**Design Patterns 2019**

**Term project report**

[](https://www.google.co.kr/url?sa=i&rct=j&q=&esrc=s&source=images&cd=&cad=rja&uact=8&ved=2ahUKEwjmr83Sw7DdAhUQKXwKHRLXAlgQjRx6BAgBEAU&url=https://ko.wikipedia.org/wiki/%ED%8C%8C%EC%9D%BC:Chung-Ang_University_logo.png&psig=AOvVaw0WcIGNrYiMRExXcTadFruQ&ust=1536672050200468)

|  |  |
| --- | --- |
| 프로젝트 명 | Jsoup 기능 확장 및 설계 개선 |
| 참여자 | 20133659 김성재 |
| 20146847 조호영 |
| 20141589 윤희성 |
| 20173298 오창균 |

**1. 프로젝트 개요**

1-1) 프로젝트(Jsoup) 소개

1-2) 프로젝트 목표   
 **2. Jsoup 설계 및 구현 조사**

2-1) overview

2-2) parser

2-3) nodes

2-4) select

2-5) etc

**3. 기능 확장 및 설계 개선**

3-1) 확장된 기능 및 설계패턴

**4. 테스트 수행 내역**

4-1) 테스트 케이스 및 결과

**5. GitHub 활동**

5-1) 활동 내역

5-2) 기여 내역

**1. 프로젝트 개요**

**1-1. 프로젝트 소개**

Jsoup은 HTML, XML 및 CSS를 파싱해주는 실용적인 Java 라이브러리이다. HTML, XML은 DOM(The Document Object Model) interface를 가지며 트리구조를 이루는 각각의 leaf를 node라고 칭한다. Jsoup은 이러한 node 형태로 되어있는 웹 상의 데이터를 추출하고 손 쉽게 조작 할 수 있는 편리한 API를 제공하고 있다.

Document doc = Jsoup.connect("http://en.wikipedia.org/").get();

log(doc.title());

Elements newsHeadlines = doc.select("#mp-itn b a");

for (Element headline : newsHeadlines) {

log("%s\n\t%s",

headline.attr("title"), headline.absUrl("href"));

}

위와 같은 간단한 코드로 손쉽게 Web상의 HTML, CSS를 추출 및 재조립할 수 있다.

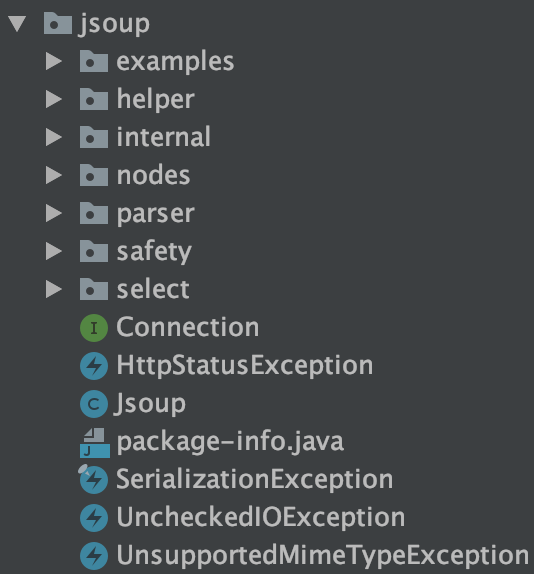
**1-2. 프로젝트 목표**

Jsoup의 아키텍쳐인 패키지 구조 및 설계 패턴 구조를 분석 및 개선하여 소프트웨어 프로젝트 아키텍쳐의 뼈대가되는 설계 패턴에 대한 이해도를 높인다. 또한 버전관리 및 협업관리 툴인 GitHub를 사용을 통해 개발에 있어서 필요한 의사소통을 능력을 개선한다.

**2. Jsoup 설계 및 구현 조사**

**2-1. overview**

Jsoup의 패키지 구조는 다음과 같다. Root directory에 API의 최초 진입점이 되는 Jsoup class가 존재하고 하위 패키지로는 helper, internal, nodes, parser, safety, select가 있다.



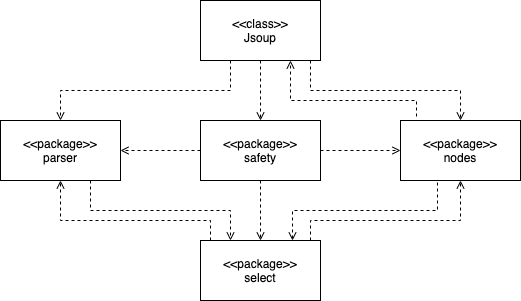
Jsoup의 최상단 클래스인 Jsoup.class가 하는 역할은 크게 다음과 같다.

* HTML을 받아서 parsing
* HTML을 파일 형태로 받아서 parsing
* Page URI를 받아서 parsing
* Parsing할 HTML 중 원하는 것들만 뽑아서 보기

텍스트, 스크린샷이(가) 표시된 사진

자동 생성된 설명

즉 Jsoup은 어떤한 형태의 HTML을 받아서 사용자가 원하는 형태로 HTML 문서를 뽑아낼 수 있게 하고 있다. 이것을 기본으로 패키지 구조를 보면 다음과 같다.



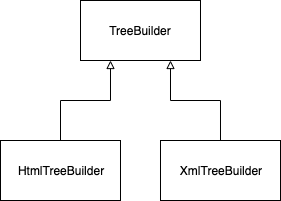
**2-2. parser**

Parser는 Html 또는 XML을 String형태로 받아서 알맞은 Tree 형태로 반환하는 역할을 하고 있는 컴포넌트입니다.

Parser는 크게 다음과 같은 3가지의 패턴으로 이루어져 있습니다.

* Template Method Pattern
* Command Pattern
* State Pattern

**2-2-1. Template Method Pattern**

****

**TreeBuilder**라는 abstract class를 **HtmlTreeBuilder**와 **XmlTreeBuilder**가 extends하고 있다. Html과 Xml의 구조를 만들 때 공통적으로 쓰이는 부분을 추상화 시켜서 추상클래스를 만들었다.

* TreeBuilder 내부의 행위를 추상화하는 abstract 메소드나 interface 필드는 없다. 그렇기 때문에 Strategy Pattern은 쓰이지 않았다.

**2-2-2. Strategy Pattern**

**TreeBuilder**를 보면 다음과 같은 abstract method와 해당 process가 **Start**인지 **End**인지에 따라서 다른 행위를 취하는 Strategy Pattern 사용을 유추해 볼 수 있다.

protected abstract boolean process(Token token);

protected boolean processStartTag(String name) {

if (currentToken == start) { // don't recycle an in-use token

return process(new Token.StartTag().name(name));

}

return process(start.reset().name(name));

}

public boolean processStartTag(String name, Attributes attrs) {

if (currentToken == start) { // don't recycle an in-use token

return process(new Token.StartTag().nameAttr(name, attrs));

}

start.reset();

start.nameAttr(name, attrs);

return process(start);

}

protected boolean processEndTag(String name) {

if (currentToken == end) { // don't recycle an in-use token

return process(new Token.EndTag().name(name));

}

return process(end.reset().name(name));

}

protected Element currentElement() {

int size = stack.size();

return size > 0 ? stack.get(size-1) : null;

}

Strategy Pattern을 확인하기 위해 **Token** **class**를 보면 **Token class**는 abstract class임을 알 수 있다. 그리고 **Tag**라는 내부 abstract static 클래스를 선언하고 **Token**을 상속하고 있다. 여기서 **Tag**는 다시 한 번, 내부 static클래스인 **StartTag**와 **EndTag**에 의해 상속된다.

final static class StartTag extends Tag {

...

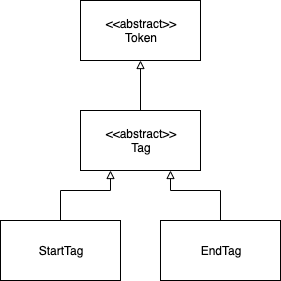
}

final static class EndTag extends Tag{

...

}

그리고 **StartTag**와 **EndTag**는 위에서 언급한 process, processStartTag, processEndTag에서 사용되고 있다. 즉, Strategy Pattern으로 Token 상태에 따라 행위를 결정하고 있다.



**2-2-3. State Pattern**

Parser에서는 **HtmlTreeBuilderState**와 **TokeniserState**에서 State Pattern을 적용하고 있다.

Template Method Pattern에서 언급한 **TreeBuilder**안에는 process라는 abstract method가 존재한다.

protected abstract boolean process(Token token);

그리고 **HtmlTreeBuilder**안에는 abstract method의 구현체가 있고 **HtmlTreeBuilderState**안의 또 다른 abstract method를 실행하고 있다.

enum HtmlTreeBuilderState {

Initial {...},

BeforeHtml {...},

BeforeHead {...},

InHead {...},

InHeadNoscript {...},

AfterHead {...},

InBody {...},

Text {...},

...

**HtmlTreeBuilderState**는 enum클래스이다. **HtmlTreeBuilderState**안에는 아래의 로직들이 있고 **process**라는 abstract 메소드가 있다. 그리고 **HtmlTreeBuilderState**의 프로퍼티들은 **process**를 구현하고 있다.

abstact process 메소드는 다음과 같으며

abstract boolean process(Token t, HtmlTreeBuilder tb);

위의 Initial 프로퍼티의 내용을 보면 다음과 같다.

Initial {

boolean process(Token t, HtmlTreeBuilder tb) {

if (isWhitespace(t)) {

return true; // ignore whitespace

} else if (t.isComment()) {

tb.insert(t.asComment());

} else if (t.isDoctype()) {

// todo: parse error check on expected doctypes

// todo: quirk state check on doctype ids

Token.Doctype d = t.asDoctype();

DocumentType doctype = new DocumentType(

tb.settings.normalizeTag(d.getName()), d.getPublicIdentifier(), d.getSystemIdentifier());

doctype.setPubSysKey(d.getPubSysKey());

tb.getDocument().appendChild(doctype);

if (d.isForceQuirks())

tb.getDocument().quirksMode(Document.QuirksMode.quirks);

tb.transition(BeforeHtml);

} else {

// todo: check not iframe srcdoc

tb.transition(BeforeHtml);

return tb.process(t); // re-process token

}

return true;

}

}

그리고 해당 state는 HtmlTreeBuilder에서 transition method로 context를 주입받고 있다.

void transition(HtmlTreeBuilderState state) {

this.state = state;

}

void resetInsertionMode() {

boolean last = false;

for (int pos = stack.size() -1; pos >= 0; pos--) {

Element node = stack.get(pos);

if (pos == 0) {

last = true;

node = contextElement;

}

String name = node.normalName();

if ("select".equals(name)) {

transition(HtmlTreeBuilderState.InSelect);

break; // frag

}

…

} else if ("html".equals(name)) {

transition(HtmlTreeBuilderState.BeforeHead);

break; // frag

} else if (last) {

transition(HtmlTreeBuilderState.InBody);

break; // frag

}

}

}

**HtmlTreeBuilderState**는 Html의 tree를 만들 때, tree node의 state에 따라(e.g. Head, Body…) **HtmlTreeBuilder**의 상태(state)가 바뀌고 있다. 즉 **process**의 행위가 바뀌도록 하는 **State Pattern**을 사용하고 있다.

**2-3. nodes**

**스크린샷이(가) 표시된 사진

자동 생성된 설명**

Nodes는 Dom tree를 그리는 node들의 요소를 그리는 역할을 한다. Tree를 그리기 위한 leaf라는 개념이 존재하며 각 leaf는 dom의 특성을 잘 담을 수 있는 클래스로 구성되어 있다.

**2-3-1. Iterator Pattern**

스크린샷이(가) 표시된 사진

자동 생성된 설명

public class Attributes implements Iterable<Attribute>, Cloneable {

...

public Iterator<Attribute> iterator() {

return new Iterator<Attribute>() {

int i = 0;

@Override

public boolean hasNext() {

return i < size;

}

@Override

public Attribute next() {

final Attribute attr = new Attribute(keys[i], vals[i], Attributes.this);

i++;

return attr;

}

@Override

public void remove() {

Attributes.this.remove(--i); // next() advanced, so rewind

}

};

}

...

}

Iterator Interface를 구현하여, Attribute를 순회하는 next, hasNext method를 오버라이딩 하였다.

**2-3-2. Prototype Pattern**

스크린샷이(가) 표시된 사진

자동 생성된 설명

public class Attributes implements Iterable<Attribute>, Cloneable {

...

@Override

public Attributes clone() {

Attributes clone;

try {

clone = (Attributes) super.clone();

} catch (CloneNotSupportedException e) {

throw new RuntimeException(e);

}

clone.size = size;

keys = copyOf(keys, size);

vals = copyOf(vals, size);

return clone;

}

...

}

public abstract class Node implements Cloneable {

...

@Override

public Node clone() {

Node thisClone = doClone(null); // splits for orphan

// Queue up nodes that need their children cloned (BFS).

final LinkedList<Node> nodesToProcess = new LinkedList<>();

nodesToProcess.add(thisClone);

while (!nodesToProcess.isEmpty()) {

Node currParent = nodesToProcess.remove();

final int size = currParent.childNodeSize();

for (int i = 0; i < size; i++) {

final List<Node> childNodes = currParent.ensureChildNodes();

Node childClone = childNodes.get(i).doClone(currParent);

childNodes.set(i, childClone);

nodesToProcess.add(childClone);

}

}

return thisClone;

}

...

}

Cloneable Interface를 구현하여 clone() method를 오버라이딩, 객체를 복사하는 Prototype패턴이 적용되었다.

**2-3-3. Composite Pattern**

스크린샷이(가) 표시된 사진

자동 생성된 설명

Element, LeafNode는 Node의 구현체이고, Element에는 자식노드를 추가 할 수 있지만 Leaf에는 자식노드를 추가 할 수 없다.

**Element 자식을 추가하는 코드**

public class Element extends Node {

public Element appendChild(Node child) {

Validate.notNull(child);

reparentChild(child);

ensureChildNodes();

childNodes.add(child);

child.setSiblingIndex(childNodes.size() - 1);

return this;

}

public Element insertChildren(int index, Collection<? extends Node> children) {

Validate.notNull(children, "Children collection to be inserted must not be null.");

int currentSize = childNodeSize();

if (index < 0) index += currentSize +1; // roll around

Validate.isTrue(index >= 0 && index <= currentSize, "Insert position out of bounds.");

ArrayList<Node> nodes = new ArrayList<>(children);

Node[] nodeArray = nodes.toArray(new Node[0]);

addChildren(index, nodeArray);

return this;

}

public Element insertChildren(int index, Node... children) {

Validate.notNull(children, "Children collection to be inserted must not be null.");

int currentSize = childNodeSize();

if (index < 0) index += currentSize +1; // roll around

Validate.isTrue(index >= 0 && index <= currentSize, "Insert position out of bounds.");

addChildren(index, children);

return this;

}

}

**LeafNode에는 자식을 추가하는 Method가 없다.**

abstract class LeafNode extends Node {

protected final boolean hasAttributes() {}

public final Attributes attributes() {}

private void ensureAttributes() {}

String coreValue() {}

void coreValue(String value) {}

public String attr(String key) {}

public Node attr(String key, String value) {}

public boolean hasAttr(String key) {}

public Node removeAttr(String key) {}

public String absUrl(String key) {}

public String baseUri() {}

protected void doSetBaseUri(String baseUri) {}

public int childNodeSize() {}

protected List<Node> ensureChildNodes() {}

}

Node의 자식인 Element에는 자식 Node를 추가, LeafNode에는 자식을 추가할 수 없게하여 Composite Pattern을 사용하고 있다.

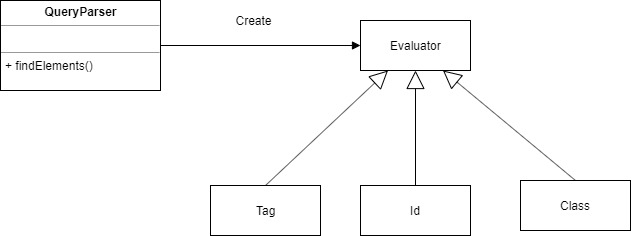
**2-4. select**

Select는 요청한 주소의 Html코드와 a, div, img, div와 같은 CSS selector들을 이용하여 Html코드에서 추출하고 싶은 태그를 추려낼 수 있게하는 컴포넌트이다.

지도, 텍스트이(가) 표시된 사진

자동 생성된 설명

**2-4-1. Factory Pattern**



public static Elements select(String query, Element root) {

Validate.notEmpty(query);

return select(QueryParser.parse(query), root);

}

사용자가 입력한 조건인 CSS Selector를 파싱하여 의미를 파악하게 도와주는 **QueryParser.parse()** 메소드는 사용자의 String 입력에 맞춰서 **QueryParser**클래스에서 생성할 클래스를 결정해 준다.

private void findElements() {

if (tq.matchChomp("#"))

byId();

else if (tq.matchChomp("."))

byClass();

else if (tq.matchesWord() || tq.matches("\*|"))

byTag();

else if (tq.matches("["))

byAttribute();

else if (tq.matchChomp("\*"))

allElements();

else if (tq.matchChomp(":lt("))

indexLessThan();

else if (tq.matchChomp(":gt("))

indexGreaterThan();

else if (tq.matchChomp(":eq("))

indexEquals();

...

}

**QueryParser.parse()**에서 **findElements()** 메소드를 통해 태그에 맞는 적합한 생성자를 찾게 한다.

private void byId() {

String id = tq.consumeCssIdentifier();

Validate.notEmpty(id);

evals.add(new Evaluator.Id(id));

}

private void byClass() {

String className = tq.consumeCssIdentifier();

Validate.notEmpty(className);

evals.add(new Evaluator.Class(className.trim()));

}

결과적으로 태그에 맞게 생성된 클래스가 evals list에 추가된다. 객체 생성을 캡슐화하고 **findElement()** 메소드를 통해 서브 클래스에서 어떤 클래스를 만들지 결정하고 있다.

**2-4-2. Strategy Pattern**

스크린샷이(가) 표시된 사진

자동 생성된 설명

public abstract class Evaluator {

protected Evaluator() {

}

/\*\*

\* Test if the element meets the evaluator's requirements.

\*

\* @param root Root of the matching subtree

\* @param element tested element

\* @return Returns <tt>true</tt> if the requirements are met or

\* <tt>false</tt> otherwise

\*/

public abstract boolean matches(Element root, Element element);

...

}

**Evaluator** 추상 클래스를 만들어 공통적인 **matches** 메소드를 캡슐화한다.

public static final class IsFirstChild extends Evaluator {

    @Override

    public boolean matches(Element root, Element element) {

        final Element p = element.parent();

        return p != null && !(p instanceof Document) && element.elementSiblingIndex() == 0;

    }

...

}

public static final class IsLastChild extends Evaluator {

    @Override

    public boolean matches(Element root, Element element) {

        final Element p = element.parent();

        return p != null && !(p instanceof Document) && element.elementSiblingIndex() == p.children().size()-1;

    }

    @Override

    public String toString() {

        return ":last-child";

    }

}

**Evaluator** 클래스를 구현하는 **Concrete** 클래스에서는 각기 다른 **matches** 로직을 구현한다.

public void head(Node node, int depth) {

if (node instanceof Element) {

        Element el = (Element) node;

        if (eval.matches(root, el))

            elements.add(el);

        }

    }

...

}

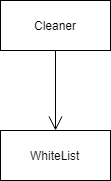
클라이언트에서는 **matches**라는 메소드명만 알고 호출하여 로직은 캡슐화 된다.

**2-5. etc**

Parser, nodes, select 패키지 외의 패키지는 root directory에 있는 클래스들과 helper, internal, safety 패키지가 있다.

각각의 패키지를 요약하는 다이어그램은 아래와 같다.

**Satefy**

****

**Internal**

**스크린샷이(가) 표시된 사진

자동 생성된 설명**

**Helper**

**스크린샷이(가) 표시된 사진

자동 생성된 설명**

**Helper - Template Pattern**

**스크린샷이(가) 표시된 사진

자동 생성된 설명**

**ChangeNotifyingArrayList** 클래스는 ArrayList를 상속받는 abstract 클래스이다. 여기에는 하나의 abstract 메소드가 존재한다.

public abstract void onContentsChanged();

이 abstract 메소드는 ArrayList내부의 값이 변경될 때 마다 호출된다.

private static final class NodeList extends ChangeNotifyingArrayList<Node> {

...

public void onContentsChanged() {

owner.nodelistChanged();

}

}

이 abstract 메소드의 실제 구현은 하위 클래스인 NodeList에서 이루어진다.

private static final class NodeList extends ChangeNotifyingArrayList<Node> {

private final Element owner;

NodeList(Element owner, int initialCapacity) {

super(initialCapacity);

this.owner = owner;

}

public void onContentsChanged() {

owner.nodelistChanged();

}

}

**3. 기능 확장 및 설계 개선**  
**3-1. 확장된 기능**

**3-1-1. Nodes-DocumentType Class 설계 개선**

**기존 코드 및 문제점**

public class DocumentType extends LeafNode {

// todo needs a bit of a chunky cleanup. this level of detail isn't needed

public static final String PUBLIC\_KEY = "PUBLIC";

public static final String SYSTEM\_KEY = "SYSTEM";

private static final String NAME = "name";

private static final String PUB\_SYS\_KEY = "pubSysKey";

private static final String PUBLIC\_ID = "publicId";

private static final String SYSTEM\_ID = "systemId";

/\*\*

\* Create a new doctype element.

\* @param name the doctype's name

\* @param publicId the doctype's public ID

\* @param systemId the doctype's system ID

\*/

     public DocumentType(String name, String publicId, String systemId) {

Validate.notNull(name);

Validate.notNull(publicId);

Validate.notNull(systemId);

attr(NAME, name);

attr(PUBLIC\_ID, publicId);

if (has(PUBLIC\_ID)) {

attr(PUB\_SYS\_KEY, PUBLIC\_KEY);

}

  attr(SYSTEM\_ID, systemId);

}

}

**문제점**

**스크린샷이(가) 표시된 사진

자동 생성된 설명**

1. PUBLIC\_KEY, SYSTEM\_KEY와 같은 변수들이 상수값으로 고정되어 있다.
2. 해당 final변수들을 다양하게 가져가려 할 때, DocumentType클래스를 수정해야 한다.
3. DocumentType객체에 PUBLIC\_KEY와 같은 상수값을 변경할 때 변경값, 현재 값이 따로 저장되어 있지 않다.

**해결 방안 및 개선된 코드 – Factory Method + Singleton**

**스크린샷이(가) 표시된 사진

자동 생성된 설명**

1. 해당 final 변수들을 설정값 클래스로 따로 분리
2. 상황에 따라 해당 기본 값 변경 시 클래스를 확장, composition으로 처리 할 수 있도록 abstract 클래스 구현
3. Abstract 클래스를 확장하여 OCP 구현
4. DocumentType 클래스로 부터의 final변수에 대한 의존성 분리, DIP 적용
5. 설정값이 추가될 때 마다 코드수정을 최소화하도록 Factory Method Pattern 적용
6. Factory Instance는 Singleton 적용

개선된 코드는 다음과 같다.

**Document.class**

public class DocumentType extends LeafNode {

private KeyStoreFactory keyStoreFactory = KeyStoreFactory.getInstance();

private KeyStore keyStore;

/\*\*

\* Create a new doctype element.

\*

\* @param name the doctype's name

\* @param publicId the doctype's public ID

\* @param systemId the doctype's system ID

\*/

public DocumentType(String name, String publicId, String systemId) {

this.keyStore = keyStoreFactory.getKeyStore("default");

Validate.notNull(name);

Validate.notNull(publicId);

Validate.notNull(systemId);

this.keyStore.setName(name);

this.keyStore.setPublicId(publicId);

this.keyStore.setSystemId(systemId);

attr(this.keyStore.getNameType(), this.keyStore.getName());

attr(this.keyStore.getPublicIdType(), this.keyStore.getPublicId());

if (has(keyStore.getPublicIdType())) {

attr(keyStore.getPublicSystemKeyType(), this.keyStore.getPublicKey());

}

attr(this.keyStore.getSystemIdType(), this.keyStore.getSystemId());

}

...

}

**KeyStory.class (abstract)**

package org.jsoup.nodes.keystore;

public abstract class KeyStore {

private String PUBLIC\_KEY\_TYPE;

private String SYSTEM\_KEY\_TYPE;

private String NAME\_TYPE;

private String PUB\_SYS\_KEY\_TYPE;

private String PUBLIC\_ID\_TYPE;

private String SYSTEM\_ID\_TYPE;

public KeyStore(){

this.PUBLIC\_KEY\_TYPE = "PUBLIC";

this.SYSTEM\_KEY\_TYPE = "SYSTEM";

this.NAME\_TYPE = "name";

this.PUB\_SYS\_KEY\_TYPE = "pubSysKey";

this.PUBLIC\_ID\_TYPE = "publicId";

this.SYSTEM\_ID\_TYPE= "systemId";

}

public String getPublicKeyType() {

return this.PUBLIC\_KEY\_TYPE;

}

public String getSystemKeyType() {

return this.SYSTEM\_KEY\_TYPE;

}

public String getNameType() {

return this.NAME\_TYPE;

}

public String getPublicSystemKeyType() {

return this.PUB\_SYS\_KEY\_TYPE;

}

public String getPublicIdType() {

return this.PUBLIC\_ID\_TYPE;

}

public String getSystemIdType() {

return this.SYSTEM\_ID\_TYPE;

}

abstract public String getPublicKey();

abstract public String getSystemKey();

abstract public String getName();

abstract public String getPublicSystemKey();

abstract public String getPublicId();

abstract public String getSystemId();

abstract public void setPublicKey(String value);

abstract public void setSystemKey(String value);

abstract public void setName(String value);

abstract public void setPublicSystemKey(String value);

abstract public void setPublicId(String value);

abstract public void setSystemId(String value);

}

**DefaultKeyStore.class**

package org.jsoup.nodes.keystore;

public class DefaultKeyStore extends KeyStore {

private String PUBLIC\_KEY;

private String SYSTEM\_KEY;

private String NAME;

private String PUB\_SYS\_KEY;

private String PUBLIC\_ID;

private String SYSTEM\_ID;

public DefaultKeyStore(){

super();

this.PUBLIC\_KEY = "PUBLIC";

this.SYSTEM\_KEY = "SYSTEM";

this.NAME = "name";

this.PUB\_SYS\_KEY = "pubSysKey";

this.PUBLIC\_ID = "publicId";

this.SYSTEM\_ID = "systemId";

}

@Override

public String getPublicKey() {

return this.PUBLIC\_KEY;

}

@Override

public String getSystemKey() {

return this.SYSTEM\_KEY;

}

@Override

public String getName() {

return this.NAME;

}

@Override

public String getPublicSystemKey() {

return this.PUB\_SYS\_KEY;

}

@Override

public String getPublicId() {

return this.PUBLIC\_ID;

}

@Override

public String getSystemId() {

return this.SYSTEM\_ID;

}

@Override

public void setPublicKey(String value) {

this.PUBLIC\_KEY = value;

}

@Override

public void setSystemKey(String value) {

this.SYSTEM\_KEY = value;

}

@Override

public void setName(String value) {

this.NAME = value;

}

@Override

public void setPublicSystemKey(String value) {

this.PUB\_SYS\_KEY = value;

}

@Override

public void setPublicId(String value) {

this.PUBLIC\_ID = value;

}

@Override

public void setSystemId(String value) {

this.SYSTEM\_ID = value;

}

}

**KeyStoreFactory.class (Singleton, Factory)**

package org.jsoup.nodes.keystore;

public class KeyStoreFactory {

private volatile static KeyStoreFactory instance = null;

private KeyStoreFactory(){};

public static KeyStoreFactory getInstance(){

if(instance == null){

synchronized (KeyStoreFactory.class){

if(instance == null){

instance = new KeyStoreFactory();

}

return instance;

}

}

return instance;

}

public KeyStore getKeyStore(String name){

if(name == "default"){

return new DefaultKeyStore();

}

return new DefaultKeyStore();

}

}

**3-1-2. Select-NodeTraversor Class 설계 개선**

**기존 코드 및 문제점**

public static Elements collect (Evaluator eval, Element root) {

Elements elements = new Elements();

NodeTraversor.traverse(new Accumulator(root, elements, eval), root);

return elements;

}

**문제점**

NodeTraversor 클래스에서 DFS 방식으로 구현된 traverse()를 직접 호출하여 알고리즘에 대한 유연성 확보가 되지 않음.

**해결 방안 및 개선된 코드 – Strategy**

지도, 텍스트이(가) 표시된 사진

자동 생성된 설명

위 클래스 다이어그램과 같이 Strategy Pattern을 적용한다.

1. MasterTraversor 인터페이스 생성
2. NodeTraversor(DFS), NodeTraversor\_BFS (BFS) 구상 클래스 생성
3. 클라이언트에서는 MasterTraversor 형태로 호출하여 알고리즘에 대한 유연성 확보

개선된 코드는 다음과 같다.

**MasterTraversor.interface**

public interface MasterTraverse {

    public void traverse(Node root);

    public static void traverse(NodeVisitor visitor, Node root) {}

}

**BFS 구상 클래스 생성 (NodeTraversor, NodeTraversor\_BFS)**

public class NodeTraversor implements MasterTraverse {}

public class NodeTraversor\_BFS implements MasterTraverse{}

**traverse() 구현 (NodeTraversor, NodeTraversor\_BFS)**

**NodeTraversor.class**

public static void traverse(NodeVisitor visitor, Node root) {

Node node = root;

int depth = 0;

while (node != null) {

visitor.head(node, depth);

if (node.childNodeSize() > 0) {

node = node.childNode(0);

depth++;

} else {

while (node.nextSibling() == null && depth > 0) {

visitor.tail(node, depth);

node = node.parentNode();

depth--;

}

visitor.tail(node, depth);

if (node == root)

break;

node = node.nextSibling();

}

}

}

**NodeTraversor.class**

public static void traverse(NodeVisitor visitor, Node root) {

Queue<Node> q = new LinkedList<Node>();

Node node = root;

int depth = 0;

q.add(root);

while(!q.isEmpty()) {

node = q.poll();

visitor.head(node, depth);

for(int i=0; i<node.childNodeSize(); i++) {

    q.add(node.childNode(i));

}

}

}

**클라이언트 호출**

public static Elements collect(Evaluator eval, Element root) {

    Elements elements = new Elements();

    //MasterTraverse for\_dfs = new NodeTraversor(new Accumulator(root, elements, eval));

    MasterTraverse for\_bfs = new NodeTraversor\_BFS(new Accumulator(root, elements, eval));

    //for\_dfs.traverse(root);

    for\_bfs.traverse(root);;

    return elements;

}

**3-1-3. Whitelist.class 설계 개선**

**문제점**

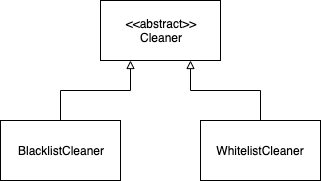
**스크린샷이(가) 표시된 사진

자동 생성된 설명**

Safety 클래스의 Cleaner를 보면 Cleaner 클래스는 whitelist를 필드로 가지고있다. Whitelist에서 설정한 Tag들을 Cleaner에서 가져와서, 설정된 Tag만을 파싱해서 가져오는 기능이다.

Whitelist를 이용하면 사용자가 원하는 Tag만 골라서 파싱할 수 있다. 하지만 만약 10개의 태그가 존재하고, 사용자는 9개의 Tag를 원한다면 “9개의 Tag를 원한다” 보다는 “1개의 Tag를 제외하겠다”가 훨씬 편하다. 그렇기 때문에 Blacklist를 만들었다.

**해결 방안 및 개선된 코드 – Strategy**

****

기존의 Whitelist와 반대되는 Blacklist를 만든다는게 기본적인 생가이지만 Whitelist 클래스는 그 존재 자체가 행위에 해당된다. 즉 추상화할 행위가 없다는 것이다. 그렇기 때문에 더 상위 개념인 Cleaner 중 공통 로직이 들어가는 copySafeNodes를 추상화시켰다.

abstract int copySafeNodes(Element source, Element dest);

그리고 사용자가 원하는 상황에 따라 Blacklist, Whitelist를 적용시킨다.

public static String cleanWhitelist(String bodyHtml, Filter filter) {

return cleanWhitelist(bodyHtml, "", filter);

}

public static String cleanBlacklist(String bodyHtml, Filter filter) {

return cleanBlacklist(bodyHtml, "", filter);

}

위와 같이 구조 개선을 하였고, 기능 구현은 TODO로 남겨놓았다. 그리고 기존의 Whitelist 기능은 다음과 같이 정상적으로 모든 테스트를 통과한다.

**텍스트이(가) 표시된 사진

자동 생성된 설명**

**4. 테스트수행 내역**  
**4-1. 테스트 케이스**

**Document – Test Case**

public class DocumentTypeTest {

@Test

public void constructorValidationOkWithBlankName() {

DocumentType fail = new DocumentType("","", "");

}

@Test(expected = IllegalArgumentException.class)

public void constructorValidationThrowsExceptionOnNulls() {

DocumentType fail = new DocumentType("html", null, null);

}

@Test

public void constructorValidationOkWithBlankPublicAndSystemIds() {

DocumentType fail = new DocumentType("html","", "");

}

@Test public void outerHtmlGeneration() {

DocumentType html5 = new DocumentType("html", "", "");

assertEquals("<!doctype html>", html5.outerHtml());

DocumentType publicDocType = new DocumentType("html", "-//IETF//DTD HTML//", "");

assertEquals("<!DOCTYPE html PUBLIC \"-//IETF//DTD HTML//\">", publicDocType.outerHtml());

DocumentType systemDocType = new DocumentType("html", "", "http://www.ibm.com/data/dtd/v11/ibmxhtml1-transitional.dtd");

assertEquals("<!DOCTYPE html \"http://www.ibm.com/data/dtd/v11/ibmxhtml1-transitional.dtd\">", systemDocType.outerHtml());

DocumentType combo = new DocumentType("notHtml", "--public", "--system");

assertEquals("<!DOCTYPE notHtml PUBLIC \"--public\" \"--system\">", combo.outerHtml());

}

@Test public void testRoundTrip() {

String base = "<!DOCTYPE html>";

assertEquals("<!doctype html>", htmlOutput(base));

assertEquals(base, xmlOutput(base));

String publicDoc = "<!DOCTYPE html PUBLIC \"-//W3C//DTD XHTML 1.0 Transitional//EN\" \"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd\">";

assertEquals(publicDoc, htmlOutput(publicDoc));

assertEquals(publicDoc, xmlOutput(publicDoc));

String systemDoc = "<!DOCTYPE html SYSTEM \"exampledtdfile.dtd\">";

assertEquals(systemDoc, htmlOutput(systemDoc));

assertEquals(systemDoc, xmlOutput(systemDoc));

String legacyDoc = "<!DOCTYPE html SYSTEM \"about:legacy-compat\">";

assertEquals(legacyDoc, htmlOutput(legacyDoc));

assertEquals(legacyDoc, xmlOutput(legacyDoc));

}

private String htmlOutput(String in) {

DocumentType type = (DocumentType) Jsoup.parse(in).childNode(0);

return type.outerHtml();

}

private String xmlOutput(String in) {

return Jsoup.parse(in, "", Parser.xmlParser()).childNode(0).outerHtml();

}

}

**Document - Result**

**스크린샷이(가) 표시된 사진

자동 생성된 설명**

**NodeTraversor – Test Case**

class CollectorTest {

    @Test

    void test\_bfs\_dfs() throws IOException {

        String h = "<html><div id=1><p>Test<p><b>Code</b></p></div><div id=2><span>Test</span></div><div></div></html>";

        String Cssquery = "div";

        Document doc = Jsoup.parse(h);

        Evaluator eval = QueryParser.parse(Cssquery);

        Elements DFS\_elements = new Elements();

        Elements BFS\_elements = new Elements();

        MasterTraverse for\_dfs = new NodeTraversor(new Accumulator(doc, DFS\_elements, eval));

        MasterTraverse for\_bfs = new NodeTraversor\_BFS(new Accumulator(doc, BFS\_elements, eval));

        for\_dfs.traverse(doc);

        for\_bfs.traverse(doc);

        assertEquals(3, DFS\_elements.size());

        assertEquals(3, BFS\_elements.size());

    }

    @Test

    void test\_bfs\_dfs2() throws IOException {

        String h = "<p><img src=foo.png id=1><img src=bar.jpg id=2><img src=qux.JPEG id=3><img src=old.gif><img></p>";

        String Cssquery = "img";

        Document doc = Jsoup.parse(h);

        Evaluator eval = QueryParser.parse(Cssquery);

        Elements DFS\_elements = new Elements();

        Elements BFS\_elements = new Elements();

        MasterTraverse for\_dfs = new NodeTraversor(new Accumulator(doc, DFS\_elements, eval));

        MasterTraverse for\_bfs = new NodeTraversor\_BFS(new Accumulator(doc, BFS\_elements, eval));

        for\_dfs.traverse(doc);

        for\_bfs.traverse(doc);

        assertEquals(5, DFS\_elements.size());

        assertEquals(5, BFS\_elements.size());

    }

    @Test

    void test\_bfs\_dfs3() throws IOException {

        String h = "<div title=foo /><div title=bar /><div /><p></p><img /><span title=qux>";

        String Cssquery = "div";

        Document doc = Jsoup.parse(h);

        Evaluator eval = QueryParser.parse(Cssquery);

        Elements DFS\_elements = new Elements();

        Elements BFS\_elements = new Elements();

        MasterTraverse for\_dfs = new NodeTraversor(new Accumulator(doc, DFS\_elements, eval));

        MasterTraverse for\_bfs = new NodeTraversor\_BFS(new Accumulator(doc, BFS\_elements, eval));

        for\_dfs.traverse(doc);

        for\_bfs.traverse(doc);

        assertEquals(3, DFS\_elements.size());

        assertEquals(3, BFS\_elements.size());

    }

}

**NodeTraversor - Result**

**스크린샷이(가) 표시된 사진

자동 생성된 설명**

**5. GitHub 활동**  
**5-1. 활동 내역**

**Issue**

**스크린샷이(가) 표시된 사진

자동 생성된 설명**

**Pull Request**

**스크린샷이(가) 표시된 사진

자동 생성된 설명**

**Summary**

스크린샷이(가) 표시된 사진

자동 생성된 설명

**5-2. 기여 내역**

**Commits**

**스크린샷이(가) 표시된 사진

자동 생성된 설명**