Федеральное государственное бюджетное
образовательное учреждение высшего образования
«Санкт-Петербургский национальный исследовательский
Академический университет Российской академии наук»
Центр высшего образования

Кафедра математических и информационных технологий

Байдин Дмитрий Алексеевич

Автоматическое извлечение шаблонов из кода

Магистерская диссертация

Допущена к защите.
Зав. кафедрой:
д. ф.-м. н., профессор Омельченко А. В.

Научный руководитель:
Шеин Р. Е.

Рецензент:
Селютин Д. С.

Санкт-Петербург
2017

SAINT-PETERSBURG ACADEMIC UNIVERSITY
Higher education centre

Department of Mathematics and Information Technology

Dmitriy Baydin

Automative template extraction

Graduation Thesis

Admitted for defence.
Head of the chair:
professor Alexander Omelchenko

Scientific supervisor:
Roman Shein

Reviewer:
Dmitriy Selutin

Saint-Petersburg
2017

Оглавление

Введение

Во время написания кода программисты часто используют одинаковые или почти одинаковые структуры. Например такие как итерация по массиву или проверка выражение на равенство null.

Отсутствие необходимости по много раз печатать одинаковые структуры способна сильно ускорить производительность программиста. Именно для этого в IDE IDEA был введен такой механизм как Live Templates.

Этот механизм позволяет нажатием нескольких клавиш вывести завязанную на эту комбинацию структуру. Программисту, возможно, придется заполнить несколько пропусков, содержание которых специфично для каждой конкретного случая использования.

Сама IDEA уже содержит достаточно обширную коллекцию таких шаблонов. Такие коллекции существуют для языка Java, Scala, Kotlin, а также для некоторых других популярных языков.

Но существует множество других языков и библиотек, для которых существуют какие-то свои часто используемые структуры, но они не представлены в коллекции шаблонов IDEA по тем или иным причинам. А также не стоит забывать про шаблоны свойственные определенному стилю программирования или даже отдельному программисту.

В IDEA существует механизм добавления шаблонов в коллекцию, но не всегда существует возможность и желание самостоятельно добавлять шаблоны. Также отсутствует возможность проверить насколько часто встречается данная структура и насколько целесообразно ее использовать.

Именно поэтому была бы полезна возможность получения часто используемых структур из кода. При этом важным требованием является независимость решения от языка.

Стоит дать определение тому, что мы будем понимать под шаблоном в дальнейшем. Шаблон это часто встречающаяся структура, которая имеет некую общую для всех реализаций часть, а также может иметь один или несколько пропусков.

Пропуск это часть структуры, которая должна быть заполнена программистом. Пропуски могут находиться в различных частях структуры. Они могут находится на месте имен переменных, в значениях литералов, типах переменных, а также на месте целых выражений. Приведем несколько примеров шаблонов:

```
for (# # : #) {
        #
}
```

Данный пример является шаблоном на языке Java для конструкции for-each. Эта структура позволяет итерироваться по коллекции. Пропуски в данной структуре обо-

значены символом “#”. Первый пропуск соответствует названию типа элементов коллекции. Второй пропуск соответствует названию переменной, в которой будет храниться текущий элемент. Третий пропуск соответствует названию коллекции, по которой будет происходить итерирование. А четвертый пропуск соответствует телу цикла.

```
# match {
  case Some(#) =>
  case None =>
}
```

Данный пример является шаблонов на языке Scala для работы с Option. Эта конструкция позволяет проверить содержит ли Option значение или нет и в зависимости от результата продолжить вычисления. Первый пропуск соответствует перменной типа Option, второй соответствует имени значения в случае если перменная не пустая.

Цель и задачи

На момент начала работы уже присутствовало решение, которое позволяло находить часто встречающиеся структуры в Java коде. Но это решение обладало рядом ключевых недостатков в частности:

1. Низкая скорость работы алгоритма.
2. Высокая сложность добавления новых параметров для фильтрации.
3. Низкое качество выдаваемых шаблонов.
4. Поддержка только языка Java.

Целью данной работы было избавление от этих недостатков и публикация плагина в репозитории JetBrains. Основные задачи этого проекта:

1. Изучение предыдущего решения.
2. Анализ предметной области. Поиск алгоритмов для поиска часто встречающихся структур.
3. Реализация или улучшение алгоритма поиска часто встречающихся структур.
4. Добавление новых параметров для фильтрации шаблонов. Добавление возможности оптимизации параметров для фильтрации.
5. Добавление поддержки для таких языков как Kotlin и Scala.

1. Анализ области

В ходе анализа области не было найдено прямых аналогов. То есть решений позволяющих искать часто встречающиеся структуры в коде независимо от используемого языка.

Поэтому было рассмотрено несколько ближайших областей. Одной из таких областей стал поиск дубликатов в коде. Различается несколько уровней похожести:

1. Код идентичен с точностью до комментариев и не влияющего на синтаксис форматирования.

2. Код идентичен с точностью до названий переменных, типов а также значений литералов.

3. К возможным различиям на предыдущем уровне добавляется возможность добавления и удаления некоторых выражений прямо внутри структуры.

4. Код должен выполнять одинаковые вычисления, но может быть по разному синтаксически реализован.

Как можно заметить, в рамках решаемой задачи нам ближе всего сходство типа 3, так как сходство этого типа больше всего подходит к данному нами определению шаблону.

Для поиска дубликатов существует множество различных подходов. Самым популярным и относительно простым подходом является подход основанный на хэшировании. [2]

Решения основанные на данном подходе используют различные техники хэширования строк. Могут применятся техники, когда хэшируется весь код, либо весь код за исключением комментариев и лишних знаков форматирования, либо код за исключением имен переменных. Это позволяет искать дубликаты принадлежащие к разным уровням похожести. Основным плюсом методов, основанных на данном подходе, является скорость их работы. Так как время работы хэширования линейно от длины строчек, а время поиска в хэш таблице армотизированная единица. Основным же недостатком метод, основанных на данном подходе, является низкая гибкость этого подхода и сложность поиска дубликатов третьего и четвертого уровня похожести.

Как раз решения основанные на хэшировании используются в IDE для поиска дубликатов. В рамках работы были проанализированы способы определения дубликатов используемые в таких IDE как IDEA, Eclipse, Visual Studio.

Существует другой класс решений основанных на AST деревьях. Одно из таких решений было использовано в первоначальной версии работы, которое было основано на деревьях похожести.[3]

Также существует целое множество методов, извлечения часто встречающихся структур в графах, частным случаем которых является поиск часто встречающихся поддеревьев. Эти методы можно разделить на два класса, по тому что считается часто встречающимся поддеревом. Основанные на количестве раз, которое встретилось поддерево во всех деревьях леса и основанные на количестве деревьев в которых содержится поддерево.[1]

Также методы можно разделить по признаку того, какая часть поддерева должна быть часто встречающейся.[5]

1. Дерево должно полностью дублироваться.

2. Индуцированное дерево. Индуцированным деревом является такое дерево, множество вершин которого является подмножеством вершин родительского дерева. А множество ребер является подмножество ребер родительского дерево, которые инцидентны множеству вершин. Другими словами, индуцированным деревом называется такое дерево, из которого были удалены некоторые вершины вместе со всеми их потомками, а ребра сохранились между всеми оставшимися вершинами.

3. Вложенное дерево. Вложенным деревом называется дерево, из которого было удалено некоторое множество вершин, а ребра могут соединять, те вершины, которые были соединены в исходном дереве, а также потомков удаленной вершины и родителей удаленной вершины.

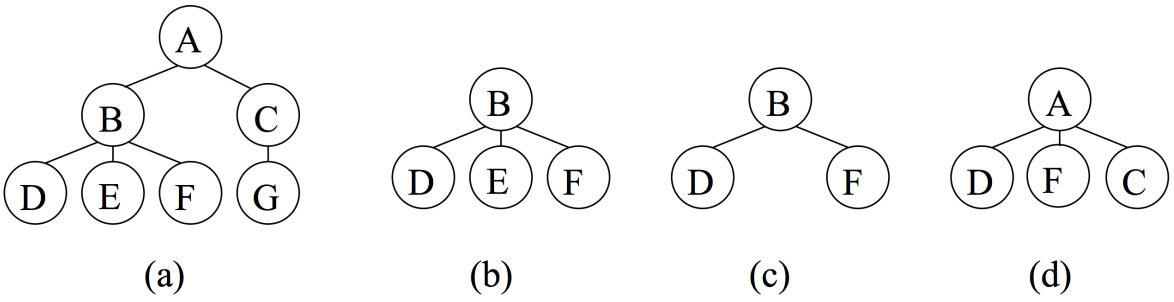


Рис. 1: Примеры поддеревьев

Также методы различаются по признаку того, важен или нет порядок следования детей вершины.

В качестве метода, который подходит для решения поставленной задачи, а именно поиска часто встречающихся поддеревьев, нужен метод основанный на количестве раз, которое встретилось дерево во всех поддеревьях. Так как структура может быть часто встречающейся даже если все ее вхождения находятся в одном файле.

Также наиболее подходящим для нами данного определения шаблона является метод поиска часто встречающихся индуцированных поддеревьев. Так как такой тип поиска сразу дает возможность отличия в различных вхождениях структуры в код отдельных имен переменных или типов, а также возможность вставки дополнительных выражений прямо посреди структуры.

Так как в AST деревьях важен порядок следования выражений, необходим метод, для которого важен порядок следования детей.

К счастью, всем этим требованиям удовлетворяет алгоритм IMB3.

2. Основная часть

2.1. Предыдущее решение

2.1.1. Описание алгоритма

Данное решение основано на такой структуре как дерево похожести. Эта структура позволяет искать часто встречающиеся структуры. Затем эти структуры подвергаются пост-обработке. Пост-обработка включает в себя удаление лишних пробелов, объединение пропусков. Затем выбираются наиболее общие структуры, другими словами структуры не являющиеся подчастью других найденных структур. На последнем этапе происходит фильтрация, то есть отбор структур по каким-то заданным сверху параметрам.

```
Def solution(astNodes)
        templates = findTemplates(astNodes)
        processedTemplates = processTemplates(templates)
        generalTemplates = removeSubTemplates(processedTemplates)
        filteredTemplates = filterTemplates(generalTemplates)
```

Мы строим отображение из типа вершины в список вершин этого типа. Для этого нам нужно один раз сделать проход в глубину по всему лесу.

Затем для каждого типа вершин мы строим дерево похожести (similarity tree).

```
Def findTemplates(astNodes)
        typeToNodes = build
        for (type in typeToNodes)
                simTree = new SimTree
                for (node in nodes)
                        simTree.add(node)
                for (node in nodes)
                        template = simNode.tryExtract(node)
                        if (template.isDefined)
                                yield template
```

С помощью каждой вершины леса AST деревьев мы пытаемся извлечь шаблон из соответствующего дерево похожести.

Дерево похожести предназначено для того, чтобы хранить все возможные варианты поддеревьев.В деревьях похожести существует два вида вершин. Это листовые и внутренние вершины.

Каждая внутренняя вершина дерева похожести соответствует определенному типу из исходного леса AST деревьев с фиксированным количеством детей.

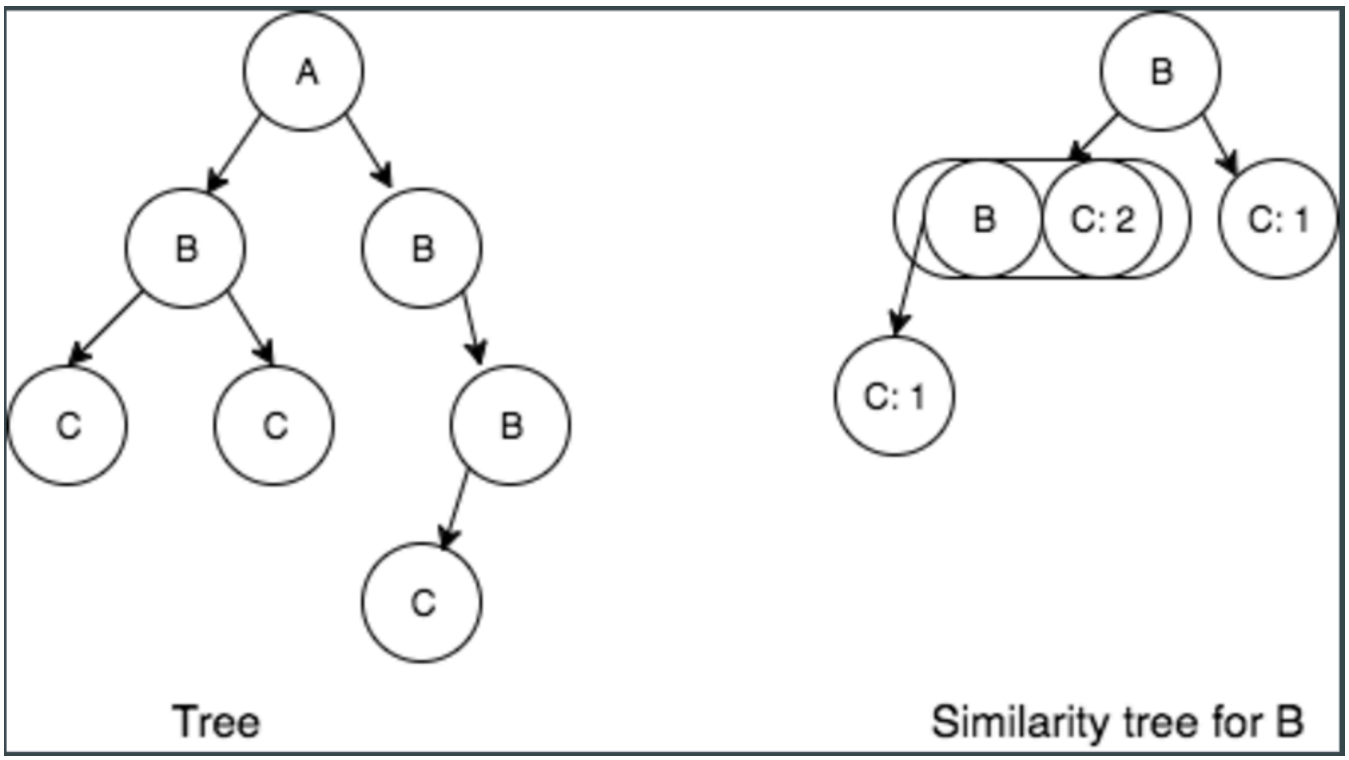


Рис. 2: Дерево похожести

Внутренние вершины деревьев похожести ссылаются на другие вершины дерева похожести. Причем если вершина соответствует вершинам с количеством детей равным k, тогда в ней будет хранится список из k отображений из типа ребенка в вершину дерева похожести. Каждое такое отображение соответствует разнообразию видов детей, находящихся на этом месте у вершины данного типа.

Каждая листовая вершина соответствует листовым вершинам определенного типа из исходного леса AST деревьев. При этом листовые вершины одного типа могут и, как правило, содержат различные варианты текста. Таким образом в листовой вершине дерева похожести хранится отоброжение из текста в количество вершин, содержащих данный текст.

Теперь более подробно рассмотрим процесс построения деревьев похожести. На первом этапе мы строим отображение из типа вершины и количества детей в список вершин такого типа и с таким количеством детей. Это делается с помощью прохода в глубину и хэш таблицы.

Каждый список вершин определенного типа соответствует корневой вершине дерева похожести.

Для каждого элемента из данного списка мы выполняем рекурсивную функцию add.

```
Def add(astNode)
        if (astNode.isLeaf)
```

```
            if (!textToCountMap.contains(astNode.text))
                    textToCountMap.put(astNode.text, 0)
            textToCountMap.put(astNode.text,
               textToCountMap.get(astNode.text) + 1)
    else
            for (child : astNode.children)
                    nodeTypeToSimTree =
                       childrens[child.index]
                    if
                       (!nodeTypeToSimTree.contains(astNode.type))
                            nodeTypeToSimTree.put(astNode.type,
                               new SimTree)
                    nodeTypeToSimTree.get(astNode.type).add(child)
```

Вначале мы проверяем, мы определяем тип вершины дерева похожести, является ли он внутренней или листовой вершиной.

Если вершина является внутренней, то мы проходим по всем детям данной вершины. И для каждого ребенка проверяем в соответствующем отображении присутствует ли для данного типа ребенка соответствующая вершина дерева похожести. В случае если в отображении такого типа нет, создаем новую вершину дерева похожести. Далее запускаем рекурсивно от вершины дерева похожести соответствующей ребенку данной вершины.

Если вершина является листовой, то мы проверяем присутствует ли текст данной вершины в отображении текста к количеству раз. Если текст не присутствует, то мы добавляем в отображение этот этот текст с коэффициентом один, иначе просто инкрементируем коэффициент.

Таким образом при построении дерева похожести выполняется обход в глубину от каждой вершины из леса AST деревьев. А каждое дерево похожести содержит все варианты поддеревьев с корнем вершин определенного типа.

Существует возможная оптимизация, которая, впрочем, не влияет на итоговую асимптотику. Мы можем использовать построенные на предыдущих шагах деревья похожести. Для этого мы должны построить ориентированный граф, вершинами в котором будут типы вершин в исходном лесе AST деревьев. А ребрами будут соединены те типы, вершины которых соединены отношением родитель-ребенок в исходном лесе AST деревьев.

Таким образом если в исходном лесе AST деревьев вершина типа A является ребенком вершины типа B, тогда в построенном нами графе вершина соответствующая типу A будет соединена с вершиной соответствующей типу B.

Стоит заметить, что данный граф не обязан быть ациклическим или не иметь петель. Таким образом, мы не всегда можем построить топологическую сортировку.

Но мы можем каким либо образом минимизировать количество деревьев, которые мы посчитаем несколько раз. Такая задача может быть решена с помощью методов динамического программирования. Далее будем предполагать, что нам известна оптимальная последовательность построения деревьев похожести.

Также мы должны будем модифицировать функцию add. Теперь, перед тем как рекурсивно запускаться, мы должны проверять построено ли уже дерево похожести для данной вершины. При этом нам понадобится функция объединяющая несколько деревьев похожести.

На каждом шаге этой функции мы имеем две вершины дерева похожести одного типа.

```
Def merge(simNode1, simNode2)
        simTree = new SimTree
        if (simNode1.isLeaf)
                textToCount = simTree.textToCount
        textToCount1 = simNode1.textToCount
        textToCount2 = simNode2.textToCount
                for ((text, count) in textToCount1)
                        textToCount.add(text, count +
                           textToCount2.get(text))
                for ((text, count) in textToCount2)
                        if (!textToCount.contains(text))
                                textToCount.add(text, count)
        else
                typeToSimNode = simTree.typeToSimNode
                typeToSimNode1 = simTree1.typeToSimNode
                typeToSimNode2 = simTree2.typeToSimNode
                for ((type, node) : typeToSimNode1)
                        if (typeToNode2.contains(type))
                                typeToSimNode.add(type,
                                   merge(node,
                                   typeToNode2.get(type)))
                        else
                                typeToSimNode.add(type, node)
                for ((type, node) : typeToSimNode2)
                        if (!typeSimToNode1.contains(type))
                                typeToSimNode.add(type, node)
```

Для листовых вершин мы объединяем отображения из текста в количество раз, которое этот текст встречается. При присутствии текста в обоих отображениях, сум-

мируем коэффициенты, иначе берем коэффициент из того отображения, где данный текст присутствует.

Для внутренних вершин объединяем отображения из типа вершины в в вершину дерева похожести. При присутствии типа в обоих отображениях, добавляем вершину которая является результатом объединения вершин из двух отображениях, иначе берем вершину из того отображения, где данный тип присутствует.

Далее следует этап извлечения шаблонов из деревьев похожести. Для начала стоит заметить, что мы не можем извлечь из каждой листовой вершины дерева похожести наиболее часто встречающийся вариант и на основе этого построить шаблон, так как можно легко придумать пример, при котором мы можем получим такую структуру, которая никогда не встречалась в исходном лесе AST деревьев. К тому же мы будем ограничены лишь одним шаблоном для каждого дерева похожести, что не всегда является хорошим результатом.

Если же мы разрешим доставать более чем один вариант из листовой вершины дерева похожести, например мы можем доставать все варианты, которые встречаются чаще чем какое то заданное значение, мы столкнемся с проблемой комбинаторного взрыва количества шаблонов получающихся с одного дерева похожести. Так как мы будем вынуждены комбинировать извлеченные варианты из одного листа со всеми возможными комбинациями полученными из других листовых вершин дерева похожести.

Таким образом, мы приходим к необходимости использовать существующие вершины AST деревьев, которые были использованы при построении дерева похожести.

```
Def tryExtract(simNode, astNode)
        if (astNode.isLeaf)
                occurrenceCount =
                   simNode.textToCount.get(astNode.text)
                if (occurrenceCount  minSupport)
                        return astNode.text
                else
                        return “ # ”
        else
                result = “”
                for (child in astNode.children)
                        typeToSimNode =
                           simNode.childrens[child.index]
                        result += tryExtract(
typeToSimNode.get(astNode.type), astNode.children[index])
                return result
```

Вначале мы проверяем является ли вершина AST дерева листовой или внутренней вершиной.

Если вершина является листовой. То мы проверяем, что текст текущей AST вершины имеет значение в отображении текста в количество раз, которое этот текст встретился, больше чем минимально заданное значение. Если текст встречается достаточно часто, то мы возвращаем этот текст в качестве результата. Иначе мы возвращаем “пропуск”.

Если же вершина является внутренней, то мы рекурсивно запускаем функцию для всех детей и конкатенируем результаты выполнения.

Таким образом мы получаем множество шаблоном. Многие из которых являются идентичными и частями других, более общих шаблонов. Для того чтобы получить только самые общие мы можем использовать обобщенное суффиксное дерево.

2.1.2. Анализ времени работы

Пусть f - внутренняя вершина, а $c(f)$ - количество потомков внутренней вершины.

Построение дерева похожести занимает время пропорциональное количеству вершин в деревьях из леса AST, участвующих в построении этого дерева похожести. Каждая внутренняя вершина является корнем какого либо дерева похожести. И для каждой внутренней вершины мы производим обход в глубину.

Таким образом построение множества деревьев похожести не оптимизированным алгоритмом занимает время $\mathcal{O}\left(\sum\limits_{f \in F} c(f)\right)$.

В случае же если мы будем использовать уже построенные деревья похожести, и при необходимости сливать. Процедура слияния занимает минимальное количество вершин из двух деревьев похожести. Минимальное время построение займет $\mathcal{O}\left(|V|\right)$. Где |V|-общее количество вершин в лесе AST деревьев.

Но можно придумать пример для которого время построения также останется равным $\mathcal{O}\left(\sum\limits_{f \in F} c(f)\right)$.

Извлечение шаблонов из деревьев похожести также занимает время равное $\mathcal{O}\left(\sum\limits_{f \in F} c(f)\right)$.

Так как мы производим обход в глубину для каждой внутренней вершины из леса AST деревьев.

2.2. Текущее решение

2.2.1. Описание алгоритма

Данное решение основано на алгоритме IMB3-Miner. Эта алгоритм позволяет искать часто встречающиеся структуры путем расширения на каждом шаге текущих

кандидатов. Затем эти структуры подвергаются пост-обработке. Пост-обработка включает в себя удаление лишних пробелов, объединение пропусков. Затем выбираются наиболее общие структуры, другими словами структуры не являющиеся подчастью других найденных структур. На последнем этапе происходит фильтрация, то есть отбор структур по каким-то заданным сверху параметрам.[4]

```
Def solution(astNodes)
        templates = findTemplates(astNodes)
        processedTemplates = processTemplates(templates)
        generalTemplates = removeSubTemplates(processedTemplates)
        filteredTemplates = filterTemplates(generalTemplates)
        return filteredTemplates
```

Подсчитываем количество вершин каждого типа и общее количество вершин. Затем мы определяем минимальное количество раз ($minSupport$), которые должен встретиться определенный шаблон, для того чтобы называться часто встречающимся. Этот параметр определяется по следующей формуле:

$$minSupport = \frac{nodeCount}{typeCount} minSupportCoefficient$$

Где $nodeCount$ - общее количество вершин, $typeCount$ - количество различных типов вершин, $minSupportCoefficient$ - настраиваемый внешний параметр, который имеет смысл - во сколько раз чаще должен встречаться шаблон, чем среднее число повторений одного типа вершины.

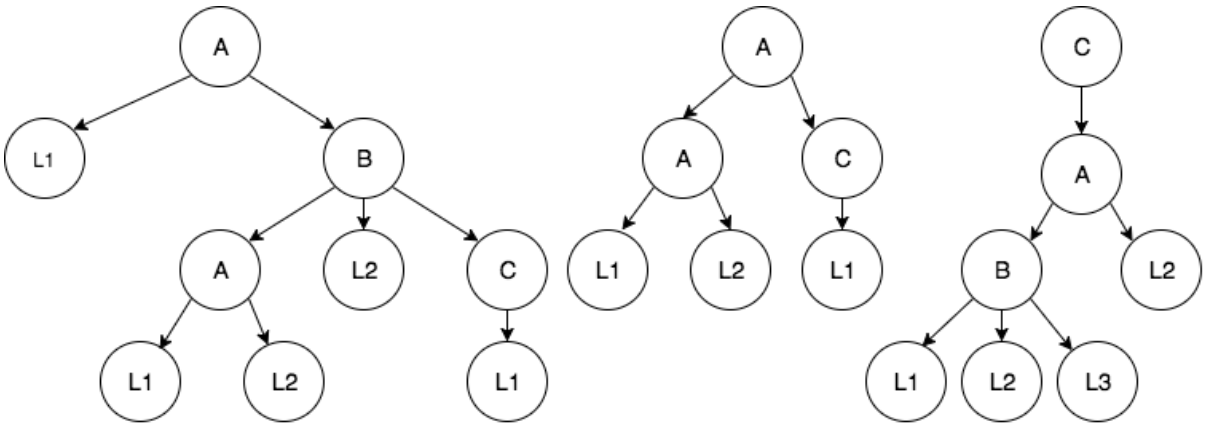
Такой способ задания $minSupport$ позволяет избежать зависимости от размера леса.

Строим список вершин, записанных при прямом обходе в глубине ($dictionary$). Списки полученные от обхода разных деревьев конкатенируем. Вместе с каждой вершиной храним ее глубину и позицию в списке самого правого наследника. При построении списка, заменяем вершины, типы которых встречаются недостаточно часто (реже чем $minSupport$) на особую метку заполнитель ($placeholder$), так как такие вершины не могут участвовать в каком либо шаблоне.

```
Def buildDictionary(astNodes)
        nodeTypeToCount = countTypeNode(astNodes)
        minSupport = calcMinSupport(nodeTypeToCount)
        dictionary = inOrderDFS(astNodes, minSupport,
           nodeTypeToCount)
        return (dictionary, minSupport)
```

Не рассматриваем вершины вместе с наследниками, чьи типы не учитываются при анализе. Это позволяет отсечь те области, в которых мы не хотим искать шаблоны.

Эта настройка специфична для каждого языка программирования. Для Java мы



Num	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
Name	A	L1	B	A	L1	L2	L2	C	L1	A	A	L1	L2	C	L1	C	A	B	L1	L2	L3	L2
Depth	0	1	1	2	3	3	2	2	3	0	1	2	2	1	2	0	1	2	3	3	3	2
Scope	8	-	8	5	-	-	-	8	-	14	12	-	-	14	-	21	21	20	-	-	-	-

Рис. 3: Пример списка вершин

не анализируем название пакета, секцию импортирования и комментарии. Так как название пакета и секция импортов в большинстве случаев генерируются автоматически, нет смысла выносить их в шаблоны. Эмпирическим путем было установлено, что комментарии также не содержат полезных шаблонов. Так как форма JavaDoc комментария генерируется по сигнатуре функции или класса, а для секции копирайтов есть специальные средства IDEA.

Примеры:

```
package edu.java.course;

import  edu.java.util.Utils;
/* Some commentary */
// Another commentary
```

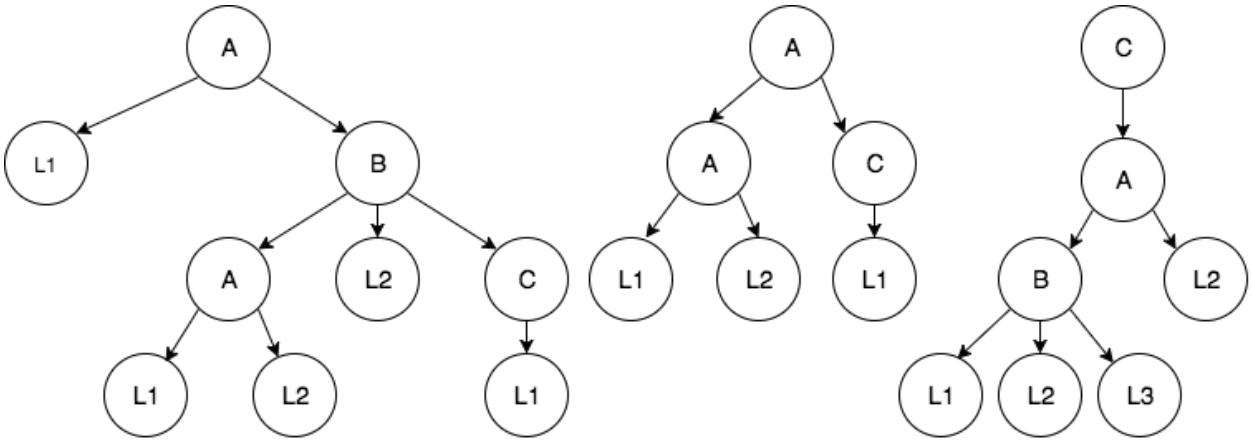
Мы отказались от идеи ограничения максимальной высоты вершины. Так как в абстрактных синтаксических деревьях вызов какого либо метода у функции является ребенком. Таким образом длинные цепочки вызовов будут иметь большую высоту и могли бы не попасть в кандидаты для шаблона, хотя должны.

Затем начинается стадия поиска шаблонов. Идея состоит в том, что мы на каждом шаге пытаемся расширить каждый часто встречающийся шаблон.[6]

```
Def findTemplates(astNodes)
        (dictionary, minSupport) = buildDictionary(astNodes)
```

```
verticalOccurrenceLists = buildVerticalLists(dictionary)
while(verticalOcurrenceLists.nonEmpty)
        occurrenceList = verticalOcurrenceLists.pop()
        (newLists, isTemplate) = extend(occurrenceList,
           minSupport)
        If (isTemplate)
                yield occurrenceList.template()
        verticalOcurrenceLists.add(newLists)
```

Для этого мы используем такую структуру данных как вертикальный список вхождений (vertical occurrence list). В данном списке хранятся все вхождения определенного шаблона в лес. Для однозначного определения вхождения необходимо хранить только позицию корня и самого правого потомка в dictionary. Каждому списку вхождений сопоставляется шаблон.



Num	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
Name	A	L1	B	A	L1	L2	L2	C	L1	A	A	L1	L2	C	L1	C	A	B	L1	L2	L3	L2
Depth	0	1	1	2	3	3	2	2	3	0	1	2	2	1	2	0	1	2	3	3	3	2
Scope	8	-	8	5	-	-	-	8	-	14	12	-	-	14	-	21	21	20	-	-	-	-

A : (0, 0), (3, 3), (9, 9), (10, 10), (16, 16)
A L1 L2 : (3, 5), (10, 12)
B # L2 # : (2, 8), (17, 21)

Рис. 4: Пример вертикального списка вхождений вершин

На первом шаге каждый шаблон имеет только одну не листовую вершину (мы предполагаем, что шаблон не может состоять только из одной листовой вершины, так как иначе шаблон будет слишком коротким). Таким образом на первом шаге мы имеем

по одному вертикальному списку вхождений для каждого типа вершин. А каждый список состоит из всех вершин данного типа.

Процедура расширения шаблона происходит следующим образом. Мы поочередно пытаемся добавить, начиная с самого левого ребенка последней добавленной вершины, следующую вершину. Порядок добавления соответствует порядку при прямом обходе в глубину. При этом мы не рассматриваем вершины, которые находятся глубже более чем на один уровень, чем уже добавленные вершины. Если новая вершина является не первой после последней добавленной, то перед ее добавлением в шаблон добавляется фиктивная вершина (placeholder). Это показывает, что шаблон содержит часть, которая не является общей для разных вхождений шаблона.

Для того чтобы понять, является ли новая вершина достаточно часто встречающейся в определенном шаблоне, мы используем механизм корзин (bucket). Корзина может быть заполненной и незаполненной. Заполненная корзина содержит как минимум minSupport вхождений, а незаполненная, соответственно, меньше. Одно вхождение может лежать в нескольких незаполненных корзинах, но только в одной заполненной, при чем если оно лежит в заполненной корзине, то в незаполненных оно лежать не может. Каждая корзина соответствует определенной вершине-кандидату на расширение.

```
Def extend(verticalOccurrenceList, minSupport)
    (completedBuckets, uncompletedBuckets) = empty, empty
    unplacedRoots = verticalOccurrenceList.roots
    while(unplacedRoots.nonEmpty)
        for(root in unplacedRoots)
            candidate = nextCandidate(root)
            if (completedBuckets.contains(candidate.encode))
                completedBuckets.add(candidate)
            if (uncompletedBuckets.contains(candidate.encode))
                uncompletedBuckets.add(candidate)
            else
                uncompletedBuckets += new Bucket(condidate)
        for (bucket : uncompletedBuckets)
            if (bucket.completed)
                completedBuckets += bucket
                for (occ : completedBuckets.occurrences)
                    unplacedRoots -= occ
    isTemplate = unplaceRoots.size >= minSupport
    return completedBuckets, isTemplate
```

На каждом шаге расширения шаблона для определенного вертикального списка

вхождений, мы поочередно для каждого вхождения добавляем следующую вершину-кандидата. На первом шаге все вхождения являются активными, то есть могут расширятся. Как только вхождение попало в заполненную корзину либо мы дошли до конца поддерева, вхождение убирается из списка активных вхождений.

Вертикальные списки вхождений из заполненных корзин идут на следующий шаг алгоритма, для последующего расширения. Если вхождений, не попавших ни в одну из заполненных корзин, больше чем minSupport, то текущий шаблон включатся в ответ алгоритма, так как это значит, что мы не смогли расширить шаблон, который является часто встречающемся по построению.

Далее следуют стадии пост-обработки и фильтрации шаблонов.

Стадия пост-обработки также является языко-зависимой. Для Java выполняются следующие оптимизации:

Сжатие placeholder- placeholderразделенные одним из следующих символов (“;”, “.”, “ “, “()”) объединяются в один.

Удаление внешних фигурных скобок.

Далее мы избавляемся от шаблонов, которые являются подстрокой других шаблонов. Для этого мы используем обобщенное суффиксное дерево. Шаблоны сортируются по убыванию длины текста. Затем последовательно ищется подстрока с таким шаблоном, если такой подстроки не найдено, то этот шаблон добавляется в обобщенное суффиксное дерево и в множество общих шаблонов.

На следующем шаге выполняется фильтрация шаблонов. На данный момент используются следующие характеристики для фильтрации:

1. Минимальная и максимальная длина текста.
2. Минимальное и максимальное количество вершин в составе шаблона.
3. Максимальное количество placeholder.
4. Максимальное соотношение количество placeholder к количеству вершин.

Реализован режим оптимиизации параметров фильтрации. Так как определение является ли шаблон полезным является в достаточной мере субъективным, было решено дать возможность определять важные шаблоны пользователям.

В данном режиме работы необходимо задать целевое значение количество шаблонов, которые пользователь хочет получить в результате работы программы. В соответствии с этим параметром будет выбран коэффициент минимального количества раз, которое должен встретиться шаблон, чтобы называться часто встречающимся. Затем поиск шаблонов будет запущен несколько раз и после каждого запуска будет корректироваться коэффициент минимального количества раз, которое должен

встретиться шаблон, чтобы называться часто встречающимся. Корректировка производиться с помощью бинарного поиска, так как утверждается, что количество найденных часто встречающихся структур монотонно возрастает с понижением коэффициента минимального количества раз, которое должен встретиться шаблон, чтобы называться часто встречающимся. Таким образом, удается подобрать значение коэффициента минимального количества раз, которое должен встретиться шаблон, чтобы называться часто встречающимся за $\mathcal{O}\left(log(n)\right)$, где n - количество возможных значений коэффициента минимального количества раз, которое должен встретиться шаблон.

После нахождения ближайшего количества шаблонов к целевому значению, наступает фаза определения важности шаблонов. Пользователь может выбрать те шаблоны, которые, по его мнению являются важными. Затем на основе этих шаблонов будут отредактированны значения коэффциентов для фильтрации в сторону значению шаблонов помеченных как важные пользователем.

После этой фазы запускается поиск шаблонов с уже отредактированными параметрами фильтрации шаблонов. Таким образом, мы получаем возможность настроить по собственному желанию параметры фильтрации и как результат результирующую выборку.

2.2.2. Анализ времени работы

Построить список вершин мы можем за линейное от количества вершин время (за один dfs).

На стадии расширения шаблонов в худшем случае мы сделаем $\mathcal{O}\left(\sum\limits_{f \in F} c(f)^2\right)$, где f - внутренние вершины, (f) - количество потомков вершины.

Так как в худшем случае на каждом шаге мы будем доходить до последнего ребенка в поисках кандидата, а добавлять кандидата с первым ребенком.

Но, к счастью, обычно кандидаты находятся на нескольких первых шагах, либо не находятся вовсе. И таким образом асимптотика становится равной $\mathcal{O}\left(\sum\limits_{f \in F} c(f)\right)$.

Последующие шаги, такие как пост-обработка, удаление под-шаблонов, фильтрация выполняются за линейное от размеров шаблонов время.

Таким образом, единственная влияющая на асимптотику фаза - это фаза расширения. Сложность алгоритма $\mathcal{O}\left(\sum\limits_{f \in F} c(f)\right)$.

2.3. Выбор наиболее общих структур

Некоторые из полученных шаблонов являются лишь частью каких-то других шаблонов. Мы же хотим получать только наиболее общие структуры.

Для этого мы используем обобщенное суффиксное дерево.

Эта структура может отвечать на запросы вида:

1. find(string): boolean- является ли stringподстрокой одной из множества строк, которые хранятся в обобщенном суффиксном дереве.

2. add(string): Unit - добавляет строку stringв множество строк дерева.

Время работы всех запросов линейно зависит от максимального размера строки хранящейся в обобщенном суффиксном дереве.

Таким образом, для того чтобы получить множество наиболее общих шаблонов, мы можем отсортировать шаблоны по убыванию размера текста шаблона.

Для каждого шаблона проверить, является ли текст шаблона подстрокой одной из строк хранящейся в обобщенном суффиксном дереве.

Если данный текст уже содержится, мы просто пропускаем его. Если же данным текст там не присутствует, мы добавляем рассматриваемый шаблон в ответ, а также добавляем его текст в дерево.

Это позволяет извлечь наиболее общие шаблоны (не являющиеся подстроками других шаблонов).

3. Результаты

Для оценки улучшения скорости работы был проведен ряд тестов. Тесты проводились на популярных проектах с открытым исходным кодом найденные на Github.

Название проекта	Количество вершин в лесу AST деревьев	Текущее решение, мс	Предыдущее решение, мс
Suffix tree	13784	43	231
Server benchmark	28885	363	681
Glide	104190	290	1589
Java Design Patterns	353308	1226	2364
OkHttp	963813	4147	11035
Spring Boot	2898363	13655	21106
Hibernate	7580079	42747	137300
Spring	8581835	57428	560286

На данном графике видно (3), что ускорение на разных проектах составляет от 2 до 10 раз. Что дает основание утверждать, что данное решение действительно ускоряет время поиска часто встречающихся структур. Были выбраны следующие проекты:

1. Suffix Tree - реализация обобщенного суффиксного дерева.

2. Server Benchmark - реализация нескольких видов синхронных и ассинхронных архитектур серверов.

3. Glide - быстрый и эффективный проект для загрузки изображений фреймворк для андроид с открытым исходным кодом для загрузки изображений, для обертывания и декодирования мультимедиа, подерживает кэширование, пулы ресурсов и простой и легкий в использовании интерфейс. Glide поддерживает извлечение, декодирование и отображение кадров видео, работу со статическим изображением и анимированными картинками. Glide включает в себя гибкий API, который позволяет разработчикам подключить практически к любой сетевой стек. По умолчанию Glide использует пользовательский HttpUrlConnection, размещенные в стеке, но также включает в себя утилиты библиотеки к проекту Google's Volley или библиотеку Square's OkHttp вместо. Основной акцент Glide на прокрутке любого списка изображений так гладко и быстро, как это воз-

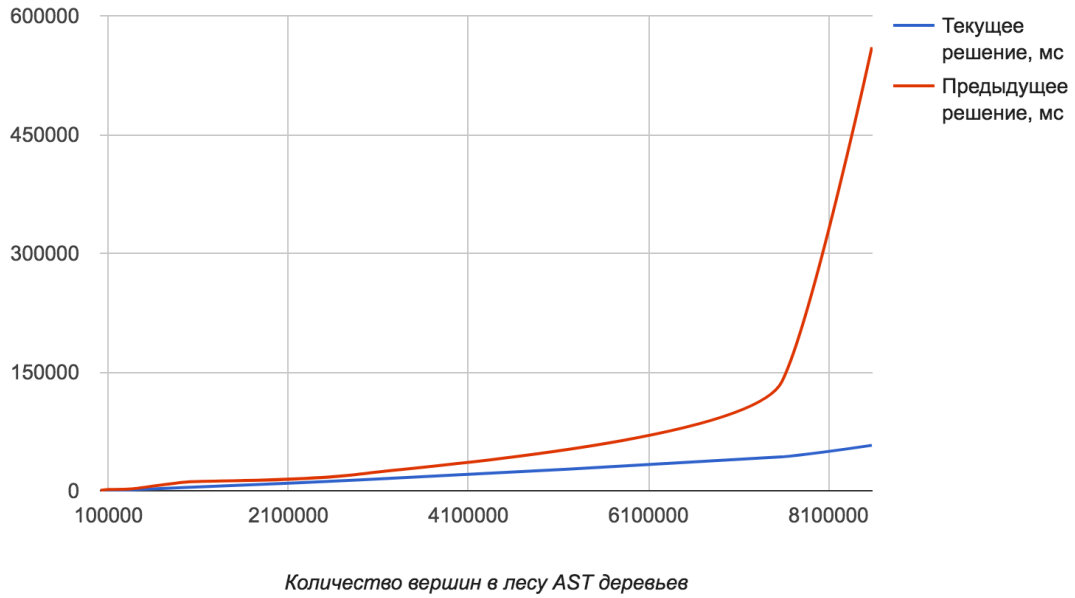


Рис. 5: Зависимость времени исполнения от числа вершин в лесу AST деревьев

можно, но Glide также эффективен практически для любого случая, когда вам нужно принести, изменение размера и отображения удаленного изображения.

4. Java Design Patterns - Шаблоны проектирования формализованные рекомендации, которые программист может использовать для решения общих проблем при разработке приложения или системы. Шаблоны проектирования могут ускорить процесс разработки, предоставляя проверенные парадигмы развития. Повторное использование шаблонов проектирования помогает предотвратить ошибки, которые могут стать причиной серьезных проблем, а также улучшить читаемость кода для программистов и архитекторов, которые знакомы с шаблонами.

5. OkHttp - HTTP & HTTP/2 клиент для Android и Java приложений.

6. Spring Boot - позволяет легко создавать Spring приложения, продуктовые приложения и сервисы с минимумом возни. Он принимает упрямый вид Spring платформе, так что новые и старые пользователи могут быстро добраться до того что им нужно. Вы можете использовать Spring Boot для создания автономного java-приложения, которые можно запустить с помощью java -jar или более традиционных способов развертывания WAR. Мы предоставляем инструмент командной строки, которая проходит Spring сценарии.

7. Hibernate - Hibernate ORM компонент/библиотека, обеспечивающая объектно-

Реляционного сопоставления (поддержка ORM) для приложений и других компонентов/библиотек. Он также обеспечивает реализацию спецификации jpa, которая является стандартизированной спецификации Java для ORM. См Hibernate.org для получения дополнительной информации.

8. Spring - Spring Framework предоставляет комплексную модель программирования и конфигурации для современных Java-приложений, на любые платформы развертывания. Ключевым элементом Spring является инфраструктурная поддержка на уровне приложения: Spring фокусируется на внутренности корпоративных приложений, так что программисты могут сосредоточиться на уровне приложений бизнес-логики, без лишних связей в специфических условиях развертывания.

Проанализируем зависимость количества найденных часто встречающихся структур от значения коэффициента минимального количества раз, которое должен встретиться шаблон, чтобы называться часто встречающимся (3).

Как видно на этом графике, зависимость ведет себя так как и предполагалось. Увеличение коэффициента минимального количества раз, которое должен встретиться шаблон, чтобы называться часто встречающимся ведет к уменьшению количества найденных шаблонов.

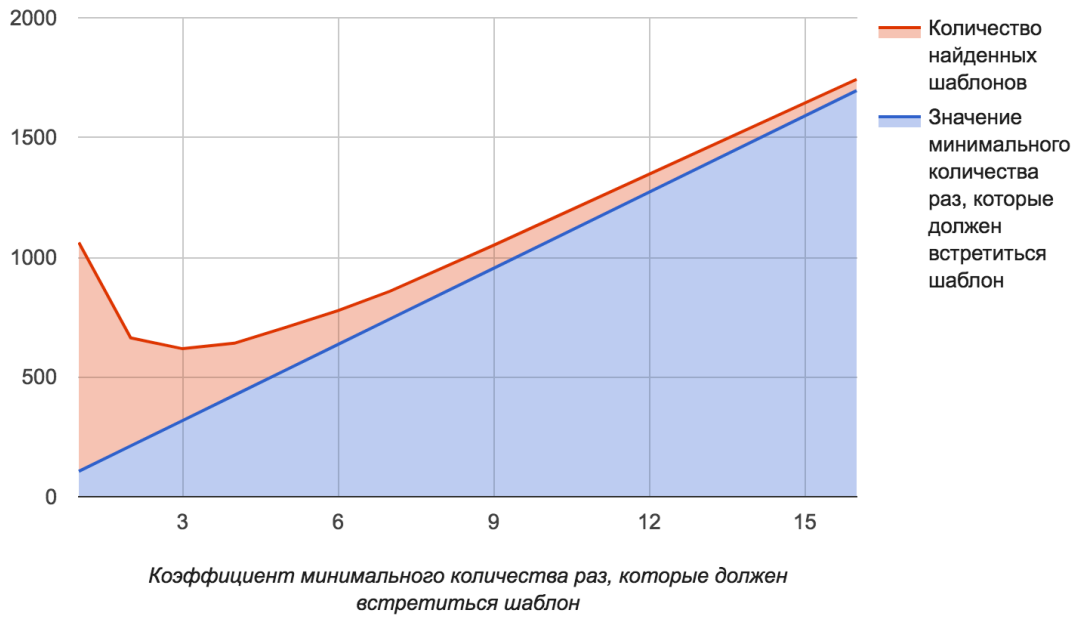


Рис. 6: Зависимость количества найденных шаблонов от минимального количества раз, которое должен встретиться шаблон, чтобы называться часто встречающимся

Рассмотрим несколько интересных результатов найденных в разных проектах. Начнем с результатов, которые были найдены в проекте Suffix Tree.

```
public # () {
    #
}
```

Данный шаблон может быть использован для создания функции с публичным уровнем доступа. Для использования данного шаблона нужно будет задать возвращаемый тип, название функции и собственно тело функции.

```
for (int i = 0; i < # ; i++) {
    #
}
```

Данный шаблон может быть использован для создания цикла по индексу. Для использования данного шаблона нужно задать количество раз, которые будет исполнятся этот цикл и собственно тело цикла. Нужно сказать, что данный шаблон уже присутствует в коллекции стандартных шаблонов для Java. Таким образом, данный результат лишь подтверждает правильность этого выбора.

```
for (String s : getSubstrings( # )) {
    #
}
```

Данный шаблон может быть использован для создания цикла for-each по всем подстрокам, какой либо строки. Для использования данного шаблона нужно задать строку, по подстрокам которой будет происходить итерация и собственно тело цикла. Этот шаблон часто используется в тестах данного проекта. Это хороший пример шаблона, который может быть полезен в конкретном проекте.

Далее рассмотрим результаты найденные в проекте Hibernate-ORM.

```
@Test
@TestForIssue( jiraKey = # )
public void # () {
    #
}
```

Данный шаблон используется для создания функций для тестов. Для использования данного шаблона нужно указать номер задачи в баг-трекере Jira, название теста, а также само тело теста.

```
private static final Logger LOGGER = LoggerFactory.getLogger( #
    .class);
```

Данный шаблон используется при создании инстанса логгера для конкретного класса. Он создает статическое финальное поле логгера, использую фабрику для логгеров. Для использования данного шаблона нужно указать название класса.

```
List< # > #  = new ArrayList<>( # );
```

Данный шаблон часто встречается при создании списков. Он создает инстанс ArrayList. Для его использования нужно задать тип элемента, который будет храниться в коллекции, названии коллекции, а также предполагаемый размер коллекции.

Далее идет список найденных шаблонов без описания.

```
try {
    #
} catch ( # ) {
    #
}
```

```
assertEquals(0, #);
```

```
assertEquals(1, #);
```

```
if ( #  == null) {
        return # ;
}
```

```
return ( #  != null ? #  : # );
```

```
assertTrue( #  instanceof # );
```

```
Collections.singletonList( # )
```

```
if ( #  instanceof # ) {
    #
} else {
    #
}
```

```
verify( # ) # ( # );


LOGGER.info( # , # );

@NotNull private final #  = new # ();
```

4. Заключение

В результате этой работы удалось реализовать алгоритм IMB3-Miner, для извлечения часто встречающихся индуцированных поддеревьев из леса. Также была проведена работа над пост-обработкой шаблонов и их фильтрацией. Добавлен режим оптимизации параметров фильтрации. Реализован выбор наиболее общих структур с помощью обобщенного суффиксного дерева. Был реализован плагин, который позволяет находить часто встречающиеся структуры. Причем делает это значительно быстрее, чем предыдущее решение.

Список литературы

[1] Aggarwal Charu C, Han Jiawei. Frequent pattern mining. — Springer, 2014.

[2] Baker Brenda S. A program for identifying duplicated code // Computing Science and Statistics. — 1993. — P. 49–49.

[3] Baxter Ira D. Clone Detection Using Abstract Syntax Trees. — 1998.

[4] IMB3-Miner: mining induced/embedded subtrees by constraining the level of embedding / Henry Tan, Tharam S Dillon, Fedja Hadzic et al. // Pacific-Asia Conference on Knowledge Discovery and Data Mining / Springer. — 2006. — P. 450–461.

[5] Jiménez Aí da, Berzal Fernando, Cubero Juan-Carlos. Frequent tree pattern mining: A survey // Intelligent Data Analysis. — 2010. — Vol. 14, no. 6. — P. 603–622.

[6] Zaki Mohammed J. Efficiently mining frequent trees in a forest // Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining / ACM. — 2002. — P. 71–80.