МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ  
УЧРЕЖДЕНИЕ ОБРАЗОВАНИЯ «ВИТЕБСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИМЕНИ П.М. МАШЕРОВА»

Факультет математики и информационных технологий  
Кафедра информатики и информационных технологий

КУРСОВАЯ РАБОТА  
по дисциплине специализации

Оптимизация HTML-верстки

Бакштай Виталий Викторович,

студент 4 курса, группы 41

Руководитель:

Оганджанян Ольга Петровна,

cтарший преподаватель кафедры ИиИТ

Витебск, 2017

Содержание

[Введение 3](#_Toc502063435)

[1 Теоретическая часть 4](#_Toc502063436)

[1.1 Постановка задачи 4](#_Toc502063437)

[1.2 Терминология 4](#_Toc502063438)

[1.3 Antlr – краткие сведения 5](#_Toc502063439)

[1.4 Обход AST 5](#_Toc502063440)

[1.5 Visitor и Listener 6](#_Toc502063441)

[1.6 Построение сокращенных имен 6](#_Toc502063442)

[1.7 Поиска имен идентификаторов и классов 7](#_Toc502063443)

[1.7.1 HTML 7](#_Toc502063444)

[1.7.2 CSS 7](#_Toc502063445)

[1.7.3 JavaScript 7](#_Toc502063446)

[1.8 Алгоритм работы приложения 8](#_Toc502063447)

[1.9 Исключительные ситуации при разборе HTML 8](#_Toc502063448)

[2 Практическая часть 9](#_Toc502063449)

[2.1 Antlr в Visual Studio 9](#_Toc502063450)

[2.2 Модуль генерации и хранения сокращенных имен 9](#_Toc502063451)

[2.2.1 Реализация алгоритма генерации имен 10](#_Toc502063452)

[2.2.2 Реализация фабрики сокращенных имен 12](#_Toc502063453)

[2.3 Пример использования фреймворка Antlr 13](#_Toc502063454)

[2.4 Модуль редактора файлов. 13](#_Toc502063455)

[2.4.1 Реализация фабрики редакторов 14](#_Toc502063456)

[2.4.2 Редактор HTML 16](#_Toc502063457)

[2.4.3 Редактор CSS 17](#_Toc502063458)

[2.4.4 Редактор JS 17](#_Toc502063459)

[2.5 Модуль обработки HTML верстки 18](#_Toc502063460)

[Заключение 20](#_Toc502063461)

[Список использованных источников 21](#_Toc502063462)

[Приложение А 22](#_Toc502063463)

[Приложение Б 23](#_Toc502063464)

# Введение

В наши дни на просторах интернета практический каждый сайт написан с использованием таких языков, как HTML, CSS, JS. Но не каждый разработчик задумывается над оптимальным написанием идентификаторов и имен классов в этих файлах. Да это и не логично на этапе разработке сайта, так как давать короткие и не говорящие имена не удобно. Но на момент размещения сайта на серверах данная информация не несет никакого смысла. Отсюда на лицо проблема с неоправданно большим размером загружаемых файлов для отображения страниц в браузере. Решение данной проблемы имеет важное теоретическое и практическое значение, так как чем меньше будет размер загружаемых файлов, тем быстрее они будут отображаться для пользователей. Поисковые системы также присваивают сайтам с меньшей скоростью загрузки более высокий ранг в списке выдачи для поискового запроса при прочих равных параметрах. На некоторых серверах (хостингах) ограничен размер хранимых данных, что также является важной проблемой.

Объект: HTML-верстка представляющая набор файлов CSS, JS, HTML.

Целью данной курсовой работы является задача оптимизации HTML-верстки представляющая набор файлов CSS, JS, HTML. Основное внимание будет уделено именованию идентификаторов и классов.

Для достижения данной цели возникают следующие задачи:

* лексический анализ файлов HTML, CSS, JS;
* построение AST (abstract syntax tree) для каждого типа файлов;
* обработка полученных AST;
* построение алгоритма выявления и сокращение имен идентификаторов и классов.

# Теоретическая часть

## Постановка задачи

Имеется HTML-верстка. Целью данной курсовой работы является задача оптимизации данной верстки представляющая набор файлов CSS, JS, HTML. А конкретнее, это сокращение именования имен идентификаторов и классов, используемых в этих трех типах файлах.

Чтобы произвести нужную оптимизацию, требуется считывать последовательно из набора файлы и разобрать их. То есть найти все упоминания имен идентификаторов и классов. После разбора необходимо сократить все найденные именования.

## Терминология

Парсинг — процесс преобразования исходного кода в структурированный вид. Типичный парсер представляет собой комбинацию лексера и парсера. Лексер группирует символы исходного кода в значащие последовательности, которые называются лексемами. После этого определяется тип лексемы (идентификатор, число, строка и т.п.). Токеном называется совокупность значения лексемы и ее типа. В примере на рисунке 1.2.1 токенами являются “*sp”, “=”, “100”*. Парсер же из потока токенов строит связную древовидную структуру, которая называется деревом разбора. В данном случае “*assign”* является одним из узлов дерева. Абстрактное синтаксическое дерево или AST — дерево разбора на более высоком уровне, из которого удалены не значимые токены, такие как скобки, запятые. Однако существуют парсеры, в которых шаг лексирования и разбора совмещены.

Для описания различных узлов AST используются правила. Объединение всех правил называют грамматикой языка. Существуют инструменты, генерирующие код под определенную платформу для разбора языков на основе грамматик. Они называются генераторами парсеров. Например, ANTLR, Bison, Coco/R [4].

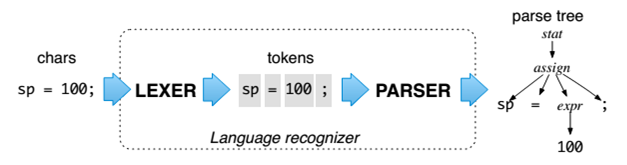


Рисунок 1.2.1 – Пример парсинга

## Antlr – краткие сведения

Для разбора файлов было решено воспользоваться фрэймворком Antlr (ANother Tool for Language Recognition – “ещё одно средство распознавания языков“). Одна из основных причин выбора, это наличие на сайте разработчика готовых файлов описания грамматики для большинства используемых языков. ANTLR позволяет создавать лексические и синтаксические анализаторы на различных языках (Java, C, C++, Python, C#, ActionScript, JavaScript, PHP) на основе грамматик [4].

Процесс создания анализатора можно описать следующим образом:

* создание грамматики для лексического анализатора
* создание грамматики для синтаксического анализатора
* отладка грамматики
* генерация классов лексического и синтаксического анализатора
* компиляция классов лексера(лексического анализатора) и парсера(синтаксического анализатора)
* использование классов для обработки текста

## Обход AST

Как известно, парсер преобразует исходный код в дерево разбора (в дерево, в котором убраны все незначимые токены), называемом AST. Существуют различные способы обработать такое дерево. Простейший — заключается в обработке дерева с помощью рекурсивного обхода потомков в глубину. Однако данный способ применим только для совсем простых случаев, в котором существует немного типов узлов и логика обработки простая. В остальных случаях, необходимо выносить логику обработки каждого типа узла в отдельные методы. Это осуществляется с помощью двух типовых подходов (паттернов проектирования): Visitor и Listener [4].

## Visitor и Listener

В Visitor для посещения потомков узла необходимо вручную вызывать методы обхода дочерних узлов. При этом если родитель имеет три потомка, и вызвать методы только для двух узлов, то часть поддерева вообще не будет обработана. В Listener же методы посещения всех потомков вызываются автоматически. В Listener существует метод, вызываемый в начале посещения узла (enterNode) и метод, вызываемый после посещения узла (exitNode). Методы Visitor, в отличие от Listener, могут возвращать объекты и даже могут быть типизированными, т.е. при объявлении \*Visitor каждый метод Visit, будет возвращать объект AstNode, который в нашем случае является общим предком для всех остальных узлов унифицированного AST.

Таким образом, код преобразования дерева с помощью Visitor получается более функциональным и лаконичным за счет того, что в нем нет необходимости хранить информацию о посещенных узлах [4].

## Построение сокращенных имен

Имена идентификаторов и классов могут начинаться только с латинских букв строчных либо прописных, а последующие символы могу содержать цифр. Так же имена с прописными буквами и строчными считаются, как два разных имени. Следовательно, генерация имен будет заключаться в последовательном переборе символов. При переборе всех символов в k-ой позиции будет произведен переход к следующему символу в k+1-ой позиции, а в текущей позиции переход к первому символу. На первую позицию налаживается ограничение, оговоренное выше.

Пусть имеется словарь, сопоставляющий некоторому числу символ.

Пусть имеется массив, содержащий набор индексов символов, представляющий текущее имя.

Алгоритм:

1. увеличить первый индекс символа на 1;
2. k=0, где k – номер текущего индекса в массиве;
3. проверить на выход k-ого индекса за границу доступных символов, если выход есть, то перейти к Шагу 4, иначе к Шагу 7;
4. обнулить k-ый индекс;
5. увеличить на единицу значение k;
6. выполнить Шаг 3;
7. преобразовать текущий массив в строчное представление используя ранее описанный словарь символов.

Выполняя данный алгоритм на одном и том же массиве можно получать не повторяющиеся имена с минимальной длинной.

## Поиска имен идентификаторов и классов

### HTML

В HTML идентификаторы хранятся в значении атрибута “id”, а имена классов в значении атрибута “class”. Имена классов записаны через пробел, поэтому их легко получить, разделив строку по пробельному символу.

### CSS

В CSS идентификаторы могут записывать в селекторах. А записываются в виде строки, где первый символ это “#”, а остальная часть, это название идентификатор. С классами аналогичная ситуация только первым символом является “.”.

### JavaScript

В JavaScript есть множество способов обращения к DOM объектам страницы. Примеры функций:

* getElementById(name:string) – получение узла с идентификатором name.
* getElementsByClassName(names:string) – получение списка узлов с именами классов names, где в names имена записаны через пробел.

Так как очень сложно отследить какая именно строка была передана в данные функции, было принято решение сокращать литеральные строки, полностью совпадающие с именами ранее сокращенных имен. В большинстве случаев разработчик передает данные в эти функции именно литералом.

Так же в JS некоторые разработчики пишут собственные функции обращения к DOM элементам HTML. И часто в эти функции передаются строки символом ‘#’ перед именем идентификатора или ‘.’ перед именем класса. Описанный случай в данной курсовой работе рассматриваться не будет, но будет учтен в дальнейшей разработке.

## Алгоритм работы приложения

Полученное в результате приложение должно делать копию существующей директории. Данный алгоритм будет рекурсивно просматривать директории, при этом оптимизируя нужные файлы. На вход подается начальная директория.

Алгоритм работы:

1. просмотреть все вложенные директории и выполнить этот же алгоритм для каждой найденной директории;
2. найти следующий файл, если нет, то перейти к Шагу 8;
3. определить тип файла;
4. если это файл типа “.js” то поместить его в очередь;
5. изменить все имена идентификаторов и классов;
6. записать измененный файл в новую директорию;
7. перейти к Шагу 2;
8. взять файл из очереди;
9. изменить имена на основании найденных имен;
10. выполнить Шаг 6;
11. перейти к шагу 8.

## Исключительные ситуации при разборе HTML

Известно, что в документе HTML можно делать вставки CSS используя тег “<style></style>”, а также вставки JS используя тег “<script></script>”. Поэтому при обработке HTML документа необходимо находить данные теги и содержимое обрабатывать содержимое так же, как и при работе с файлами CSS и JS.

# Практическая часть

Для решения поставленных задач был выбран язык программирования C#, а средой разработки Visual Studio.

## Antlr в Visual Studio

Так как средой разработки была выбрана Visual Studio необходимо автоматизировать процесс генерации классов лексера и парсера при изменении граматики. Данную возможность предлагает Visual Studio. Для работы с Antlr достаточно установить через менеджер управлением пакетов NuGet пакет “Antlr4”. Это возможно сделать через графический интерфейс NuGet или выполнить команду из листинга 2.1.1.

Листинг 2.1.1 – Команда для установки пакетов Antlr

Install-Package Antlr4

Установив данный пакет в окне добавления элемента в проект появится дополнительные пункты, связанные с Antlr, такие как “Antlr 4 Lexer”, “Antlr 4 Parser”, “Antlr 4 Combine”. Последний из представленных создает файл, описывающий одновременно и лексер и парсер грамматики.

После создания одного из представленных файлов Visual Studio будет используя установленный пакет компилировать файл в классы реализующих лексический анализ, это файл \*Lexer, и синтаксический анализ, это файл \*Parser, где \* это название файлов, описывающих граматику. Данная компиляция будет происходить автоматически при сохранениях файла граматики.

## Модуль генерации и хранения сокращенных имен

Для реализации данного модуля была разработана следующая диаграмма классов.

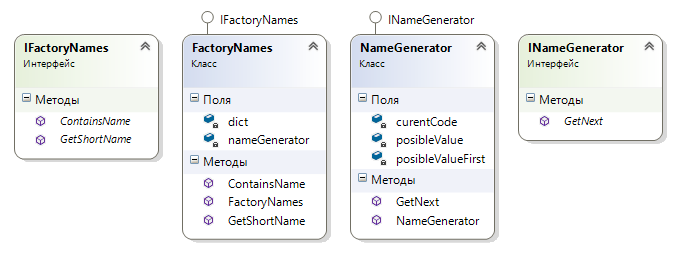


Рисунок 2.2.1 – Диаграмма классов

Используя такой подход классы использующие интерфейсы, описанные в диаграмме, не зависят от конкретной реализации.

### Реализация алгоритма генерации имен

На основании алгоритма, описанного в пункте 1.6 был реализован класс NameGenerator, представленный в листингах 2.2.1.1 - 2.2.1.3.

Листинг 2.2.1.1 – Реализация класса NameGenerator

class NameGenerator : INameGenerator {

//возможные значения для первого символа

private char[] posibleValueFirst;

//возможные значения для остальных символов

private char[] posibleValue;

private IList<int> curentCode;

public NameGenerator() {

//реализация конструктора см Листинг 2.2.1.2

}

//метод получения следующего имени

public string GetNext() {

//реализация конструктора см Листинг 2.2.1.3

}

}

Листинг 2.2.1.2 – Реализация конструктора класса NameGenerator

public NameGenerator() {

//заполнение массивов возможных значений

curentCode = new List<int>();

var tempVal = new List<char>();

for (char i = 'a'; i <= 'z'; i++) {

tempVal.Add(i);

}

for (char i = 'A'; i <= 'Z'; i++) {

tempVal.Add(i);

}

posibleValueFirst = tempVal.ToArray();

for (char i = '0'; i <= '9'; i++) {

tempVal.Add(i);

}

posibleValue = tempVal.ToArray();

//начальное значение

curentCode.Add(0);

}

Листинг 2.2.1.3 – Реализация метода GetNext

public string GetNext() {

StringBuilder sb = new StringBuilder();

int i = 0;

sb.Append(posibleValueFirst[curentCode[i]]);

i++;

while (i < curentCode.Count) {

sb.Append(posibleValue[curentCode[i]]);

i++;

}

int cur = 0;

if (curentCode[cur] + 1 == posibleValueFirst.Length) {

curentCode[cur] = 0;

cur++;

}

while (cur < curentCode.Count

&& curentCode[cur] + 1 == posibleValue.Length) {

curentCode[cur] = 0;

cur++;

}

if (cur == curentCode.Count) {

curentCode.Add(-1);

}

curentCode[cur]++;

return sb.ToString();

}

Класс NameGenerator реализует интерфес INameGenerator описанный в листинге 2.2.1.4.

Листинг 2.2.1.4 – Определение интерфейса INameGenerator

public INameGenerator () {

string GetNext();

}

Данный интерфейс имеет метод GetNext(). При вызове которого будет возвращаться следующее сгенерированное сокращенное название.

Ниже приведен пример имен, которые генерирует класс.

Листинг 2.2.1.5 – Пример имен

[a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u,v,w,x,y,z,A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P,Q,R,S,T,U,V,W,X,Y,Z,aa,ba,ca,da,ea,fa,ga,ha,ia,ja,ka,la,ma,na,oa,pa,qa,ra,sa,ta,ua,va,wa,xa,ya,za,Aa,Ba,Ca,Da,Ea,Fa,Ga,Ha,Ia,Ja,Ka,La,Ma,Na,Oa,Pa,Qa,Ra,Sa,Ta,Ua,Va,Wa,Xa,Ya,Za,ab,bb,cb]

### Реализация фабрики сокращенных имен

После того как будут найдены нужные имена, необходимо где-то хранить новые имена идентификаторов и классов. Для данной задачи был реализован класс FactoryNames, который описан в листинге 2.2.2.1, реализующий интерфейс IFactoryNames описанный в листинге 2.2.2.2.

Листинг 2.2.2.1- Реализация класса FactoryNames

public class FactoryNames : IFactoryNames {

private Dictionary<string, string> dict

= new Dictionary<string, string>();

private INameGenerator nameGenerator;

public FactoryNames(INameGenerator nameGenerator) {

this.nameGenerator = nameGenerator;

}

public string GetShortName(string name) {

if (dict.ContainsKey(name)) {

return dict[name];

}

string genValue = nameGenerator.GetNext();

dict.Add(name, genValue);

return genValue;

}

public bool ContainsName(string name) {

return dict.ContainsKey(name);

}

}

Листинг 2.2.2.2 – Определение интерфейса IFactoryNames

public interface IFactoryNames {

string GetShortName(string name);

bool ContainsName(string name);

}

Данный интерфейс имеется два метода.

* GetShortName(name) - возвращает сокращенное название;
* ContainsName(name) - проверяет запрашивалось ли сокращенное имя для данной строки.

В классе FactoryName метод GetShortName(name) работает следующим образом. Если данный метод не выполнялся для запрашиваемого имени, то генерируется новое, используя реализацию интерфейса INameGenerator, иначе возвращается значения из внутреннего словаря.

## Пример использования фреймворка Antlr

В Листинге 2.3.1 показан пример использования фреймворка Antlr, используя сгенерированные лексер и парсер для языка JavaScript.

Листинг 2.3.1 – Пример использования Antlr

//получение содержимое файла

string input = File.ReadAllText(fileName);

//получения потока символов

AntlrInputStream inputStream = new AntlrInputStream(input);

//разбор потока символов на лексемы

var lexerJs = new JsLexer(inputStream);

//получение потока токенов

CommonTokenStream commonTokenStream;

commonTokenStream = new CommonTokenStream(lexerJs);

//составление из лексем AST

var parserJs = new JsParser(commonTokenStream);

//получения корня AST

var root = parserJs.program();

//Создание объект класса JsVisitor который

//наследуется от JsParserBaseVisitor

var visitor = new JsVisitor();

//запуск обхода всего дерева

visitor.Visit(root);

## Модуль редактора файлов.

Для обработки файлов была выбрана следующая архитектура.

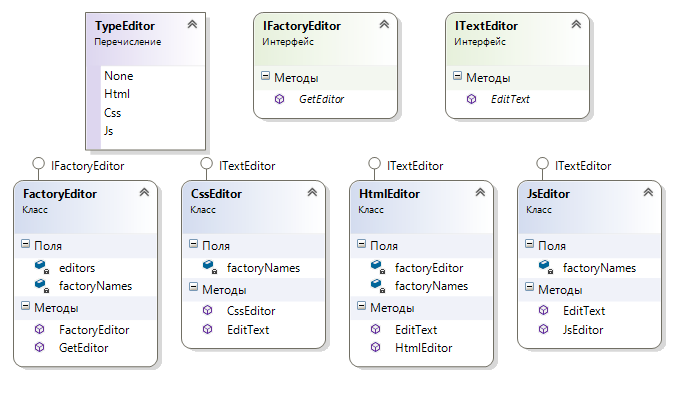


Рисунок 2.4.1 – Диграмма классов модуля обработки файлов.

Как видно из диаграммы классов класс HtmlEditor содержит ссылку на IFactoryeditor, так как в момент обработки HTML может потребоваться обрабатывать вставки кода JS и CSS.

### Реализация фабрики редакторов

В зависимости от типа нужного редактора, фабрика вернет реализацию интерфейса ITextEditor.

Данный класс использует перечисление, описанное в Листинге 2.4.1.1.

Листинг 2.4.1.1 – Типы поддерживаемых редакторов файла

enum TypeEditor {

None = 0,

Html = 1,

Css = 2,

Js = 3

}

Листинг 2.4.1.2 – Реализация фабрики редакторов

class FactoryEditor : IFactoryEditor {

Dictionary<TypeEditor, ITextEditor> editors;

IFactoryNames factoryNames;

public FactoryEditor(IFactoryNames factoryNames) {

this.factoryNames = factoryNames;

editors = new Dictionary<TypeEditor, ITextEditor>();

editors.Add(

TypeEditor.Css,

new CssEditor(factoryNames)

);

editors.Add(

TypeEditor.Html,

new HtmlEditor(factoryNames, this)

);

editors.Add(

TypeEditor.Js,

new JsEditor(factoryNames)

);

}

public ITextEditor GetEditor(int type) {

TypeEditor typeEditor = (TypeEditor)type;

if (editors.ContainsKey(typeEditor)) {

return editors[typeEditor];

}

return null;

}

}

### Редактор HTML

Листинг 2.4.1 – Класс обработки HTML документа

class HtmlEditor : ITextEditor {

IFactoryNames factoryNames;

IFactoryEditor factoryEditor;

public HtmlEditor(IFactoryNames factoryNames

,IFactoryEditor factoryEditor) {

this.factoryNames = factoryNames;

this.factoryEditor = factoryEditor;

}

public string EditText(string text) {

AntlrInputStream inputStream = new AntlrInputStream(text);

ITokenSource lexer;

IVisitorTree visitor;

IChangeTokenSource editorTokens;

lexer = new HtmlLexer(inputStream);

visitor = new HtmlVisitorChangeAtributeValue(

factoryNames,lexer.TokenFactory);

editorTokens = new BaseHtmlEditTokens(visitor);

lexer = editorTokens.Edit(lexer);

visitor = new HtmlVisitorEditorScriptTag(

factoryNames,lexer.TokenFactory,factoryEditor);

editorTokens = new BaseHtmlEditTokens(visitor);

lexer = editorTokens.Edit(lexer);

visitor = new HtmlVisitorEditStyleTag(

factoryNames,lexer.TokenFactory,factoryEditor);

editorTokens = new BaseHtmlEditTokens(visitor);

lexer = editorTokens.Edit(lexer);

visitor = new HtmlVisitorAddSpace(lexer.TokenFactory);

editorTokens = new BaseHtmlEditTokens(visitor);

lexer = editorTokens.Edit(lexer);

CommonTokenStream cs = new CommonTokenStream(lexer);

cs.Fill();

return cs.GetText();

}

}

### Редактор CSS

Листинг 2.4.1 – Класс обработки CSS документа

class CssEditor : ITextEditor {

IFactoryNames factoryNames;

public CssEditor(IFactoryNames factoryNames) {

this.factoryNames = factoryNames;

}

public string EditText(string text) {

AntlrInputStream inputStream = new AntlrInputStream(text);

ITokenSource lexer;

IVisitorTree visitor;

IChangeTokenSource editorTokens;

lexer = new CssLexer(inputStream);

visitor = new CssVisitorEditNameSelector(

factoryNames,

lexer.TokenFactory

);

editorTokens = new BaseCssEditTokens(visitor);

lexer = editorTokens.Edit(lexer);

CommonTokenStream cs = new CommonTokenStream(lexer);

cs.Fill();

return cs.GetText();

}

}

### Редактор JS

Листинг 2.4.1 – Класс обработки JS документа

class JsEditor : ITextEditor {

IFactoryNames factoryNames;

public JsEditor(IFactoryNames factoryNames) {

this.factoryNames = factoryNames;

}

public string EditText(string text) {

AntlrInputStream inputStream = new AntlrInputStream(text);

ITokenSource lexer;

IVisitorTree visitor;

IChangeTokenSource editorTokens;

lexer = new JsLexer(inputStream);

visitor = new JsVisitorChangeLiteralString(

factoryNames, lexer.TokenFactory);

editorTokens = new BaseJsEditTokens(visitor);

lexer = editorTokens.Edit(lexer);

CommonTokenStream cs = new CommonTokenStream(lexer);

cs.Fill();

return cs.GetText();

}

}

## Модуль обработки HTML верстки

Класс Manager описанный в листингах 2.5.1 – 2.5.4 реализует алгоритм из пункта 1.8.

Листинг 2.5.1 – Описание класса Manager

class Manager {

IFactoryEditor factoryEditor;

//очередь для обработки JS файлов

List<Action> queue;

public Manager() {

factoryEditor = new FactoryEditor(

new FactoryNames(new NameGenerator())

);

queue = new List<Action>();

}

//стартовая точка работы класса

//pathFolder – обрабатываемая директория

public void DoRefactor(string pathFolder) {

//реализация метода в Листинге 2.5.2

}

//рекурсивная функция обработки директории

private void DoRefactor(DirectoryInfo rootFolder,

DirectoryInfo mirrorFolder) {

//реализация метода в Листинге 2.5.3

}

//определение типа файла

static public TypeEditor GetTypeEditor(string pathFile) {

//реализация метода в Листинге 2.5.4

}

}

Листинг 2.5.2 – Реализация метода DoRefactor(pathFolder)

public void DoRefactor(string pathFolder) {

DirectoryInfo root = Directory.CreateDirectory(pathFolder);

if (root.Parent == null) return;

string pathParentOfRoot = root.Parent.FullName;

string pathMirorFolder = string.Concat(

pathParentOfRoot,

"\\",

root.Name,

"\_min");

DirectoryInfo mirror = new DirectoryInfo(pathMirorFolder);

mirror.Create();

DoRefactor(root, mirror);

foreach (var item in queue) {

item();

}

queue.Clear();

}

Листинг 2.5.3 – Реализация метода DoRefactor(rootFolder,mirrorFolder)

private void DoRefactor(DirectoryInfo rootFolder,

DirectoryInfo mirrorFolder) {

foreach (var dir in rootFolder.GetDirectories()) {

string pathNextMirror = mirrorFolder.FullName + "\\" + dir.Name;

DirectoryInfo nextMirror = new DirectoryInfo(pathNextMirror);

nextMirror.Create();

DoRefactor(dir, nextMirror);

}

foreach (var file in rootFolder.GetFiles()) {

TypeEditor type = GetTypeEditor(file.Name);

var editor = factoryEditor.GetEditor((int)type);

string pathMirrorFile = mirrorFolder.FullName + "\\" + file.Name;

if (editor == null) {

file.CopyTo(pathMirrorFile);

continue;

}

Action write = () => {

Console.WriteLine(file.FullName);

string inputText = File.ReadAllText(file.FullName);

string resultText = editor.EditText(inputText);

File.WriteAllText(pathMirrorFile, resultText);

};

if (type == TypeEditor.Js) {

queue.Add(write);

} else {

write();

}

}

}

Листинг 2.5.4 – Реализация метода GetTypeEditor(pathFile)

static public TypeEditor GetTypeEditor(string pathFile) {

int indexPoint = pathFile.LastIndexOf('.');

if (indexPoint == -1) {

return TypeEditor.None;

}

string typeS = pathFile.Substring(indexPoint + 1);

switch (typeS.ToUpper()) {

case "CSS":

return TypeEditor.Css;

case "HTML":

return TypeEditor.Html;

case "JS":

return TypeEditor.Js;

default:

return TypeEditor.None;

}

}

# Заключение

В ходе выполнения курсовой работы проводился анализ CSS, HTML, JS файлов. Целью курсовой работы являлась оптимизация HTML-верстки путем сокращения именования идентификаторов и классов.

В курсовой работе представлены ряд алгоритмов, позволяющих решить данные задачи.

Результатом курсовой работы является приложение, которое производит валидацию выше упомянутых файлов, анализирует и сокращает именование идентификаторов и классов.

В дальнейшем планируется ввести следующие изменения и усовершенствования приложения:

* увеличение производительности данного приложения;
* улучшение обработки JS-файлов.
* добавить дополнительные минификации с файлами.

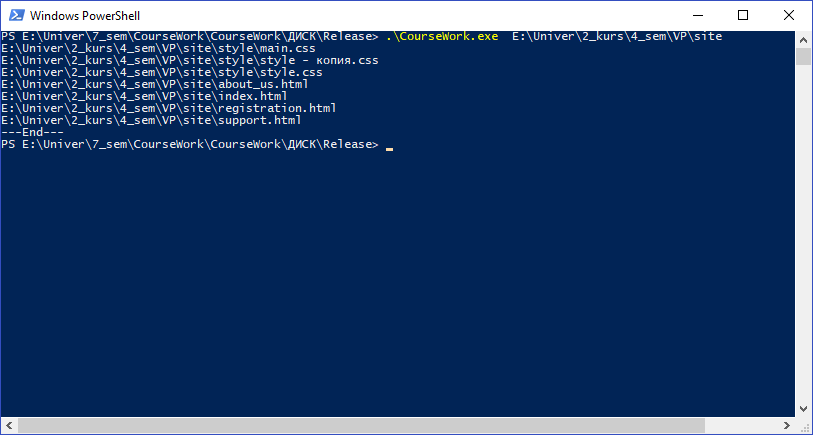
# Список использованных источников

1. Cascading Style Sheets [Электронный ресурс]. - Режим доступа: https://www.w3.org . Дата доступа: 10.12.2017.
2. Дэвид Макфарланд, CSS3: The Missing Manual 3-е издание. — 3-е. — Санкт-Петербург, 2014. — С. 608 — ISBN 978-5-496-00428-2
3. HTML5, CSS3 и JavaScript. Исчерпывающее руководство. — 4-e — Эксмо, —2014. — С. 528 — ISBN 978-5-699-67603-3
4. Кочуркин И. Обработка древовидных структур и унифицированное AST [Электронный ресурс]. – Режим доступа: <https://www.securitylab.ru/analytics/481706.php> . Дата доступа 26.12.2017.
5. JavaScript. Подробное руководство. — 1-е — Символ-Плюс, — 2008.—

С. 992 — ISBN 978-5-93286-103-5

# Приложение А

Пример запуска приложения



# Приложение Б

Содержимое дискового носителя

1. \Source - файлы исходного кода проекта.
2. \Release - разработанное приложение
3. Записка.docx - пояснительная записка к курсовой работе.