ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

"НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО"
(УНИВЕРСИТЕТ ИТМО)

Факультет программной инженерии и компьютерной техники

Направление (специальность) — 09.04.04 Программная инженерия

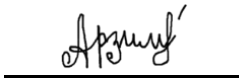
Образовательная программа — Веб-технологии

Дисциплина — Проектирование и анализ языков веб-решений

Курсовой проект (работа)

ТЕМА: Разработка графа зависимостей программных модулей

ВЫПОЛНИЛ

Студент группы Р41081  Арзиманова К. А.

№ группы подпись, дата ФИО

ПРОВЕРИЛ к.п.н, доцент _____ Государев И. Б.

ученая степень, должность подпись, дата ФИО

САНКТ-ПЕТЕРБУРГ

2021 г.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

В роли преобразователей исходного кода программы в машинный выступают либо интерпретаторы, либо компиляторы. Программный код, представляющий собой обычный текст, проходит этап парсинга – превращения обычного текста в иерархическую структуру данных, называемую абстрактным синтаксическим деревом AST (Abstract Syntax Tree). На многих стадиях жизненного цикла программного обеспечения, анализ исходного кода программы является важной частью, как разработки, так и дальнейшей поддержки продукта. Это помогает изначально создавать качественный программный продукт, оценивать степень оттестированности программы и оптимизировать задачу регрессионного тестирования. Такой анализ удобно проводить на основе графа зависимостей модулей программного продукта. В данной курсовой работе рассмотрим алгоритм построения графа зависимостей исходной программы в виде абстрактного синтаксического дерева для программ, состоящих из модулей системы Node.js. В таком дереве вершинами будут являться модули программы, а дугами их интерфейсы.

1 МОДУЛЬНОЕ ПРОГРАММИРОВАНИЕ

Для модульного программирования характерна декомпозиция исходного задания на отдельные задачи, которые представляются в виде программных модулей. Каждый модуль реализует отдельную функциональность программы и имеет публичный (public) интерфейс, доступный для использования в других модулях. Данный подход уменьшает зону ответственности разработчика модуля по сравнению с разрабатываемой программой, ускоряет процесс разработки, упрощает процессы отладки и тестирования. Кроме того, такая архитектура программного обеспечения уменьшает количество изменений, вызванных редактированием или расширением спецификаций, или отдельных частей исходного кода. Для написания качественного кода на языке объектно-ориентированного программирования соблюдение перечисленных ранее аспектов в архитектуре можно достичь, если придерживаться принципов *SOLID*-программирования. Аббревиатура *SOLID* включает первые буквы названий пяти следующих основных принципов современного объектно-ориентированного программирования, предложенных Робертом Мартином.

1. *S: Single Responsibility Principle* (принцип единственной ответственности): каждый класс должен решать только одну задачу.
2. *O: Open-Closed Principle* (принцип открытости-закрытости): программные элементы (классы, методы, функции) должны быть открыты только для расширения, но не для модификации.
3. *L: Liskov Substitution Principle* (принцип подстановки Барбары Лисков (принцип заменяемости): классы-наследники можно использовать вместо родительских, не нарушая работу программы.
4. I: *Interface Segregation Principle* (принцип разделения интерфейса): интерфейсы должны быть узкоспециализированными, для того чтобы конкретный клиент использовал только нужные ему методы.

5. *D: Dependency Inversion Principle* (принцип инверсии зависимостей):

- должна быть выстроена четкая иерархия классов (предки не должны зависеть от потомков);
- связь между классами должна осуществляться через интерфейсы, а не напрямую;
- в интерфейсах и абстрактных классах не должно быть конкретных реализаций кода, только сигнатура;
- интерфейсы должны реализовывать конкретные классы.

Придерживаясь принципов *SOLID*-программирования, разработчик способен создать качественную, легко расширяемую программу, удобную для тестирования и дальнейшего сопровождения. Поскольку модули в архитектуре программы очень эффективны, то современные языки объектно-ориентированного программирования добавляют поддержку модульной структуры программ в свои стандарты.

В данном курсовом проекте рассматривается реализация модульного подхода *Node.js* для языка программирования *JavaScript* [1]. В настоящий момент в *Node.js* поддерживаются два типа модулей:

- система модулей, появившаяся в ES6 стандарте языка *JavaScript* – *ESM* [2];
- система модулей, появившаяся в 0.4 версии *Node.js* – *CommonJS* [3].

Каждый файл исходной программы, написанной на *Node.js*, является отдельным модулем, одного из перечисленных типов, между *EMS* и *CommonJS*-модулями поддерживается совместимость. Для предоставления публичного интерфейса используется синтаксические конструкции следующего вида:

1. В случае *CommonJS*:

```
module.exports = {
a: () => {...},
b: someConstant,
```

```
};
```

2. В случае *ESM*:

```
export const a = () => {...};
export const b = someConstant;
```

В данном примере, файл a.js, содержащий представленный выше программный код является модулем. А предоставляемые по ключам a и b методы – подмодулями данного модуля.

Для получения доступа к объявленному в модуле интерфейсу используются следующие синтаксические конструкции *require* и *import*:

1. В случае *CommonJS*:

```
const all = require('a.js'); // получаем доступ ко всем подмодулям a.js
const {b} = require('a.js'); // получаем доступ к подмодулю b
```

2. В случае *ESM*:

```
import all from 'a.js'; // получаем доступ ко всем подмодулям a.js
import {b} from 'a.js'; // получаем доступ к подмодулю b
```

При разработке программы с помощью модулей становится возможным построение граф зависимостей. Связи между различными зависимостями берутся из инструкций *import* или *require*, которые используются в коде.

В процессе разработки и сопровождении программы граф зависимостей используется для оптимизации отбора тестов при регрессионном тестировании [4]. В ходе регрессионного тестирования специалист по тестированию проверяет внесенные программистом изменения. Для такого вида тестирования специалист использует уже разработанные тестовые наборы и сценарии, на которых была обнаружена исправляемая ошибка и, возможно, дополнительно разрабатывает новые регрессионные тесты, если при исправлении была изменена / добавлена / удалена какая-либо функциональность программного кода. Затем тестировщик пробует воспроизвести ошибку каким-либо другим способом и обязательно тестирует последствия исправлений, т.к. исправления, возможно, внесли новые ошибки в код, который до этого исправно работал. Для повторного прогона отбирать тесты можно по построенному дереву зависимостей, оставив в дереве только вершины, соответствующие измененным модулям программы и вершины, соответствующие модулям, связанным с ними интерфейсами.

Для удобства отслеживания процесса изменения кода каждая версия программного продукта, отличающаяся от предыдущей версии исправлениями, должна иметь различные имена, показывающие название функционального модуля и его вариативность. После проведения последнего успешного тестирования программного продукта вносить изменения в программный код запрещено, поэтому готовым коммерческим продуктом, представляемым заказчику, является последняя успешно протестированная версия кода.

2 АБСТРАКТНЫЕ СИНТАКСИЧЕСКИЕ ДЕРЕВЬЯ

Абстрактное синтаксическое дерево (далее – *AST*) – конечное дерево, в котором вершины описывают операторы языка, а листья – операнды, т.е., представляют собой граф зависимостей (управления) программы. Рассмотрим процесс парсинга исходного кода программы, представляющий собой перевод текстового представления исходного кода в дерево AST. Данный процесс состоит из двух фаз:

- лексический анализ;
- синтаксический анализ.

2.1 Лексический анализ

Введем понятие морфологии, как множества, состоящего из языковых термов с описанием правил их использования. Морфология представляет собой описание доступных языковых конструкций исходного языка, в случае с языками программирования является частью спецификации языка программирования.

Лексический анализ – процесс, на вход которого подается строковое представление исходной структуры, в результате возвращается упорядоченный список *токенов. Токен* – элемент из множества, заданного исходной морфологией языка программирования.

Рассмотрим следующую конструкцию произвольного языка: *n*n*.

Результатом лексического анализа заданной конструкции является список токенов:

```
[
{ type: { ... }, value: "n", start: 0, end: 1, loc: { ... } },
{ type: { ... }, value: "*", start: 2, end: 3, loc: { ... }
}, { type: { ... }, value: "n", start: 4, end: 5, loc: { ...
} },
]
```

Каждый токен содержит информацию о:

- типе содержащегося терма – *type*;
- расположении терма в исходном текстовом представлении – *loc*;

• занимаемых позициях в линейном символьном представлении – *start, end.*

2.2 Синтаксический анализ

Синтаксический анализ – процесс, на вход которого подается упорядоченный список токенов, как результат – возвращается построенное дерево AST.

На этапе синтаксического анализа токены группируются в узлы дерева, между которыми задается продиктованная морфологией иерархическая связь.

Рассмотрим ранее объявленную конструкцию: *n*n*.

Произведем лексический анализ данной конструкции, используя морфологию Node.js, а затем, к получившемуся набору токенов, применим синтаксический анализ.

В качестве морфологии используем estree спецификацию: https://github.com/estree/estree. Нас будут интересовать следующие узлы:

1. Узел *ImportDeclaration*, соответствующий программному коду, описанному в ESM системе модулей:

```
interface ImportDeclaration <: ModuleDeclaration {
type: "ImportDeclaration";
    specifiers: [ ImportSpecifier | ImportDefaultSpecifier |
ImportNamespaceSpecifier ];     source: Literal;},
```

где *source* содержит информацию о пути в файловой системе до запрашиваемого модуля; specifiers множество запрашиваемых подмодулей.

2. Узел *CallExpression*, соответствующий программному коду, описанному в *CommonJS* системе модулей:

```
extend interface CallExpression {      callee: Expression | Super;},
```

где *callee* содержит информацию о вызове функции, если узел *CallExpression* имеет тип *Identifier*:

```
interface Identifier <: Expression, Pattern{type:"Identifier";name:
string;}
```

где параметр *name* со значением *require* будет соответствовать программному коду в *CommonJS* системе модулей.

3 ПОСТРОЕНИЕ ГРАФА ЗАВИСИМОСТЕЙ МОДУЛЕЙ ПРОГРАММЫ

3.1 Постановка задачи

Пусть существует множество F – множество файлов исходной программы и T – множество узлов абстрактного синтаксического дерева.

Введем функцию $p(f)\colon F \to T$, представляющую собой синтаксический *парсер* (переводчик текстового представления исходного кода в дерево *AST*). Применяя функцию *p(f)* для каждого элемента из множества модулей исходной программы F заполняем множество вершин соответствующего ей абстрактного дерева T. Необходимо построить граф $G := (V, E)$, где V – множество вершин графа G (конкретное имя модуля в файловой структуре), E – множество дуг графа (интерфейсы между модулями, будем хранить имена связанных с этой вершиной модулей). Для построения графа G введем функцию $n(f)\colon F \to V$, возвращающую имя файла в файловой структуре. Применив функцию *n(f)* к каждому элементу из множества F, заполним множество вершин V графа G. Зададим функцию $l(t)\colon T \to E$, возвращающую имена связанных с данной вершиной дерева T модулей. Применяя функцию *l(f)* к каждому из узлов дерева T заполняем множество дуг E графа G.

3.2 Описание алгоритма

Необходимо найти в построенном синтаксическом дереве модуля все узлы с вызовами *require* и *import*, отвечающих за использование интерфейсов других модулей.

Пусть на каждом из уровней вложенности исходного дерева содержится n_i вершин.

Тогда общее кол-во вершин $V = \sum^{n}{}_{i}$, $i = 0$, где k – количество уровней в исходном дереве.

В общем случае для поиска узлов, отвечающих за вызов *require* и *import* придется совершить полный обход дерева. Если рассматривать приложение, программный код которого использует только *ESM* систему модулей, то согласно спецификации языка программирования, гарантируется факт того, что вызов *import* всегда находится в корневой области видимости. Это важная информация, поскольку позволяет нам ограничить уровни для поиска. Таким образом, вместо обхода всего множества вершин: $\{n_0, n_1, \ldots, n_k\}$ достаточно будет обойти множество, состоящее из $\{n_0, n_1\}$, где n_0 является множеством, состоящим из одного элемента – корневого узла. Теперь задача сводится к тому, чтобы обойти всех детей корневого узла.

Именно поэтому в современном программировании становится актуальным стиль разработки программ, приводящий исходный код приложения от *CommonJS* к *ESM*-системе модулей. Исходя из вышеописанного, задачу построения управляющего графа можно разбить на две этапа: преобразование программы из совокупности *CommonJS* и *ESM*-модулей к программе, состоящей только из *ESM*-модулей; построение графа зависимостей.

Для преобразования исходного кода в модульную структуру *ESM* предлагается следующий *алгоритм*:

1. По алгоритму обхода графа в ширину *BFS (breadth-first search)* находим в исходном коде программы все узлы *CallExpression*, такие, где *callee* соответствует вызову *require*.
2. Создаем узел *ImportDeclaration* с параметром *source* найденным в пункте 1.
3. Дописываем данный узел в корневую область видимости, делая его ребенком узла *Program*.
4. Удаляем найденные в пункте 1 узлы.
5. Перезаписываем исходный файл.

Пример преобразования

Исходный программный файл *a.js:*

```
function a() { return b};
const b = require('c');
```

Изменения в *AST* дереве:

```
          (Statements
              {+(Import)+}
                (Function
                  (Identifier)
                  (StatementBlock
                    (Return
                      (Identifier))))
              {-(VariableDeclaration
                {-(JavaScriptRequire
                  {-(Identifier)-})-})-})
```

Символом «+» помечены добавленные узлы, символом «-» помечены удаленные узлы. Сгенерированный в результате преобразования новый программный код:

```
import b from 'c';
function a() { return b};
```

3.3 Программная реализация

Для удобной работы с иерархической структурой, содержащей в себе различные классы узлов, было решено использовать паттерн *Visitor*. Каждый узел *AST*-дерева является потомком класса *ASTNode*, в котором реализован метод принятия класса, имплементирующего интерфейс *IVisitor*. Такое решение позволяет реализовать логику обработки каждого типа узлов отдельно от класса и для каждой из двух исходных задач разработать свой класс:

1. *ESMPatcherVisitor* – реализует логику преобразования исходного кода, работает с узлами следующих классов: *ImportDeclaration*, *CallExpression*, *Program.*
2. *GraphBuilderVisitor* – реализует логику построения графа зависимостей, работает с узлами следующих классов: *ImportDeclaration*, *Program.*

Алгоритм построения графа:

1. По алгоритму обхода графа в ширину *BFS (breadth-first search)* проходим по списку детей узла типа *Program*, являющихся объектами класса *ImportDeclaration* и заполняем множество *V* вершин графа.

2. Из поля *source* получаем имя используемого модуля и в множество ребер *E* графа *G* добавляем соответствующее ребро (имя используемого модуля).

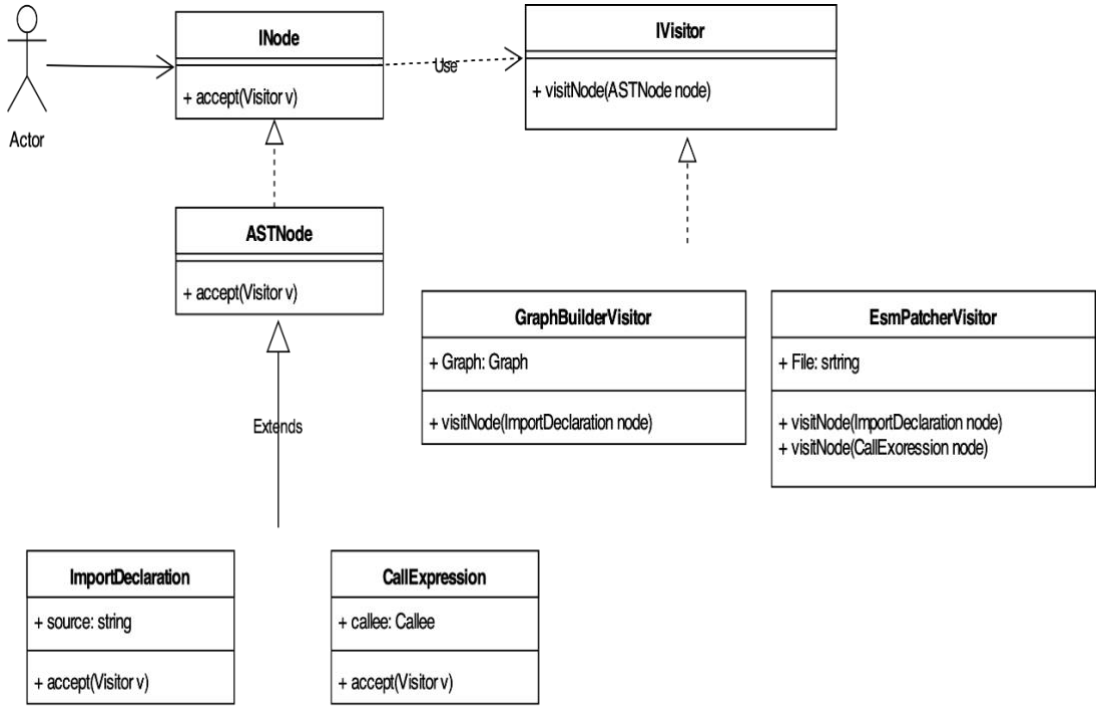


Рисунок 1 - UML-диаграмма классов программной реализации

4 РЕЗУЛЬТАТЫ АПРОБАРЦИИ ПРЕДЛАГАЕМЫХ АЛГОРИТМОВ

Вычислительные эксперименты проводились на машине со следующими характеристиками:

- процессор: 2,7 GHz Intel Core i5;
- память: 16 GB 1867 MHz DDR3.

Характеристики программного кода, для которого выполнялось построение графа:

- количество модулей: 2296;
- использованные системы модулей: *ESM* и *CommonJS*.

4.1 Процесс проведения экспериментов

Для каждого из двух вариантов комбинаций модулей (*ESM* и *CommonJS* или только *ESM*) в исходной программе было проведены 50 запусков, в таблице 1 приведено среднее арифметическое времени построения графа зависимостей программы по результатам всех запусков.

Таблица 1 - Результаты экспериментов

Использованные модульные системы	**Время выполнения программы, с**
ESM + CommonJS	237
ESM	125

Этап перехода от архитектуры программы *ESM + CommonJS* к архитектуре *ESM* при помощи алгоритма преобразования позволил ускорить работу построения графовой модели зависимостей модулей для конкретного программного кода в 1,9 раза.

4.2 Параллельная реализация алгоритма

Попробуем реализовать алгоритм обхода списка файлов с помощью распараллеливания вычислений. Для возможности распределенных вычислений применим *MapReduce* подход. Процесс *MapReduce* состоит из двух шагов: *Map* и *Reduce*.

1. *Map*: на этом шаге происходит построение дерева *AST* из получаемого строкового представления исходного кода.

2. *Reduce*: главный узел получает результаты рабочих узлов, производит последовательное объединение множеств ребер, так как данные множества являются непересекающимися. Роль узла могут выполнять как потоки, если вычисления происходят в рамках одного физического сервера, так и отдельные физические сервера, если речь идет о распределенных вычислениях.

Вычислительные эксперименты по распараллеливанию проводились на машине, обладающей следующими характеристиками:

- процессор: 2,7 GHz Intel Core i5;
- память: 16 GB 1867 MHz DDR3;
- использовался *NodeJS v12.4.0*;
- для работы с потоками использовался встроенный модуль *Worker Threads1*. Характеристики программного кода, для которого строился граф: использованные системы модулей *ESM*.

Таблица 2 - Результаты экспериментов в случае параллельной реализации алгоритма

Количество *ESM* модулей в программе	**Время работы, с 1 поток**	**Время работы, с 2 потока**	**Время работы, с 4 потока**
1000	87	64	67
2000	219	122	92
3000	347	182	112

Полученные результаты показывают, что реализация алгоритма через большое число параллельных потоков не всегда выгодна с точки зрения временных характеристик. Для программ, состоящих из порядка 1000 модулей, увеличение числа параллельных потоков до четырех привело к увеличению (ухудшению) времени работы по сравнению с двух потоковой реализацией (64 с и 67 с соответственно). Для программ, состоящих из порядка 2000 модулей, время выполнения сократилось в 1,7 раза для распараллеливания двумя потоками, а вот увеличение числа параллельных

потоков до четырех улучшило время относительно двух параллельной версии незначительно, лишь на 24 %. Для 3000 модулей процесс распараллеливания сократил время для двух потоков в 1,9 и дальнейшее увеличение числа потоков до четырех привело улучшению времени лишь на 38 %.

Следовательно, увеличение числа параллельных потоков для реализации предлагаемого алгоритма оправданно лишь в случае большого числа модулей *ESM*.

5 ЗАДАЧА ПОИСКА ЦИКЛИЧЕСКИХ ЗАВИСИМОСТЕЙ В ГРАФЕ

Стоит отметить, что *Node.js* не поддерживает автоматическое разрешение циклических зависимостей, а также является не компилируемым. Поэтому выявить ошибку циклической зависимости можно только во время работы самого приложения, что может являться узким местом в стабильности программного обеспечения. Решим задачу нахождения циклических зависимостей в *AST*.

5.1 Постановка задачи и описание алгоритма

Необходимо найти циклы или подтвердить их отсутствие в построенном графе зависимостей модулей.

Задача сводится к поиску цикла в ориентированном графе, для ее решения был использован алгоритм обхода графа в ширину *BFS* (breadth-first search) с запоминанием посещенных вершин. Для визуализации построенных графов была использована библиотека *graphviz*.

5.2 Результаты

Рассмотрим простейший пример циклической зависимости. Исходный проект состоит из 4 файлов:

```
File: a.js
require(“./b.js”);
File: b.js
require(“./c.js”);
File: c.js
require(“./a.js”);
File: d.js
require(“./a.js”);
```

Построенный граф (рисунок 2):

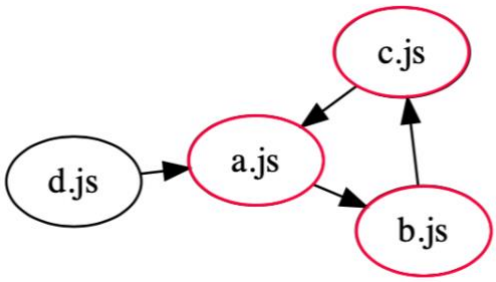


Рисунок 2 - Иллюстрация циклических зависимостей модулей

Красным визуально отмечаются узлы в построенном графе зависимостей модулей, отвечающие за модули, участвующие в циклической зависимости. Консольный вывод программы в таком случае (рисунок 3):

```
Кол-во вершин: 4
Кол-во ребер: 4
Время выполнения: 451мс
Циклические зависимости:
a.js -> b.js -> c.js
```

Рисунок 3 - Консольный вывод программы

Заключение

В курсовой работе был рассмотрен метод представления программного кода, использующего модульную структуру, в виде графа зависимостей модулей программы на основе построения и анализа абстрактных семантических деревьев. Проведен ряд вычислительных экспериментов, реализующих предлагаемые алгоритмы. Результаты показали улучшение временных метрик построения графа зависимостей программ, состоящих из *ESM*-модулей по отношению к программам, состоящим, как из *ESM*, так и *Common.JS*-модулей. Это говорит об эффективности предлагаемого алгоритма построения дерева на основе преобразований модульной структуры программы. Было проведено исследование параллельной реализации алгоритма. Можно достоверно утверждать, что реализация алгоритма даже через два параллельных потока приводит к более эффективной работе, как показано на таблице 2. Однако, говорить о том, что большее, чем два, число параллельных потоков всегда дает больший выигрыш по времени реализации неправильно. В дальнейших исследованиях предполагается использовать алгоритм построения графа, как составную часть алгоритма по оптимизации отбора регрессионных тестов из исходного множества для их прогона на измененной части программы.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Wagner, J. Reduce JavaScript Payloads with Tree Shaking / J. Wagner, 2018. – Режим доступа: https://developers.google.com/web/fundamentals/performance/optimizing-javascript/tree-shaking/ (дата обращения 27.04.2021).
2. Srinuvasu, M.A. Class-Oriented Model Graph Design Based on Abstract Syntax Tree / M.A. Srinuvasu // International Journal of computer sciences and engineering. – 2016. – №2. – Т. 7. – С. 157-168.
3. Breslav, A. DSL development based on target meta-models. Using AST transformations for automating semantic analysis in a textual DSL framework / A. Breslav, 2008. – Режим доступа: https://arxiv.org/ftp/arxiv/papers/0801/0801.1219.pdf (дата обращения 27.04.2021).
4. Сидорова, Е.В. Динамическое тестирование программного обеспечения / Е.В. Сидорова. – Нижний Новгород: НГТУ им. Р.Е. Алексеева, 2019. – 83 с.
5. Алексеев, В. Е. Графы и алгоритмы. Структуры данных. Модели вычислений / В.Е. Алексеев, В.А. Таланов. - М.: Бином. Лаборатория знаний, Интернет-университет информационных технологий, 2014. – 291 с
6. Орлов, С. А. Программная инженерия. Учебник / С.А. Орлов. - М.: Питер, 2016. – 640 с
7. Программная платформа NODE.JS [Электронный ресурс]. – Режим доступа: https://crispersoft.com/ru/node-js/ (Дата обращения: 27.04.2021)
8. Aydin, Olgun R web scraping quick start guide / Aydin, Olgun // Packt publishing limited. – 2018. – С. 114
9. Семакин, И. Г. Основы алгоритмизации и программирования / И.Г. Семакин, А.П. Шестаков. - М.: Academia, 2013. – 400 с.
10. Ивутин А.Н., Ларкин Е.В. Прогнозирование времени выполнения алгоритма // Известия Тульского государственного университета. Технические науки. Тула: Изд-во ТулГУ, 2013. Вып. 3. С. 301 – 315.

11. Богаченко Н.Ф., Файзуллин Р.Т. Автоматы, грамматики, алгоритмы. Омск: Изд-во «Наследие. Диалог-Сибирь», 2006. 145 с.

12. Дараган Е.И. Метод выявления информационных связей в программном обеспечении // Молодой ученый. 2012. No12. С. 125-13