МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ  
УЧРЕЖДЕНИЕ ОБРАЗОВАНИЯ «ВИТЕБСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИМЕНИ П.М. МАШЕРОВА»  
  
Факультет математики и информационных технологий  
Кафедра прикладного и системного программирования

КУРСОВАЯ РАБОТА

по дисциплине специализации

**МИНИФИКАЦИЯ CSS-ФАЙЛОВ**

Бакштай Виталий Викторович,

студент 3 курса, группы 31

Руководитель:

Сергеенко Сергей Владимирович,

старший преподаватель кафедры ПиСП

Витебск, 2016

Содержание

[Введение 3](#_Toc469962584)

[ТЕОРЕТИЧЕСКАЯ ЧАСТЬ 4](#_Toc469962585)

[1.1 Постановка задачи 4](#_Toc469962586)

[1.2 Комментарии, лишние пробелы, переходы на новую строку. 4](#_Toc469962587)

[1.3 Переопределение свойств. 5](#_Toc469962588)

[1.4 Использование сокращенных конструкций при задании свойств. 6](#_Toc469962589)

[1.5 Выгодное написание свойств. 6](#_Toc469962590)

[1.6 Алгоритм минификации CSS-файла. 7](#_Toc469962591)

[1.7 Алгоритм поиска описания селекторов. 7](#_Toc469962592)

[1.8 Алгоритм оптимизации селектора. 8](#_Toc469962593)

[1.9 Алгоритм объединения селекторов в группы. 8](#_Toc469962594)

[1.10 Алгоритм выгодного написания свойств. 9](#_Toc469962595)

[ПРАКТИЧЕСКАЯ ЧАСТЬ 10](#_Toc469962596)

[2.1 Описание работы приложения. 10](#_Toc469962597)

[2.3 Классы хранения селекторов. 10](#_Toc469962598)

[2.4 Лексический анализ CSS-файла. 12](#_Toc469962599)

[2.4.1 Поиск описания селекторов 12](#_Toc469962600)

[2.4.2 Удаление комментариев 13](#_Toc469962601)

[2.4.3 Реализация считывания селекторов . 13](#_Toc469962602)

[2.4.4 Реализация сохранения селекторов. 15](#_Toc469962603)

[2.6 Реализация минификации Storage. 18](#_Toc469962604)

[2.6.1 Реализация объединения селекторов в группы. 18](#_Toc469962605)

[2.6.2 Реализация оптимизации селекторов. 19](#_Toc469962606)

[Сравнительная характеристика приложения. 20](#_Toc469962607)

[Заключение 22](#_Toc469962608)

[Список использованной литературы 23](#_Toc469962609)

[Приложение 24](#_Toc469962610)

[Содержимое дискового носителя 24](#_Toc469962611)

# Введение

В наши дни на просторах интернета уже не встретишь такого сайта, который не использует каскадные таблицы для оформления сайта. Но не каждый разработчик задумывается над оптимальным написанием стилей в CSS-файлах. 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.

Причинами больших размеров CSS-файлов являются:

* комментарии, лишние пробелы, переходы на новую строку;
* переопределение однотипных свойств в селекторе;
* использование избыточных конструкций при задании свойств;
* при большом количестве селекторов сложно уследить за наиболее оптимальным написанием свойств.

Объект: каскадные таблицы стилей.

Целью данной курсовой работы является задача минификации CSS-файла в байтовом размере.

Для достижения данной цели возникают следующие задачи :

* анализ CSS-файла;
* выявление не оптимально определенных стилей;
* построение алгоритма минификации CSS-файла;
* разработка приложение, которое реализует алгоритм минификации.

# Теоретическая часть

## Постановка задачи

Имеется CSS-файл. Целью курсовой работы является минификация данного файла в байтовом размере.

Чтобы минифицировать CSS-файл, требуется считать его и разобрать. То есть найти все селектора и их свойства в исходном файле. После разбора каждый селектор следует рассмотреть отдельно и по возможности соответствующим алгоритмом.

Приведем различные ситуации, которые требуют оптимизации:

* комментарии, лишние пробелы, переходы на новую строку;
* переопределение свойств;
* использование сокращенных конструкций при задании свойств;
* выгодное написание свойств.

## 1.2 Комментарии, лишние пробелы, переходы на новую строку.

Каждый разработчик форматирует написанный код так, как ему удобно его читать. В большинстве случаев шаблон описывается, как показано в Листинге 1.2.1

Листинг 1.2.1 Пример описание свойств селектора

div {

color: *black*;

padding: *10px*;

…

}

В большинстве случаев разработчик использует табуляции, переходы на новую строку, повторяющиеся пробелы. Все эти элементы «мертвый груз», который не требуется для корректного применения стилей к странице. Отсюда следует, что необходимо удалить их там, где это возможно сделать в исходном файле.

Каждый разработчик пишет комментарии при написании стилей для html-страницы. Но комментарии нужны только для того, чтобы повысить читаемость CSS-файла, а на отображение страницы они никакой роли не играют. Следовательно необходимо удалить все комментарии в исходном файле.

Также часто встречается, разработчики ставят закрывающий символ свойства в последнем свойстве селектора.

Листинг 1.2.2 Пример с лишним закрывающим символом свойства

div {

color: *black*;

padding: *10px*;

}

Для парсера CSS браузера этот селектор будет обрабатываться также, как и в Листинге 1.2.3

Листинг 1.2.3 Пример без закрывающего символа свойства

div {

color: *black*;

padding: *10px*

}

Отсюда можно сделать вывод, что этот символ можно опустить в последнем свойстве селектора.

## **1.3 Переопределение свойств**.

Встречаются так же ситуации, когда переопределяются уже написанные свойства для селекторов.

Листинг 1.3.1 Пример переопределения свойства селектора

div {

padding-top:*10px*

}

*/\* какой-то код \*/*

div {

padding-top:*20px*

}

Видно, что первую запись можно опустить. Отсюда возникает еще одна задача: убрать свойства селекторов, которые были переопределены.

Также, зачастую, некоторые свойства указываются по частям тогда, когда можно было использовать более краткую запись.

Листинг 1.3.2 - Пример задания свойства

div {

padding:*10px 20px 30px;*

padding-left:*40px;*

}

*/\*менее часто\*/*

div {

padding-top:*10px;*

padding-right:20px;

padding-bottom:*30px*;

padding-left:*40px*

}

Данные свойства селектора можно записать с помощью одного свойства (см Листинг 1.3.3)

Листинг 1.3.3 - Пример задания свойства

div {

padding:*10px* *20px* *30px* *40px*

}

Следовательно, нужно учитывать это при разработке приложения.

## 1.4 Использование сокращенных конструкций при задании свойств.

Цвет разработчики задают так, как им удобно. Это словами (red, green, black и др.), функциями rgb ( rgb(255,0,0), rgb(255,255,255) и др.), hsl (hsl(10%,20%30%) и др.), шестнадцатеричным кодом (#d3d3d3,#f00 и др). Но это не всегда выгодно в плане занимаемого места в файле. Так, например, цвет заданный словом mediumspringgreen выгоднее записать шестнадцатеричным кодом #00fa9a. Данную особенность также нужно учесть в приложении.

## 1.5 Выгодное написание свойств.

При большом количестве селекторов сложно уследить за наиболее оптимальным написанием свойств, так как одинаковые свойства в селекторах можно объединить.

Например селектора в Листинге 1.5.1 можно объединить по свойству *padding-top: 10px* как показано в Листинге 1.5.2

Листинг 1.5.1 – Селектора с одинаковыми свойствами

**p**{

padding-top: *10px*;

color: *red*

}

**div**{

padding-top: *10px*;

color: *green*

}

Листинг 1.5.2 – Пример объединения селекторов

**p, div**{

padding-top: *10px*

}

**p**{

color: *red*

}

**div**{

color: *green*

}

## 1.6 Алгоритм минификации CSS-файла.

* считывание CSS-файла;
* поиск описания селекторов;
* минификация каждого селектора возможными способами;
* формирование из селекторов групп;
* оптимизация каждой группы отдельно;
* запись полученных данных в новый CSS-файл.

## 1.7 Алгоритм поиска описания селекторов.

Стоит учитывать, что есть блоки, в которых могут быть вложены селектора. Следовательно, каждый раз для разбора по селекторам будет браться часть файла, в котором одинаковое количество открывающих и закрывающих скобок.

В каждой части будет производиться поиск общей конструкции селектора: поиск открывающей фигурной скобки ‘{’ , поиск закрывающей фигурной скобки ‘}’. Часть до первой закрывающей скобки будет содержать имена селекторов. Разбив первую часть по запятой, получим список имен селекторов, для которых применяются свойства. Вторая часть будет содержать свойства. В этой части производим поиск общей конструкции записи свойства: поиск символа ‘:’, поиск закрывающего символа ‘;’. Часть до ‘:’ будет содержать имя свойства. Часть между символами ‘:’ и ‘;’ содержит значение свойства.

Алгоритм:

* считывание следующего блока;
* получение селектора из блока;
* разбор селектора на две части, часть, где имена и где свойства;
* получение списка имен и списка свойств.

## 1.8 Алгоритм оптимизации селектора.

После того как получили список селекторов и список свойств для них, каждый селектор нужно отдельно оптимизировать.

Оптимизация заключается в нахождение свойств, которые можно объединить в более общее свойство. Возможны два случая, когда можно объединить свойства.

* указано общее свойство и указаны более конкретные значения этого свойства;
* указаны все составляющие общего свойства.

Далее, когда определены свойства. Необходимо объединить эти свойства в том порядке, котором они расположены в селекторе. Удалить свойства, которые объединяли, и добавить новое, которое получилось после объединения.

## 1.9 Алгоритм объединения селекторов в группы.

Так как в основании алгоритма оптимального написания свойств лежит метод перебора всевозможных комбинаций селекторов, очень важно уменьшить количество селекторов для оптимизации за один раз. Таким образом нужно объединять полученные селектора в группы.

*Алгоритм объединения*. Пусть есть список селекторов, и список в котором будут помечаться те селектора, которые уже использовали для формирования групп.

1. выбирается из списка селектор, который не помечен как использованный в списке меток;
2. создание группы;
3. добавление селектора в группу;
4. пометить селектор как использованный;
5. для каждого селектора, свойства которого есть в выбранном селекторе, выполнить пункт 3;
6. выполнять пункт 1 пока не будут помечены все селектора.

Таким образом, можно выделить группы, селектора в которых есть смысл объединять.

## 1.10 Алгоритм выгодного написания свойств.

Данный алгоритм будет реализовываться методом полного перебора. Он заключается в полном переборе всевозможных объединений селекторов из группы.

*Алгоритм.* Предположим, что имеется группа селекторов (Группа 1) .

1. выбираются селектора из Группы 1;
2. находится пересечение свойств;
3. находится объединение имен селекторов;
4. формируется новый селектор, в котором список имен из пункта 3, а свойства пункт 2;
5. если полученный будет занимать меньше места, чем свойства участвующие в объединении по отдельности, то объединенный селектор записывается в новую группу;
6. из Группы 1 у селекторов, полученных из пункта 3, удаляются свойства полученных из пункта 2;
7. из Группы 1 удаляются селектора не содержащие свойств;
8. пункт 1 повторяется до тех пор, пока в Группе 1 будут селектора.

Так как временная сложность алгоритма полного перебора очень высокая и при большом количестве селекторов в группе его использовать не целесообразно. Чтобы уменьшить количество операций можно убрать заведомо неоптимальные пересечения свойств. Для этого можно построить граф. Каждая вершина это селектор, а связь между вершинами означает, что пересечение данных селектором дает не пустой селектор. Из каждой вершины находится наиболее выгодный путь, где вершины это селектора и есть те селектора, которые выбираются для пункта 1. Так же следует исключить прохождение по одному и тому же пути дважды. За одинаковый путь будет приниматься тот путь, у которого один и тоже набор селекторов.

# 2. Практическая часть

## 2.1 Описание работы приложения.

Для работы приложения указывается CSS-файл. С помощью класса FileReader указанный файл считывает и записывается в строку. Далее создается объект класса CSSTextAnalysis. Этот класс отвечает за лексический анализ CSS-файла. После объект класса CSSTextAnalysis передается в статический метод analyze(CSSTextAnalysis) метод класса CSSAnalysis. Данный метод возвращает объект класса Storage . Storage – класс хранения всех селекторов и их свойств. Далее создается объект класса CSSOptimizer, в конструктор которого передается объект класса Storage,и вызывается метод toOptimize(). Данный метод выполняет группировку свойств. Получившиеся данные записываются в файл классом FileWriter.

## 2.3 Классы хранения селекторов.

Хранение селектора реализуют классы Selector, Property, PropertiesSelector, SelectorsProperty.

Selector – класс для хранения имени селектора.

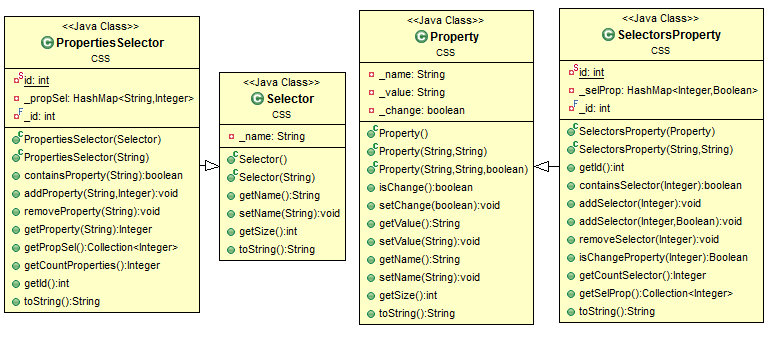
Property – класс для хранения свойства .

PropertiesSelector – хранение именн свойств и их значений для определенного селектора. Для переопределения свойств.

SelectorsProperty – хранение селекторов для свойства. Чтоб была возможность определить, можно изменять свойство для определенного селектора или нет.

Для более удобной работы каждый объект класса PropertiesSelector и SelectorsProperty имеет уникальный id, который присваивается при создании объекта автоматически.

Рис. 2.3.1 – UML диаграмма классов.

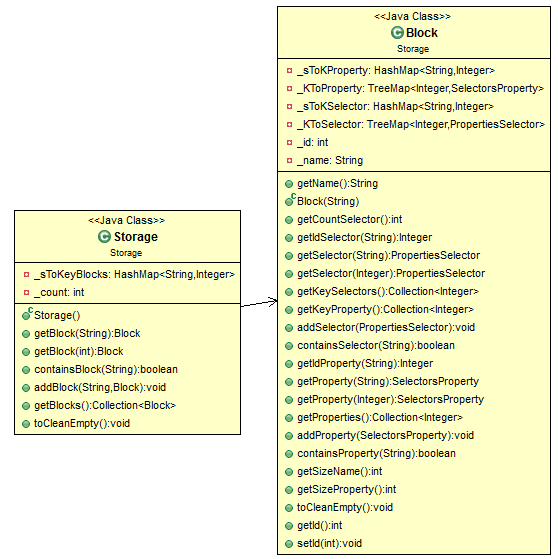


Хранение всех селекторов реализуют два класса:

Storage – хранит в себе блоки селекторов. Чтобы разделить основные стили и @media.

Block – хранит в себе HashMap объектов SelectorsProperty, где ключ это строка “имя свойства + значение свойства”. И HashMap объектов PropertiesSelector, где ключ это строка “имя селектора”.

Рис. 2.3.2 – UML диаграмма классов Storage, Block.



## **2.4 Лексический анализ CSS-файла.**

### 2.4.1 Поиск описания селекторов

Поиск будет осуществляться с помощью регулярных выражений. Сначала производится поиск описания полностью. А именно какой-то текст, открывающая фигурная скобка и закрывающая фигурная скобка. Составим регулярное выражение: \s\*([^{]+)\{\s\*([^}]\*)}\s\*. Теперь разберем: \s\* - пропуск всех пробелов, [^{]+ - название селектора к которому будут применятся свойства, [^}]\*- само описание селектора. Тут в регулярном выражении выделено две группы. Первая группа – это имена селекторов. Вторая – это свойства. Далее на основании полученных данных получим имена селекторов. Для этого тоже используем регулярное выражение ([^,{]+),? . Здесь каждый раз будет возвращаться одно из имен селекторов, для которых применяются свойства. Также из второй группы напишем регулярное выражения для получения одного свойства. Получим \s\*([^:]+):\s\*([^;!]+)\s\*(!important)?\s\*? .Здесь [^:]+ - имя свойства, [^;!]+ - значение этого свойства, так же тут отдельная группа для !important, чтобы понять можно ли изменять это свойство для селектора или нет.

### 2.4.2 Удаление комментариев

Поиск многострочного комментария можно реализовать с помощью регулярного выражения \/\\*.\*?\\*\/. На первый взгляд вопросительный знак кажется лишним, так как звездочка сама по себе допускает отсутствие символа. Но на самом деле данная последовательность включает ленивую квантификацию. Интерпретатор в данном случае начнет поиск с первого символа подстроки, пошагово наращивая длину строки поиска. Дойдя до первой последовательности, которой заканчивается регулярное выражение («\\*/» — окончание комментария), интерпретатор остановит дальнейший поиск, так как ленивая квантификация говорит ему искать совпадение минимальной длины.

После нахождения комментария можно заменить на пустую строку.

### 2.4.3 Реализация считывания селекторов .

Считывание реализует класс CSSTextAnalysis. Данный класс использует регулярные выражжения написанные в пункте 2.4.1 и пункте 2.4.2.

Реализация класса показана в Листингах 2.4.3.1 - 2.4.3.6

Листинг 2.4.3.2 – Реализация конструктора.

public CSSTextAnalysis(String text) {//text – текст файла

text = text.replaceAll("\\/\\\*.\*?\\\*\\/", "");//удаление комен

\_text = text;

}

Листинг 2.4.3.2 – Реализация метода Поиска символа.

private int indexOf(String str, int ch, int start, int end) {

int index = -1;

while (start < end) {// поиск { в промежутке

if (str.charAt(start) == ch) {

index = start;

break;

}

start++;

}

return index;

}

Листинг 2.4.3.3 – Реализация метода получения следующего блока.

public String nextBlock() {

int iStart = \_text.indexOf('{', iLast);

int iEnd = \_text.indexOf('}', iStart + 1);

String nameBlock = "default";

if (iStart != -1) {

int iMid = indexOf(\_text, '{', iStart + 1, iEnd);

if (iMid != -1) {

while (iMid != -1 && iEnd > iMid) {

iEnd = \_text.indexOf('}', iEnd + 1);

iMid = indexOf(\_text, '{', iMid + 1, iEnd);

}

Matcher name = \_nameSelector.matcher(\_text.substring(iLast, iStart));

name.find();

nameBlock = name.group();

iLast = iStart;

}

iEnd++;

\_textBloks = \_text.substring(iLast, iEnd);

iLast = iEnd;

\_matSelector = \_selector.matcher(\_textBloks);

return nameBlock;

}

return null;

}

Листинг 2.4.3.4 – Реализация метода получения следующего селектора.

public boolean nextSelector() {

boolean res = \_matSelector.find();

if (res) {

String names = \_matSelector.group(1);

names = names.trim()//

.replaceAll("\\s+", " ") // пробелы

.replaceAll("\\s?,\\s?", ","); // пробелы вокруг ,

\_matNameSelector = \_nameSelector.matcher(names);

\_matProperty = \_property.matcher(\_matSelector.group(2));

}

return res;

}

Листинг 2.4.3.5 – Реализация метода получения следующего имени селектора.

public Selector nextName() {

boolean res = \_matNameSelector.find();

if (res) {

String str = \_matNameSelector.group(1);

Selector sel = new Selector(str);

return sel;

} else {

return null;

}

}

Листинг 2.4.3.6 – Реализация метода получения следующего свойства селектора.

public Property nextProperty() {

boolean res = \_matProperty.find();

if (res) {

String str1 = \_matProperty.group(1).trim();

String str2 = \_matProperty.group(2).trim();

Property prop = new Property(str1, str2);

prop.setChange(\_matProperty.group(3) == null);

return prop;

} else {

return null;

}

}

### 2.4.4 Реализация сохранения селекторов.

Сохранение селекторов реализует метод analyze(CSSTextAnalysis) класса CSSAnalysis.

Листинг 2.4.4.1 Реализация метода

public static Storage analyze(CSSTextAnalysis cta) {

Storage storage = new Storage();

HashSet<Integer> selectors = new HashSet<Integer>();

String nameBlock = cta.nextBlock();

Block block = null;

while (nameBlock != null) {

if (storage.containsBlock(nameBlock)) {

block = storage.getBlock(nameBlock);

} else {

block = new Block(nameBlock);

storage.addBlock(nameBlock, block);

}

while (cta.nextSelector()) {

CSS.Selector sel = cta.nextName();

while (sel != null) {

// если не существует

if (!block.containsSelector(sel.toString())) {

// создание селектора

PropertiesSelector ps = new PropertiesSelector(sel);

block.addSelector(ps);//добавление в блок

selectors.add(ps.getId());// пометка в selectors

} else {

// иначе просто пометка selectors.add(block.getIdSelector(sel.getName()));

}

sel = cta.nextName();

}

CSS.Property prop = cta.nextProperty();

while (prop != null) {

int id;

RuleProperty rule = null;

String nameProperty;

nameProperty = prop.getName();

// Получение класса для обработки свойства

rule = Rules.getRuleProperty(nameProperty);

if (rule != null) {

prop = rule.toConvert(prop);

}

// проверка есть ли такое свойство в блоке

if (!block.containsProperty(prop.toString())) {

// создание свойства

SelectorsProperty sp = new SelectorsProperty(prop);

block.addProperty(sp);// добавляем

// получение id свойсва

id = sp.getId();

} else {

// получение id свойсва

id = block.getIdProperty(prop.toString());

}

// цикл по именам в селекторе

for (Integer idSel : selectors) {

// есть ли в селекторе такое же свойство

if (block.getSelector(idSel)

.containsProperty(nameProperty)) {

// получение id свойства которое было в селекторе

int temp = block.getSelector(idSel)

.getProperty(nameProperty);

// можно ли его изменять

if (block.getProperty(temp)

.isChangeProperty(idSel) || !prop.isChange()) {

// удаление ссылки на селектор

SelectorsProperty pre = block.getProperty(temp);

pre.removeSelector(idSel);

// добавление свойства в селектор

block.getSelector(idSel)

.addProperty(nameProperty, id);

// добавление ссылки на селектор в свойстве

block.getProperty(id)

.addSelector(idSel, prop.isChange());

}

} else {

// если нет, то добавление селектору свойство block.getSelector(idSel).addProperty(nameProperty, id);

// добавление ссылку на селектор в свойстве

block.getProperty(id).addSelector(idSel, prop.isChange());

}

}

prop = cta.nextProperty();

}

selectors.clear();

}

nameBlock = cta.nextBlock();

}

for (Block bl : storage.getBlocks()) {

for (Integer integer : bl.getKeySelectors()) {

PropertiesSelector sel = bl.getSelector(integer);

//Минификация селектора

GroupRules.toCombine(sel, bl);

}

}

//удаление всех пустых селекторов и свойств

storage.toCleanEmpty();

return storage;

}

**2.5 Реализация минификации селекторов.**

Создадим абстрактный класс RuleProperty(см. Листинг 2.5.1)

Листинг 2.5.1 Описание абстрактоного класса RuleProperty.

public abstract class RuleProperty {

protected String \_name;//имя свойства

public String getName(){

return \_name;

}

//оптимизация свойства

public abstract CSS.Property toConvert(CSS.Property p);

//p2 налаживается на p1

public abstract CSS.Property toOverlay(CSS.Property p1, CSS.Property p2);

}

Далее наследуем классы, в которых будут определены методы для минификации.

Так же создадим класс Rules который будет хранить все эти правила. И в зависимости от того какое свойство, будет возвращать нужный класс.

Листинг 2.5.2 Реализация класса Rules.

public class Rules {

private static HashMap<String, RuleProperty> \_rules;

static {

\_rules = new HashMap<String, RuleProperty>();

// добавление правил

TRBL trbl = new TRBL("padding");

\_rules.put("padding", trbl);

\_rules.put("padding-top", trbl);

\_rules.put("padding-right", trbl);

\_rules.put("padding-bottom", trbl);

\_rules.put("padding-left", trbl);

trbl = new TRBL("margin");

\_rules.put("margin", trbl);

\_rules.put("margin-top", trbl);

\_rules.put("margin-right", trbl);

\_rules.put("margin-bottom", trbl);

\_rules.put("margin-left", trbl);

Color color = new Color("color");

\_rules.put("color", color);

color = new Color("background-color");

\_rules.put("background-color", color);

}

public static RuleProperty getRuleProperty(String name) {

return \_rules.get(name);

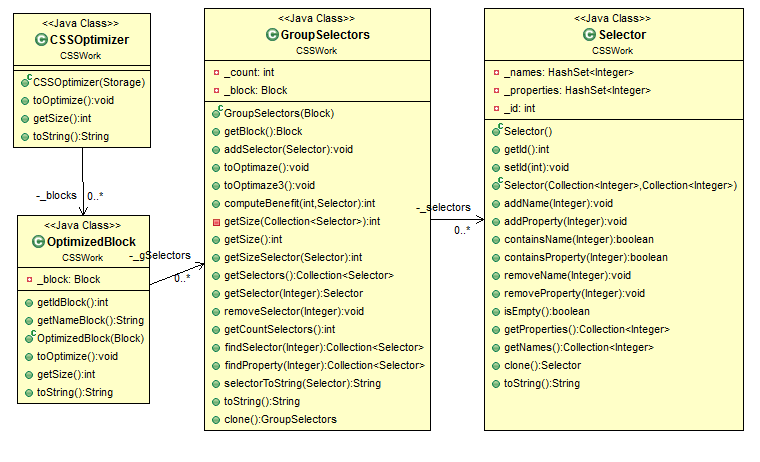
}

}

## 2.5 Реализация минификации Storage.

Для минификации используются классы . OptimizedBlock, GroupSelectors, Selector, CSSOptimizer.

Рис. 2.5.1 – UML диаграмма классов.



Сперва создается объект класса CSSOptimizer в конструктор которого передается объект Storage. И для каждого объекта Block из Storage создается OptimizedBlock в конструктор которого передается объект Block. И для каждого объекта OptimizedBlock вызывается метод toOptimize(). В данном методе происходит объединение селекторов в группы.

### 2.5.1 Реализация объединения селекторов в группы.

Объединение селекторов в группы реализовано в методе toOptimize() класса OptimizedBlock.Данный метод формирует массим объектов класса GroupSelectors.

Листинг 2.5.1.1 – Реализация Объединения.

LinkedList<GroupSelectors> gSelectors = new LinkedList<>();

HashSet<Integer> mark = new HashSet<>();

Stack<Integer> stack = new Stack<>();

for (Integer i : \_block.getKeySelectors()) {

if (!mark.contains(i)) {

stack.push(i);

GroupSelectors gs = new GroupSelectors(\_block);

while (!stack.isEmpty()) {

Integer id = stack.pop();

Selector sel = new Selector();

sel.addName(id);

mark.add(id);

for (Integer idProp : \_block.getSelector(id).getPropSel()){

sel.addProperty(idProp);

for (Integer idSel : \_block.getProperty(idProp).getSelProp()) {

if (!mark.contains(idSel)) {

stack.push(idSel);

mark.add(idSel);

}

}

}

gs.addSelector(sel);

}

\_gSelectors.add(gs);

}

}

### 2.5.2 Реализация оптимизации селекторов.

После создания групп каждая группа оптимизируется отдельно.

Листинг 2.5.2.1 – Реализация Объединения.

for (GroupSelectors gs : \_gSelectors) {

int count = gs.getCountSelectors();

if (count != 1) {

gs.toOptimaze();

count = gs.getCountSelectors();

if (count != 1) {

Optimise op = new Optimise();

GroupSelectors newGs = op.toOptimize(gs);

if (gs.getSize() > newGs.getSize()) {

gs = newGs;

}

}

}

gSelectors.add(gs);

}

## 2.6 Работа с приложением.

Написанное приложение не имеет графического интерфейса. Взаимодействие с приложение осуществляется через командную строку. Оно принимает единственный параметр – это путь к файлу который нужно минифицировать. Выходной файл будет создан в той же папке ,где находился исходный, и будет иметь имя с окончанием *.min.css* .

Листинг 2.6.1 - Пример команды запуска приложения.

java -jar CSSO.jar file.css

Экран работы программы можно посмотреть в Приложение A.

# 3. Сравнительная характеристика приложения.

Для примера возьмем CSS файл небольшого размера .

Листинг 3.1 – Содержимое CSS-файла.

a {

display: block;

color: red

}

p {

display: block;

color: yellow

}

h1 {

display: block;

color: blue

}

Минифицируем CSS-файл с помощью нашей программы и аналогичных ресурсов в интернете (css.github.io/csso/csso.html).

Листинг 3.2 – Вывод написанного приложения.

a,p,h1{display:block}a{color:red}p{color:#ff0}h1{color:blue}

Исходный размер: 130

Итоговый размер: 60

Выгода:70 ( 53.84% )

Листинг 3.3 – Вывод стороннего ресурса.

a,h1,p{display:block;color:red}h1,p{color:#ff0}h1{color:#00f}

Original: 128 bytes

Compressed: 61 bytes (47.66%)

Saving: 67 bytes (52.34%)

Важным параметром является “Итоговый размер”. Для написанного приложения 60 байт, для стороннего 61 байт. Как видно на небольшом размере файла написанный алгоритм минификации более выгодный.

Далее для примера оптимизируем CSS-файл vk.com.

Листинг 3.4 – Вывод написанного приложения.

Исходный размер: 158129

Итоговый размер: 126893

Выгода:31236 ( 19.75% )

Листинг 3.5 – Вывод стороннего ресурса.

Original: 158128 bytes

Compressed: 131990 bytes (83.47%)

Saving: 26138 bytes (16.53%)

Видно, что написанное приложение получило более высокую выгоду.

Так же есть и минусы. Если взять пример из Листинга 3.1 и заменить имена селекторов на более длинные, то получатся следующие данные.

Листинг 3.6 – Содержимое CSS-файла.

very-long-name1 {

display: block;

color: red

}

very-long-name2 {

display: block;

color: yellow

}

h1 {

display: block;

color: blue

}

Минифицируем CSS-файл с помощью написанной программы и аналогичных ресурсов в интернете (css.github.io/csso/csso.html).

Листинг 3.7 – Вывод написанного приложения.

very-long-name1{display:block;color:red}very-long-name2{display:block;color:#ff0}h1{display:block;color:blue}

Исходный размер: 156

Итоговый размер: 109

Выгода:47 ( 30.12% )

Листинг 3.8 – Вывод стороннего ресурса.

very-long-name1{display:block;color:red}h1,very-long-name2{display:block;color:#ff0}h1{color:#00f}

Original: 156 bytes

Compressed: 98 bytes (62.82%)

Saving: 58 bytes (37.18%)

Как видно, что чем меньше длина имен селекторов, тем выгоднее приложение минифицирует селектора.

# Заключение

В ходе выполнения курсовой работы проводился анализ CSS-файла. Целью курсовой работы являлась его минификация в байтовом размере. Были выявлены следующие проблемные места в CSS-файле:

* комментарии, лишние пробелы, переходы на новую строку;
* переопределение однотипных свойств в селекторе;
* использование избыточных конструкций при задании свойств;
* при большом количестве селекторов сложно уследить за наиболее оптимальным написанием свойств.

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

Результатом курсовой работы является приложение, которое минифицирует CSS-файл в байтовом размере. Эффективность работы приложения составляет в среднем порядка 30%.

Проведена сравнительная характеристика работы приложения с аналогичными WEB-сервисами. Созданное приложение по своим характеристикам ничем не уступает данным сервисам, а в некоторых случаях дает лучший результат.

В дальнейшем планируется ввести следующие изменения и усовершенствования приложения:

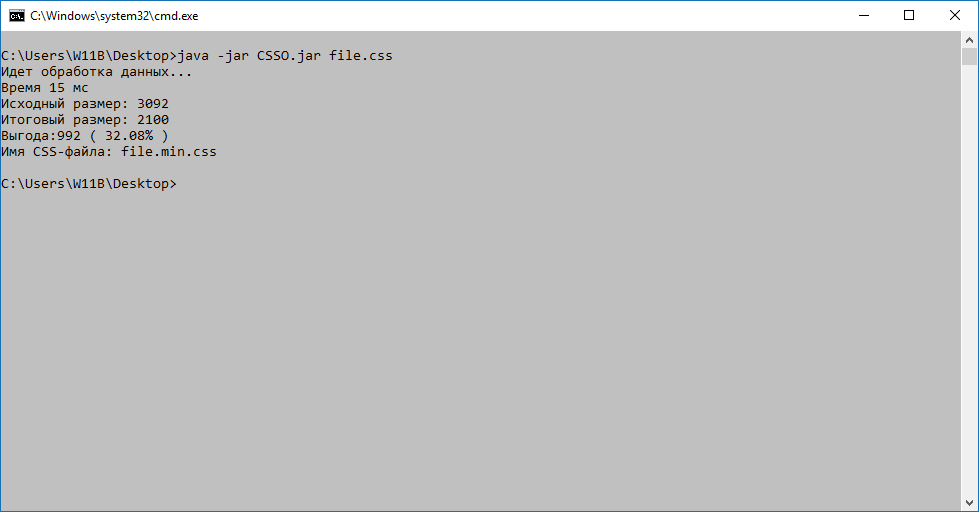
* реализовать синтаксический анализ селекторов;
* улучшить алгоритм минификации CSS-файла;
* реализовать оптимизацию в совокупности с HTML-страницами.

# Список использованной литературы

1. Cascading Style Sheets [Электронный ресурс]. - Режим доступа: https://www.w3.org . Дата доступа: 10.11.2016.
2. Дэвид Макфарланд, CSS3: The Missing Manual 3-е издание. — 3-е. — Санкт-Петербург, 2014. — С. 608 — ISBN 978-5-496-00428-2
3. Полный перебор. Wikipedia [Электронный ресурс]. - Режим доступа: <https://ru.wikipedia.org/wiki/Полный_перебор>. Дата доступа: 10.11.2016.
4. CSSO – CSS optimizer [Электронный ресурс]. - Режим доступа: <https://css.github.io/csso/csso.html>. Дата доступа: 10.11.2016.
5. CSS оптимизатор [Электронный ресурс]. - Режим доступа: <http://alexvaleev.ru/cssoptimizer/>. Дата доступа: 10.11.2016.

# Приложение А

**Пример запуска приложения**



# Приложение Б

**Содержимое дискового носителя**

1. \CSSO - файлы исходного кода проекта.
2. CSSO.jar – разработанное приложение.
3. Записка.docx - пояснительная записка к курсовой работе.