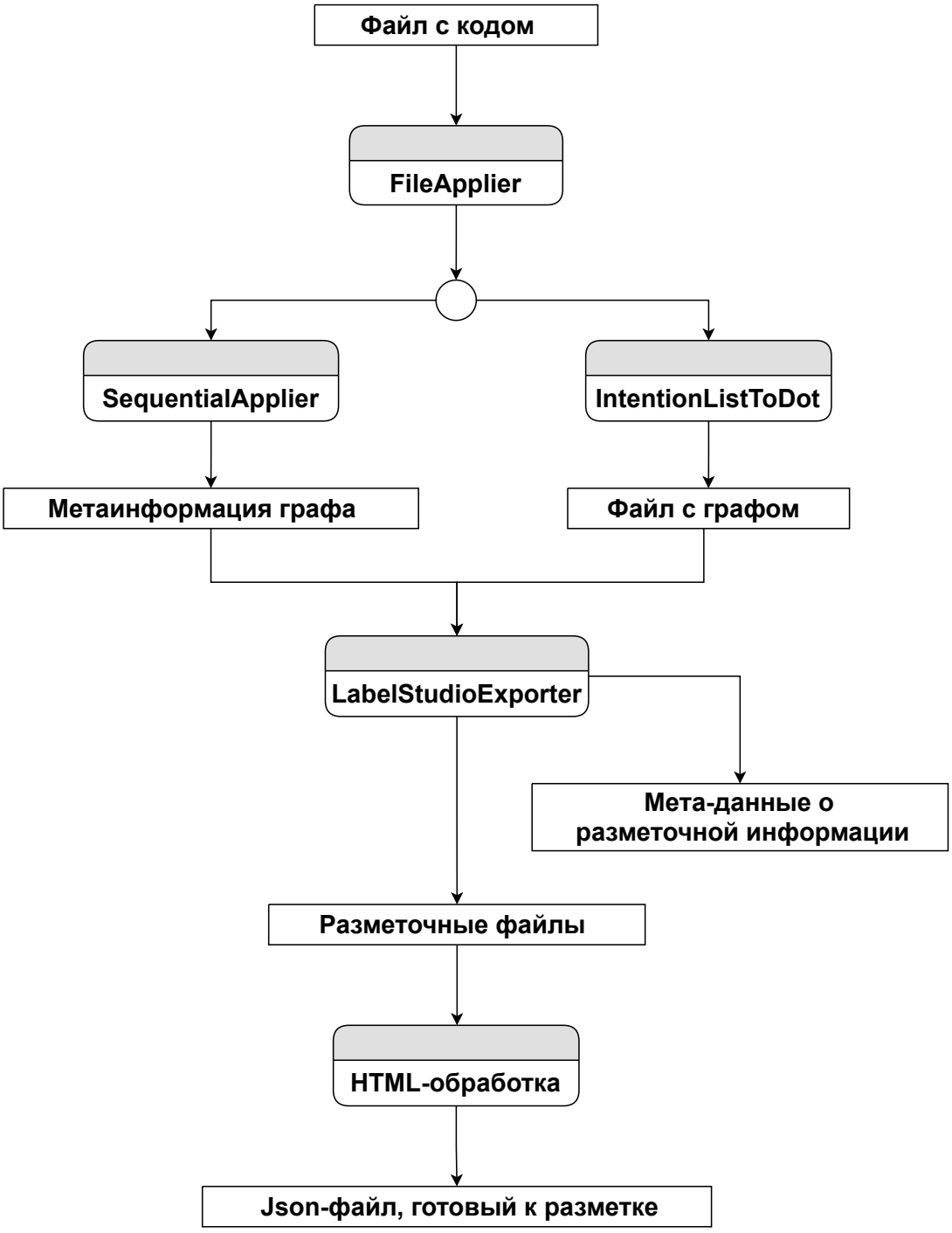

Рис. 5: Диаграмма потока данных

также выделение участка кода вокруг места применения намерения с целью дальнейшей разметки. Для получения намерений, применимых в выбранной позиции, используется класс *CurrentPositionHandler*. Он запрашивает у среды разработки список намерений, отфильтровывает те, которые меняют семантику кода или требуют ввода дополнительной информации от пользователя (фильтрация таких намерений про-

исходит с помощью созданного вручную списка), а затем вызывает для каждого из них метод *isAvailable()* для определения тех намерений, которые можно применить в выбранной позиции. При построении графа намерений этот процесс повторяется для каждого полученного варианта кода. Полученные на данном этапе данные о соответствии между вершинами графа и участками кода записываются в файл. Затем полученный граф, записанный в формате DOT, выгружается на диск с помощью класса *IntentionListToDot*.

В качестве среды разметки был выбран фреймворк LabelStudio [33], предоставляющий широкие возможности для конфигурации разметки. Особенно удобной функциональностью оказалась возможность генерации графического представления HTML-кода внутри разметочных примеров. Она позволила сделать фрагменты кода более читаемыми для программиста с помощью добавления подсветки синтаксиса и места применения намерения. Из графа, полученного на предыдущем шаге, извлекаются пары, где первый элемент — неизмененный код, а второй — полученный применением одного или нескольких намерений. Данный класс также занимается ограничением количества таких пар (в дальнейшем мы будем называть их “заданиями”) с целью сокращения числа однотипных заданий при разметке. Полученная заготовка экспортируется в *.json* файл.

Стоит отметить, что полученный на предыдущем шаге код еще не является материалом для разметки. Он еще не оформлен в HTML и содержит в себе технические метки. Для создания HTML-представления полученных пар был разработан скрипт на языке Python с использованием библиотеки pygments [34]. Полученный в результате *.json* файл готов к загрузке в систему и разметке.

2.2. Подготовка данных для обучения

В качестве кодовой базы для генерации примеров для разметки использовался проект с открытым исходным кодом guava [35], который является набором библиотек от компании Google. После обработки ин-

струментом для извлечения данных был получен 2631 пример, которые затем были размечены. Данные были разделены на 10 групп и разметка производилась шестью разными людьми — тремя сотрудниками компании JetBrains и тремя студентами кафедры системного программирования СПбГУ. Это позволило снизить влияние субъективного восприятия каждого человека на итоговую модель. Разница в опыте среди размечающих также позволила в итоге сделать модель менее заточенной под определенную группу людей и их предпочтения. Для дополнительной оценки работы модели были использованы другие проекты — dbeaver [36], инструмент для управления базами данных, и presto [37] — инструмент для SQL запросов на больших данных. По ним было сгенерировано и размечено 2037 и 500 примеров соответственно. Тестирование по другим проектам проводилось, чтобы понять, как модель, обученная на одном проекте, покажет себя на других проектах. Это тестирование важнее, чем тестирование на одном и том же проекте, так как моделирует применение плагина в реальной жизни, когда обученная нами модель используется на коде из других проектов. Низкий показатель при этом тестировании может сказать нам о том, что наша модель переучилась конкретно на том коде, что присутствовал в обучающем проекте.

2.2.1. Цепочки намерений

Отдельно стоит упомянуть вклад цепочек намерений в результаты разметки. Из 2631 размеченных примеров в 576 случаях размечающие посчитали, что преобразование улучшает код, что составляет почти 22%. Среди этих примеров 46% пришлось на преобразования, полученные в результате применения нескольких намерений (Рис. 6). На гистограмме можно увидеть, что с увеличением количества примененных намерений падает количество положительно размеченных. Это ожидаемо, так как во-первых, большое количество намерений меняет код слишком сильно, а во-вторых, существует меньше мест, где можно применить большое число намерений. Самые длинные цепочки, размеченные как хорошие, имели длину 6, таких оказалось 4 штуки.

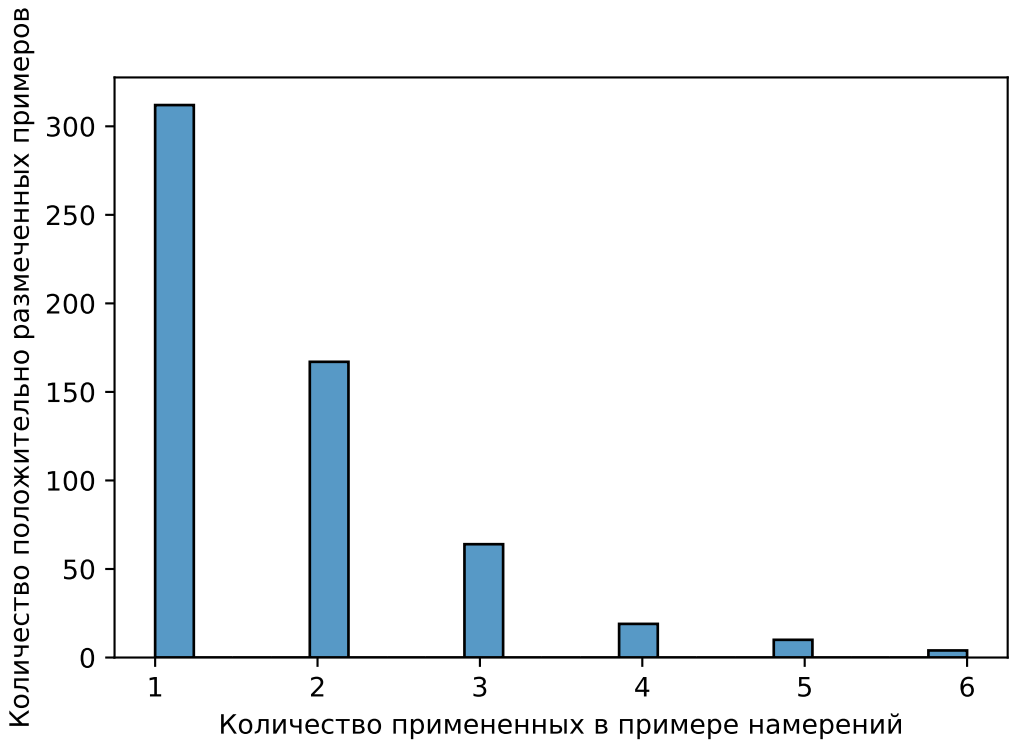

Рис. 6: Гистограмма количества положительно размеченных примеров относительно количества примененных преобразований.

2.3. Векторизация кода

Векторизация кода производилась с помощью 9 метрик (Таблица 2), которые были подобраны эмпирически на основании данных о том, как существующие в среде разработки намерения изменяют код. Выбирались те метрики, которые могут достаточно хорошо описать изменения, производимые намерениями, которые было решено поддержать в разрабатываемом плагине (Таблица 1). Например, на Рис. 7 представлено намерение, переносящее аргументы на новые строки. Такое преобразование выглядит плохо, когда аргументы занимают мало места. Однако, если каждый из аргументов достаточно длинный, то вся строка становится нечитаемой и такой перенос улучшает код. Метрика, подсчитывающая среднюю длину аргумента в символах как раз и сообщит модели, стоит ли применять такое преобразование. По каждой паре вариантов кода подсчитываются два набора абсолютных значений, а также третий, в который записывается разница между этими двумя наборами. Это делается для того, чтобы модель могла оценить, как изменился

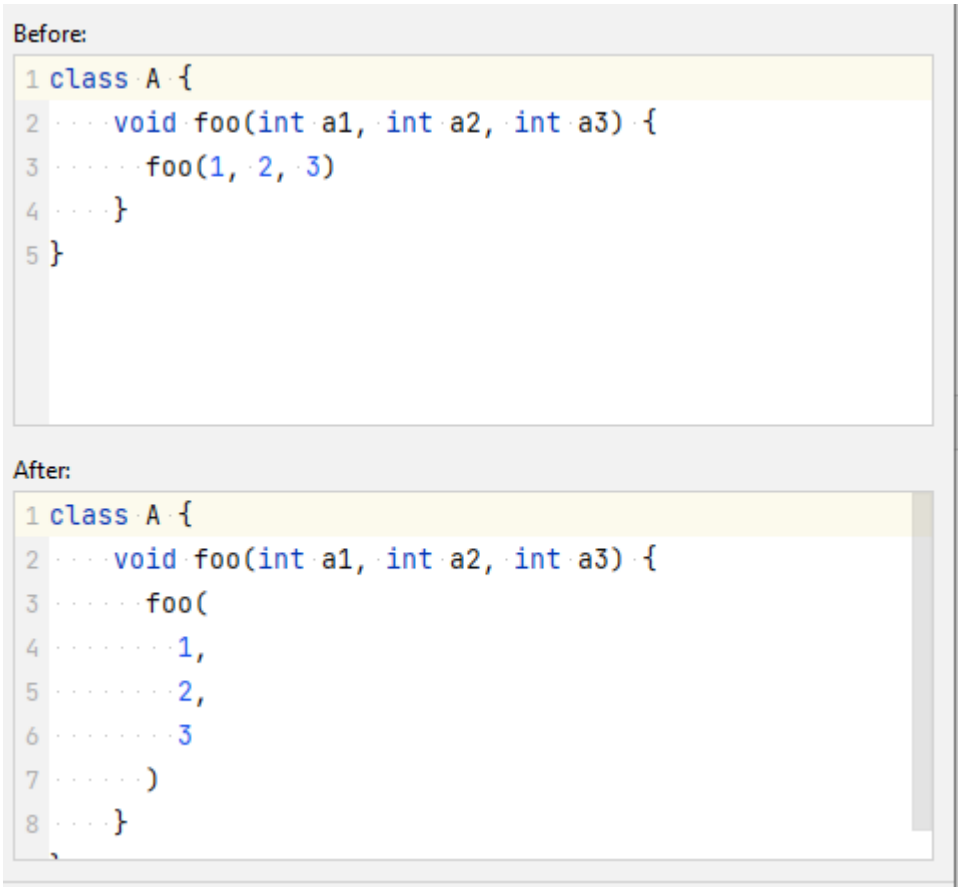

Рис. 7: Пример намерения “Put arguments on separate lines”

код в результате преобразования. В итоге каждая пара вариантов кода характеризуется 27 значениями метрик. При обучении модель не будет знать, каким способом был получен тот или иной вариант кода, в результате чего она должна будет научиться разделять преобразования, основываясь лишь на метриках. Большое количество метрик не должно ввести модель в заблуждение, так как тестируемые нами модели достаточно сложны и способны научиться распознавать несколько групп хороших преобразований.

Группа намерений	**Примеры намерений из группы**
Намерения для строк	‘Replace String.format() with concatenation’, ‘Replace concatenation with formatted output’
Намерения для исключений	‘Surround with try-with-resources block’, ‘Detail exceptions’
Намерения для условий	‘Merge nested ifs’, ‘Merge sequential ifs’
Намерения, меняющие длину строки	‘Put arguments on one line’, ‘Put arguments on separate lines’
Арифметические намерения	‘Compute constant value for subexpression’, ‘Inline increment/decrement’

Таблица 1: Примеры поддерживаемых намерений

Название метрики	Описание метрики
AverageLengthOfParameterNames	Средняя длина имени параметра. Используется для векторизации намерений с переносом аргументов на новую строку, уменьшением ширины кода.
IndentationsNumber	Количество отступов во фрагменте. Используется для векторизации намерений с переносом аргументов на новую строку, изменением структуры кода.
MaxLineLengthInsideExpression	Длина самой длинной строки, которую затрагивает выражение. Используется для векторизации намерений с переносом аргументов на новую строку.
NestingDepthMetric	Глубина вложенности выражения. Используется для векторизации намерений, изменяющих структуру вложенности.
NumberOfEmptyLinesMetric	Число пустых строк во фрагменте. Влияет на визуальное восприятие.
NumberOfLineBreaks-InsideExpression	Число переносов строки внутри выражения. Распознает намерения с переносом аргументов на новую строку.
NumberOfLines	Число строк в файле. Измеряет преобразования, уменьшающие или увеличивающие число строк.
NumberOfParametersMetric	Число параметров в выражении. Используется для векторизации намерений с переносом аргументов на новую строку.
NumberOfPlusesMetric	Число знаков “плюс” во фрагменте. Используется для векторизации намерений, связанных с конкатенацией.

Таблица 2: Описание используемых метрик

2.4. Метрики качества

Так как данная работа стремится улучшить пользовательский опыт при взаимодействии со средой разработки, мы стремимся предлагать пользователю только хорошие варианты преобразований. Другими словами, наша модель должна предлагать пользователю преобразование

только если в нем уверена, а все спорные варианты она должна отсекать. Таким образом, ключевой характеристикой модели в данной задаче является ее точность (*precision*). В общем случае точность вычисляется по следующей формуле:

$$precision = \frac{TP}{TP + FP}$$

В нашем случае TP (*True Positives*) — число верно предложенных пользователю вариантов (тех, в которых преобразованный вариант лучше изначального), а $TP + FP$ (*True Positives + False Positives*) — общее число предложенных пользователю вариантов. Таким образом, большая точность будет говорить нам, что пользователю предлагаются хорошие варианты.

Помимо точности, влияние имеет полнота (recall). Она, однако же, не так важна, и мы будем использовать ее только для оценки среднего гармонического. В общем случае она вычисляется по формуле:

$$recall = \frac{TP}{TP + FN}$$

В нашем случае TP (*True Positives*) — число верно предложенных пользователю вариантов, а $TP + FN$ (*True Positives + False Negatives*) — число вариантов, которые мы должны были предложить пользователю.

Для одновременного контроля точности и полноты предлагается использовать F-меру, или их среднее гармоническое, которое вычисляется по формуле:

$$F\text{-}measure = 2 * \frac{precision * recall}{precision + recall}$$

2.5. Модели

Был проведен ряд экспериментов с моделями машинного обучения с помощью библиотеки Weka [31]. Для тестирования были отобраны модели, являющиеся de facto стандартами при решении задач классификации на табличных данных, в том числе те, что упоминались в обзоре. Подсчитывались описанные выше метрики при оценке внутри проек-

та (обучающая выборка и тестирующая выборка получены из проекта guava), а также при оценке cross-project (обучающая выборка получена из проекта guava, а тестирующая из другого проекта). В итоге была выбрана модель RandomForest, как показавшая себя лучше всего на cross-project оценке.

	Внутри проекта		Cross-project dbeaver		Cross-project presto	
Модель	Precision	F-мера	Precision	F-мера	Precision	F-мера
Random Forest	**0.87**	0.86	**0.86**	**0.80**	0.67	0.69
J48	**0.87**	**0.87**	0.80	0.79	**0.70**	0.70
AdaBoostM1	0.80	0.81	0.74	0.74	0.64	0.66
Naive Bayes	0.74	0.75	0.78	0.78	**0.70**	**0.71**

Таблица 3: Результаты тестирования моделей

3. Плагин к IntelliJ IDEA

3.1. Особенности реализации плагина

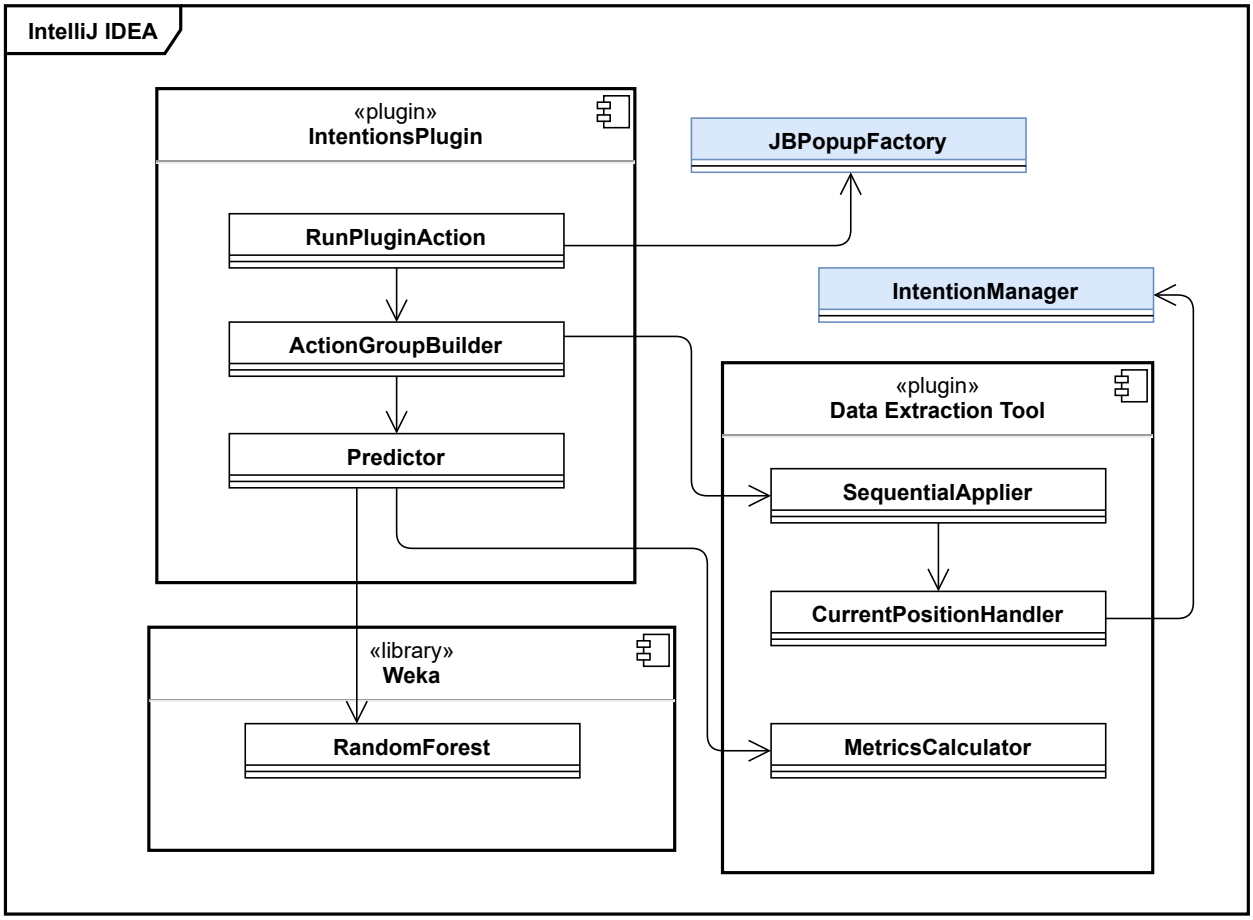

Рис. 8: Диаграмма компонентов плагина. Синим выделены использованные классы IntelliJ Platform.

Так как конечному плагину необходимо производить часть из тех действий, что делал инструмент для сбора данных (подсчет метрик, построение графа намерений), оба эти плагина взаимосвязаны (Рис. 8). Работа плагина выглядит следующим образом (Рис. 9). За взаимодействие с пользователем отвечает класс *RunPluginAction*. Он вызывается при нажатии необходимой комбинации клавиш. Затем этот класс запускает класс *ActionGroupBuilder*, который запускает класс *SequentialApplier* для построения графа намерений. Варианты кода из графа намерений отправляются в класс *Predictor*, в котором от них считаются метрики с помощью класса *MetricsCalculator* и производится предсказание с по-

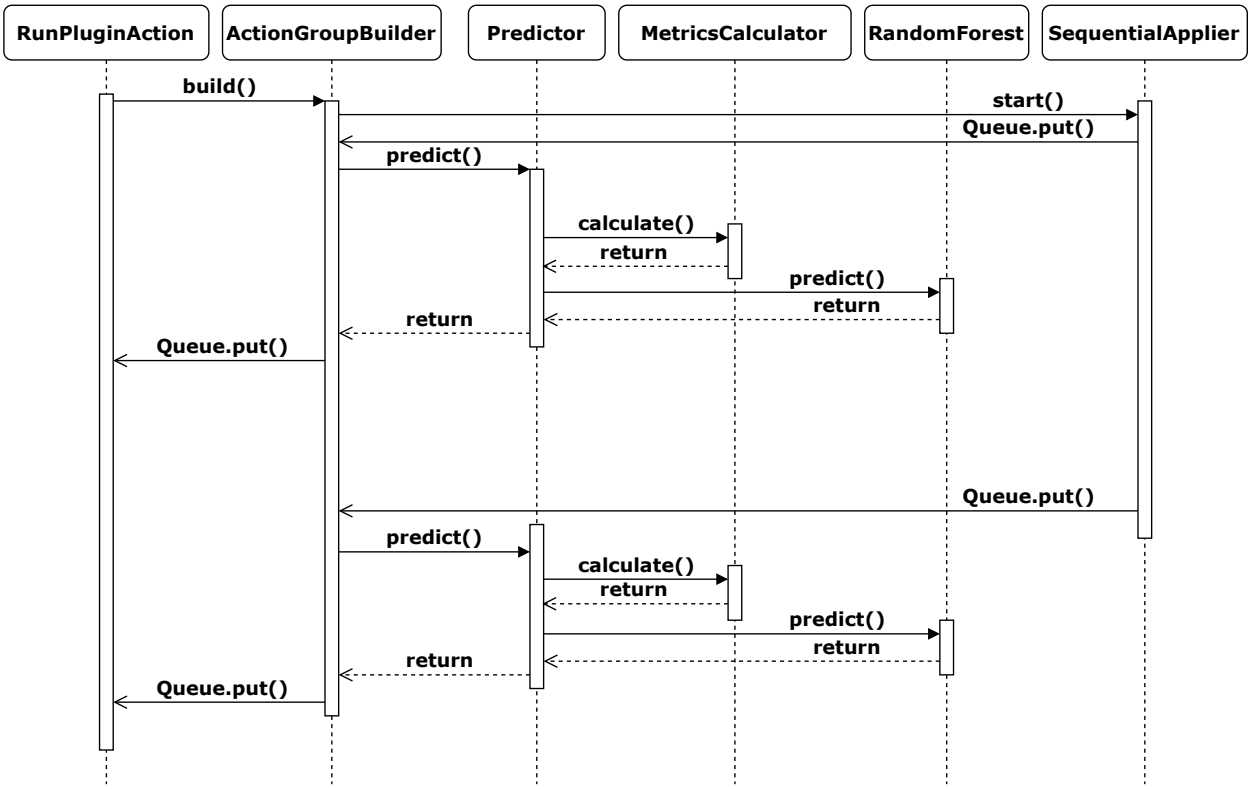

Рис. 9: Алгоритм работы плагина.

мощью модели. Результаты возвращаются в класс *RunPluginAction* и происходит демонстрация списка пользователю.

Изначально плагин планировалось запускать в синхронном режиме, описанном выше, выдавая сразу весь список намерений, которые мы хотим предложить пользователю. Однако, в процессе работы оказалось, что такое решение работает слишком долго, часто более нескольких секунд, в результате чего пользователю кажется, что приложение зависло. Было выяснено, что такое большое время работы обусловлено большим количеством проверок возможности применения намерения. Избежать таких проверок или каким либо образом их оптимизировать оказалось невозможно, так как это потребовало бы переписывания каждого конкретного намерения, что не представляется возможным. Для решения этой проблемы было решено сделать работу плагина асинхронной, а предлагаемый пользователю список динамически обновляемым, чтобы пользователь мог видеть процесс работы плагина и принимать решения о дальнейших действиях еще во время его работы.

Работа плагина изменилась следующим образом (Рис. 9). Класс

RunPluginAction запускает класс *ActionGroupBuilder*, а сам встает в режим ожидания объектов из блокирующей очереди. Класс *ActionGroupBuilder* сначала запускает класс *SequentialApplier*, который строит граф намерений, а в процессе построения каждый новый полученный вариант кода асинхронно возвращает через блокирующую очередь. Эти варианты отправляются в класс *Predictor*, в котором от них считаются метрики и производится предсказание с помощью модели. Результаты асинхронно возвращаются в класс *RunPluginAction* и происходит обновление показываемого пользователю списка. В результате данного изменения первые варианты демонстрируются пользователю уже спустя 150 миллисекунд. Общее время работы на сложных примерах увеличилось с 1.5 секунд до 3 из-за накладных расходов на асинхронность и невозможности слишком часто производить обновление пользовательского интерфейса, однако теперь пользователю не кажется, что приложение зависло.

3.2. Интеграция с инструментом предпросмотра

В IntelliJ IDEA реализован механизм предпросмотра намерений, который позволяет пользователю увидеть, к чему приведет применение намерения без изменения самого кода. Для интеграции с этим механизмом, необходимо передать в класс *IntentionPreviewPopupUpdateProcessor* объект, реализующий интерфейс *IntentionAction*. Однако же это невозможно сделать, используя объекты, наследуемые от класса *AnAction*, которые находятся в демонстрируемом пользователю списке. Для решения этой проблемы был создан класс *ActionWithIntention*, который наследуется от абстрактного класса *AnAction* и содержит в себе поле с переменной типа *IntentionAction*. Использование этого класса позволяет нам извлекать объект типа *IntentionAction* и использовать механизм предпросмотра намерений (Рис. 10).

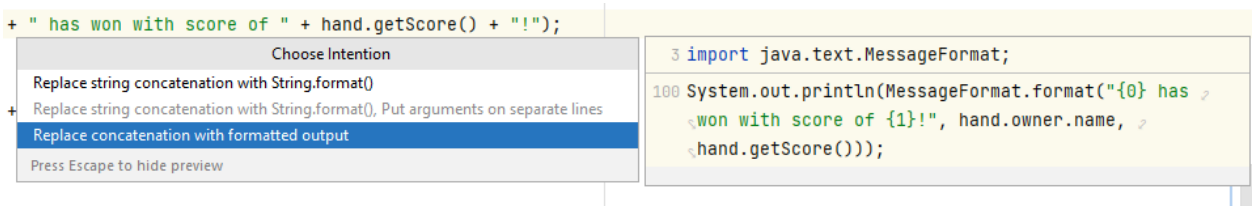

Рис. 10: Использование механизма предпросмотра в плагине.

4. Апробация

4.1. Методология

В апробации плагина поучаствовали пять опытных программистов (один программист с опытом 2 года, три программиста с опытом 3 года и один с опытом 4 года). Им предлагалось поиспользовать плагин в своей работе в течение недели, а затем ответить на несколько вопросов о своих впечатлениях (Таблица 4). Вопросы касались их мнения о плагине, скорости его работы, простоте использования. Также в опросе присутствовал открытый вопрос о сценариях, в которых плагин показал себя хорошо.

Вопрос	Результаты
Насколько легко вам было использовать плагин?	1 x Очень легко, 4 x Скорее легко
Как бы вы оценили качество советов, предлагаемых плагином?	1 x Очень полезны, 3 x Скорее полезны, 1 x Затрудняюсь ответить
Как часто вы принимали рекомендации плагина и были удовлетворены результатом?	4 x Часто, 1 x Редко
Расскажите, в каких ситуациях плагин показал себя хорошо, а в каких плохо? (необязательный открытый вопрос)	3 ответа
Замечали ли вы проблемы с производительностью IDE во время работы плагина?	3 x Нет, 1 x Скорее нет, 1 x Затрудняюсь ответить
Будете ли вы использовать плагин в дальнейшем?	3 x Скорее всего буду, 1 x Скорее всего не буду, 1 x Затрудняюсь ответить
Советы, впечатления, пожелания, предложения (необязательный открытый вопрос)	4 ответа

Таблица 4: Результаты опроса.

4.2. Отзывы пользователей

В целом пользователи оценили плагин положительно. Все участники сообщили о том, что у них не возникло проблем с использованием плагина. 80% положительно оценили качество советов, предлагаемых им. Такой же процент участников сообщил, что они часто принимали рекомендации плагина и были ими довольны. 60% участников сообщили, что они скорее всего будут использовать плагин в дальнейшем. Стоит отметить, что один из участников, оценивших плагин отрицательно, пытался использовать его в коде на языке Kotlin. Таким образом, гипотеза о том, что плагин сработает и для этого языка, не подтвердилась.

В открытых вопросах участники отмечали хорошее качество работы плагина на намерениях, связанных со строками. Также пользователи оценили возможность использования инструмента предпросмотра. Так, один из участников заявил:

Идея хорошая, мне нравится, что плагин быстро правит

код, делает его читабельнее. <...> Здорово, что превью работает, это удобно, можно посмотреть результат и выбрать лучшее предложение.

Среди недостатков плагина пользователи отмечали большое количество ситуаций, в которых плагин не находил возможностей для рефакторинга, а также некоторые проблемы с пользовательским дизайном:

<...> Надпись о том, что рефакторинги не были найдены, мелькает на секунду в углу и пропадает. Следить за прогрессом в углу экрана не очень удобно.

Заключение

В ходе данной работы было разработано расширение для среды разработки IntelliJ IDEA, предлагающее программисту наиболее подходящие изменения кода в выбранной позиции. Ключевым нововведением данной работы является использование для этого методов машинного обучения, а также возможность применять сразу цепочки намерений. Это позволяет программисту работать быстрее и эффективнее.

В процессе работы были достигнуты следующие результаты:

- Разработан инструмент для извлечения вариантов кода из кодовой базы. С его помощью удалось обработать несколько проектов и подготовить данные для разметки.
- Проведены эксперименты с несколькими моделями и выбрана наилучшая. Полученная модель была интегрирована в решение.
- Разработан плагин, предлагающий пользователю возможные варианты преобразований. Реализована интеграция с механизмом предпросмотра.
- Проведена апробация на пользователях. Получены в целом положительные отзывы, по результатам опроса локализованы некоторые недостатки плагина, которые планируется исправить в будущем.

Код проекта доступен по ссылкам:

- Инструмент для извлечения вариантов кода — `https://github.com/SacredArrow/idea-intentions-plugin`
- Плагин — `https://github.com/SacredArrow/usableIntentionsPlugin`

Список литературы

[1] Stack Overflow. Developer Survey Results 2019. — https://insights.stackoverflow.com/survey/2019. — 2019. — [Online; accessed November 24, 2020].

[2] Pierre Carbonnelle. PYPL index. — https://pypl.github.io/IDE.html. — 2020. — [Online; accessed November 24, 2020].

[3] Murphy G. C., Kersten M., Findlater L. How are Java software developers using the Elipse IDE? // IEEE Software. — 2006. — Vol. 23, no. 4. — P. 76–83.

[4] Amann S., Proksch S., Nadi S., Mezini M. A Study of Visual Studio Usage in Practice // 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER). — Vol. 1. — 2016. — P. 124–134.

[5] JetBrains. IntelliJ IDEA. — https://www.jetbrains.com/ru-ru/idea/. — 2020. — [Online; accessed November 24, 2020].

[6] Fowler Martin. Refactoring: Improving the Design of Existing Code. — Boston, MA, USA : Addison-Wesley, 1999. — ISBN: 0-201-48567-2.

[7] Haas Roman, Hummel Benjamin. Deriving Extract Method Refactoring Suggestions for Long Methods // Software Quality. The Future of Systems- and Software Development / Ed. by Dietmar Winkler, Stefan Biffl, Johannes Bergsmann. — Cham : Springer International Publishing, 2016. — P. 144–155.

[8] Silva Danilo, Terra Ricardo, Valente Marco. JExtract: An Eclipse Plug-in for Recommending Automated Extract Method Refactorings. — 2015. — 06.

[9] Palomba Fabio, Panichella Annibale, De Lucia Andrea et al. A textual-based technique for Smell Detection // 2016 IEEE 24th International Conference on Program Comprehension (ICPC). — 2016. — P. 1–10.

[10] Yue Ruru, Gao Zhe, Meng Na et al. Automatic Clone Recommendation for Refactoring Based on the Present and the Past. — 2018. — 1807.11184.

[11] Schapire Robert E. A Brief Introduction to Boosting // Proceedings of the 16th International Joint Conference on Artificial Intelligence - Volume 2. — IJCAI'99. — San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 1999. — P. 1401–1406.

[12] Fontana F. A., Zanoni M., Marino A., Mäntylä M. V. Code Smell Detection: Towards a Machine Learning-Based Approach // 2013 IEEE International Conference on Software Maintenance. — 2013. — P. 396–399.

[13] Tempero E., Anslow C., Dietrich J. et al. The Qualitas Corpus: A Curated Collection of Java Code for Empirical Studies // 2010 Asia Pacific Software Engineering Conference. — 2010. — P. 336–345.

[14] Liu Hui, Xu Zhifeng, Zou Yanzhen. Deep learning based feature envy detection. — 2018. — 09. — P. 385–396.

[15] Mikolov Tomas, Chen Kai, Corrado Greg, Dean Jeffrey. Efficient Estimation of Word Representations in Vector Space. — 2013. — 1301.3781.

[16] Whale G. Identification of Program Similarity in Large Populations // The Computer Journal. — 1990. — 01. — Vol. 33, no. 2. — P. 140–146. — https://academic.oup.com/comjnl/article-pdf/33/2/140/1092084/330140.pdf.

[17] Luo Lannan, Ming Jiang, Wu Dinghao et al. Semantics-Based Obfuscation-Resilient Binary Code Similarity Comparison with Applications to Software and Algorithm Plagiarism Detection // IEEE Transactions on Software Engineering. — 2017. — Vol. 43, no. 12. — P. 1157–1177.

[18] Roy C. K., Cordy J. R. NICAD: Accurate Detection of Near-Miss Intentional Clones Using Flexible Pretty-Printing and Code Normalization // 2008 16th IEEE International Conference on Program Comprehension. — 2008. — P. 172–181.

[19] Joy M., Luck M. Plagiarism in programming assignments // IEEE Transactions on Education. — 1999. — Vol. 42, no. 2. — P. 129–133.

[20] Luo Lannan, Ming Jiang, Wu Dinghao et al. Semantics-Based Obfuscation-Resilient Binary Code Similarity Comparison with Applications to Software Plagiarism Detection // Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. — FSE 2014. — New York, NY, USA : Association for Computing Machinery, 2014. — P. 389–400. — Access mode: https://doi.org/10.1145/2635868.2635900.

[21] Sajnani H., Saini V., Svajlenko J. et al. SourcererCC: Scaling Code Clone Detection to Big-Code // 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE). — 2016. — P. 1157–1168.

[22] Baxter I.D., Yahin A., Moura L. et al. Clone detection using abstract syntax trees // Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272). — 1998. — P. 368–377.

[23] Jiang Lingxiao, Misherghi Ghassan, Su Zhendong, Glondu Stephane. DECKARD: scalable and accurate tree-based detection of code clones. — 2007. — 06. — P. 96–105.

[24] Milutin Anton. Software code metrics. — https://www.viva64.com/en/a/0045/. — 2009. — [Online; accessed December 8, 2020].

[25] Varela Alberto, Perez-Gonzalez Hector, Martinez Francisco, Soubervielle-Montalvo Carlos. Source Code Metrics: A Systematic Mapping Study // Journal of Systems and Software. — 2017. — 04. — Vol. 128.

[26] McCabe T. J. A Complexity Measure // IEEE Transactions on Software Engineering. — 1976. — Vol. SE-2, no. 4. — P. 308–320.

[27] Cotroneo Domenico, Natella Roberto, Pietrantuono Roberto. Predicting aging-related bugs using software complexity metrics // Performance Evaluation. — 2013. — Vol. 70, no. 3. — P. 163 – 178. — Special Issue on Software Aging and Rejuvenation. Access mode: http://www.sciencedirect.com/science/article/pii/S0166531612000946.

[28] Muthukumaran K., Rallapalli Akhila, Murthy N. L. Bhanu. Impact of Feature Selection Techniques on Bug Prediction Models // Proceedings of the 8th India Software Engineering Conference. — ISEC '15. — New York, NY, USA : Association for Computing Machinery, 2015. — P. 120–129. — Access mode: https://doi.org/10.1145/2723742.2723754.

[29] Arcelli Fontana Francesca, Mäntylä Mika, Zanoni Marco, Marino Alessandro. Comparing and experimenting machine learning techniques for code smell detection // Empirical Software Engineering. — 2015. — 06. — Vol. 21.

[30] Breiman Leo. Random Forests // Mach. Learn. — 2001. — Oct. — Vol. 45, no. 1. — P. 5–32. — Access mode: https://doi.org/10.1023/A:1010933404324.

[31] Appendix B - The WEKA workbench // Data Mining (Fourth Edition) / Ed. by Ian H. Witten, Eibe Frank, Mark A. Hall, Christopher J. Pal. — Morgan Kaufmann, 2017. — P. 553–571. — Access mode: https://www.sciencedirect.com/science/article/pii/B9780128042915000246.

[32] Di Nucci D., Palomba F., Tamburri D. A. et al. Detecting code smells using machine learning techniques: Are we there yet? // 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER). — 2018. — P. 612–621.

[33] heartex.ai. Label Studio. — https://labelstud.io/. — 2019. — [Online; accessed December 11, 2020].

[34] Brandl Georg. Pygments. — https://pygments.org/. — 2006. — [Online; accessed December 11, 2020].

[35] Google. guava. — https://github.com/google/guava. — 2020.

[36] dbeaver. dbeaver. — https://github.com/dbeaver/dbeaver. — 2020.

[37] prestodb. presto. — https://github.com/prestodb/presto. — 2020.