

# Design Patterns Term Project

# JSoup 기능 확장 및 설계 개선

20165289 김지민

20172151 이성민

20161493 하태윤

## JSoup 개요

### Jsoup: Java HTML Parser

JSoup은 HTML 문서에 저장된 데이터를 구문 분석, 추출 및 조작하도록 설계된 오픈 소스 Java 라이브러리다. Jsoup은 다음과 같은 기능을 한다.

- 1) URL, 파일, 문자열을 소스로 하여 HTML을 파싱한다.
- 2) DOM 구조를 추적하거나 CSS 선택자를 사용하여 데이터를 찾아 추출해낸다.
- 3) 문서 내의 HTML 요소, 속성, 텍스트를 조작할 수 있다.
- 4) 사용자가 입력한 데이터로부터 XSS 공격을 방지하기 위해서 안전한 화이트 리스트 방식으로 지정된 태그만 남기고 나머지는 제거할 수 있다.
- 5) 깔끔한 형태의 HTML을 출력할 수 있다.

### Open source

JSoup은 MIT 라이선스에 따라 배포되는 오픈소스 라이브러리다.

### JSoup 사용법

JSoup을 사용하기 위해서는 <http://jsoup.org/download>에서 다운로드 받은 후 해당 자바 프로젝트에 jar 파일을 추가해야 한다. Jsoup을 사용할 파일에 `org.jsoup.Jsoup`, `org.jsoup.nodes.Document`, `org.jsoup.select.Elements` 등 필요한 패키지를 import 해준다.

## JSoup 설계 및 구현 조사

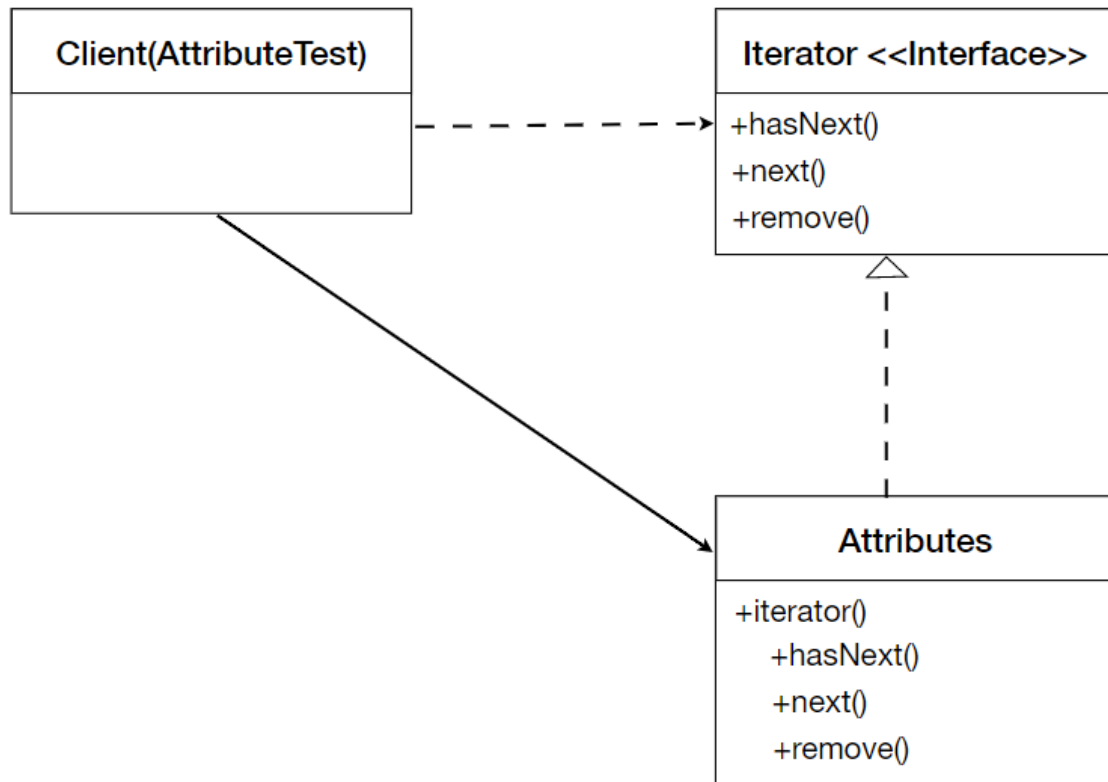
### 설계 Overview

JSoup은 JSoup class 내의 `connect` 함수를 사용하여 제공된 url과 연결한다. 그 후 `get()`을 사용하여 해당 url 내의 html 문서를 parsing한 후 Document 객체에 넣는다. Document의 `select` 함수를 사용하여 원하는 값을 선택하여 Elements 객체에 저장한다.

## 적용되어 있는 설계패턴 소개

- Iterator Pattern: 자료구조에 상관없이 객체 접근 방식을 통일시키고자 할 때 사용된다.

1)



판단근거1: Attributes class가 Iterator interface를 implement하여 hasNext(), next(), remove()를 override한다.

판단근거2: Attributes class의 iterator()는 iterator의 method들을 override하는 동시에 create한다.

판단근거3: AttributeTest class에서 Attribute 객체를 사용하여 implement한 iterator를 사용한다.

판단근거2

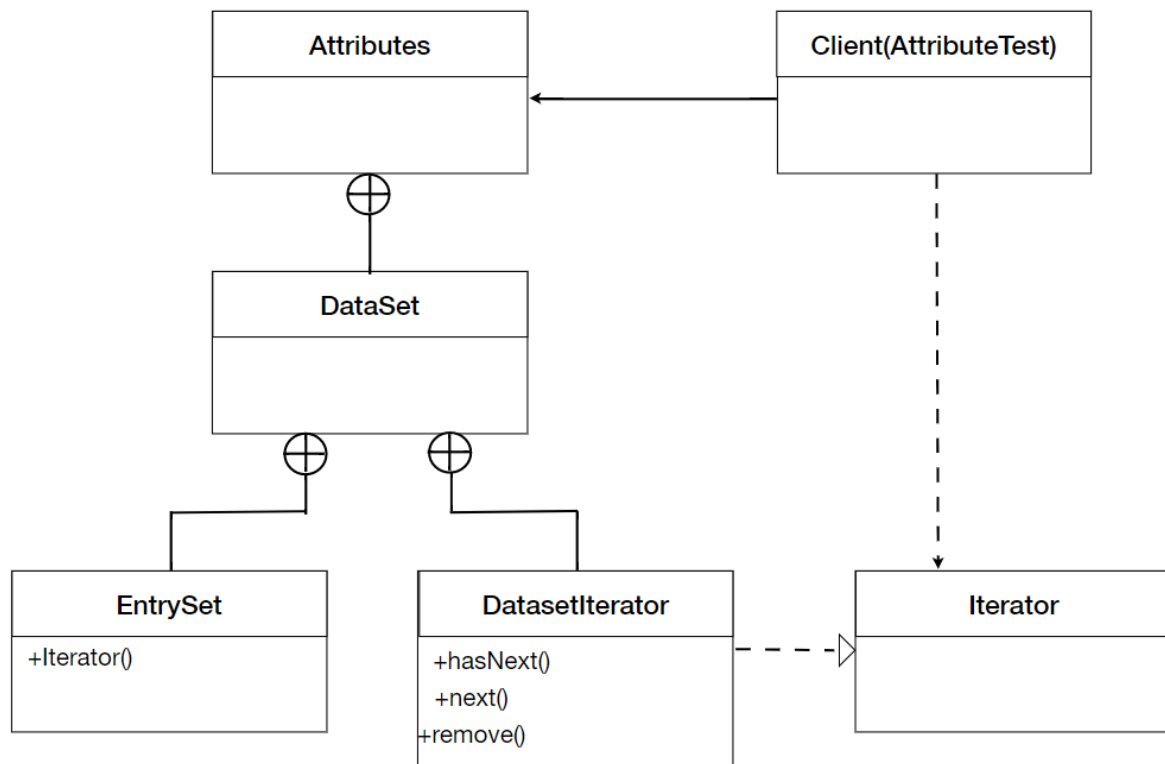
```
public Iterator<Attribute> iterator() {  
    return new Iterator<Attribute>() {  
        int i = 0;  
  
        @Override  
        public boolean hasNext() { return i < size; }  
  
        @Override  
        public Attribute next() {  
            final Attribute attr = new Attribute(keys[i], vals[i], parent: Attributes.this);  
            i++;  
            return attr;  
        }  
    }  
    판단근거1  
    @Override  
    public void remove() { Attributes.this.remove(--i); // next() advanced, so rewind }  
};  
}
```

@Test

```
public void testIteratorRemovable() {  
    Attributes a = new Attributes();  
    a.put("Tot", "a&p");  
    a.put("Hello", "There");  
    a.put("data-name", "Jsoup");  
    assertTrue(a.containsKey("Tot"));  
  
    Iterator<Attribute> iterator = a.iterator();  
    Attribute attr = iterator.next();  
    assertEquals("Tot", attr.getKey());  
    iterator.remove();  
    assertEquals(2, a.size());  
    attr = iterator.next();  
    assertEquals("Hello", attr.getKey());  
    assertEquals("There", attr.getValue());  
}
```

판단근거3

2)



판단근거1: DatasetIterator class에서 iterator interface를 implement하여 hasNext(), next(), remove()를 override한다.

판단근거2: EntrySet class의 iterator()에서 DatasetIterator 객체를 생성한다.

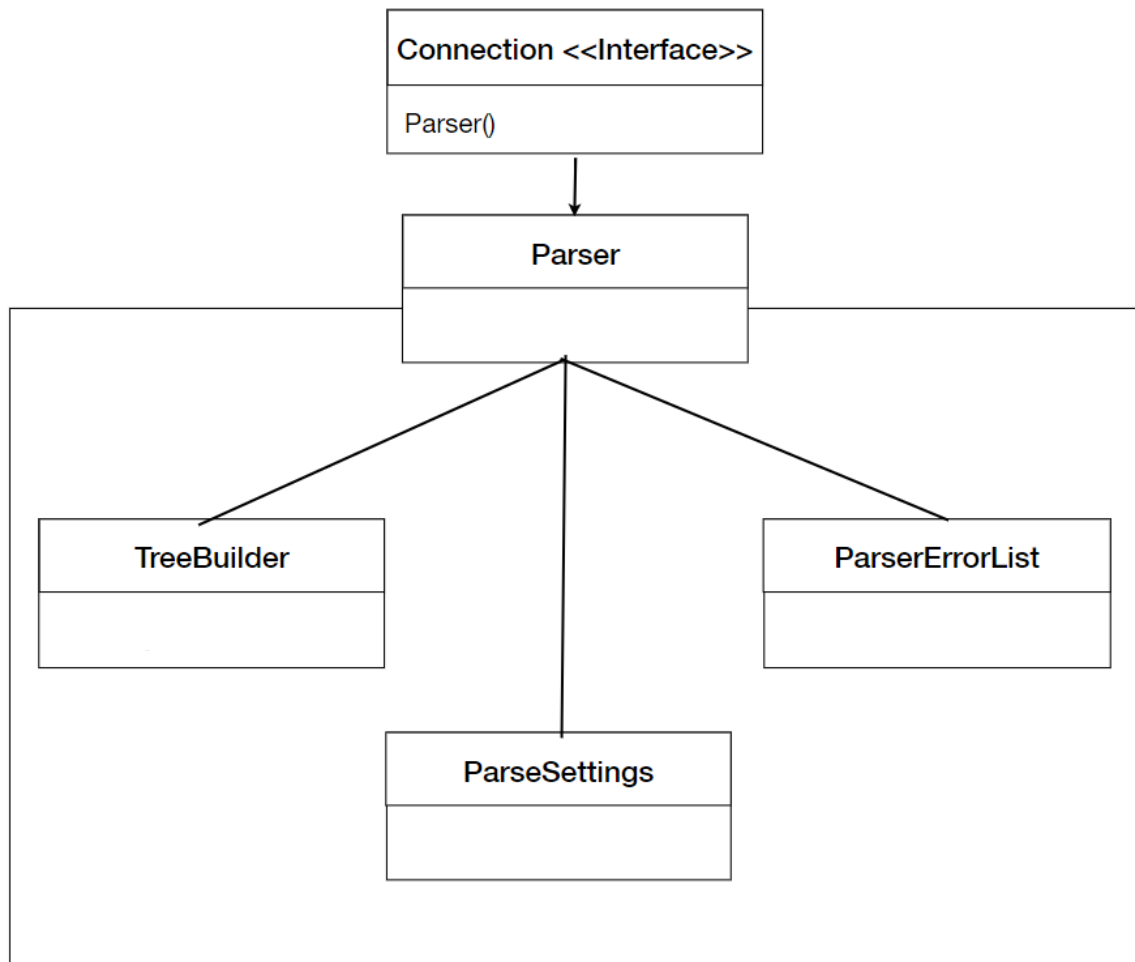
판단근거1

```
private class DatasetIterator implements Iterator<Map.Entry<String, String>> {
    private Iterator<Attribute> attrIter = attributes.iterator();
    private Attribute attr;
    public boolean hasNext() {
        while (attrIter.hasNext()) {
            attr = attrIter.next();
            if (attr.isDataAttribute()) return true;
        }
        return false;
    }
    public Entry<String, String> next() {
        return new Attribute(attr.getKey().substring(dataPrefix.length()), attr.getValue());
    }
    public void remove() { attributes.remove(attr.getKey()); }
}
```

```
private class EntrySet extends AbstractSet<Map.Entry<String, String>> {  
  
    @Override  
    public Iterator<Map.Entry<String, String>> iterator() {  
        return new DatasetIterator();  
    }  
  
    @Override  
    public int size() {  
        int count = 0;  
        Iterator iter = new DatasetIterator();  
        while (iter.hasNext())  
            count++;  
        return count;  
    }  
}
```

판단근거2

- Façade Pattern: 단순화된 인터페이스를 통해 서브시스템을 더 쉽게 사용할 수 있도록 하기위한 용도로 쓰인다.



판단근거1: Parser class는 Treebuilder, ParseErrorList, ParseSettings 객체를 선언하여 사용한다.

판단근거2: Connection interface는 Parser 객체를 선언하여 parser의 기능을 간편하게 사용한다.

```

package org.jsoup.parser;

import ...

/**
 * Parses HTML into a {@link org.jsoup.nodes.Document}. Generally best to use one of the more convenient parse methods
 * in {@link org.jsoup.Jsoup}.
 */
public class Parser {
    private TreeBuilder treeBuilder;
    private ParseErrorList errors;
    private ParseSettings settings;

    /**
     * Create a new Parser, using the specified TreeBuilder
     * @param treeBuilder TreeBuilder to use to parse input into Documents.
     */
    public Parser(TreeBuilder treeBuilder) {
        this.treeBuilder = treeBuilder;
        settings = treeBuilder.defaultSettings();
        errors = ParseErrorList.noTracking();
    }
}

```

**판단근거1**

```

TokeniserState.java x HtmlTreeBuilderState.java x HtmlTreeBuilder.java x Parser.java x Connection.java x Connection x
/**
 * Specify the parser to use when parsing the document.
 * @param parser parser to use.
 * @return this Request, for chaining
 */
Request parser(Parser parser);

/**
 * Get the current parser to use when parsing the document.
 * @return current Parser
 */
Parser parser();

/**
 * Sets the post data character set for x-www-form-urlencoded post data
 * @param charset character set to encode post data
 * @return this Request, for chaining
 */
Request postDataCharset(String charset);

/**
 * Gets the post data character set for x-www-form-urlencoded post data
 * @return character set to encode post data
 */

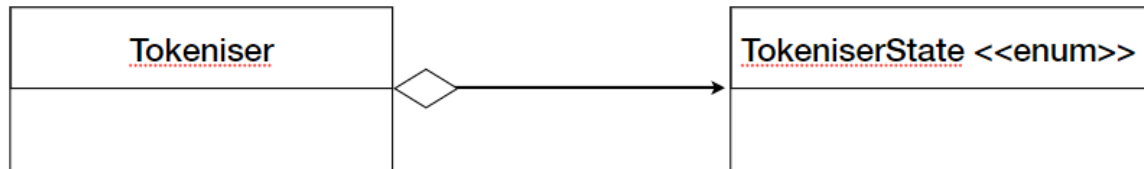
```

**판단근거2**



- **State Pattern**: 객체가 특정 상태에 따라 행위를 달리하는 상황에서 상태를 객체화하여 상태가 행동할 수 있도록 위임하는 패턴이다.

1)



판단근거1: TokeniserState에 Tokeniser의 state들을 enum으로 정의하고 state마다 read()를 정의하였다.

판단근거2: Tokeniser class에서 TokeniserState 객체를 선언하여 상태가 변할 때마다 각 state의 read()를 사용한다.

판단근거3: HTMLTreeBuilder class에서 tagName에 따라 Tokeniser의 state를 변환시킨다.

```

/** States and transition activations for the Tokeniser. */
enum TokeniserState {
    Data {...},
    CharacterReferenceInData {...},
    RCDATA {...},
    CharacterReferenceInRCDATA {...},
    Rawtext {...},
    ScriptData {...},
    PLAINTEXT {...},
    TagOpen {...},
    EndTagOpen {...},
    TagName {...},
    RCDATAlessthanSign {...},
    RCDATAEndTagOpen {...},
    RCDATAEndTagName {...},
    RawtextlessthanSign {...},
    RawtextEndTagOpen {...},
    RawtextEndTagName {
        void read(Tokeniser t, CharacterReader r) {
            handleDataEndTag(t, r, Rawtext);
        }
    },
};

```

판단근거1

```

0x0090, 0x2018, 0x2019, 0x201C, 0x201D, 0x2022, 0x2013, 0x2014,
0x202C, 0x2122, 0x0161, 0x203A, 0x0153, 0x009D, 0x017E, 0x0178,
};

static {
    Arrays.sort(notCharRefCharsSorted);
}

private final CharacterReader reader; // html input
private final ParseErrorList errors; // errors found while tokenising

private TokeniserState state = TokeniserState.Data; // current tokenisation state

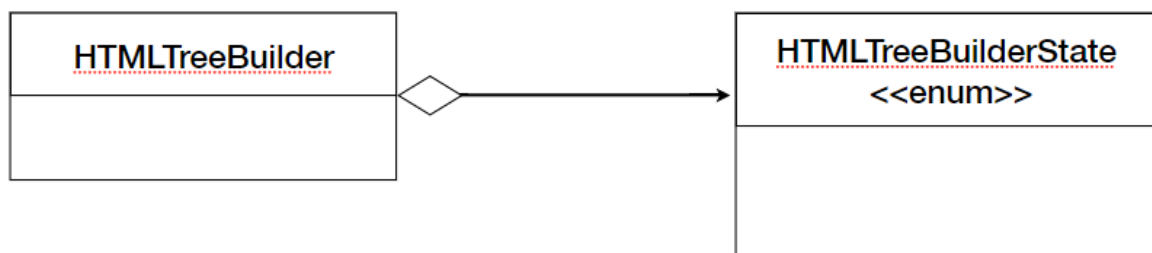
```

판단근거2

```
String contextTag = context.tagName();
if (StringUtil.in(contextTag, ...haystack: "title", "textarea"))
    tokeniser.transition(TokeniserState.Rcdata);
else if (StringUtil.in(contextTag, ...haystack: "iframe", "noembed", "noframes", "style", "xmp"))
    tokeniser.transition(TokeniserState.Rawtext);
else if (contextTag.equals("script"))
    tokeniser.transition(TokeniserState.ScriptData);
else if (contextTag.equals("noscript"))
    tokeniser.transition(TokeniserState.Data); // if scripting enabled, rawtext
else if (contextTag.equals("plaintext"))
    tokeniser.transition(TokeniserState.Data);
else
    tokeniser.transition(TokeniserState.Data); // default
```

판단근거3

2)



판단근거1: HTMLTreeBuilderState에 HTMLTreeBuilder의 state들을 enum으로 정의하고 state마다 함수를 정의하였다.

판단근거2: HTMLTreeBuilder class에서 HTMLTreeBuilderState 객체를 선언하여 노드의 이름에 따라 HTMLTreeBuilderState의 state를 변화시켜준다.

```

enum HtmlTreeBuilderState {
    Initial {
        boolean process(Token t, HtmlTreeBuilder tb) {...}
    },
    BeforeHtml {
        boolean process(Token t, HtmlTreeBuilder tb) {...}

        private boolean anythingElse(Token t, HtmlTreeBuilder tb) {...}
    },
    BeforeHead {
        boolean process(Token t, HtmlTreeBuilder tb) {...}
    },
    InHead {...},
    InHeadNoscript {
        boolean process(Token t, HtmlTreeBuilder tb) {...}

        private boolean anythingElse(Token t, HtmlTreeBuilder tb) {...}
    },
    AfterHead {
        boolean process(Token t, HtmlTreeBuilder tb) {...}

        private boolean anythingElse(Token t, HtmlTreeBuilder tb) {...}
    },
}

```

판단근거1

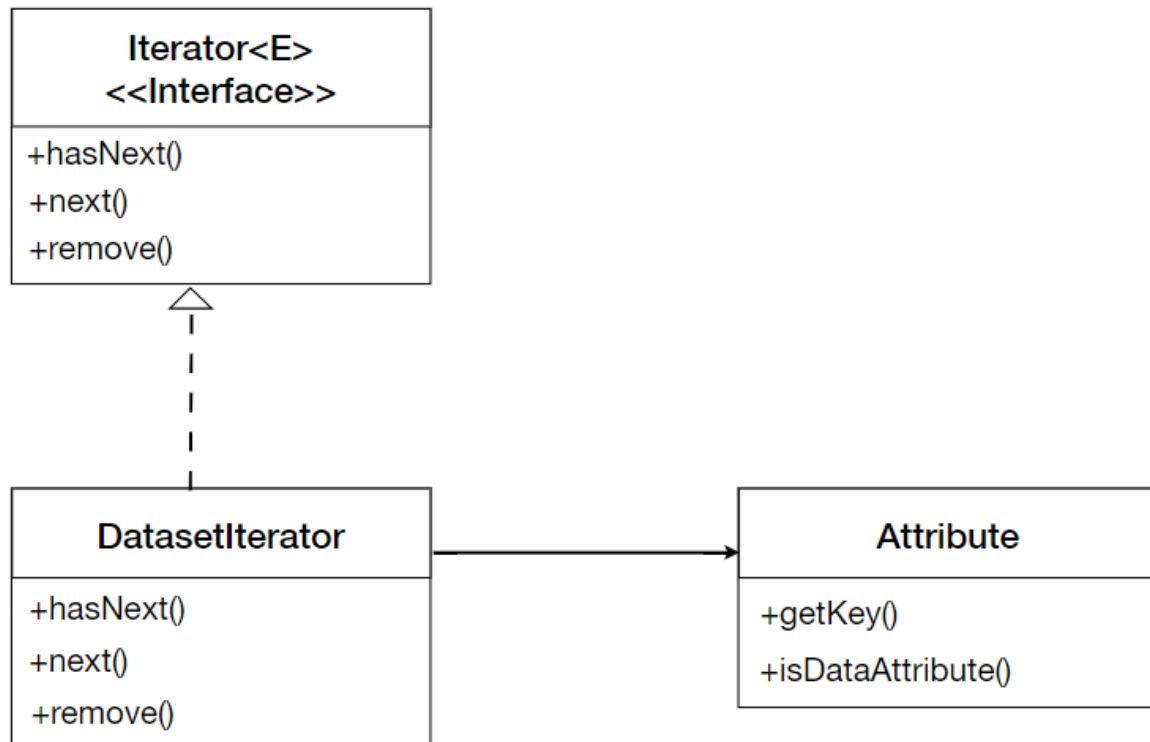
```

String name = node.normalName();
if ("select".equals(name)) {
    transition(HtmlTreeBuilderState.InSelect);
    break; // frag
} else if (("td".equals(name) || "th".equals(name) && !last)) {
    transition(HtmlTreeBuilderState.InCell);
    break;
} else if ("tr".equals(name)) {
    transition(HtmlTreeBuilderState.InRow);
    break;
} else if ("tbody".equals(name) || "thead".equals(name) || "tfoot".equals(name)) {
    transition(HtmlTreeBuilderState.InTableBody);
    break;
} else if ("caption".equals(name)) {
    transition(HtmlTreeBuilderState.InCaption);
    break;
} else if ("colgroup".equals(name)) {
    transition(HtmlTreeBuilderState.InColumnGroup);
    break; // frag
}

```

판단근거2

- Adapter Pattern: 호환성이 없는 인터페이스 때문에 함께 동작할 수 없는 클래스들이 함께 작동하도록 해준다.



판단근거1: DatasetIterator class는 Iterator<E> interface를 implement한다.

판단근거2: DatasetIterator class는 Attribute 객체를 선언하여 override한 `hasNext()`, `next()`, `remove()`에서 Attribute의 함수 `getKey()`, `isDataAttribute()`를 사용한다.

```
private class DatasetIterator implements Iterator<Map.Entry<String, String>> {
    private Iterator<Attribute> attrIter = attributes.iterator();
    private Attribute attr;
    public boolean hasNext() {
        while (attrIter.hasNext()) {
            attr = attrIter.next();
            if (attr.isDataAttribute()) return true;
        }
        return false;
    }

    public Entry<String, String> next() {
        return new Attribute(attr.getKey().substring(dataPrefix.length()), attr.getValue());
    }

    public void remove() { attributes.remove(attr.getKey()); }
}
```

판단근거1

```
private class DatasetIterator implements Iterator<Map.Entry<String, String>> {
    private Iterator<Attribute> attrIter = attributes.iterator();
    private Attribute attr;
    public boolean hasNext() {
        while (attrIter.hasNext()) {
            attr = attrIter.next();
            if (attr.isDataAttribute()) return true;
        }
        return false;
    }

    public Entry<String, String> next() {
        return new Attribute(attr.getKey().substring(dataPrefix.length()), attr.getValue());
    }

    public void remove() { attributes.remove(attr.getKey()); }
}
```

판단근거2

```
protected boolean isDataAttribute() {
    return isDataAttribute(key);
}
```

```
public String getKey() {
    return key;
}
```

## 🌀 팀이 수행한 기능 확장과 설계 개선

### 확장된 기능

#### 1. Jsoup을 이용하여 parsing한 html을 태그의 형식에 맞게 md형식의 파일로 바꿔주는 기능

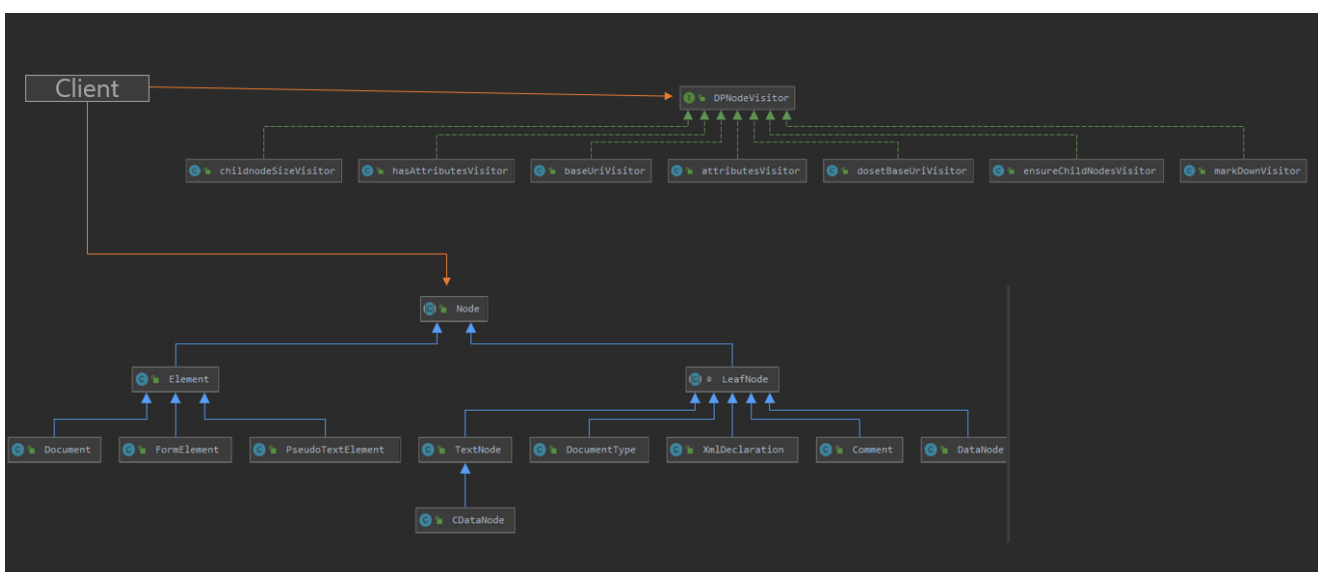
- 1) Jsoup을 이용하여 html 파일을 받아와서 htmltree를 build한다.
- 2) 기존의 select할 때 사용하던 traverse를 이용하여, 모든 트리를 traverse하면서 html의 내용을 md 파일에 write한다.
- 3) Html 웹페이지를 jsoup을 이용하여 md파일로 변환하여, 내 로컬에 웹페이지가 md 형식의 파일로 저장된다,

#### 2. Image Node를 생성하여 Html Tree가 build될 때 자동으로 html 파일에 포함된 사진을 로컬에 저장시켜주는 기능

- 1) Image Builder를 이용하여, Image의 링크만을 담고 있는 Image node class 생성
- 2) Tree가 생성될 때, image node의 링크를 통해 웹에 존재하는 이미지를 자동으로 로컬에 저장.

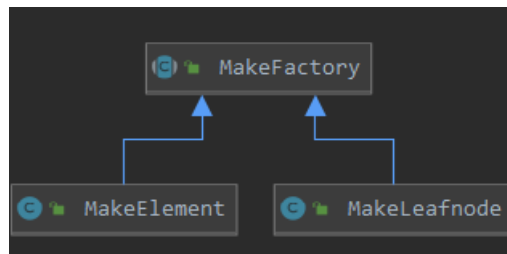
### 설계 개선 내용

#### 설계 1: Visitor Pattern



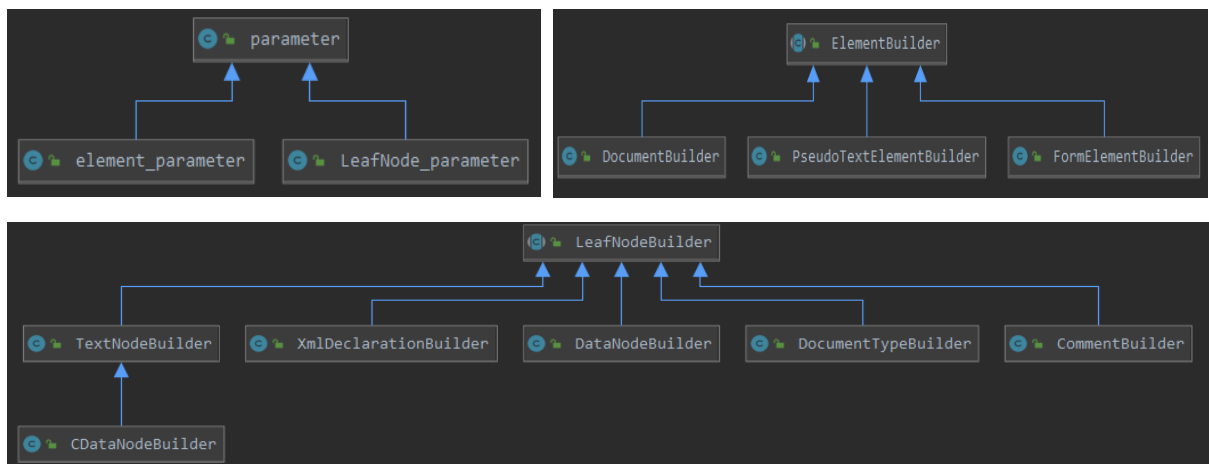
Jsoup의 Node package에서 Abstract class Node를 상속받는 Element와 LeafNode가 생성되고, Element와 LeafNode를 상속받는 Concrete class들이 위와 같이 생성된다. 이들의 공통된 로직을 응집하고 확장을 용이하게 하기 위해 DPNodeVisitor라는 Visitor interface를 생성하였다. 그리고 각각의 기능을 수행하는 Concrete Visitor(attributesVisitor, childnodeSizeVisitor, dosetBaseUriVisitor, ensureChildNodeVisitor, hasAttributesVisitor, basUriVisitor)들을 생성하였다. 이를 통해 Node에서 abstract로 구현되어 element와 leafnode에서 구현하는 공통된 로직들을 visitor로 응집시켰다.

## 설계 2: Factory Pattern



MakeElement, MakeLeafnode class는 abstract class인 MakeFactory를 상속받아 노드를 반환하는 createNode 함수를 구현한다. 각각의 class는 parameter를 통해 전달받은 데이터의 값에 따라 해당하는 constructor를 사용하여 element와 leafnode를 생성한다.

## 설계 3: Builder Pattern



Element class를 상속받은 FormElement, Document, PseudoTextElement와 LeafNode를 상속받은 TextNode, CDataNode, XmlNodeDeclaration, DataNode, Comment, DocumentType에 대한 각각의



builder class를 생성하였다. 각각의 builder class는 ElementBuilder, LeafNodeBuilder를 상속받아 abstract method들을 구현한다. Builder class들은 constructor들의 parameter를 product로 생성하며 client인 MakeElement, MakeLeafNode class는 director인 ElementDirector와 LeafNodeDirector를 통해 product에 접근한다.

## 상세한 변경 내용 설명 및 기존 설계/코드와의 비교

### 설계 1: Visitor Pattern

- 기존 설계 / 코드

```
// public abstract Attributes attributes();
```

[ Node class내의 abstract class ]

```
@Override
public final Attributes attributes() {
    ensureAttributes();
    return (Attributes) value;
}
```

[ LeafNode내의 abstract class 구현부분 ]

```
@Override
public Attributes attributes() {
    if (!hasAttributes())
        attributes = new Attributes();
    return attributes;
}
```

[ Element내의 abstract class 구현부분 ]

- 변경 설계 / 코드

1. 두 가지 기능을 DPNodeVisitor를 상속받은 attributesVisitor로 분리.

```

package org.jsoup.nodes;

public class attributesVisitor implements DPNodeVisitor {

    @Override
    public Object visit(Element element) {
        if (!(boolean)(element.accept(new hasAttributesVisitor()))
            element.attributes = new Attributes();
        return element.attributes;
    }

    @Override
    public Object visit(LeafNode leafnode) {
        leafnode.ensureAttributes();
        return (Attributes) leafnode.value;
    }
}

```

2. Element와 LeafNode에 Visitor를 실행가능한 accept 메소드를 추가.

```

public abstract Object accept(DPNodeVisitor v);

```

[ node class의 abstract method ]

```

public Object accept(DPNodeVisitor v) { return v.visit( leafnode: this); }

```

```

public Object accept(DPNodeVisitor v) { return v.visit( element: this); }

```

[ Element 와 Leafnode에 abstract 메소드 구현 ]

3. 다른 class에서 element와 leafnode에 구현된 메소드 대신 accept를 이용하여 visitor 적용.

Element.attributes() => element.accept(new attributesVisitor()) 사용.

```

List<org.jsoup.nodes.Attribute> values = element.attributes().asList();

```

```

List<org.jsoup.nodes.Attribute> values =
    ((Attributes)(element.accept(new attributesVisitor()))).asList();

```

Attributes 외의 나머지 5개의 abstract 메소드도 위와 같은 방식으로 visitor로 수정하여, element와 leafnode를 상속받은 concrete class 들은 accept 메소드를 이용하여 visitor를 통해 다양한 기능을 수행할 수 있다.

## 설계 2: Factory Pattern

### 기존 [HTMLTreeBuilder.java](#)

```
void insert(Token.Comment commentToken) {  
    Comment comment = new Comment(commentToken.getData());  
    insertNode(comment);  
}
```

### 기존 [XMLTreeBuilder.java](#)

```
void insert(Token.Comment commentToken) {  
    Comment comment = new Comment(commentToken.getData());  
    Node insert = comment;  
    if (commentToken.bogus && comment.isXmlDeclaration()) {  
        // xml declarations are emitted as bogus comments (which is right)  
        // so we do a bit of a hack and parse the data as an element to put  
        XmlDeclaration decl = comment.asXmlDeclaration(); // else, we could  
        if (decl != null)  
            insert = decl;  
    }  
    insertNode(insert);  
}
```

## MakeLeafNode class

```
public class MakeLeafnode extends MakeFactory<LeafNode_parameter> {
    DataNode a = null;
    DocumentType b = null;
    XmlDeclaration c = null;
    CDataNode d = null;
    Comment e = null;
    TextNode f = null;

    @Deprecated
    public Node createnode(LeafNode_parameter para) {
        if (para.type.equals("DataNode")) {
            if (para.getBaseUrl() == null) {
                a = new DataNode(para.getData());
                return a;
            } else {
                a = new DataNode(para.getData(), para.getBaseUrl());
                return a;
            }
        } else if (para.type.equals("DocumentType")) {
            b = new DocumentType(para.getName(), para.getPubSysKey(), para.getPublicId(), para.getSystemId(), para.getBaseUrl());
            return b;
        } else if (para.type.equals("XmlDeclaration")) {
            if (para.getBaseUrl() == null) {
                c = new XmlDeclaration(para.getName(), para.getisProcessingInstruction());
                return c;
            } else {
                c = new XmlDeclaration(para.getName(), para.getBaseUrl(), para.getisProcessingInstruction());
                return c;
            }
        }
    }
}
```

## MakeElement class

```
public class MakeElement extends MakeFactory<element_parameter>{
    PseudoTextElement a = null;
    Document b = null;
    FormElement c = null;

    public Node createnode(element_parameter para){
        // Node node = null;
        if (para.type.equals("PseudoTextElement")) {
            a = new PseudoTextElement(para.getTag(),para.getBaseUri(),para.getAttributes());
            return a;
        }
        else if (para.type.equals("Document")){
            b = new Document(para.getBaseUri());
            return b;
        }
        else if (para.type.equals("FormElement")) {
            c = new FormElement(para.getTag(),para.getBaseUri(),para.getAttributes());
            return c;
        }
        else {
            return null;
        }
    }
}
```

기존 코드는 HTMLTreeBuilder와 XMLTreeBuilder class에서 comment, xmldeclaration 등 각각의 node를 직접 생성하였다. Factory Pattern을 적용한 후에는 abstract class인 MakeFactory에서 하위 클래스에 노드를 생성하여 반환하는 createnode 함수를 abstract로 정의하여 MakeElement, MakeLeafNode에서 node를 생성하였다. 각각의 class는 parameter를 통해 전달받은 값으로 해당하는 생성자를 사용하여 노드를 생성한 후 반환한다.

### 설계 3: Builder Pattern

#### 변경된 HTMLTreeBuilder.java

```
void insert(Token.Comment commentToken) {  
  
    LeafNodeDirector leafN = new LeafNodeDirector();  
    LeafNodeBuilder Comment= new CommentBuilder( _type: "Comment",commentToken.getData());  
  
    leafN.setLeafNodeBuilder(Comment);  
    leafN.constructparameter();  
    LeafNode_parameter Cparams = leafN.getelement();  
  
    MakeLeafnode Cfactory = new MakeLeafnode();  
  
    Comment comment = (Comment) Cfactory.createnode(Cparams);  
    insertNode(comment);  
}
```

#### 변경된 XMLTreeBuilder.java

```
void insert(Token.Comment commentToken) {  
    LeafNodeDirector leafN = new LeafNodeDirector();  
    LeafNodeBuilder Comment= new CommentBuilder( _type: "Comment",commentToken.getData());  
  
    leafN.setLeafNodeBuilder(Comment);  
    leafN.constructparameter();  
    LeafNode_parameter Cparams = leafN.getelement();  
  
    MakeLeafnode Cfactory = new MakeLeafnode();  
  
    Comment comment = (Comment) Cfactory.createnode(Cparams);  
    Node insert = comment;  
    if (commentToken.bogus && comment.isXmlDeclaration()) {  
        // xml declarations are emitted as bogus comments (which is right for html, but not xml)  
        // so we do a bit of a hack and parse the data as an element to pull the attributes out  
        XmlDeclaration decl = comment.asXmlDeclaration(); // else, we couldn't parse it as a decl, so leave as a comment  
        if (decl != null)  
            insert = decl;  
    }  
    insertNode(insert);  
}
```

## LeafNode\_parameter class

```
package org.jsoup.nodes;

public class LeafNode_parameter extends parameter{
    private String data;
    private String baseUrl;
    private String text;
    private String name;
    private String publicId;
    private String systemId;
    private String pubSysKey;
    private boolean IsProcessingInstruction;

    public LeafNode_parameter(String type) { super.type = type; }

    public void setPubSysKey(String pubSysKey) {
        this.pubSysKey = pubSysKey;
    }

    public String getPubSysKey() { return pubSysKey; }

    public void setData(String data) { this.data = data; }

    public void setBaseUrl(String baseUrl) {
        this.baseUrl = baseUrl;
    }

    public void setText(String text) {
        this.text = text;
    }
}
```

## CommentBuilder class

```
public class CommentBuilder extends LeafNodeBuilder {
    String data;
    String baseUri = null;
    public CommentBuilder(String _type, String _data) {
        super.type = _type;
        data = _data;
    }
    public CommentBuilder(String _type, String _data, String _baseUri){
        super.type = _type;
        data = _data;
        baseUri = _baseUri;
    }
    public void buildPubsyskey() { para.setPubSysKey(null); }
    public void buildData(){
        para.setData(data);
    };
    public void buildbaseUri(){
        para.setBaseUrl(baseUri);
    };
    public void buildText() { para.setText(null); };
    public void buildName() { para.setName(null); };
    public void buildPublicId() { para.setPublicId(null); };
    public void buildSystemId() { para.setSystemId(null); };
    public void buildIsProcessingInstruction() { para.setProcessingInstruction(false); };
}
```

LeafNodeBuilder와 ElementBuilder는 abstract class로 각각 자식 클래스에서 구현해야 할 method들을 abstract로 선언하였다. LeafNode\_parameter와 Element\_parameter class는 parameter를 상속받아 매개변수로 사용되는 변수들을 선언하고, 각 변수에 대한 get(), set()함수를 구현하였다. CommentBuilder 등의 9개 Builder class들은 각각의 부모 클래스의 builder 클래스를 상속받으며, 생성자의 매개변수 종류에 따라 parameter를 사용해 변수의 값을 셋팅한다.

기존의 HTMLTreeBuilder class와 XMLTreeBuilder class에서는 클래스 내에서 직접 Comment 객체를 생성하였다. 그러나 Builder Pattern을 적용한 후에는 builder와 director를 통해 comment 객체를 생성한다.

## 확장기능 1

DPNodeVisitor 를 구현함으로써 element 와 leafnode 내의 구조를 변경하지 않고, visitor만을 추가함으로써 새로운 기능을 확장할 수 있게 되었다. 따라서 위에서 언급한 html파일을 md파일로 만들어주기 위한 새로운 visitor를 생성하여 기존 코드를 수정하지 않고 확장된 기능을 추가할 수 있다.

### 1. Document class에 새로운 기능을 수행하는 markdown() 메소드를 추가

```
public void markdown(String filename) throws IOException {  
    String result = null;  
    FileWriter fw = new FileWriter(filename);  
  
    Elements elements = new Elements();  
    result = MarkNodeTraversor.traverse( root: this);  
    fw.write(result);  
    fw.close();  
}
```

Markdown 메소드는 html 파일의 내용을 기록할 md파일을 불러온 후, MarkNodeTraversor를 이용하여 html내용을 md파일 형식에 맞게 파일에 write한다.

### 2. MarkNodeTraversor class 생성

Marknodetraversor class 는 기존 방식과 같은 방식으로 html tree를 traverse 하지만, 기존에 의 traverse 하면서 사용자가 입력한 쿼리문과 일치하는 element 만 반환한 nodeTraversor class와 달리, element를 돌면서 모든 element를 markdownVisitor을 이용하여 형식에 맞는 String으로 반환해주는 기능이 추가되었다.



```

while (node != null) {
    content = ((String) (node.accept(new markDownVisitor())));
    if (content != null) {
        if (result != null) {
            result = result + content;
        } else
            result = "" + content;
    }
}

```

[ 캡처하지 않은 아래 부분은 기존의 traverse와 동일 ]

3. MarkdownVisitor class : markdown 기능을 수행하기 위해 element의 태그에 맞는 형식으로 해당 태그에 내용을 String으로 반환해준다.

```

public class markDownVisitor implements DPNodeVisitor{

    @Override
    public Object visit(Element element) {
        if(element.tag.toString() == "h1") {
            return "#" + element.text() + "\n";
        } else if(element.tag.toString() == "h2") {
            return "##" + element.text() + "\n";
        } else if(element.tag.toString() == "h3") {
            return "###" + element.text() + "\n";
        } else if(element.tag.toString() == "h4") {
            return "####" + element.text() + "\n";
        } else if(element.tag.toString() == "br") {
            return "<br/>\n\n";
        } else if(element.tag.toString() == "img") {
            return " \n.toString()+")\n";
        } else if(element.tag.toString() == "li") {
            if(element.parentNode().toString().startsWith("<ol>"))
                return "*" + element.text() + "\n";
            else if(element.parentNode().toString().startsWith("<ul>"))
                return "1. " + element.text() + "\n";
            return null;
        }
    }
}

```

1.

```

@Override
public Object visit(LeafNode leafnode) {
    String content = leafnode.value.toString();
    content = content.replaceAll( regex: "\t", replacement: "").replaceAll( regex: "\n", replacement: "").trim();

    if(leafnode.parentNode().toString().startsWith("<body>")&&!content.equals("")&&content!=null){
        return " \n" + content;
    }
    return null;
}
}

```

- 1) Element visit : 태그를 비교하여 태그안의 내용을 해당 태그에 맞는 md형식으로 변환하여 string으로 return.
- 2) LeafNode visit : 태그가 없는 바디의 내용은 Leafnode에서 받아와야 함으로, leafnode visit를 이용하여 String으로 return.

## 확장기능 1 테스트 결과.


Design Pattern Test Page 01

주소창: 주의 요함 | 34.84.237.242:8080/test3

정규화(Normalizati...

# 초코 공룡쿠키 만들기

간편하지만 정성이 가득한 만들기 세트를 만나보세요.



### 재료

- 하트쿠키믹스
- 달걀 노른자 1개
- 버터 40g
- 우유 20ml
- 다크커버쳐초콜릿 30g
- 초코펜(화이트/3개)
- 공룡쿠키커터 3종세트

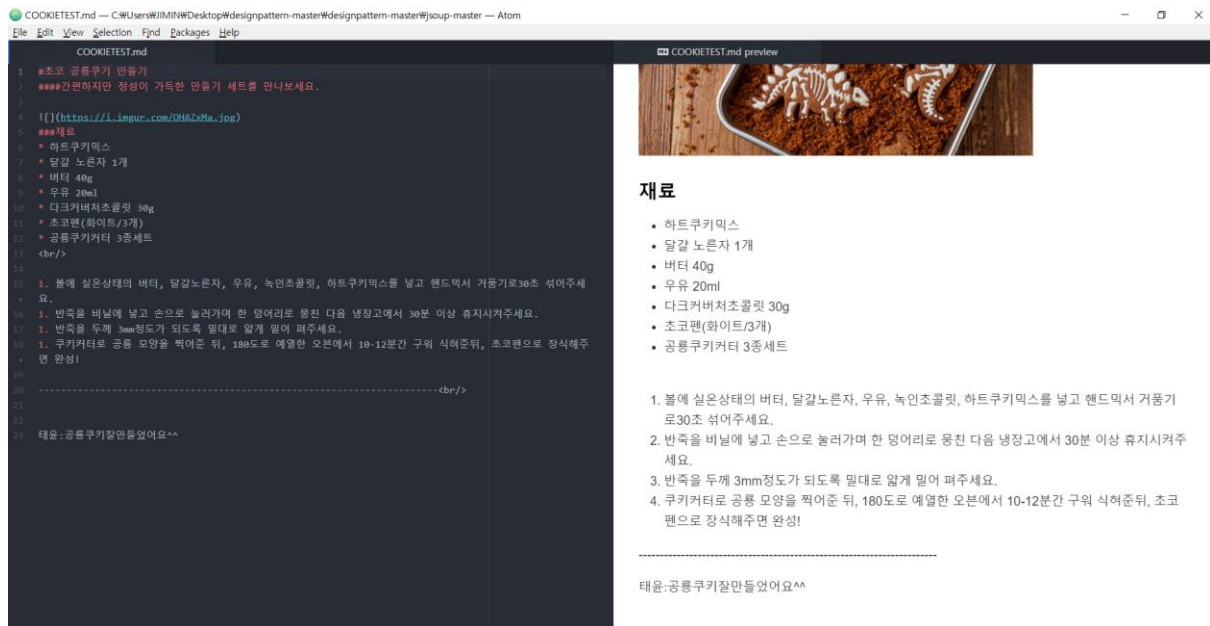
1. 볼에 실온상태의 버터, 달걀노른자, 우유, 녹인초콜릿, 하트쿠키믹스를 넣고 핸드믹서 거품기로30초 섞어주세요.
2. 반죽을 비닐에 넣고 손으로 눌러가며 한 덩어리로 뭉친 다음 냉장고에서 30분 이상 휴지시켜주세요.
3. 반죽을 두께 3mm정도가 되도록 밀대로 얇게 밀어 펴주세요.
4. 쿠키커터로 공룡 모양을 찍어준 뒤, 180도로 예열한 오븐에서 10-12분간 구워 식혀준뒤, 초코펜으로 장식 해주면 완성!

---

태윤 : 공룡 쿠키 잘 만들었어요 ^^

```
doc.markdown("COOKIETEST.md");|
```

<MAIN 안에 TEST CODE>



« 바탕 화면 » designpattern-master-taeyun » designpattern-master » jsoup-master					jsoup-master
	이름	수정한 날짜	유형	크기	
	.idea	2019-12-07 오전 1...	파일 폴더		
	.settings	2019-12-07 오전 1...	파일 폴더		
	bin	2019-12-07 오전 1...	파일 폴더		
src	src	2019-12-07 오전 1...	파일 폴더		
target	target	2019-12-07 오전 1...	파일 폴더		
	.project	2019-12-06 오전 7...	PROJECT 파일	1KB	
	CHANGES	2019-12-06 오전 7...	파일	55KB	
	COOKIETEST	2019-12-07 오전 1...	MD 파일	1KB	
	image1	2019-12-07 오전 1...	JPG 파일	398KB	
	jsoup.iml	2019-12-07 오전 1...	IML 파일	2KB	
	LICENSE	2019-12-06 오전 7...	파일	2KB	
	pom	2019-12-06 오전 7...	XML 문서	10KB	
	README	2019-12-06 오전 7...	MD 파일	3KB	

< 생성된 MD 형식의 파일 >

## 확장기능 2

1. Image를 로컬에 저장하기 위해, Image의 링크를 담고 있는 LeafNode를 상속받는 Image Node를 생성.

```
public class image extends LeafNode {  
  
    public static image createFromEncoded(String encodedData, String baseUri) {  
        String data = Entities.unescape(encodedData);  
        LeafNodeDirector leaf = new LeafNodeDirector();  
        LeafNodeBuilder datanode = new imageBuilder( _type: "image", data);  
  
        leaf.setLeafNodeBuilder(datanode);  
        leaf.constructparameter();  
        LeafNode_parameter params = leaf.getelement();  
  
        MakeLeafnode factory = new MakeLeafnode();  
  
        return (image) factory.createnode(params);  
    }  
}
```

[ image node 내부에서 director와 builder를 이용하여 node를 생성하는 메소드 ]

2. Image Node를 만들기 위해 생성된 imageBuilder에는 이미지를 저장하는 기능을 하는 StoreImage() 메소드를 추가. 생성자에서 storeImage()를 실행함으로써, Image 노드가 생성될 때마다 이미지를 저장할 수 있다.

```
public class imageBuilder extends LeafNodeBuilder {  
    String data;  
    String baseUri = null;  
    int Count = 0;  
    public imageBuilder(String _type, String _data) {  
        super.type = _type;  
        data = _data;  
        StoreImage();  
    }  
    public imageBuilder(String _type, String _data, String _baseUri){  
        super.type = _type;  
        data = _data;  
        baseUri = _baseUri;  
        StoreImage();  
    }  
}
```

[ 생성자에 추가된 StoreImage() 메소드 ]

```

public void StoreImage(){
    Count++;
    URL url = null;
    InputStream in = null;
    OutputStream out = null;

    try {

        url = new URL(data);
        in = url.openStream();
        out = new FileOutputStream( name: "Image"+Count+".jpg");

        while(true){
            //read image
            int data = in.read();
            if(data == -1){
                break;
            }
            //write image
            out.write(data);
        }
        in.close();
        out.close();
    } catch (Exception e) {
        e.printStackTrace();
    }
}

```

[ StoreImage 메소드 ]

3. Image class 생성 후, Html tree를 생성할 때 Tokenizer state 내에서 태그 중 img를 인식하여 트리에 노드를 추가한다.

```

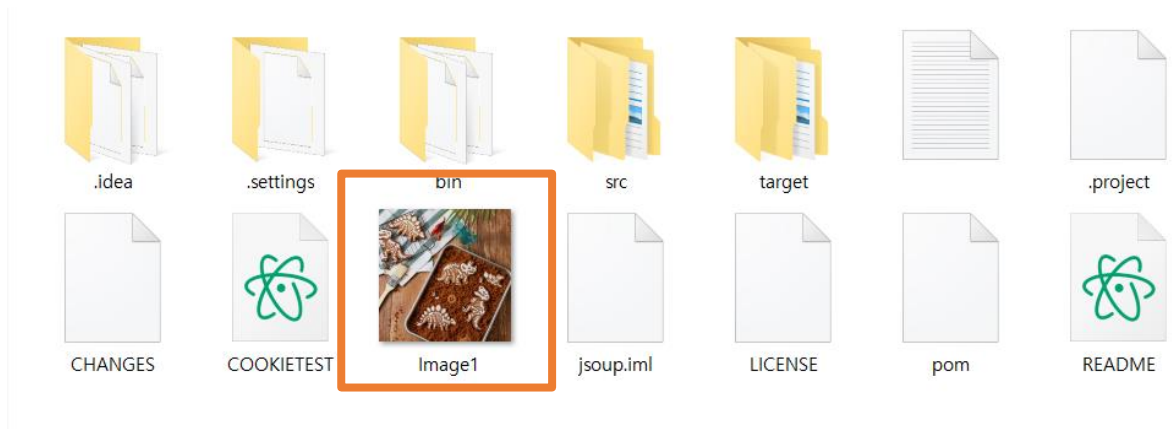
if(t.tagPending.tagName.equals("img")){

    LeafNodeDirector leaf = new LeafNodeDirector();
    LeafNodeBuilder image = new imageBuilder( _type: "image",value.toString());
    leaf.setLeafNodeBuilder(image);
    leaf.constructparameter();
    LeafNode_parameter params = leaf.getelement();

    MakeLeafnode factory = new MakeLeafnode();
    org.jsoup.nodes.image a = (image) factory.createnode(params);
}

```

## 확장기능 2 테스트 결과



[ 위에서 테스트한 웹페이지를 JSOUP을 통해 파싱하였을 때 생성된 Image1 파일 ]

## 도입된 설계 패턴 및 설계 원칙과 적용 이유

### - 설계 1

- 확장에 대해 열려 있어야 하고 수정에 대해서는 닫혀 있어야 한다는 Open Closed Principle을 Visitor Pattern을 이용하여 적용.
- 적용 이유
  - 1) Node들의 구조를 변화시키지 않고, 새로운 기능을 추가 가능하기 때문에 연산에 대한 확장을 용이하게 하기 위함.
  - 2) 공통된 로직들을 Visitor class에 응집하게 하여 디버깅 혹은 분석에 어려움을 줄이기 위함.

### - 설계 2

설계 2에는 Factory Pattern을 도입하였다. Factory Pattern은 node를 생성할 때 element와 leafnode인지에 따라 각각의 factory에서 node를 효율적으로 생성하기 위해 적용하였다.

### - 설계 3

설계 3에는 Builder Pattern을 도입하였다. Builder Pattern은 Factory Pattern을 적용시키면서 클래스의 constructor가 오버로드 되어 있는 경우 factory에서 매개변수에 따라 구분하여 객체를 생

성할 수 없는 상황을 해결하기 위해 적용하였다.

## ✓ 테스트 내역

### 테스트 케이스 (JUnit 테스트 코드)

- Builder Pattern 테스트 코드 예시

```
public class XmlDeclarationBuilderTest {

    @Test
    public void buildbasUri() {
        LeafNodeBuilder xmldeclaration = new
        XmlDeclarationBuilder("XmlDeclaration", "name",null,true);
        xmldeclaration.createNewParameter();
        xmldeclaration.buildbasUri();
        assertEquals(null,xmldeclaration.getParameter().getBaseUrl());
    }

    @Test
    public void buildName() {
        LeafNodeBuilder xmldeclaration = new
        XmlDeclarationBuilder("XmlDeclaration", null,"baseuri",true);
        xmldeclaration.createNewParameter();
        xmldeclaration.buildName();
        assertEquals(null,xmldeclaration.getParameter().getName());
    }

    @Test
    public void buildIsProcessingInstruction() {
        LeafNodeBuilder xmldeclaration = new
        XmlDeclarationBuilder("XmlDeclaration", "name","baseuri",false);
        xmldeclaration.createNewParameter();
        xmldeclaration.buildIsProcessingInstruction();

        assertEquals(false,xmldeclaration.getParameter().getisProcessingInstruction());
    }
}
```

LeafNodeBuilder class를 사용하여 XmlDeclarationBuilder("XmlDeclaration","name",false)를 생성하였다. XmlDeclarationBuilder의 createNewParameter()를 사용하여 parameter를 생성한 후 set()을 사용하여 변수의 값을 설정해주었다. 테스트는 Junit의 assertEquals함수를 사용하여 임의로 입력한 name과 생성된 객체의 name이 같은 값인지 확인하였다.

- Factory Pattern 테스트 코드 예시

```
@Test
public void createnode() {
    LeafNodeDirector leaf = new LeafNodeDirector();
    LeafNodeBuilder xmlDeclaration = new
XmlDeclarationBuilder("XmlDeclaration","xml",false);
    try {
        leaf.setLeafNodeBuilder(xmlDeclaration);
        leaf.constructparameter();
    } catch (Exception e1){
        System.out.println(e1);
    }
    LeafNode_parameter params = leaf.getelement();

    MakeLeafnode factory = new MakeLeafnode();
    XmlDeclaration decl;
    try {
        decl = (XmlDeclaration) factory.createnode(params);
    } catch (Exception e2){
        System.out.println(e2);
    }

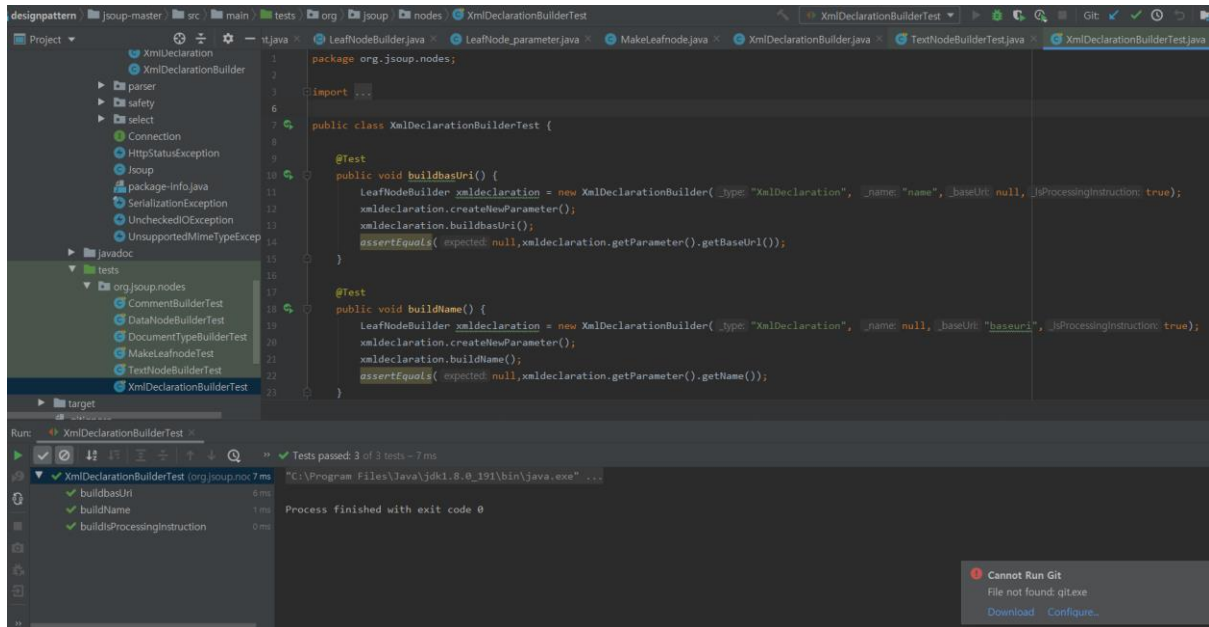
    assertEquals(false,xmlDeclaration.getParameter().getisProcessingInstruction());
    assertEquals("xml", xmlDeclaration.getParameter().getName());
}
```

LeafNodeDirector 객체와 LeafNodeBuilder 객체를 생성하였다. Director class의 setLeafNodeBuilder()와 constructparameter()를 호출하여 error가 발생하는지 확인하였다. 또한 createnode()를 사용하여 node를 반환하는 부분을 try catch를 사용하여 확인하였다. factory 역할을 하는 MakeLeafNode class를 통해 node가 제대로 완성되는지 확인하기 위해 생성된 노드의 값과 처음에 생성할 때 입력해주었던 값을 assertEquals()를 사용하여 확인하였다.



## 테스트 케이스 적용 결과 (스크린 샷)

- Builder Pattern 테스트 케이스 적용 결과



## ≡ GitHub 프로젝트 활용 요약

팀의 github 프로젝트 주소

<https://github.com/hataeyunn/designpattern.git>

팀원별 기여를 잘 나타낼 수 있는 각종 자료

하태윤: Factory Pattern과 추가기능 구현

김지민: Visitor Pattern과 추가기능 구현

이성민: Builder Pattern과 test code 구현