СТАТЬИ

УДК 004.42

АРХИТЕКТУРА ПРОГРАММНОЙ СИСТЕМЫ АВТОМАТИЗАЦИИ РЕФАКТОРИНГА КОДА НА ОСНОВЕ МЕТОДА КОМБИНАТОРНЫХ ПАРСЕРОВ

Алпатов А.Н., Соловьев Р.В.

ФГБОУ ВО «МИРЭА – Российский технологический университет», Москва, e-mail: aleksej01-91@mail.ru

Рефакторинг программного кода является важной частью современного процесса разработки. Рефакторинг программного кода позволяет не только снизить конечную стоимость производства и обслуживания программного обеспечения за счет исправления возможных ошибок, но и выполнить анализ качества кода всей программной системы или отдельных частей. В результате проведённого анализа предметной области в данной работе выдвинуто предположение, что использование парсер-комбинаторов в методах рефакторинга является более предпочтительным и достаточным для решения большинства повседневных практических задач, нежели, например, рефакторинг на основе регулярных выражений или реализация полноценного синтаксического анализа с построением абстрактного синтаксического дерева (abstract syntax tree – AST). В данной работе представлена архитектура программной системы рефакторинга кода на основе модифицированного метода парсеров комбинаторов. Предложен способ использования заполнителей в шаблонах синтаксического поиска и анализа, с возможностью введения соответствующих ограничений. Также продемонстрирован способ развёртывания предлагаемого решения в среде непрерывной интеграции и непрерывной доставки (CI/CD), реализованной на основе популярных и широко используемых на практике программных средств, что позволяет упростить и значительно снизить стоимость внедрения системы автоматизированного рефакторинга в реальный процесс разработки программного обеспечения.

Ключевые слова: рефакторинг, комбинаторный парсер, язык Дика, абстрактное синтаксическое дерево, просмотр кода

SOFTWARE SYSTEM ARCHITECTURE FOR CODE REFACTORING AUTOMATION BASED ON COMBINATORIAL PARSER METHOD

Alpatov A.N., Solovev R.V.

MIREA – Russian Technological University, Moscow, e-mail: aleksej01-91@mail.ru

Code refactoring is an important part of the modern development process. Program code refactoring allows not only to reduce the final cost of production and maintenance of software by correcting possible errors, but also to analyze the quality of the code of the entire software system or individual parts. As a result of the analysis of the subject area, in this paper it is suggested that the use of a combinator parser in refactoring methods is more preferable and sufficient for solving most everyday practical problems than, for example, refactoring based on regular expressions or the implementation of a full-fledged parsing with the construction of abstract syntax tree (abstract syntax tree – AST). This paper presents the architecture of a code refactoring software system based on a modified combinator parser method. A method is proposed for using placeholders in templates for syntactic search and analysis, with the possibility of introducing appropriate restrictions. Also, a method for deploying the proposed solution in a continuous integration and continuous delivery (CI / CD) environment implemented on the basis of popular and widely used software tools is demonstrated, which makes it possible to simplify and significantly reduce the cost of introducing an automated refactoring system into a real software development process.

Keywords: refactoring, combinatorial parser, Dyck language, abstract syntax tree, code review

Необходимость постоянной доработки программного обеспечения в условиях стремительно развивающихся технологий приводит к росту сложности и объема кодовой базы, заложенной внутри разрабатываемых систем, в результате чего большая часть ресурсов команды разработки направлена на сопровождение и поддержку существующего программного обеспечения, а не на разработку нового.

Сопровождение (поддержка) программного обеспечения включает в себя управление состоянием системы: не только добавление нового, но и обеспечение работоспособности уже реализованного функционала, а также исправление ошибок. Компании вынуждены подстраиваться, делая все больший акцент на качестве программного обеспечения, увеличивая количество затрачиваемых ресурсов на тестирование и рефакторинг.

Высокие темпы развития технологий и необходимость быстрого реагирования на изменяющиеся условия привели к развитию и популяризации гибких методологий разработки, позволяющих быстро вносить изменения и предоставлять их конечному пользователю. Традиционным является подход «ручного» рефакторинга, с применением инструментов анализа качества кода и автоматическим форматированием в среде разработки. Такой подход может стать одним из «узких мест» в процессах компании.

Необходимость автоматизации рефакторинга привела к появлению многих инструментов и подходов, но из-за сложности реализации подобные инструменты имеют множество проблем и недостатков, связанных с удобством и сложностью использования в силу специфичности языков.

Подобные инструменты реализованы либо на основе регулярных выражений, такие как sed [1] или JSCodeshift [2], в которых выполняются простые текстовые подстановки, либо на построении абстрактного синтаксического дерева, такие как clang-reformat [3] и Putout [4].

Реализации, основанные на регулярных выражениях, имеют довольно низкую скорость работы, а также сложно воспринимаемый и ограниченный синтаксис, связанный с особенностями описания самих регулярных выражений. Основным недостатком данного подхода является отсутствие поддержки сбалансированных групп (например, для анализа скобок).

Использование AST подхода позволило бы выполнять необходимый анализ, но для применения подобных инструментов необходимы довольно глубокие знания в области синтаксических анализаторов. Также существенным недостатком использования абстрактных синтаксических деревьев является сложность сценариев редактирования деревьев с большим количеством узлов, что проявляется также в виде отсутствия таких инструментов для многих языков программирования, особенно не самых популярных, что опять же связано со сложностью данного подхода [5].

Все большую популярность приобретают функциональные языки программирования, использование которых позволяет по-новому решать существующие задачи, применяя подходы и методы, характерные именно для функционального программирования. Одним из таких мощных инструментов является парсер-комбинатор, ставший одним из основных подходов при написании парсеров. Реализация системы рефакторинга на основе данного метода позволит по-новому взглянуть на задачу автоматизации рефакторинга [7, 3].

Парсер-комбинаторы как основа построения систем рефакторинга

Зачастую в реальных условиях разработки процесс рефакторинга не требует промежуточных представлений исходного кода, как в случае использования AST подхода, а выполняемые преобразования должны быть удобны для чтения и повторного использования, чего нельзя сказать об инструментах, в основе которых регулярные выражения.

Большинство преобразований, выполняемых при рефакторинге, не требуют информации о типе той или иной переменной, стеке выполнения и даже самом языке программирования. Подобный рефакторинг может быть выполнен путем анализа и обработки исходного кода программы как обычного текста. Разбор необходимых конструкций может быть выполнен с помощью парсер-комбинаторов, основной идеей которых является построение парсеров как функций высшего порядка, которые могут быть составлены из более простых парсеров.

В качестве примера может быть рассмотрена замена конструкций ветвления на специальные операторы, повышающие читаемость кода на языке C#.

На рис. 1 показан процесс замены ветвления на тернарный оператор, который вычисляет логическое выражение и в зависимости от полученного значения true или false и возвращает результат одного из двух соответствующих выражений: левого, если выражение равно true, и правого, если false.

Как видно из рис. 1, эквивалентная конструкция *if else* может быть более выразительно и компактно представлена с помощью тернарного оператора.

По аналогии, на рис. 2 показана замена ветвления на оператор присваивания с объединением с NULL, который связывает правый и левый операнды по значению только в том случае, если в левосторонний аргумент операции вносится значение null.

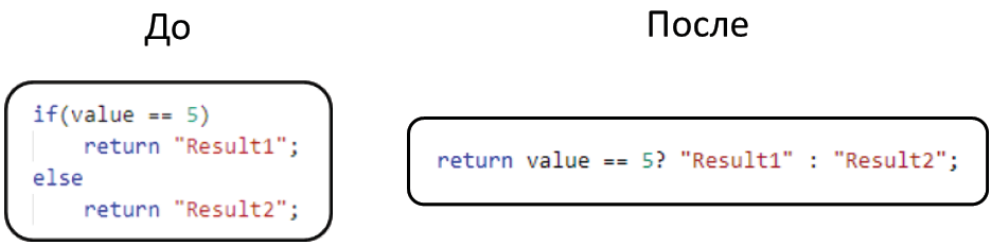

Рис. 1. Пример замены ветвления на тернарный оператор

До

```
if(value == null)
{
    value = new List<int>();
}
```

После

```
value ??= new List<int>();
```

Рис. 2. Пример замены ветвления на оператор присваивания с NULL объединением

До

```
if(tmp == null)
{
    value = tmp;
}
else
{
    value = new List<int>();
}
```

После

```
value = tmp ?? new List<int>();
```

Рис. 3. Пример замены ветвления на оператор объединения с NULL

На рис. 3 показана замена ветвления на нулевой оператор объединения с NULL, который возвращает свой правый аргумент, только если левый аргумент равен null или «не определено», в противоположном случае возвращается левый аргумент.

Большинство инструментов, основанных на регулярных выражениях, не могут быть использованы для сопоставления подобных шаблонов, так как содержат сбалансированные группы в виде скобок и многострочные конструкции с группировками.

Решение, основанное на использовании AST инструментов и на посещении абстрактного синтаксического дерева, является слишком избыточным для подобной задачи.

Математическая модель рефакторинга программного кода

Зачастую при решении той или иной задачи разработки использование самого языка программирования является неудобным для описания реализуемого решения. С целью более наглядного и удобного отображения специфики задачи разрабатываются специальные предметно-ориентированные языки (domain-specific language, DSL). Самыми известными примерами DSL являются SQL [8] и Regex [9].

Как было сказано выше, основу процесса рефакторинга в разрабатываемой системе составляют шаблоны поиска и замены, описываемые пользователем, на основе разработанного предметно-ориентированного языка.

В качестве основы предлагаемого DSL используется язык Дика (Dick language). Язык Дика является простым контекстно-свободным языком, представляющим из себя набор вложенных выражений, имеющих открывающий и закрывающий разделители.

Формальная грамматика, разбирающая значение, может быть представлена кортежем

$$S = < T, S, A, \varepsilon_s >, \quad (1)$$

где T – конечный набор символов – терминалов;

S – конечное множество нетерминалов;

A – конечное множество правил вида $R \leftarrow \varepsilon \mid R \in S$

ε_s – выражение для разбора и синтаксического анализа.

Таким образом, язык Дика может быть описан в виде выражения

$$S \rightarrow \varepsilon \mid \text{‘(‘S’)’S} , \quad (2)$$

где ε – пустая строка;

S – нетерминал;

‘(‘S’)’ – терминальные символы: открывающий и закрывающий разделители [10].

Для удобного представления вводятся дополнительные расширения:

1. Список разделителей состоит из элементов: ‘(‘S’)’, ‘[‘S’]’, ‘{‘S’}’.

2. Множество нетерминалов состоит из обычных строк, являющихся токенами языков программирования.

Полученный DSL позволяет описать шаблоны для большинства современных языков программирования и может быть описан с помощью грамматики, представленной в листинге (рис. 4).

Предложенная грамматика содержит четыре основных типа элементов:

− placeholder – идентификатор, обозначающий «динамическую» часть шаблона, т.е. ту, которая будет подставлена во время разбора шаблона;

− string_literal – строка, состоящая из любых символов, кроме символов разделителей и символов пробела;

− whitespace – любое количество пробельных символов: перенос строки, пробел, символа возврата каретки;

− comment – однострочный или многострочный комментарий.

В силу того, что в данной работе рассматривается вопрос создания автоматизированных инструментов для рефакторинга программного кода, необходим соответствующий критерий оценки проведённого в автоматизированном режиме рефакторинга. Для этих целей в данной работе предлагается сложная оценка качества рефакторинга кода.

Как было уже отмечено выше, рефакторинг программного кода, выполненный над некоторой структурой кода, не должен приводить к изменению внешнего поведения его логики, то есть для других структурных элементов исходного кода, а также непосредственного для самого разработчика, логика функционирования той части кода, которая будет подвергнута рефакторингу, как до момента модификации, так и после должна сохраняться. В зависимости от характера рефакторинга, в том числе и от выбранного метода рефакторинга, и сложности исходного кода, объём вносимых изменений может существенно отличаться. Так в реальных условиях разработки к одной и той же логической структуре могут быть применены как один из базовых методов рефакторинга, например изменение сигнатуры метода, так и их последовательная совокупность, например удаление параметра и перемещение метода. В рамках разрабатываемой системы объём и совокупность применяемых методов формируются пользовательскими шаблонами поиска и замены. Таким образом, оценка рефакторинга может быть следующей:

$$\vartheta = \sum_{i=0}^{n} \varphi_i , \qquad (3)$$

где φi ∈ [0..1] – фактор применения соответствующего метода рефакторинга к участку кода (0, если метод применён, иначе – 1);

n – количество методов рефакторинга, поддерживаемых автоматизированной системой.

При этом задачей оптимизацией в данном случае будет $f(\vartheta) \rightarrow \max$.

Шаблоны поиска с заполнителями в задаче рефакторинга

Большинство простых преобразований кода основано на сопоставлении токенов, определяемых в строго заданном структурном контексте. Для описания подобных шаблонов может быть использована простая грамматика, основанная на заполнителях (placeholder) и текстовой части, содержащей конструкции анализируемого языка программирования.

Для выполнения рефакторинга, с помощью полученной грамматики описываются шаблоны поиска и замены.

Примеры, рассмотренные выше с описанными шаблонами поиска и замены, представлены на рис. 5.

```
grammar = term EOF
term = '(' term ')' | '{' term '}' | '[' term ']' | term + | token
token = placeholder | string_literal | whitespaces | comment
placeholder = '$' identifier '$'
string_literal = <escaped string>
whitespace = [“\n\r” | '\n' | ' ‘] +
comment = <single or multiline comment>
```

Рис. 4. Листинг – Грамматика предложенного DSL

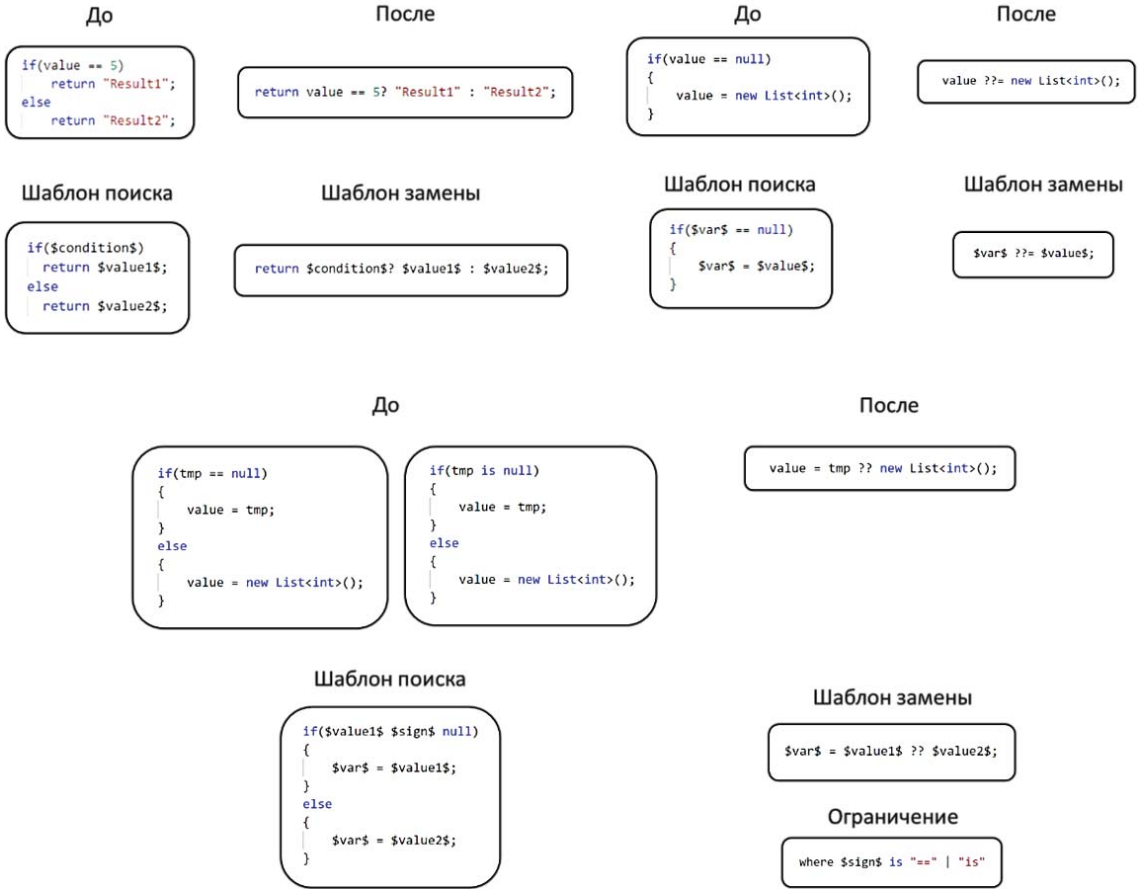

Рис. 5. Пример шаблонов поиска и замены

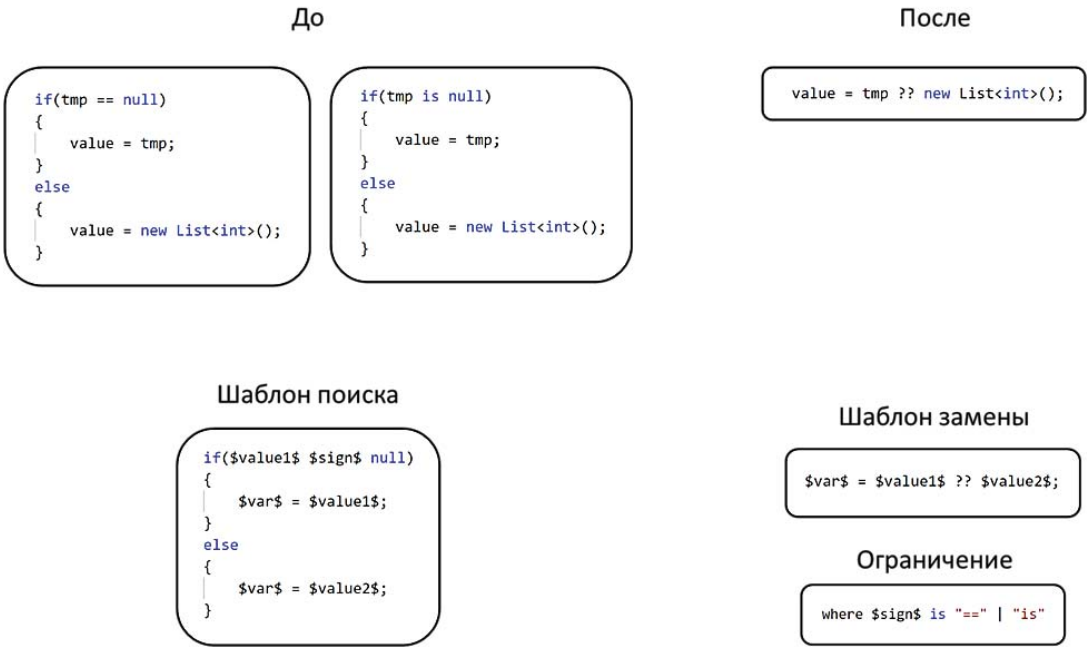

Рис. 6. Шаблоны поиска и замены с ограничениями

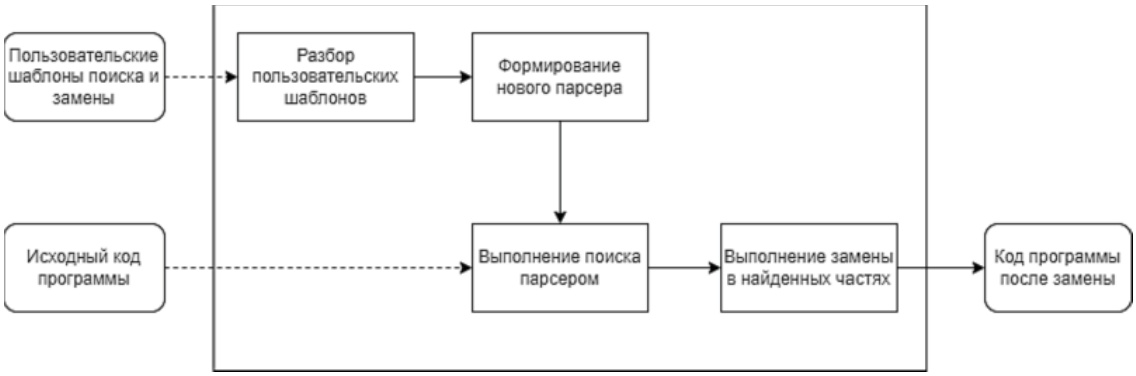

Рис. 7. Обобщённая схема работы комбинаторного парсера

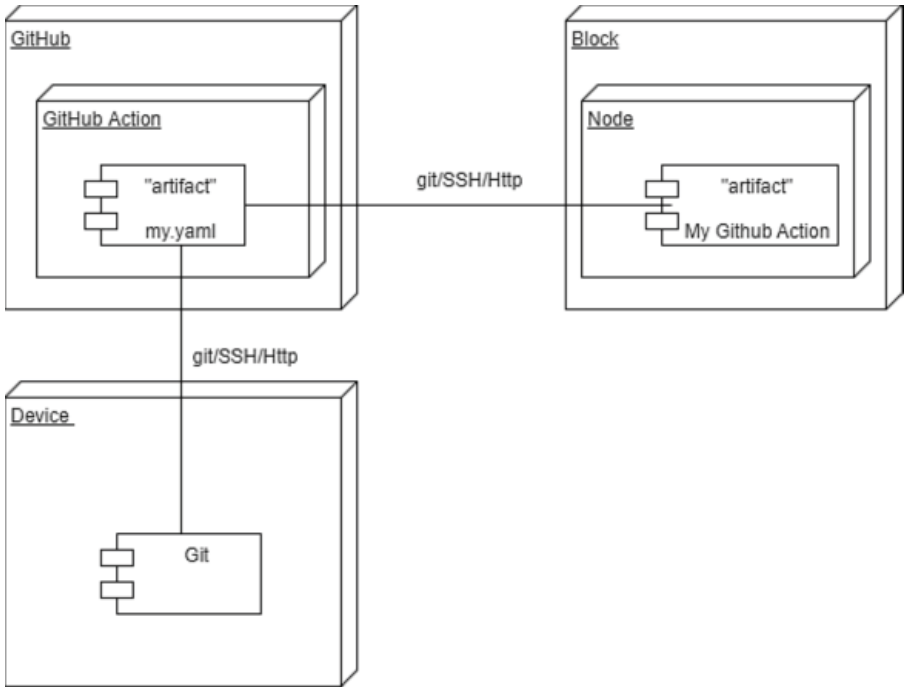

Рис. 8. Диаграмма развёртывания предлагаемой системы

Для реализации дополнительной логики могут быть указаны ограничения для заменителей, содержащие регулярные выражения и комбинированные фильтры. Использование ограничений позволит ограничить и расширить количество частей, удовлетворяющих шаблонам поиска. На рис. 6 показан пример ввода дополнительных ограничений для заменителей в шаблонах поиска и замены.

Программная система автоматизированного рефакторинга

Как было сказано выше, основной частью разрабатываемой системы является парсер-комбинатор, использование которого позволит построить парсеры из других, более простых парсеров.

На основе данного подхода может быть создан парсер, являющийся синтаксическим анализатором для подаваемых на вход структурных шаблонов пользователя и результатом работы которого является новый парсер, с помощью которого может быть выполнен поиск и замена в анализируемом исходном коде.

Основные правила разбора шаблона для формирования нового парсера:

– Наличие одного пробельного символа в шаблоне интерпретируется как неограниченное количество пробелов в анализируемом коде. Подобное поведение облегчает

работу с шаблонами и позволяет не следить за строгостью соблюдения при выполнении шаблона. При необходимости данное поведение может быть изменено

− Строковые литералы интерпретируются как строго заданная последовательность символов, необходимая в исходном коде для успешного сопоставления с шаблоном.

− Наличие в шаблоне открывающего разделителя означает строгое наличие закрывающего парного разделителя, для соответствия определения языка Дика. Пара разделителей в шаблоне гарантирует наличие такой же пары в тексте.

− Возможное содержимое заполнителя (placeholder) определяется заданным в шаблоне ограничением. В случае отсутствия ограничений содержимое заполнителя определяется как набор любых строковых литералов и пробельных символов, имеющих разделители согласно определению языка Дика.

− Наличие заполнителей с одинаковым именем в шаблоне интерпретируется как строго совпадающие части в анализируемом тексте.

Результатом разбора пользовательского шаблона является парсер, по которому затем выполняется анализ заданного исходного кода. Затем для областей кода, соответствующих шаблону поиска, выполняется операция перезаписи, согласно заданному шаблону замены.

Обобщенная схема работы комбинаторного парсера представлена на рис. 7.

Предлагаемое решение реализуется в виде библиотеки классов, которая затем может быть использована в виде GitHub action, запускаемого в рамках выполнения процессов CI/CD в репозитории проекта. Шаблоны поиска и замены описываются в конфигурационном файле, находящемся в репозитории вместе с исходным кодом.

Диаграмма развертывания программных артефактов предложенной системы представлена на рис. 8.

Заключение

В данной работе предложен способ реализации системы автоматизированного рефакторинга на основе метода комбинаторного парсера, что значительно упрощает процесс синтаксического анализа в современных языках программирования и, следовательно, процесс рефакторинга. Предложенный способ даёт возможность построения более сложного синтаксического анализатора на основе совокупности простых в реализации комбинаций парсеров, которые будут являться входными аргументами комбинатора синтаксического анализатора, предлагаемого в данной работе. Предложенное решение встраивается в существующие и широко используемые механизмы непрерывной интеграции и доставки, за счёт использования популярных сервисов, таких как системы версионного контроля Git, репозитория GitHub, сервиса системы CI/CD GitHub Actions и облачной платформы Azure с развернутым Node.js, что позволит упростить процесс внедрения предложенного решения автоматизированного рефакторинга в существующий технологический цикл разработки промышленного программного обеспечения.

Список литературы

1. GNU sed [Электронный ресурс]. URL: https://www.gnu.org/software/sed/ (дата обращения: 11.02.2022).

2. Ni W., Sunshine J., Le V., Gulwani, S., Barik, T. reCode: A Lightweight Find-and-Replace Interaction in the IDE for Transforming Code by Example. In The 34th Annual ACM Symposium on User Interface Software and Technology. 2014. P. 258–269.

3. Van Tonder R., Le Goues C. Lightweight multi-language syntax transformation with parser parser combinators. Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation. 2019. P. 363–378.

4. Putout [Электронный ресурс]. URL: https://github.com/coderaiser/putout (дата обращения: 11.02.2022).

5. Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, Martin Monperrus. Fine-grained and accurate source code differencing. Proceedings of the 29th ACM/IEEE international conference on Automated software engineering. 2014. P. 313–324.

6. Abstract Syntax Tree [Электронный ресурс]. URL: https://ps-group.github.io/compilers/ast/ (дата обращения: 14.02.2022).

7. Классические парсер-комбинаторы на Python [Электронный ресурс]. URL: https://habr.com/ru/post/317304/ (дата обращения: 14.02.2022).

8. SQL – computer language [Электронный ресурс]. URL: https://www.britannica.com/technology/SQL (дата обращения: 14.02.2022).

9. Everything you need to know about Regular Expressions [Электронный ресурс]. URL: https://towardsdatascience.com/everything-you-need-to-know-about-regular-expressions-8f622fe10b03 (дата обращения: 18.02.2022).

10. Liebehenschel J. Lexicographical generation of a generalized dyck language. SIAM Journal on Computing. 2003. Vol. 32. No. 4. P. 880–903.