**설계패턴 Team Project**

**‘Enhancing Jsoup’** 

**Subject** 설계 패턴

**Professor** 이찬근 교수님

**Name** 박준영 (20143846)

손창우 (20146518)

양동욱 (20141095)

이강희 (20142748)

**목차**

1. **Jsoup 패턴/구조 분석**

* Decorator 패턴
* Visitor 패턴
* Builder 패턴 & Strategy 패턴
* State 패턴
* Observer 패턴
* Iterator 패턴
* Composite 패턴
* Factory Method

1. **새로 추가할 기능**

* getAppliedStyle()

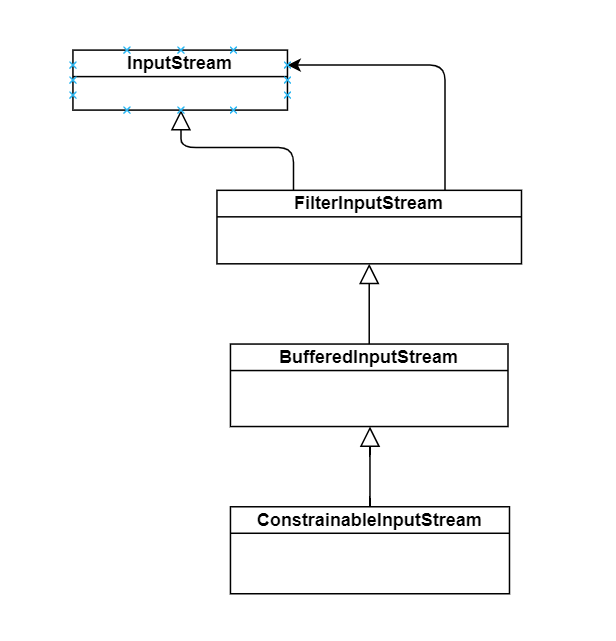
1. **구현 방법**

* Created
* Modified

1. **Test**

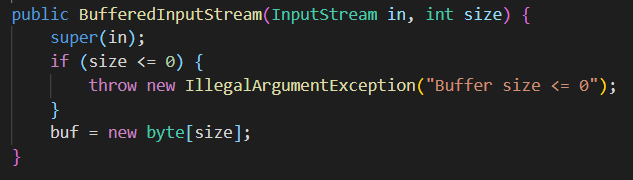
**1. Jsoup 패턴/구조 분석**

**1) Decorator 패턴**

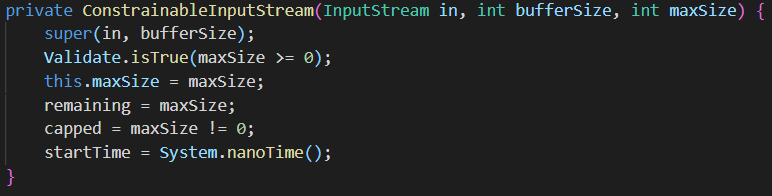




- 기본적으로 BufferedInputStream은 Java에서 제공하는 Decorator 패턴의 클래스이다. JSoup에서 BufferedInputStream을 상속하여 Client에서 InputStream이 전달하는 크기를 제한할 수 있는 기능을 가진 Decorator를 구현하였다.

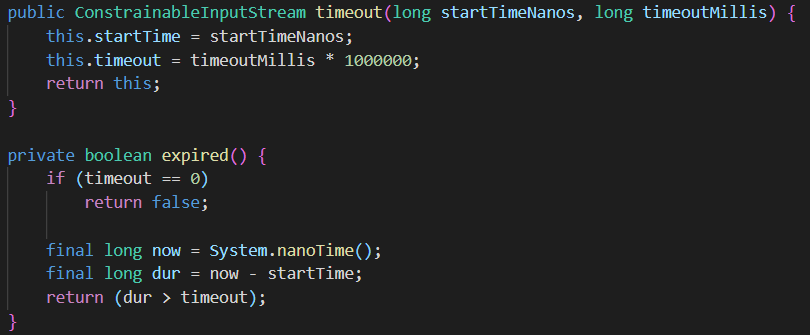


**(1) BufferedInputStream의 생성자**

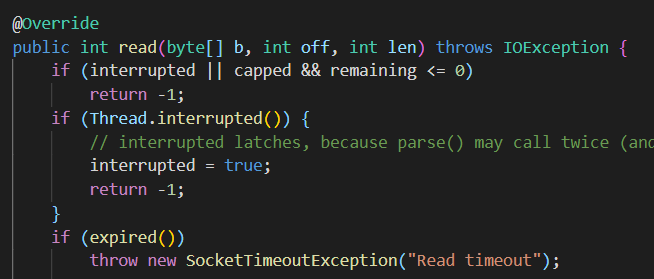


**(2) ConstrainableInputStream의 생성자**

- (1), (2) 두 생성자를 비교해보면 추가적인 기능으로 읽어들이는 Buffer의 Max size를 Client가 직접 설정할 수 있음을 알 수 있다.

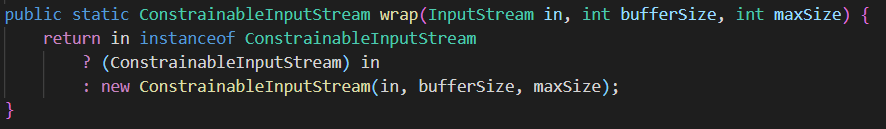


**(3) timeout, expired 메소드**



**(4) inputStream의 read 메소드를 Override**

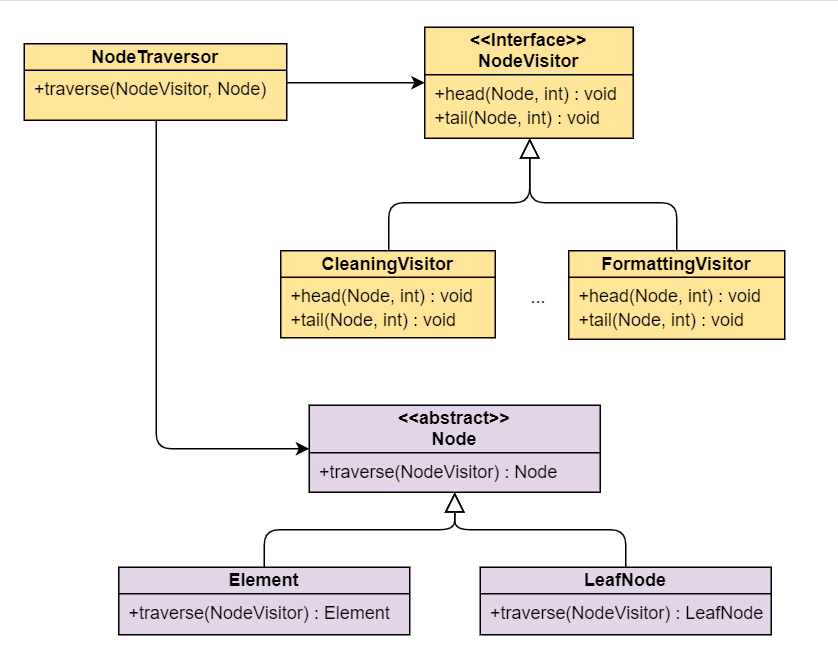
- (3), (4) 그림을 보면 Time out이라는 기능이 추가되었음을 알 수 있다. inputStream을 받아들일 때 (3)의 expired함수를 통해 SocketTimeoutException을 처리한다.



**(5) wrap 메소드**

(5) wrap은 Override나 Overloadding이 아닌 새로 정의된 메소드이다. inputStream을 상속하여 만들어진 객체들을 ConstrainableInputStream으로 변형시켜 리턴한다. 이를 통해 Decorator 패턴을 적용하고 있음을 직접적으로 알 수 있다.

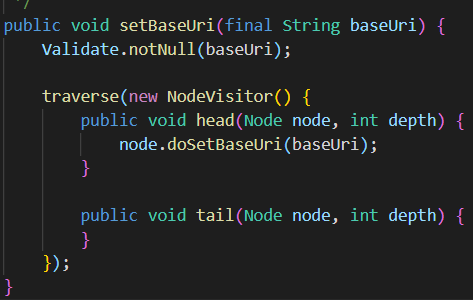
**2) Visitor 패턴**



- NodeTraversor 내의 public static인 traverse를 호출할 때 탐색을 시작할 Node와 탐색할 때 사용될 함수 head, tail이 Override된 NodeVisitor를 매개변수로 전달한다.

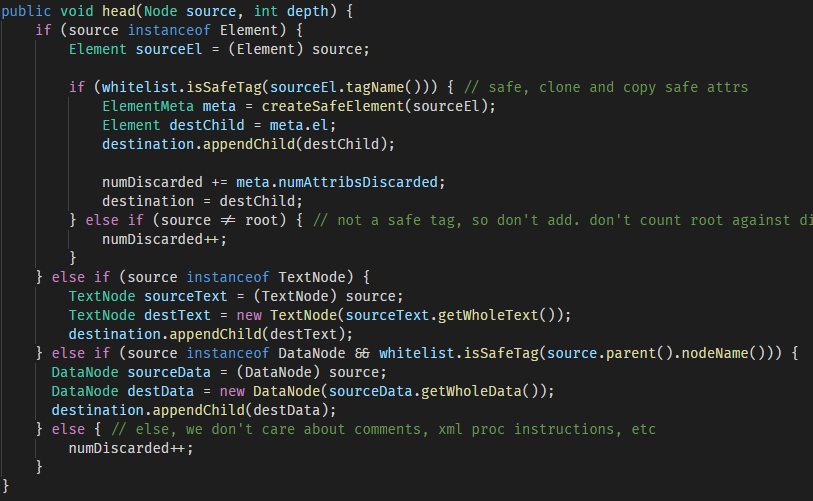
- Concrete Visitor들은 HtmlToPlainText, Node등 클래스들의 곳곳에 있으며 아래의 사진과 같이 사용될 때 바로 재사용되는 등 각각의 필요성에 따라 새롭게 정의된다.

- 주목해야할 점으로, 일반적인 Visitor의 경우 순회하고자 하는 각 concrete node에 따라서 visitor의 operation이 구현되어 있어야 하지만, JSoup에서는 Visitor가 Node로 추상화하여 받은 다음에 instanceof를 통해 runtime에서 type을 확인하고, 이에 따른 동작을 분기하여 수행한다.



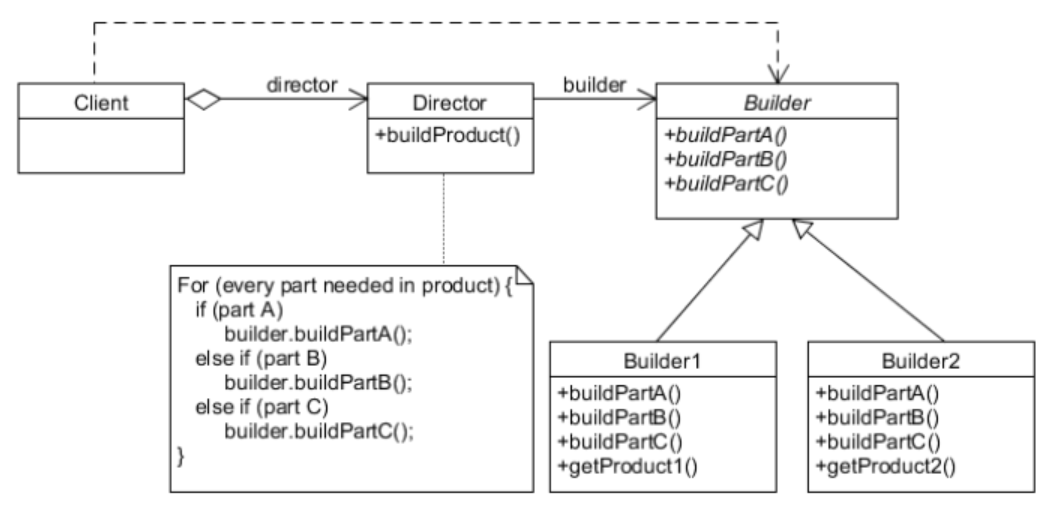
(1) 코드 곳곳에서 새롭게 만들어지는 Concrete한 Visitor

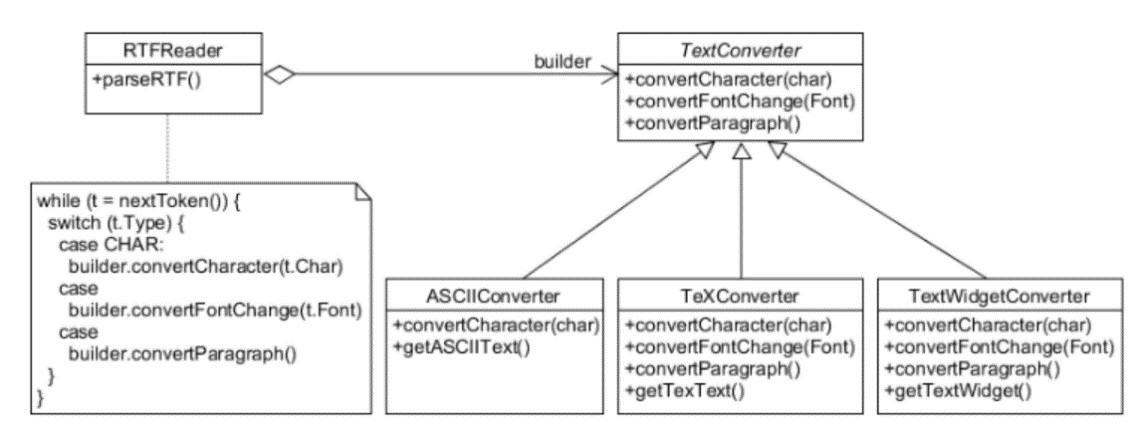
- Visitor 패턴을 통해서 Client가 Element, LeafNode의 기존 구성요소에 영향을 끼치지 않으면서head(), tail() 두 메소드를 원하는 대로 정의할 수 있다.



(2) Visitor의 Operation 구현 방식

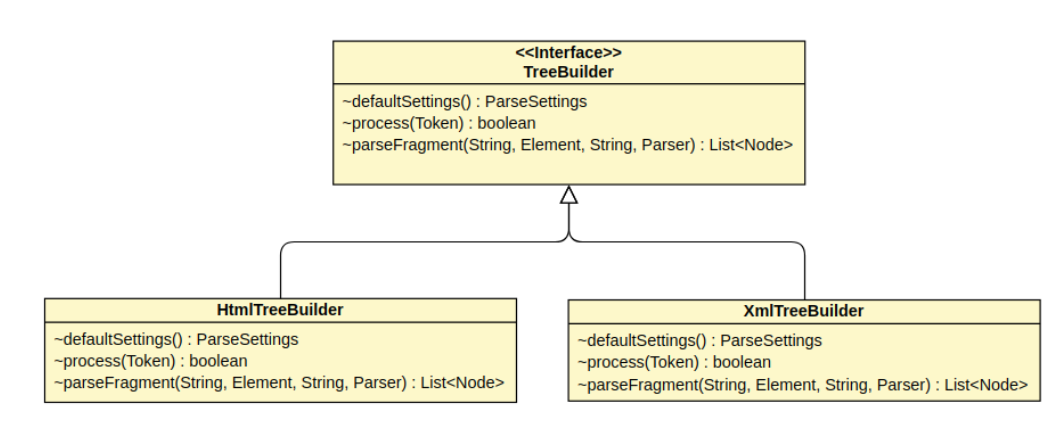
* 위 CleaningVisitor의 예시를 보면, Element, TextNode, DataNode 각각의 operation을 구현한 것이 아닌, head() 내에서 instanceof를 통해 type을 구분하여 코드를 분기하고 있음을 알 수 있다.
* 이는 DOM의 각 노드가 head와 tail로 구분되어 있고, 이 Node에 따라서 operation을 수행하기 위해 응용에 맞게 visitor의 설계를 변경하여 적용한 사례이다.

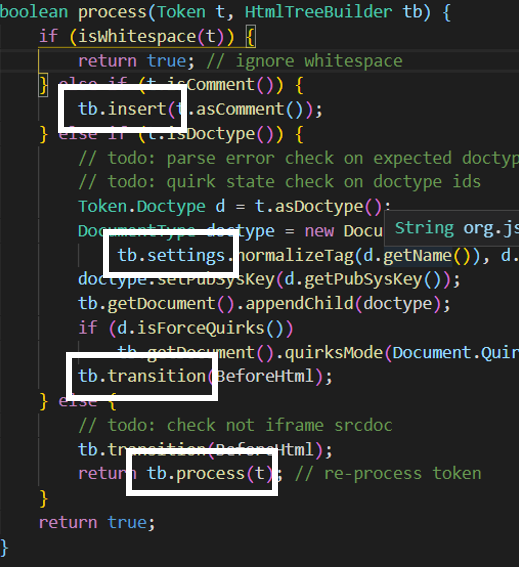
**3) Builder 패턴 & Strategy 패턴**

**(1) Director가 있는 Builder 패턴**

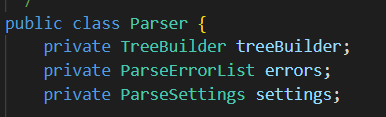
**(2) Director가 없는 Builder패턴**

- 위의 그림(1)과 같이 Director가 있는 Builder 패턴은 사용하는 Client가 Builder안이 어떠한 메소드들로 이루어져 있는지 알 수 없다. 반면에, 그림(2)와 같은 Builder 패턴은 Composition과 Delegation을 이용한 전략 패턴의 모습을 보이며, 각 경우에 따라 builder의 메소드를 명시적으로 호출한다. 어떠한 메소드들로 이루어져 있는지 알고 있다고 할 수 있다.

(**3) Jsoup에서의 Builder**

- Jsoup에서 형성되어있는 Builder는 그림 (2)와 같은 형태임을 좌측의 코드를 보면 알 수 있다. (HtmlTreeBuilderState.java)

- parse()로 파싱할 document의 종류(ex. xml, html)에 따라 TreeBuilder를 선택하므로 Builder패턴과 전략패턴이 동시에 적용되었음을 알 수 있다.



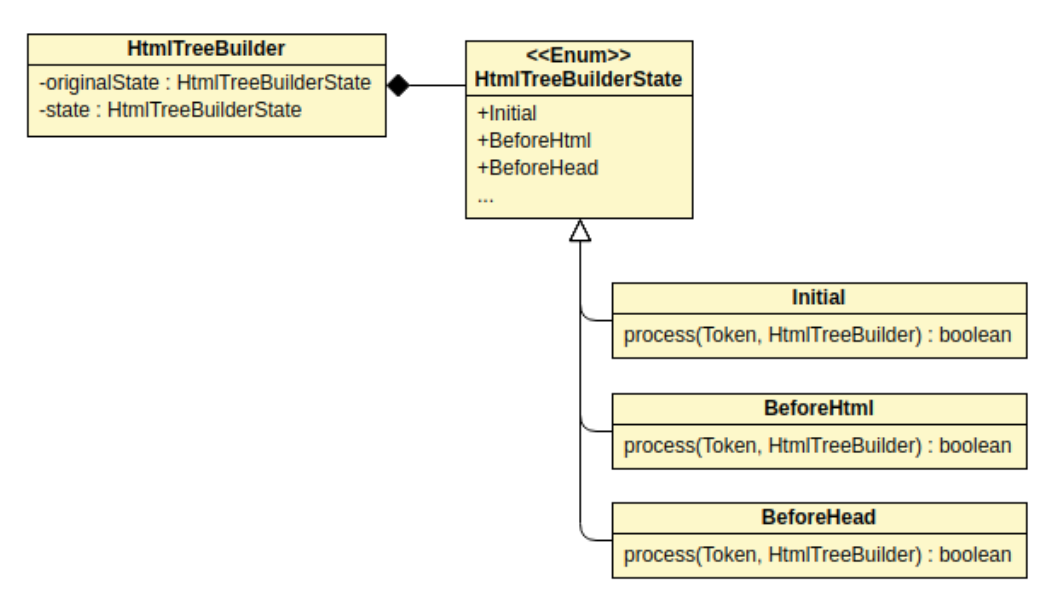
**(4) Parser 클래스가 treeBuilder를 소유하고 있음**



**(5) TreeBuilder내의 abstract 메소드**

- Parser가 Concrete TreeBuilder (HtmlTreeBuilder, XmlTreeBuilder)에게 트리를 생성하는 것은 위임하고, 각 Concrete Builder들은 parser에서 요청하는 방식에 따라 자신의 parseFragment, process 메소드를 수행한다. 따라서 Client가 원하는 양식을 전략적으로 선택하여 문서를 읽을 수 있다.

**4) State 패턴**

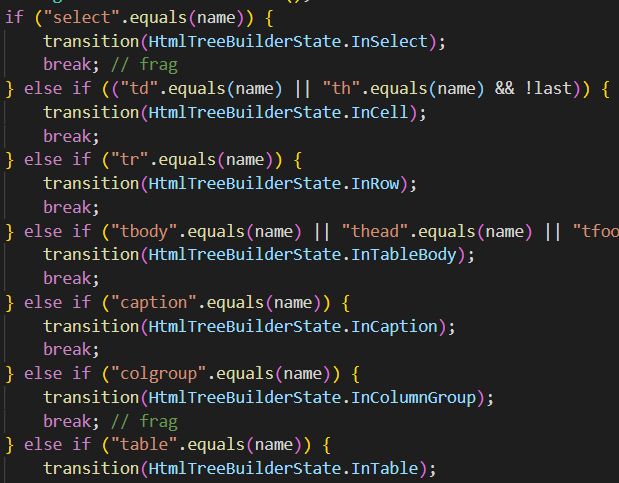


- 각 State 별로 Class를 선언하지 않고, **Enum을 이용해서 각 상태의 method를 overriding하여 구현하였다.** 따라서 모든 상태는 process(Token, HtmlTreeBuilder)의 method를 가지고, 상태에 맞는 operation을 수행한다.

- HtmlTreeBuilder내부에 originalState, state변수가 선언되어 있으며 초기화 함수에서 originalState는 Initial, state는 null로 초기화된다.

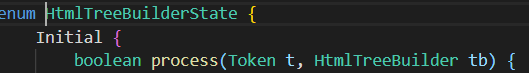
- state의 종류는 다음과 같다.

Initial, BeforeHtml, BeforeHead, InHead, InHeadNoscript, AfterHead, InBody, Text, InTable, InTableText, InCaption, InColumnGroup, InTableBody, InRow, InCell, InSelect, InSelectInTable, AfterBody, InFrameset, AfterFrameset, AfterAfterBody, AfterAfterFrameset, ForeignContent



**(1) HtmlTreeBuilder 내부의 resetInsertionMode 메소드**

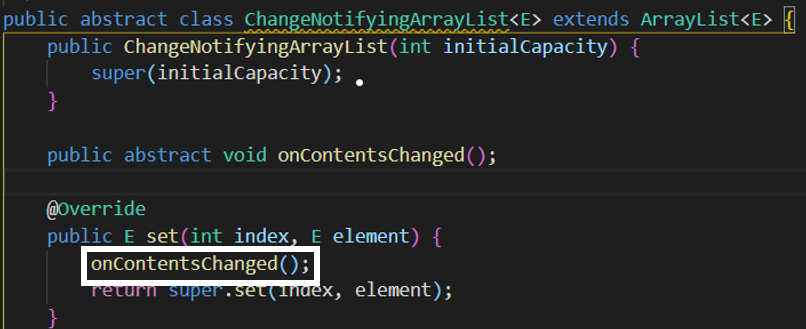
- state 전환은 HtmlTreeBuilder 내의 resetInsertionMode 메소드(그림 (1))를 호출함으로써 이루어진다. 예를 들어 <tbody>와 같은 태그 안에 있는 경우 state를 InRow로 전환한다.



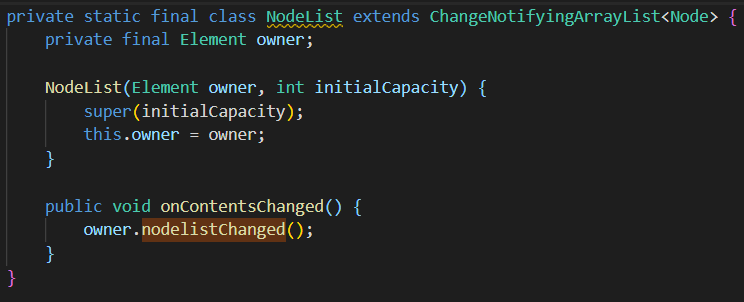
**(2) 각 state별로 Override 되는 process함수**

- 각 state 별로 process 메소드를 Override함으로써 state때마다(여기에선 태그의 종류) 다른 기능이 일어나도록 한다.

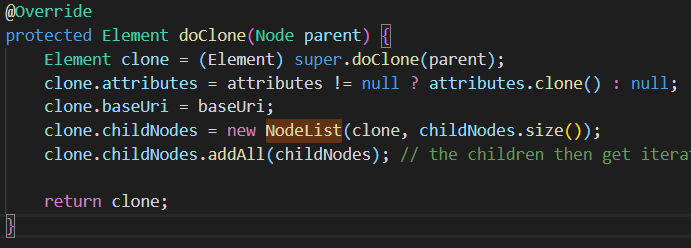
**5) Observer 패턴**



**(1) ArrayList의 메소드를 호출할 때마다 notify 할 수 있도록 구현되어있다.**

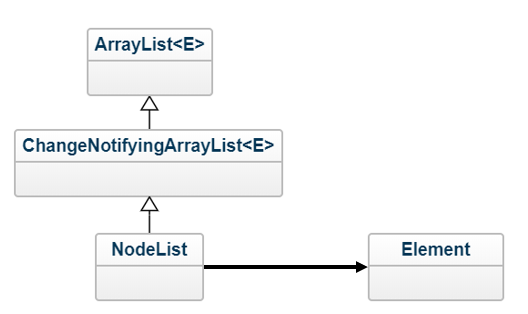


**(2) Observer 패턴에서 Subject 역할을 하는 NodeList**



**(3) notify 되는 사례(NodeList의 addAll 메소드 호출)**

- ChangeNotifyingArrayList형에 add, set등의 구성요소를 변경하는 메소드를 실행시키면 Clone(Node)이 Override했던 nodelistChanged 메소드가 실행된다. Clone(Node)의 childNode들이 NodeList로 생성됨으로써 childNodes들이 변할 때, 예를 들자면 마지막 사진의 addAll 메소드를 실행시키면 clone(Node)의 nodelistChanged()함수가 실행된다.

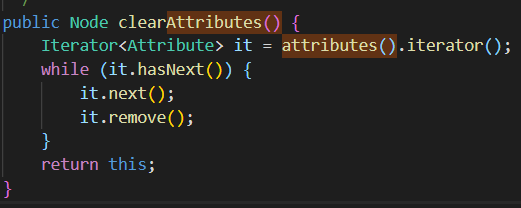


(4) Class 다이어그램

- Subject는 NodeList이며, Observer는 NodeList를 childNodes로 가지고 있는 각 Element자료형 들이다.

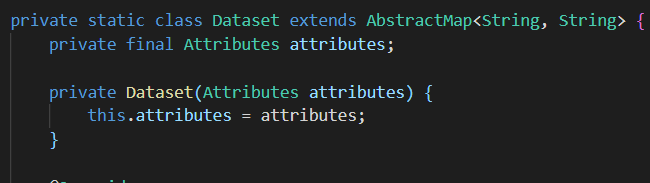
- Element 내에 있는 childNodes에 add, set등의 메소드를 실행하면 NodeList형인 childNodes는 Subject인 Element에게 notify함으로써 Element내의 nodelistChanged메소드를 호출한다.

**6) Iterator 패턴**



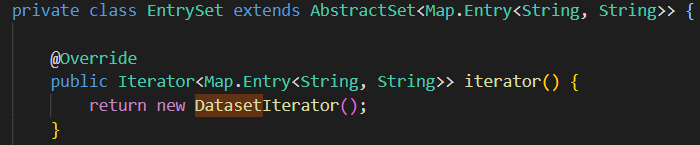
**(1) iterator를 호출하는 예시**

- Node를 상속한 Element에 Attributes 변수가 있고, Attributes는 Iterable<Attribute>를 implement한다. attributes().iterator()를 하면 예를 들어 Element안에있는 Iterator<Attribute>형인 attributes가 반환된다. 이를 통해 Iterator 패턴을 이용함을 알 수 있다.



**(2) class Dataset (attributes가 iterable을 implement한 class이다.)**

- Attributes의 내부 클래스인 Dataset을 탐색하기 위해서 DatasetIterator를 구현하고 있음을 볼 수 있다.



**(3) createIterator 메소드 역할을 담당하는 메소드(EntrySet은 Dataset의 innerClass)**

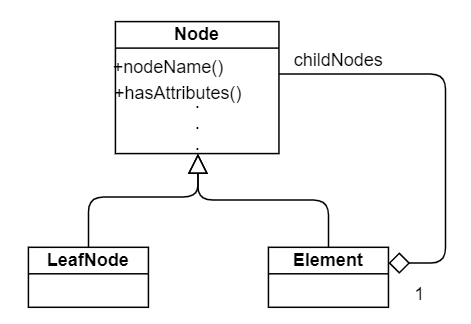
- createIterator 메소드를 내부의 EntrySet class에서 구현한 것을 확인할 수 있다.



**(4) Dataset에 해당하는 iterator class DatasetIterator**

- DatasetIterator를 생성함으로써 Dataset의 Attributes의 iterator가 hasNext, next, remove 메소드의 대상이 된다.

**7) Composite 패턴**

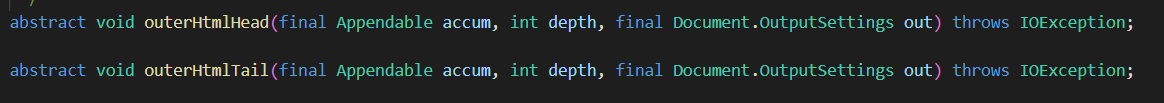


|  |
| --- |
|  |

**(1) Node형을 상속하고 있는 LeafNode와 Element**

- Node의 interface를 통해서 Element인지, LeafNode인지 모르게 이용할 수 있다. Abstract로 되어진 nodeName(), hasAttributes(), outerHtmlHead(), outerHtmlTail() 등의 interface를 공유하고 있다.

- Element가 Node형 List를 가지고 있으므로 트리형 자료구조를 이루고 있음을 알 수 있다.

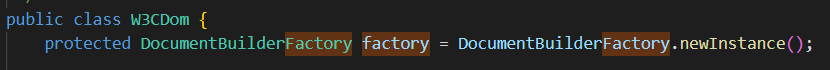


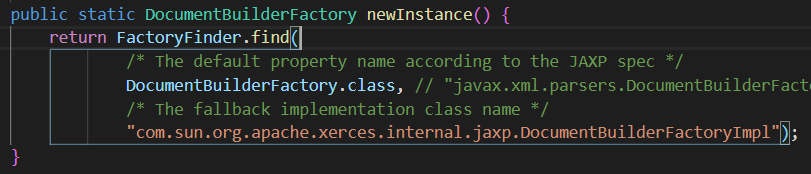
**(2) Safety를 위한 공유 메소드 예외처리**

- abstract를 통해 공유하는 메소드들 중 LeafNode에서 쓰이지 않을 것들은 throw IOException등을 통해 예외처리 되어있다. (안전성을 위해서)

- 그림 (1)을 보면 LeafNode의 경우 Empty List를 가지고 있어서 전체를 훑을 때 추가적인 예외처리 없이 Empty인지만 검사하면 되도록 구현해 놓았음을 확인할 수 있다

**8) Factory Method 패턴**





- Create문을 다른 객체에 미룸으로써 Factory Method Pattern이 적용되었음을 알 수 있다.

- W3CDom 클래스 내부에서 DocumentBuilderFactory로 Concrete하게 선언하였기 때문에 다른 Factory로의 확장은 불가능하지만 FactoryFinder.find 메소드를 보아 다른 Factory의 newInstance도 저 함수를 통해 호출해낼 수 있을 것으로 추측된다.

- DocumentBuilderFactory부터는 java에서 제공되는 class들이다.

**2. 새로 추가할 기능**

Jsoup은 HTML을 분석을 도와주는 라이브러리이다. 단순히 원하는 Element를 찾는 목적으로 사용할 수 있지만, 해당 Elements에 적용된 스타일을 똑같이 가져오고 싶다면 웹 브라우저에서 개발자 도구 등을 이용해 style속성을 직접 찾아 적용해 보아야만 하는 불편함이 있다. Element에 적용되는 inline 속성을 확인할 수 있지만, 이는 element에 직접 적용된 style에 불과하고, 실제 적용되는 style를 확인할 수 없다.

우리는 link tag를 통해 실제로 호출하는 css, document 내에 존재하는 style tag, 그리고 자신에게 적용되는 style을 반환하는 getAppliedStyle()을 구현하여, Client가 필요하면 해당 element에 적용되는 Css style을 가져올 수 있는 기능을 설계 패턴을 적용하여 추가하고자 한다.

**3. 구현 방법**

1. **Created**

* **org.jsoup.parser.CssParser.java**

Document에 존재하는 모든 style에 대한 정보를 분석한다.

public static HashMap<String, String> parse(Document doc)

Document내 모든 Style 속성 가져와 Map으로 return한다.

style의 우선순위를 고려하여 우선순위가 낮은 속성부터 높은 속성 순으로 탐색한다. 첫째, link 태그 css 파일에 저장된 style정보를 찾아 저장한다. 둘째, style 태그안에 존재하는 style을 찾아주었으며 마지막으로 inline으로 들어간 style속성을 찾아 중복되는 속성값은 덮는 방식으로 저장한다.

* **org.jsoup.style.Styler.java – 적용된 패턴: Visitor Pattern**

원본 Document에 대해 각 Element에서 자신에게 지원되는 css rule을 새로운 Document를 생성하여 반환한다.

private final class StylingVisitor implements NodeVisitor

기존의 Document와 CssParser로 분석한 Style을 이용하여 각 Element가 자신에게 적용될 Style을 갖는 새로운 Document를 만들어서 반환한다. Tree형식을 갖는 Document의 각 Node를 순회하며 일치하는 Style을 갖도록 하기위해 NodeVisitor를 상속하는 StylingVisitor를 내부 클래스로 선언하여, 각 element에 style를 적용여부를 판단하는 operation을 수행한다.

**@Override**

* + public void head(Node source, int depth)

Head는 해당 Node를 처음 방문하였을 때 불려지는 함수이다. 해당 Node가 style 속성을 가질 수 있는 Element라면 자신에게 적용될 수 있는 Style을 갖도록 해준다. 그렇지 않다면 변경점이 없으므로 기존의 Node상태를 가져온다.

* + public void tail(Node source, int depth)

Tail은 자식 Node를 모두 탐색 후 마지막으로 Node를 방문할 때 불려지는 함수이다. 해당 Node가 속한 SubTree는 모두 탐색을 마쳤으므로, 상위 Node부터 탐색을 진행할 수 있도록 현재 수행중인 Node를 parentNode를 가리키게 한다.

private void copyStyledNodes(Element source, Element dest)

StylingVisitor를 생성하고 적용할 원본 Document와 결과 Document를 설정한다. NodeTraversor를 이용해 원하는 Visitor방식으로 결과 Document를 만든다

* **org.jsoup.Jsoup.java**

public static Document applyExternalStyle(Document doc)

사용자가 이용할 Jsoup 인터페이스를 위해 추가하였다. 스타일을 적용하고 싶은 Document 넘기면 적용된 스타일을 갖는 Document를 return해준다. 이 인터페이스를 통해 사용자는 스타일링이 적용된 Document를 사용하고 싶지 않을 시 불필요한 오버헤드를 감당하지 않도록 한다.

1. **Modified**

* **org.jsoup.node.Element.java**

적용된 스타일에 대한 정보를 각 Element에서 갖고 있는다. List<String> styles를 멤버변수로 가지며 Client의 접근을 위해getter 와 setter를 가진다. Client가 applyExternalStyle() 를 호출하지 않는다면, 생성되지 않을 정보이므로, style 파싱이 수행될 때까지 ArrayList를 생성을 지연한다.

**4. Test**

추가적으로 구현한 두개의 Class에 대해 unit test module을 만들어 테스팅 하였다.

1. CssParserTest
   * simpleBehaviorTest

Style tag를 갖는 Document에 대한 CssParser Unit test

* + simpleInlineTest

Inline attribute로 style을 갖는 Document에 대한 CssParser Unit test

1. StylerTest
   * simpleBehaviorTest

Origin Document로부터 Style이 입혀진 Document를 생성하는 Styler에 대한 Unit Test

* 테스팅 결과:

-------------------------------------------------------

T E S T S

-------------------------------------------------------

Running org.jsoup.helper.DataUtilTest

Tests run: 15, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.219 sec

Running org.jsoup.helper.HttpConnectionTest

Tests run: 23, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.037 sec

Running org.jsoup.helper.W3CDomTest

Tests run: 6, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.356 sec

Running org.jsoup.integration.ConnectTest

2019-12-06 18:50:44.592:INFO::main: Logging initialized @973ms

2019-12-06 18:50:44.640:INFO:oejs.Server:main: jetty-9.2.28.v20190418

2019-12-06 18:50:44.688:INFO:oejs.ServerConnector:main: Started ServerConnector@1b6e1eff{HTTP/1.1}{0.0.0.0:44617}

2019-12-06 18:50:44.688:INFO:oejs.Server:main: Started @1069ms

2019-12-06 18:50:45.307:INFO:oejs.ServerConnector:main: Stopped ServerConnector@1b6e1eff{HTTP/1.1}{0.0.0.0:0}

Tests run: 32, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.81 sec

Running org.jsoup.integration.ParseTest

Tests run: 12, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.155 sec

…

Running org.jsoup.parser.CssParserTest

Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.007 sec

…

Running org.jsoup.styler.StylerTest

Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0 sec

Results :

Tests run: 698, Failures: 0, Errors: 0, Skipped: 5

[INFO] ------------------------------------------------------------------------

[INFO] BUILD SUCCESS

[INFO] ------------------------------------------------------------------------

[INFO] Total time: 11.797 s

[INFO] Finished at: 2019-12-06T18:50:46+09:00

[INFO] ------------------------------------------------------------------------

테스트 결과를 통해 기능을 추가하면서 다른 class의 기능을 손상시키지 않았고, 우리의 기능도 동작하는 지 여부를 확인하는 단위테스트를 통과하였다.