Санкт-Петербургский государственный университет
Математико-механический факультет

# Отчет о курсовом проекте

YetAnotherClipper

*Выполнил:* Толстопятов Всеволод Андреевич

Санкт-Петербург, май 2014

# 1 Введение

## 1.1 Постановка задачи

В качестве курсовой работы был выбран проект YetAnotherClipper: плагин для Google Chrome, который позволяет сохранять интересные статьи и материалы из интернета на Яндекс.Диск в удобочитаемом виде. Для этого необходимо разработать алгоритм извлечения статей из документов, написать плагин для браузера и backend-сервер, который извлекал бы статью и сохранял еҷ на диск. Более того, получившееся приложение должно быть отказоустойчивым, масштабируемым и легко поддерживаемым.

## 1.2 Требования

1. Корректность. Даже в случае критических ситуаций, система не должна прекращать свою работоспособность, а все такие ситуации должны быть тщательно залогированы.
2. Масштабируемость. Система должна иметь адекватное время отклика для пользователя, и, если количество пользователей резко возрастает, должен быть легкий способ отмасштабироваться, не потеряв работоспособность.
3. Don't repeat yourself.
4. Тестируемость. Должна быть возможность независимо тестировать отдельно взятые компоненты системы. Работоспособность системы должна быть тщательно проверена.

## 1.3 Обзор выбранных инструментов

В качестве плагина для браузера использовался javascript, так как других вариантов попросту нету. В качестве сервера была выбрана Java, т.к. одна из основных целей была получить опыт промышленной разработки и изучить новые технологии, для чего Java с множеством enterprise-библиотек, таких как экосистема Spring, Log4j, etc. прекрасно подходит.

# 2 Обзор существующих алгоритмических решений

## 2.1 Обзор существующих аналогов

К сожалению, в сети почти нет open source page-clipping алгоритмов. Наиболее известные коммерческие продукты: readability (расширение для браузера), getpocket (расширение для браузера, которое позволяет сохранять статьи на внутренний сервер getpocket, чтоб прочесть их потом), safari reader mode, justext.

## 2.2 justext

Алгоритм под New BSD License, написанный студентом в качестве его диплома. Предназначен не для извлечения статьи, но для извлечения любого текста из HTML-документа. Имеется реализация на python: https://code.google.com/p/justext/ Данный алгоритм имеет простую идею сегментации: разбить HTML-страницу на группы блоков, которые предназначены для визуального форматирования браузеров, например BLOCKQUOTE, CAPTION, P, PRE. Так же берется в расчет идея гомогенности блоков, то есть чаще всего можно определить, содержит ли блок текст, посмотрев на количество ссылок и осмысленного текста в нем. Осмысленность текста это количество в нем заданного количества слов из словаря для данного языка.

## 2.3 Safari Reader

Хотя и Safari является закрытым коммерческим продуктом, благодаря тому, что код алгоритма исполняется в браузере на стороне клиента, есть возможность его увидеть с помощью бага, описанного, например, здесь: http://blog.manbolo.com/2013/03/18/safari-reader-source-code Я не смог до конца разобрать алгоритм по следующим причинам: 1200 строк на javascript'е без единого комментария, более сотни созданных массивов без единого вызова splice(). Алгоритм использует следующие идеи:

1. Для некоторых нод DOM-дерева считается расстояние Левенштейна, чтоб определить, относится ли текст к заголовку статьи
2. Алгоритм медленно отрезает по одной ноде DOM-дерева, пока не останется только статья (квадратичное время работы)
3. Некоторые идентификаторы, такие как #disqus_thread захардкожены и алгоритм их сразу пропускает.
4. Алгоритм проверяет все ссылки, ведущие из данного документа, чтоб найти следующие страницы статьи, если они есть

### 2.4 Заключение

Так как готовой реализации нет, то было решено взять лучшие идеи из тех открытых алгоритмов, что имеются, дополнить некоторыми своими и сравнивать результат работы с коммерческими продуктами.

## 3 Алгоритм

### 3.1 Размышления и идеи

Так как готовых реализаций алгоритма, которые бы меня устроили, не было, решено было позаимствовать некоторые идеи из существующих и добавить несколько новых. Расстояние Левенштейна сразу было отметено, так как связь между заголовком и самой статьей не всегда присутствует, а вводить какие-то особенные эвристики для того, чтоб это учитывать, мне не хотелось. Однако идея сразу же пропускать некоторый тэги и идентификаторы мне показалась довольно разумной, благо почти все разделы комментариев содержат css-атрибут вроде *comment*. Из justext была взята идея гомогенности текста. Так же, из гомогенности следует еще более важная идея - кластерность. Похожие куски документа будут скорее детьми одного родителя в DOM-дереве, чем разных. Поэтому я решил ввести некоторый вес для элементов DOM-дерева, чтоб потом извлекать ноды с наибольшим весом. Таким образом, будут отброшены блоки с ссылками на, например, похожие статьи, но не будут отброшены блоки с, например, ссылками в конце статьи.

### 3.2 Итоговый алгоритм

1. Изначально определяем списки идентификаторов: “точно не элемент статьи”, “скорее всего не элемент статьи”, “скорее всего элемент статьи”
2. Начинаем обходить DOM-дерево и помечать узлы, содержащие в себе только визуальные элементы (p, pre, blockquote, a etc.) или textNode (W3C)
3. Для всех помеченных элементов считаем вес по некоторой метрике. Так как используется идея гомогенности, то этот же вес прибавляется отцу в дереве и половина этого веса добавляется деду.
4. Извлекается элемент с наибольшим весом.
5. Если в элементе содержится достаточное количество текста и запятых, то это статья. Если же нет, то утверждается, что статьи в документе нет и возвращается весь документ.

### 3.3 Метрика

Так как метрика сама по себе является эмпирической характеристикой документа, то точные коэффициенты в данной статье не приводятся, а подбирались они методом научного тыка с проверкой на популярных ресурсах. Изначально, для каждого тэга вводится некоторый вес, который может быть как положительным, так и отрицательным. Те тэги, что чаще отвечают за текст и предназначены для визуализации, получают больший вес, чем другие. А всевозможные iframe, ссылки и H1 получают наименьший вес. После этого происходит проверка, не является ли тэг или его атрибуты элементами списков “(не) элемент статьи” и в зависимости от этого прибавляется или отнимается существенный вес. Благодаря добавлению веса еще и родителю с дедом, получается оптимальная

кластеризация. Если добавлять вес только родителю, то на некоторых тестах начинались отвратительные выбросы, которые сводили на нет всю идею алгоритма, а при добавлении четверти прадеду существенно замедлялась работа, а выигрыша не было совсем.

### 3.4 Тест алгоритма

Были проведены тесты на всех многих популярных ресурсах: хабрахабр, The New York Times, всевозможные новостные порталы и блоги на популярных cms. На всех был получен хороший результат, сравнимый с коммерчискими решениями и который удовлетворял требованию алгоритма, поэтому данный пункт своей курсовой работы я считаю успешно выполненым.

### 3.5 Недостатски

Как и любой эмпирический алгоритм без машинного обучения, он не лишен некоторых недостатков. Во-первых, практически невозможно определить, какая из ссылок в документе ведет на следующую страницу. Статьи в многостраничном формате сейчас не встречаются, но для сервисов онлайн чтения книг это более чем актуально. Так же замечена проблема с извлечением заголовка: алгоритм не использует никакие наблюдения по этому поводу и просто берет title у документа. Поэтому на некоторых ресурсах заголовок получался наподобие “Обзор java.util.concurrent.* / Блог компании Luxoft / Хабрахабр”, когда как коммерческие продукты выдавали “Обзор java.util.concurrent.* ”

### 3.6 Итог

Алгоритм показал себя дееспособным, а его результат сравним с коммерческими продуктами. В процессе его разработки было перепробовано большое количество различных подходов, был получен опыт тестирования не только на одном документе (что в начале меня очень огорчало), а так же опыт не просто написания кода, а создания чего-то нового.

## 4 Бэкэнд и его архитектура

### 4.1 Выбор инструментов

Так как нужно было написать не просто “заглушку”, которая исполняла бы алгоритм, а полноценную рабочую систему, то сначала нужно было правильно выбрать инструменты и приблизительно спроектировать систему. Сначала была выбрана система сборки Maven и механизм авторизации oAuth 2.0. Так как проект был большим, то собирать его простой командой из консоли уже не получалось, а количество зависимостей росло вместе с количеством компонент сервера. Maven был выбран из-за простоты результирующей сборки (весь проект собирается одной командой) и его репозиториям, из которых он сам же во время сборки выкачивал все зависимости.
В качестве парсера HTML-документов был выбран jsoup, так как выбор был в основном между ним и написанием парсера своими руками, что является отдельной трудоеской задачей.
Так как веб-сервер должен был быть асинхронным, то была выбрана Jetty, как легковесный сервер и контейнер сервлетов, в противовес Netty, который не поддерживает Servlet API 3.0 и предназначен для event-driven систем.
Log4j был выбран в качестве логера “из коробки”, потому как приходилось разбираться с огромным количеством новых технологий, а slf4j является массивной надстройкой над log4j.
В качестве DI-контейнера был выбран Spring IoC, который используется в любом enterprise-проекте, когда как Google Guice довольно редко встречается за пределами Google.
Spring JDBC - еще один элемент экосистемы Spring, который использовался в качестве прослойки между базой данных в качестве кэша и основным приложением. Так как проект часто собирался на разных машинах и тестировался, то в качестве базы данных для кэша была выбрана in-memory база данных Derby, так как иметь все время работающий сервес с, например, MySQL не представлялось возможным.
И JUnit 4 для тестирования: хоть он и работает гораздо медленнее, чем TestNG, но гораздо проще в использовании, а также поддерижвает кастомные runner’ы, что необходимо для контекстов Spring’a.

### 4.2 Процесс разработки и выбор архитектурных решений

Очевидно, что далеко не все технологии были выбраны сразу. В этом параграфе будет описан процесс становления архитектуры и появления тех или иных технологий.
Изначально были выбраны Jetty, Jsoup и Maven. Так как Jetty асинхронный и легковесный, то внутри он поддерживает свой собственный пул потоков, которые выделяются под обращения к сервлетам и использовать эти же потоки еще и для самого алгоритма было бы нецелесообразно и могло повести к замедлению системы. Поэтому был заведен CachedThreadPool и все обращение проксировались ему. Так как количество зависимостей от всевозможных классов увеличивалось, то был внедрен Spring, а lifecycle запроса выглядел так: поток Jetty принимал запрос, проверял валидность параметров, а после проксировал его абстракции над ThreadPool'ом. Пул задач через фабрику создавал по параметрам запроса непосредственно Task для самого пула потоков.
Жизнь задачи в пуле потоков выглядит так: через фасад проверяется сначала кэш, потом извлекается статья, из имеющихся ресурсов собирается готовый документ, сохраняется в кэш и передается прокси Яндекс.Диска, который сохранял статью на диск. Параллельно, в отдельном потоке, существовал демон, который заведовал кэшом и чистил его по таймстампу раз в некоторый интервал времени. На моменте, когда вместо алгоритма была заглушка, сервис уже иногда начинал выдавать какие-то ошибки, поэтому был изучен и добавлен log4j.
После разработки и реализации алгоритма встал вопрос о том, как представлять результирующую статью. Пришлось разобраться в css, чтоб результирующий документ можно было настраивать (ширина текста в статьее, размер шрифтов, выравнивания) и были добавлены абстракции для создания объектов результирующей статьи.
В процессе разработки писались тесты, а так же был найден баг в SDK Яндекс.Диска с последующим репортом. Благодаря независимости системы, масштабируемость достигается на сетевом уровне поднятием нескольких машин и сетевым балансировщиком.

### 4.3 Что не сделано

1. Несмотря на тестируемость системы, благодаря DI, количество unit-тестов недостаточно велико
2. Кроме профилирование в VisualVM, не было написано никаких бенчмарков
3. Очень неточно определено количество возможных запросов в секунду (на грубых тестах это примерно 30)

## 5 Итог

В ходе работы было изучено множество технологий, получен опыт промышленной разработки, а также опыт создания новых алгоритмов, а получившийся продукт можно считать завершенным.