

CAU-CSE

Design Pattern – Term Project



20153283 이정원

20153734 이정민

20152577 서재훈

목차

1. Jsoup 개요
2. Jsoup 설계 및 구현 조사
3. 설계 개선
4. 확장 기능
5. 테스트 수행 내역
6. GitHub 활동 요약

1.Jsoup 개요

jsoup 은 DOM 을 사용하여 HTML, XML 문서에 저장된 데이터를 구문 분석, 추출 및 조작하도록 설계된 오픈 소스 Java 라이브러리

2.Jsoup 설계 및 구현 조사

[*설계 Overview](#)

Jsoup 기능 접근을 위해 public Jsoup 클래스를 core public access point 로 둔다.

Jsoup 은 클라이언트가 Jsoup.Connect 함수를 통해 HTTP Request 와 Response 를 만들어 내며 그 과정에서 해당하는 사이트의 Html 혹은 Xml 을 가져와 파싱하는 과정이 있다. 유저는 Connect 함수를 통해 url 을 넘겨주기만 하면 통신에 대한 과정을 알 필요 없이 해당 url 에 대한 결과인 document 를 얻어 낼 수 있다. 유저가 Connect 라는 함수만을 불러 document 를 얻어낸다는 점에서 Facade 패턴이 적용되었다고 판단했다.

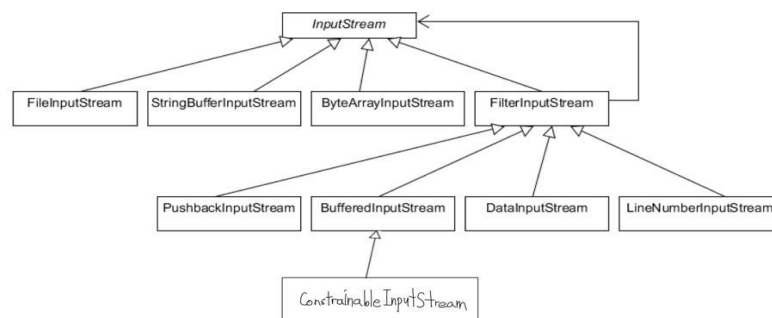
Facade 패턴은 내부의 복잡한 과정을 알 필요 없고, 제공된 간단한 인터페이스를 통해 내부 작업을 수행 할 수 있게 만들어진 패턴을 의미하며 이 다음부터는 Facade 안에 숨겨진 세부적인 코드에 대해 적용된 설계 패턴을 소개하려 한다.

*적용된 설계 패턴 소개 - 이름, 클다, 판단 근거, 소스코드와 매핑

i. Decorator Pattern

- ConstrainableInputStream.java - wrap() 메서드 (BufferedInputStream 을 extends)

최초의 Input Stream 객체 만들고 여러가지 Input Stream 으로 Decorate 하면서
최종 결과물을 만들어 냄

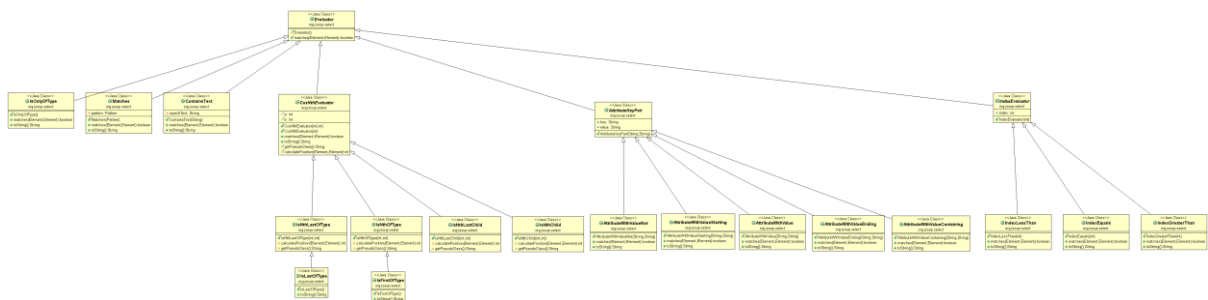


- Evaluator.java

State: Evaluator

Context: FirstFinder, Accumulator

request 메서드: head()



ii. Visitor Pattern

- NodeVisitor.java, NodeTraversor.java

NodeVisitor Interface 가 Visitor 에 매칭 되고 Accumulator class, CleanVisitor class 등이 Concrete visitor 에 매칭.

Node class 가 패턴의 Element, Traverse method 가 패턴의 accept method 에 대응, Element class 와 LeafNode class 는 Concrete Element 이다.

Node 자체가 추가될 염려가 없으므로 visitor 에서 Elements 가 증가함에 따라, 수정하게 될 걱정을 하지 않아도 된다.

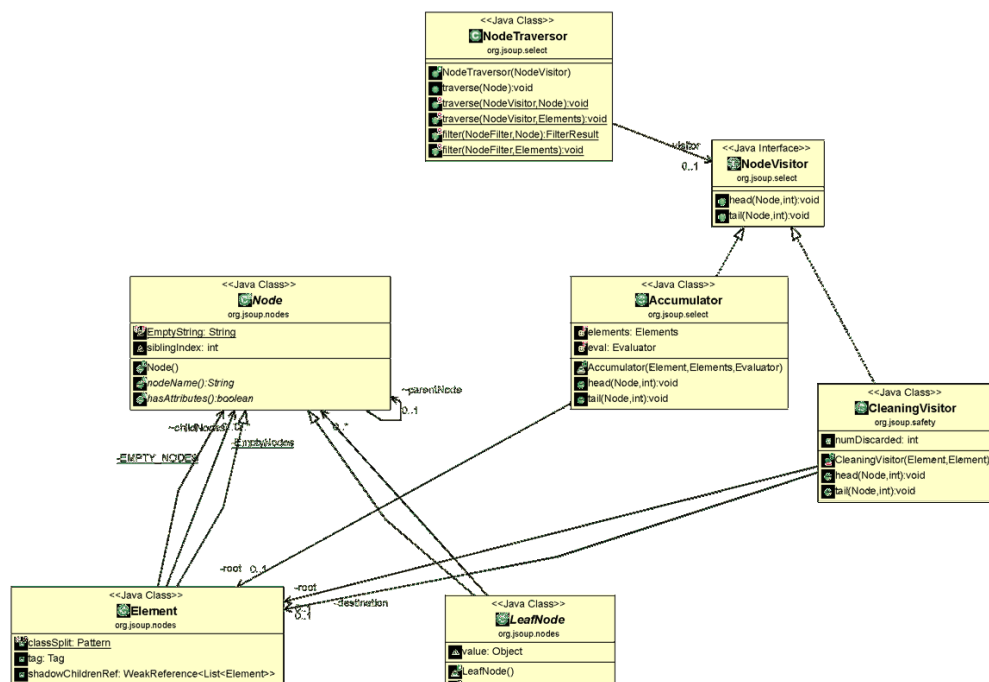
NodeVisitor 에서 visit 이 선언되어 있지 않고, 각각의 비지터는 visit 을 구현하지 않고 Node Traversor 에서 visit 에 해당하는 traverse 를 구현해 놓았다.

traverse 하는 동작이 런타임에 바뀔 수 있으므로, (cleaning, 등)

* 특이사항

- Concrete visitor 를 호출 시에 생성하여 사용 nodes>Element>wholeText(), text()

- Visitor 와 Adapter 패턴이 함께 적용되어 Accumulator 등 Concrete Visitor 들이 Concrete Element 인 Element 를 직접 알고있다.



iii. Builder Pattern

- TreeBuilder.java

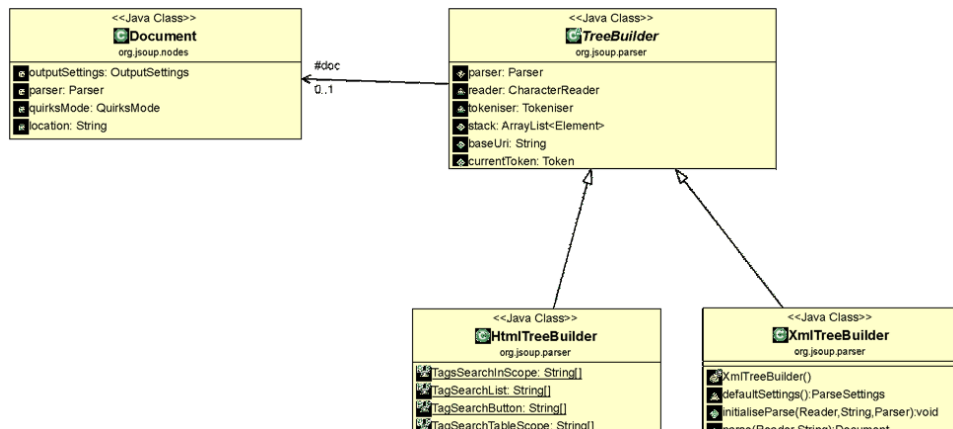
TreeBuilder 에서 만드는 Part 들(ex) Element, Token 등)을 만들어준다. Client 는 Part 들에 대해선 모르지만 TreeBuilder 를 통해 완성된 Tree 를 볼 수 있다.

Director: Parser

abstract Builder 클래스: TreeBuilder

Concrete Builder: HtmlTreeBuilder, XmlTreeBuilder

Client: Document



iv. Strategy Pattern

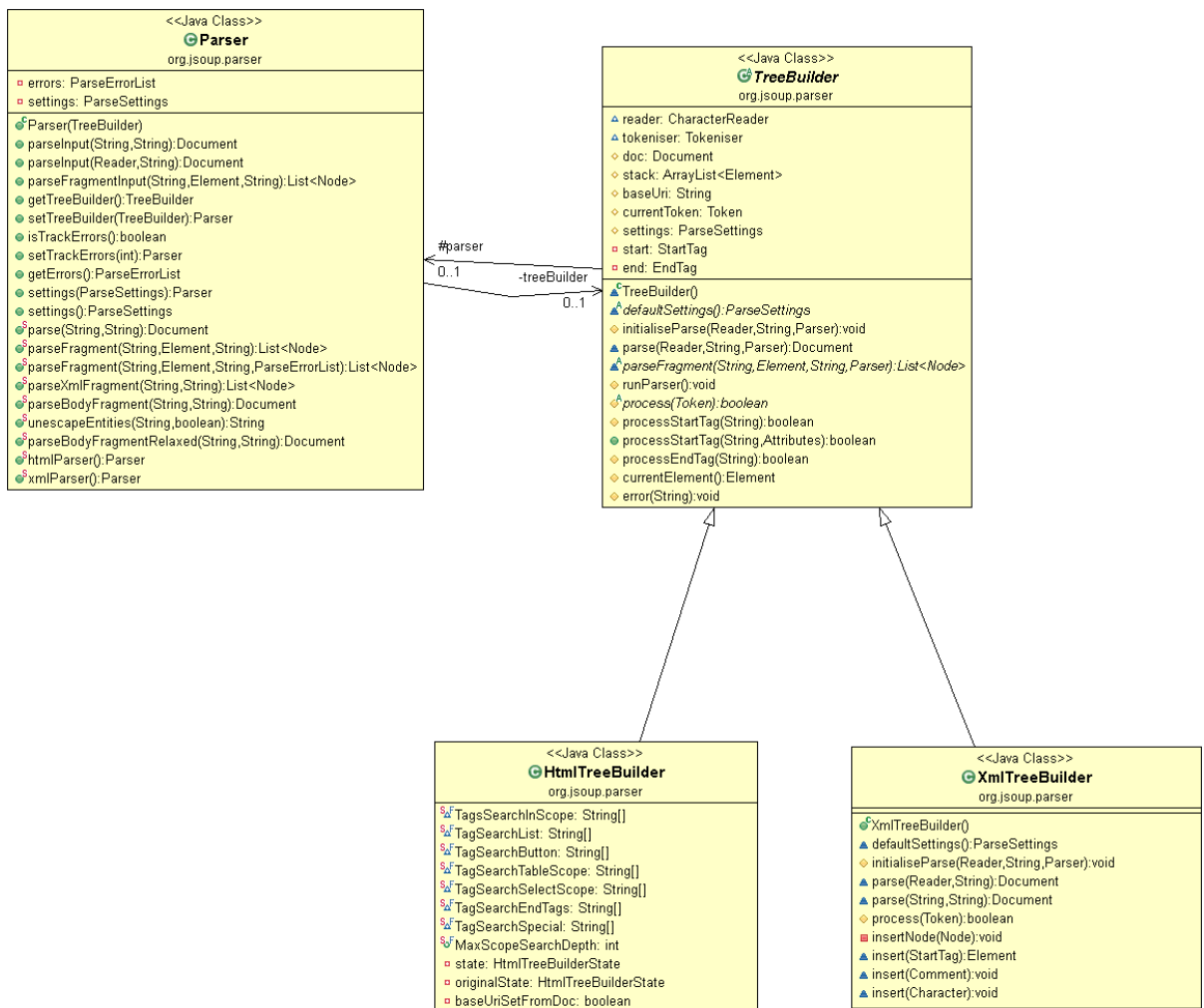
- Parser.java, TreeBuilder.java, HtmlTreeBuilder.java, XmlTreeBuilder.java

State: TreeBuilder (abstract class)

Context: Parser

Concrete State: HtmlTreeBuilder, XmlTreeBuilder

request 메서드: TreeBuilder.parseFragmentInput()



V. Observer Pattern

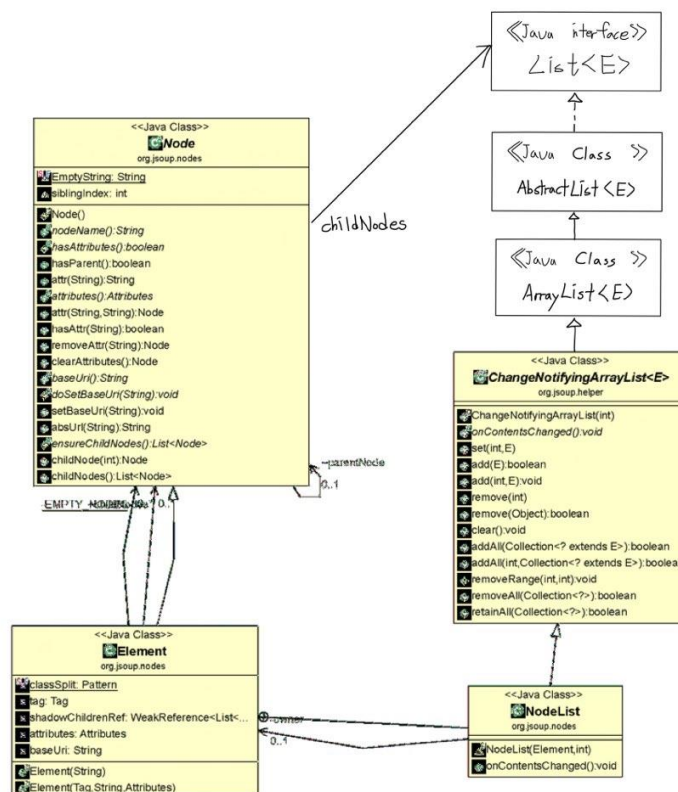
- ChangeNotifyingArrayList.java

abstract Observer 클래스: ChangeNotifyingArrayList

Concrete Observer: NodeList

Subject: Element

onContentsChanged()



Vi. State Pattern, Singleton Pattern

- TokeniserState.java, HtmlTreeBuilderState.java

State interface 를 따로 두지 않고, State 클래스 안에 handleReq 메서드들과 Enum 을 통한 Singleton 으로 State 들을 같이 정의해봤다.

-> Enum 형 Tokeniser State : 별도의 concrete 없음, 이렇게 구현했을때, context 에서 conditional 이 없을 경우 불필요한 클래스를 만드는 것 보다 바람직한 구현이라고 함.

<https://stackoverflow.com/questions/10752157/state-pattern-vs-enum>

Tokeniser State

Context: Tokeniser

request 메서드: read()

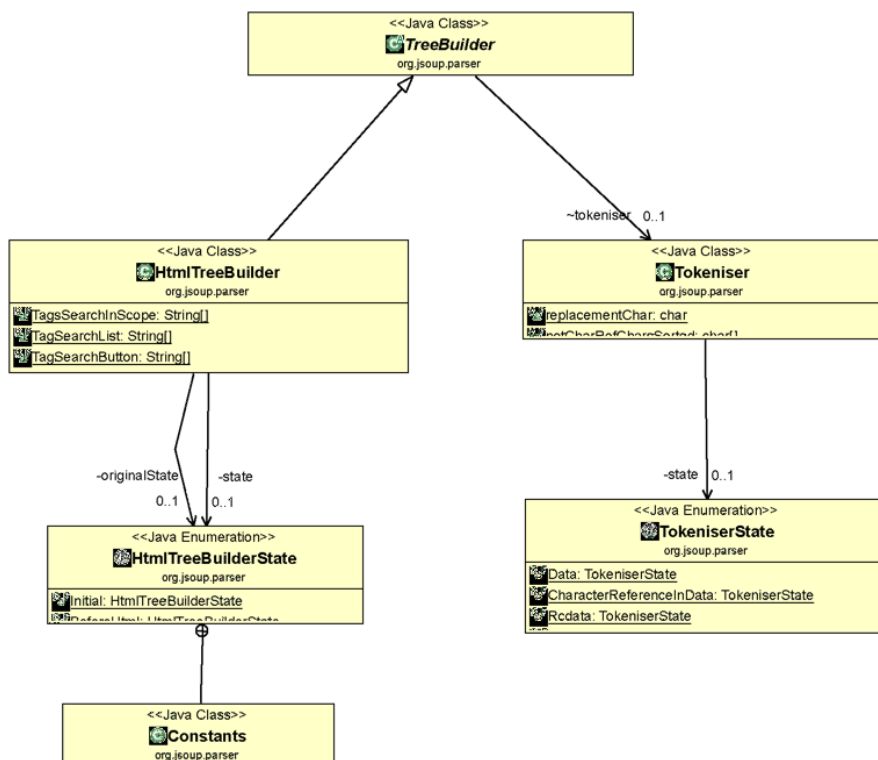
Client: TreeBuilder

HtmlTreeBuilder State

Context: HtmlTreeBuilder

request 메서드: process()

Client: TreeBuilder.runparser()



Vii. Adapter Pattern

- TreeBuilder.java, HtmlTreeBuilder.java, HtmlTreeBuilderState.java

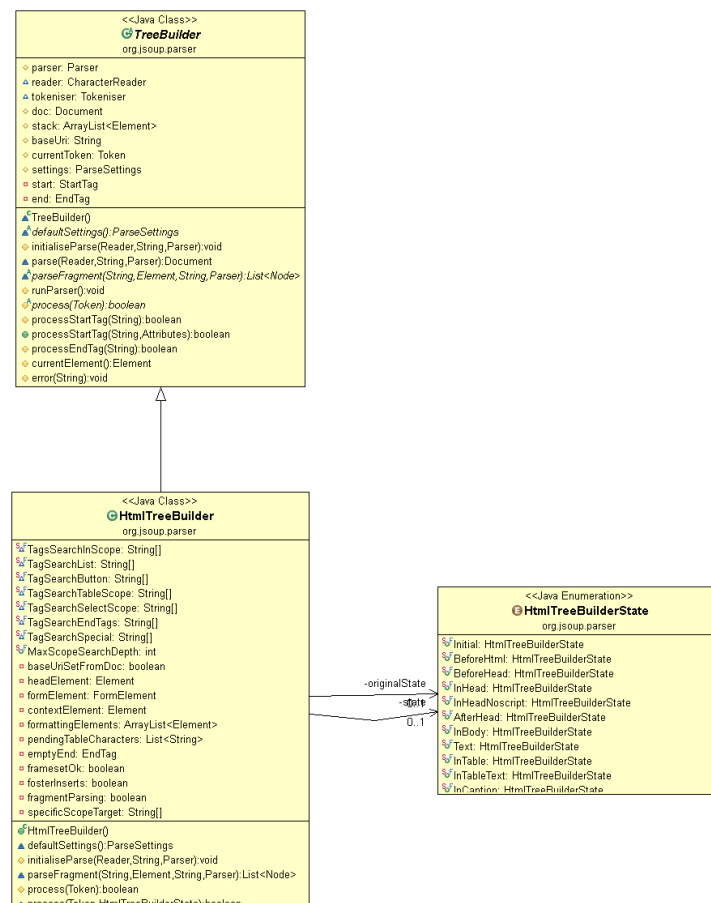
TreeBuilder: Target(interface)

HtmlTreeBuilder: Adapter

HtmlTreeBuilderState: Adaptee

TreeBuilder.process(): request 메서드

HtmlTreeBuilderState 에 있는 Enum class 들: Client



- Node.java, Element.java, Attributes.java

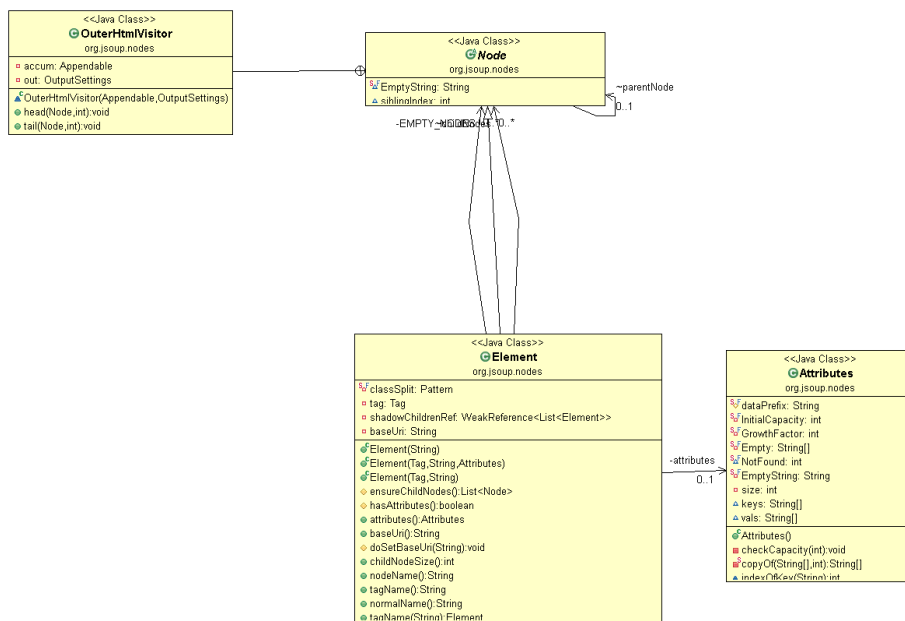
Node: Target(interface)

Element: Adapter

Attributes: Adaptee

Node.outerHtmlHead(): request 메서드

Node.OuterHtmlVisitor.class: Client:



- Node.java, Element.java, Tag.java

Node: Target(interface)

Element: Adapter

Tag: Adaptee

Node.nodeName(), Node.outerHtmlHead(), outerHtmlTail(): request 메서드

Node.head(): Client



- Response.class, Request.class (HttpConnection 내장클래스)

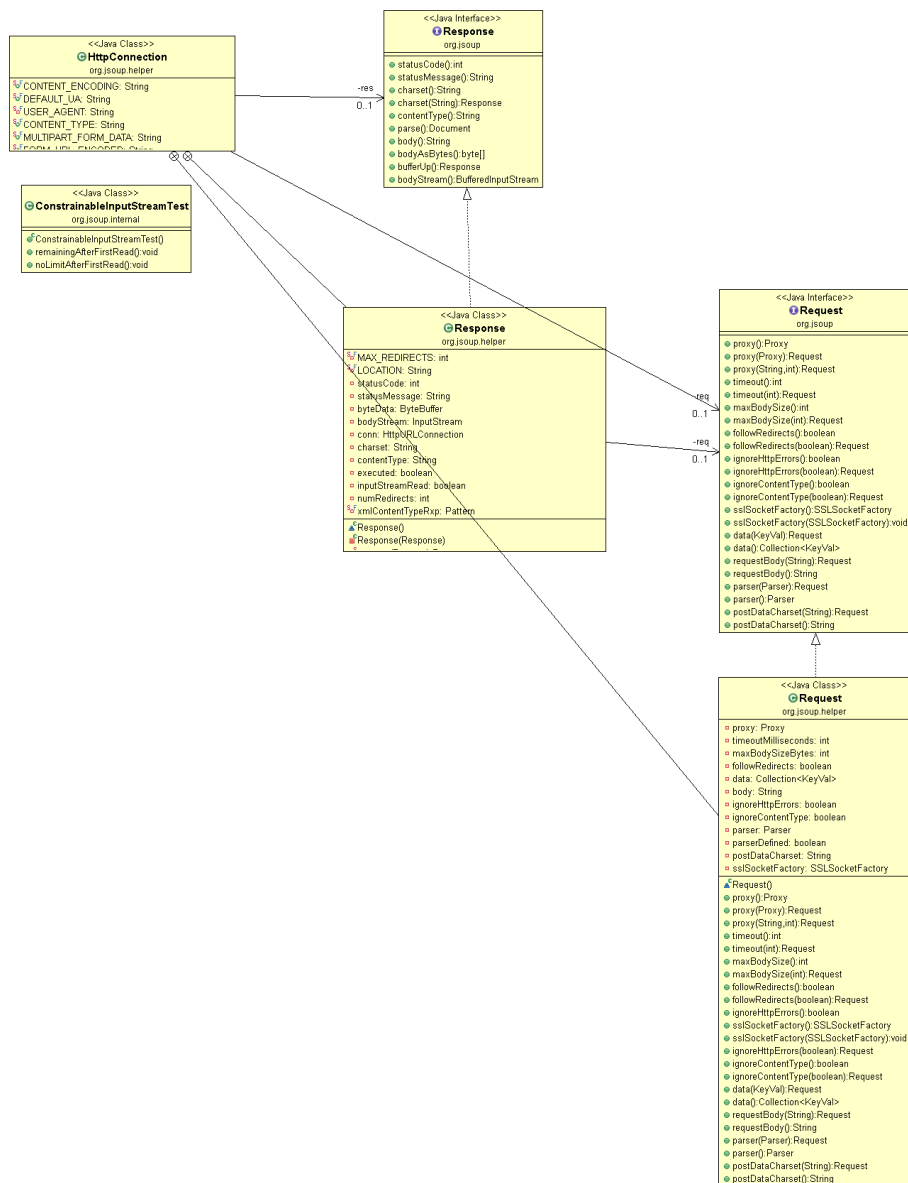
Response(Interface): Target

Response(concrete): Adapter

Request(Interface), Request(Concrete): Adaptee

Response.parse(), Response.bodyStream(): request 메서드

HttpConnection.get(), post(), ConstrainableInputStreamTest(): Client



- TreeBuilder.java, HtmlTreeBuilder.java, Element.java

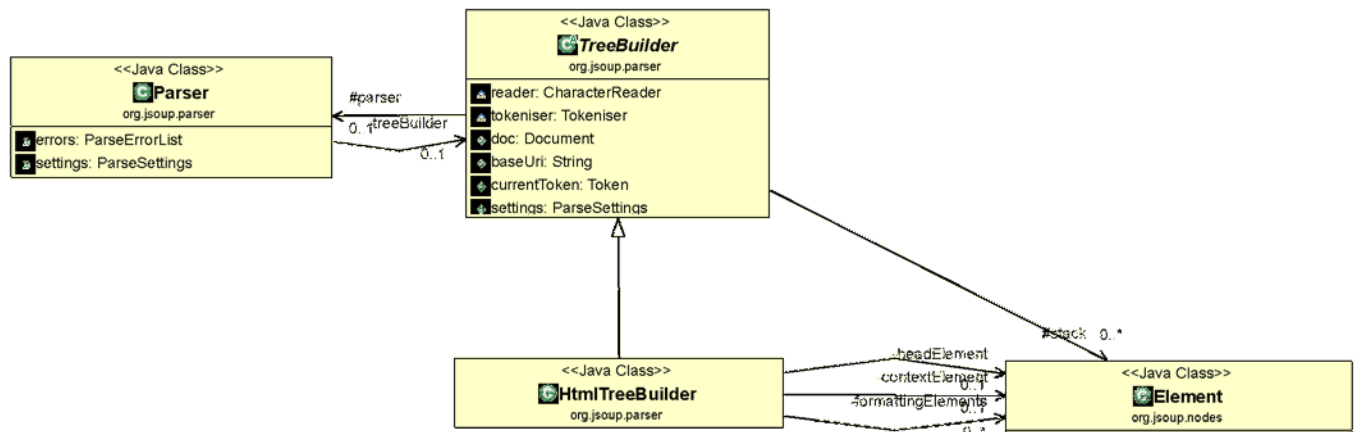
TreeBuilder: Target

HtmlTreeBuilder: Adapter

Element: Adaptee

TreeBuilder.parseFragment(): request 메서드

Parser: Client



- NodeVisitor.java, Element.java, CleaningVisitor.class (Element 내장클래스) 관련

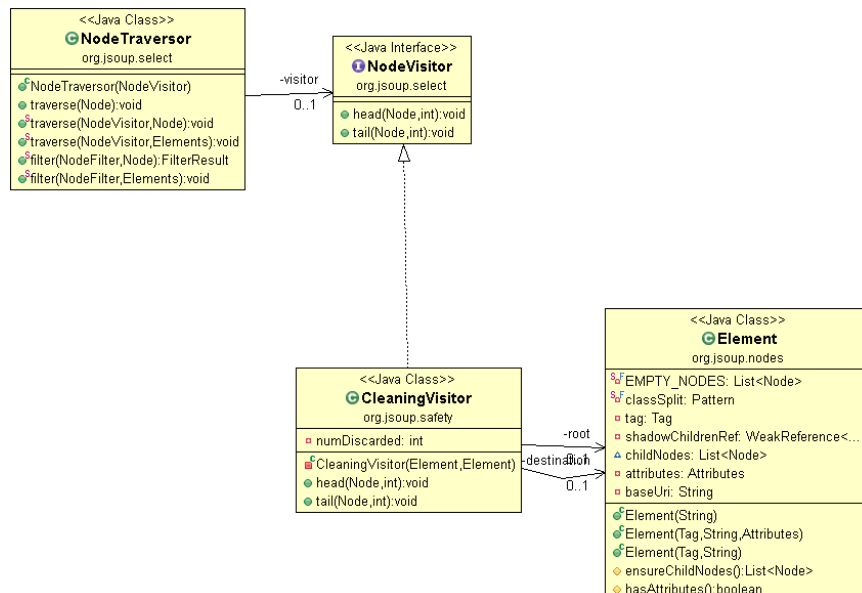
NodeVisitor(Interface): Target

CleaningVisitor: Adapter

Element: Adaptee

NodeVisitor.head(), tail(): request 메서드

NodeTraversor: Client



- Connection.java, HttpURLConnection.java

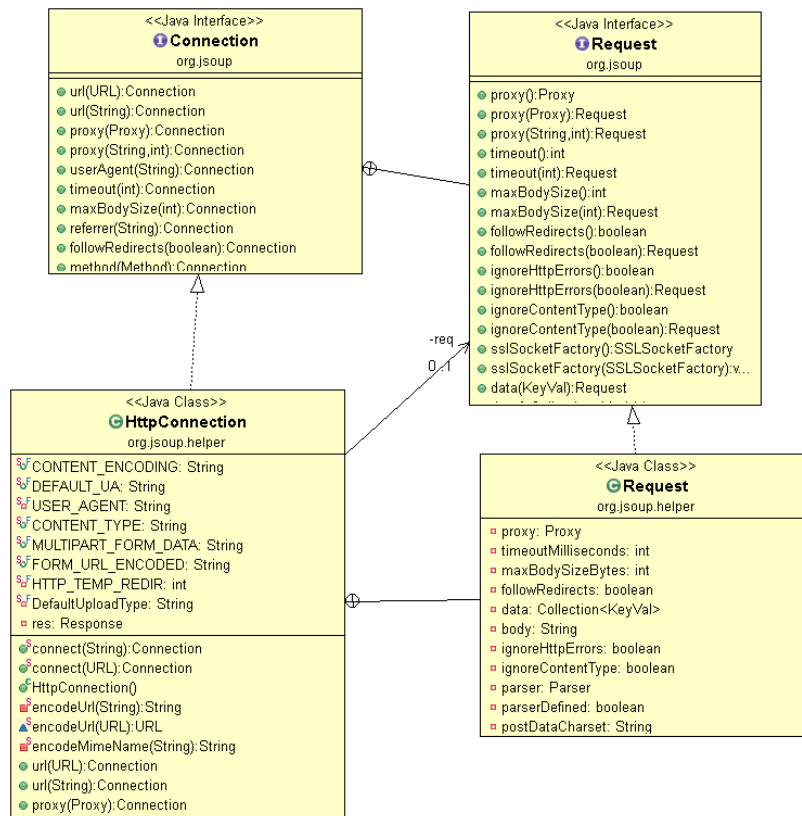
Connection(Interface): Target

HttpURLConnection: Adapter

Request(interface), Request(concrete): Adaptee

Connection.url(), proxy() ... : request 메서드

사용자: Client



- NodeVisitor.java, Accumulator.class (Collector 내장클래스), Elements.java

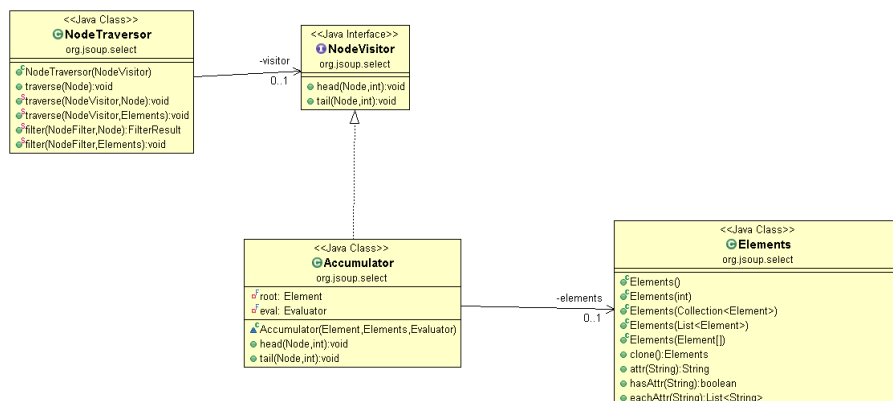
NodeVisitor(interface): Target

Accumulator: Adapter

Elements: Adaptee

NodeVisitor.head(): request 메서드

NodeTraversor: Client



- Connection.java, HttpURLConnection.java, Response.class

Connection: Target

HttpURLConnection: Adapter

Response(Interface), Response(concrete): Adapter

Connection.get(), post(): request 메서드



- NodeFilter.java, FirstFinder.class (Collector 내장클래스), Evaluator.java

NodeFilter(interface): Target

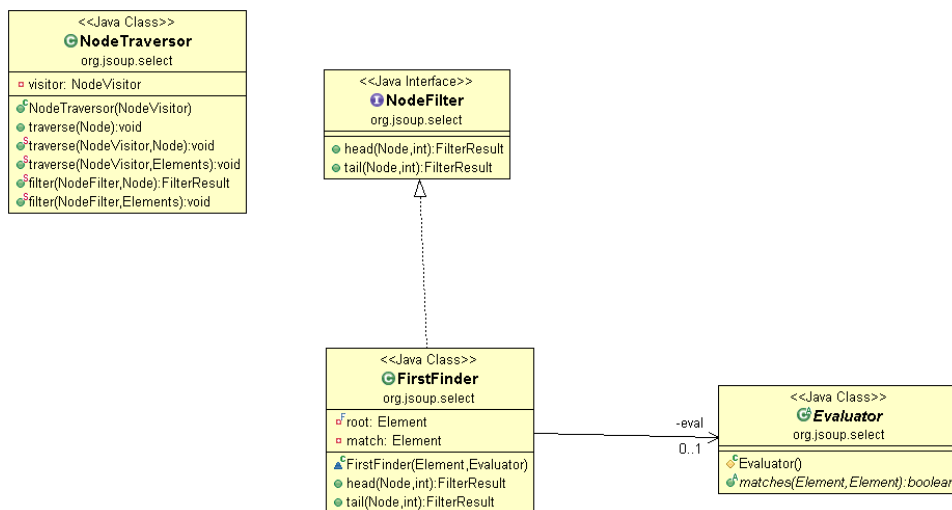
FirstFinder: Adapter

Evaluator: Adaptee

NodeFilter.head(): request 메서드

NodeTraversor: Client

NodeTraversor 안에서 NodeVisitor 같이 변수로 NodeFilter 를 가지고 있지는 않지만 filter() 메서드에서 parameter 로 사용함



- NodeVisitor.java, Accumulator.class, Evaluator.java

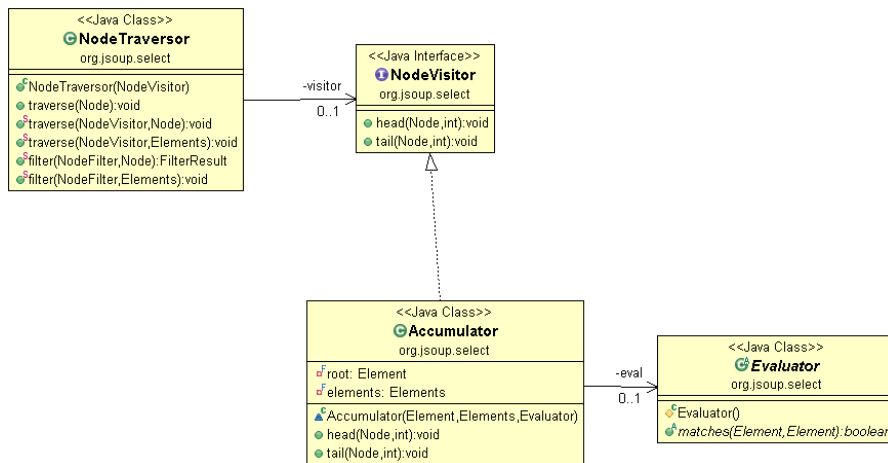
NodeVisitor: Target

Accumulator: Adapter

Evaluator: Adaptee

NodeVisitor.head(): request 메서드

NodeTraversor: Client



Viii. Bridge Pattern

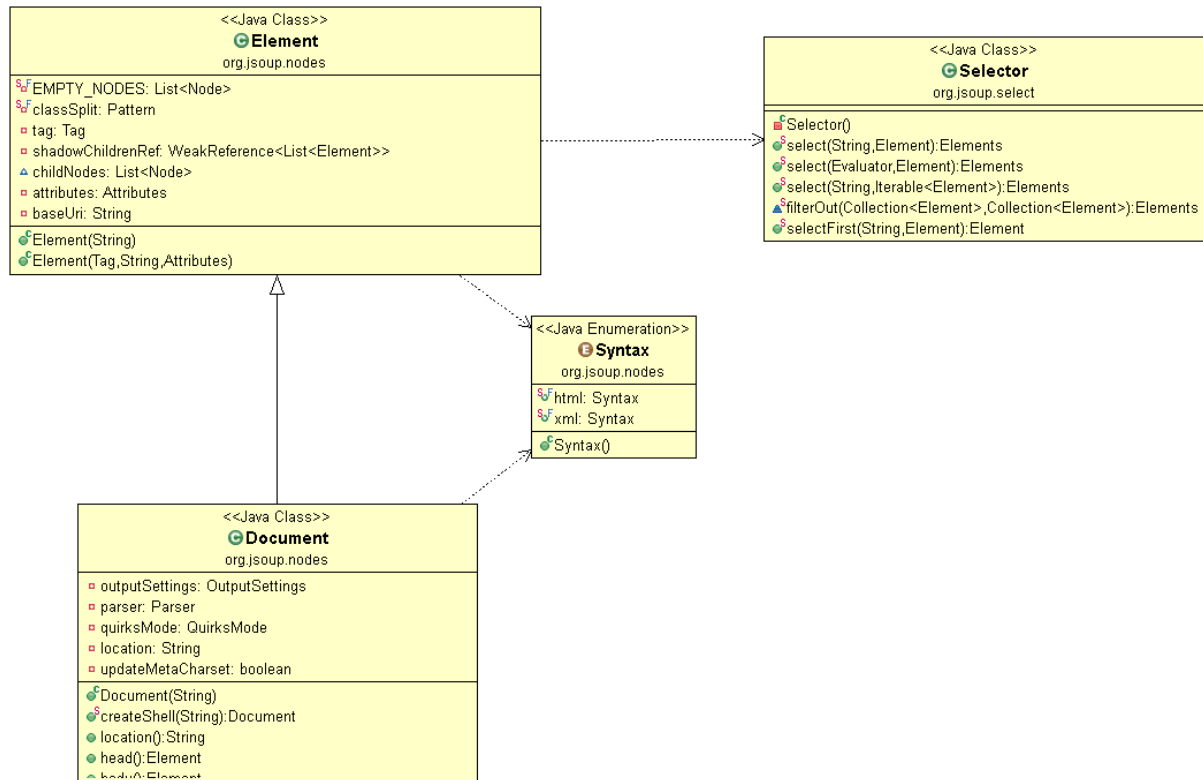
- Document.java, Element.java, Selector.java

Abstraction-Element

Refined abstraction (Document+syntax)

operation() -select(String CSSQuery)

Implementor - Selector



3. 설계 개선



기존의 Parser 클래스와 Tree Builder 클래스 간의 클래스 다이어그램이다.

기존 Parser 클래스에서 발견한 문제점은 다음과 같다.

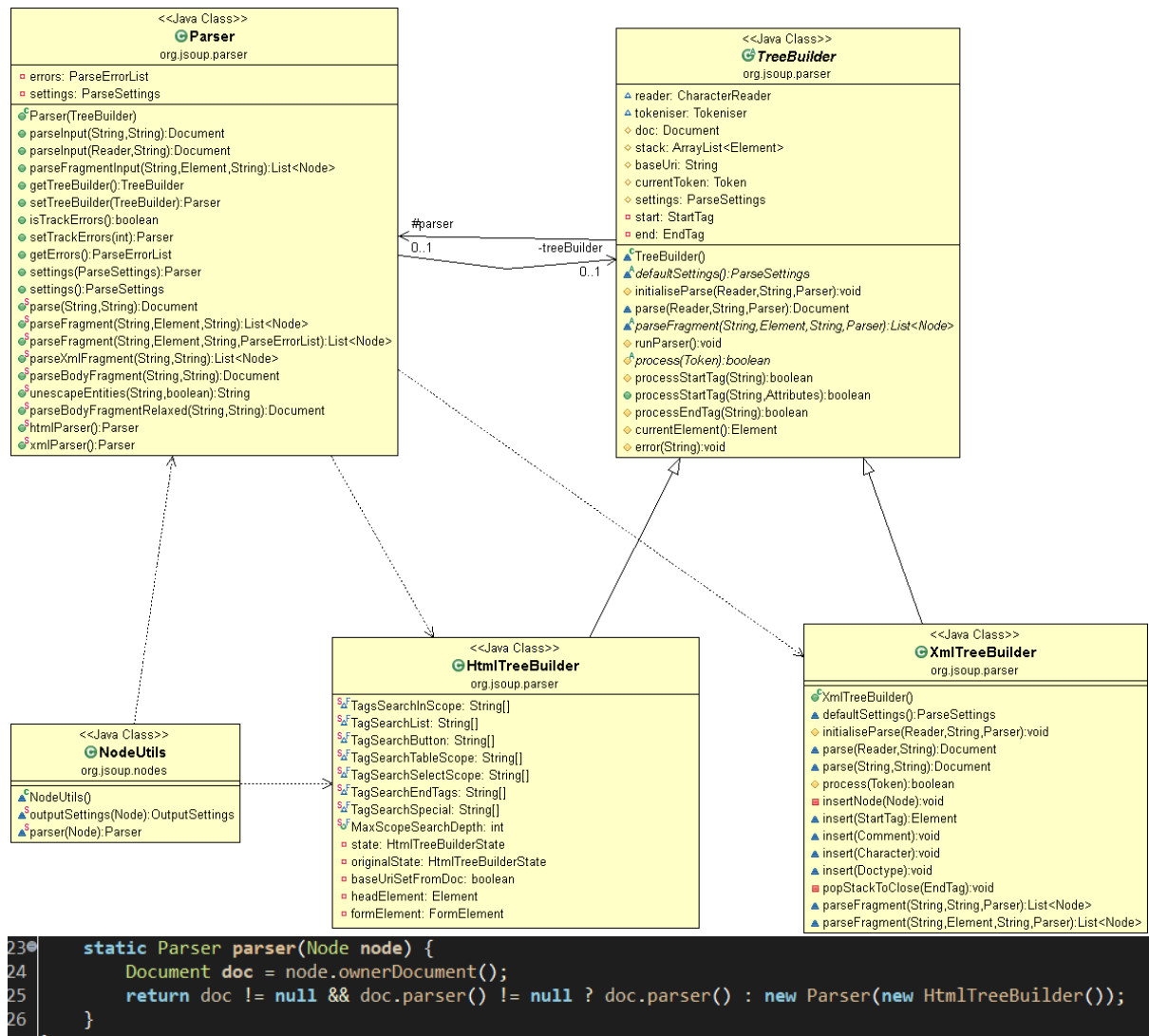
```

105 public static Document parse(String html, String baseUri) {
106     TreeBuilder treeBuilder = new HtmlTreeBuilder();
107     return treeBuilder.parse(new StringReader(html), baseUri, new Parser(treeBuilder));
108 }

```

Parser클래스는 concrete Tree Builder 클래스를 알고 있고, 명시적으로 new를 통해 클래스를 생성한다. 이로 인해 Parser클래스는 concrete Tree Builder에 대한 종속성이 생기게 된다.

또한, Parser클래스의 역할인 파싱과 Tree Builder를 생성하는 과정을 모두 Parser클래스에서 수행하는 것은 SRP에 위배된다.



분석과정에서 발견했던 추가적인 문제점은 다음과 같다.

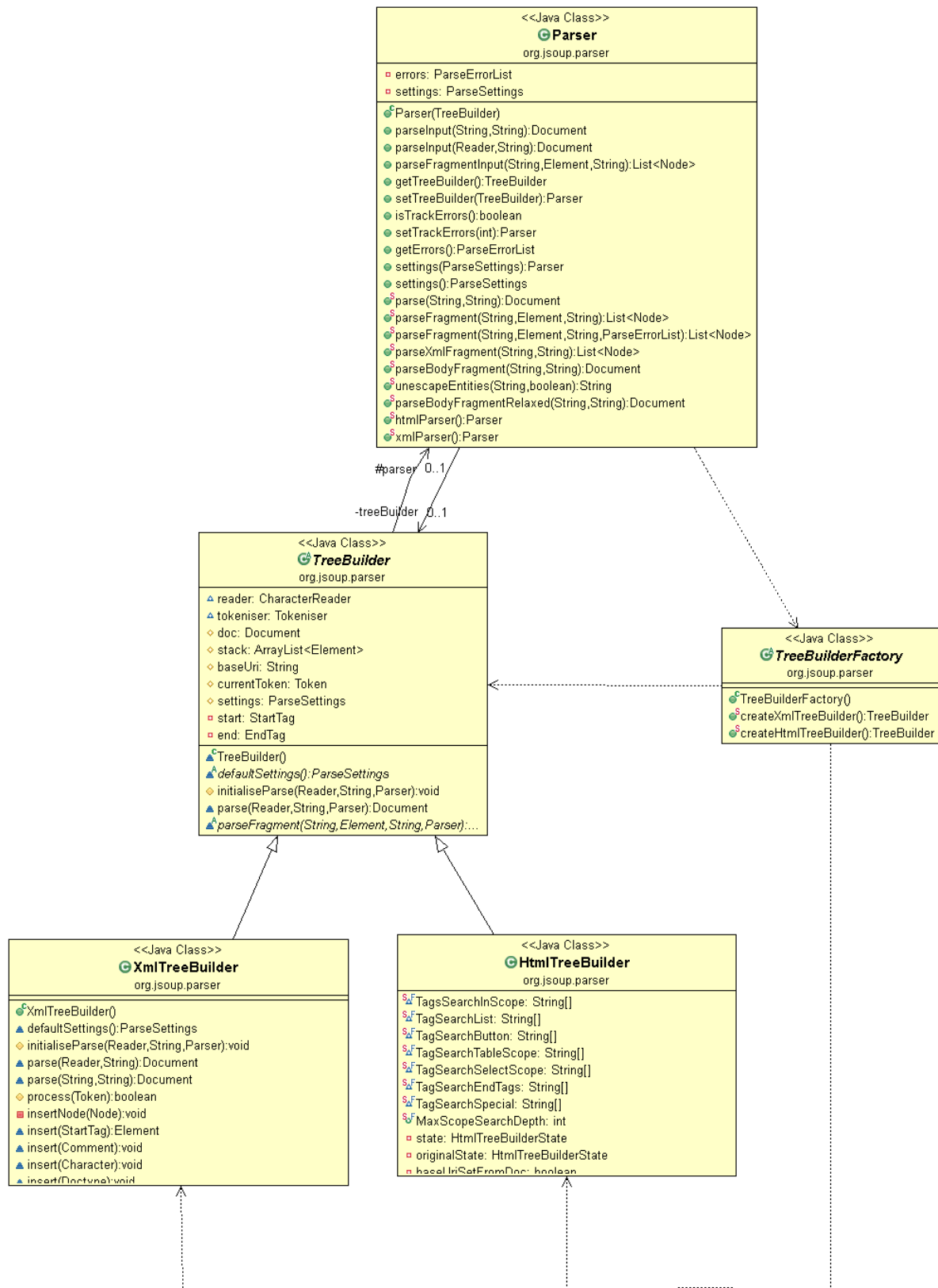
Parser클래스를 사용하는 NodeUtils 클래스에서 Parser를 생성하기 위해 concrete Tree Builder 클래스를 매개변수로 전달하는 과정이 있다.

다른 패키지에 위치한 NodeUtils 클래스가 Parser를 생성하기 위해 불필요하게 concrete Tree Builder까지 알게 되고 패키지간 커플링도 높아지게 되는 문제점이 있었다.

위와 같은 기존의 문제점을 개선하기 위해 클래스 생성과 관련된 패턴인 Factory 패턴을 적용하고자 하였다.

가장 먼저 고려했던 패턴은 클래스 생성과 관련된 Factory method 패턴이었다. 상속을 통해 Concrete Product(HtmlTreeBuilder, XmlTreeBuilder)에 대해 존재했던 Client(Parser)의 Dependency를 Abstract Product(TreeBuilder)로 한정하고, Factory Interface에 종속성을 주어 DIP를 적용하고자 하였다. 하지만, 기존 코드의 경우, parser 클래스내에 위치한 static method에서 Tree Builder를 생성하는 구조였다. 해당 구조에서 Factory method도 static으로 지원되어야 했고, 그 결과 상속관계를 활용할 수 없었기 때문에 이를 반영하는 것이 원천적으로 불가능하였다.

따라서, 기존 구조에서 DIP를 적용하는 것을 포기하고, Factory의 상속 구조가 필요 없는 Simple Factory 패턴을 적용하였다. 패턴이 적용된 다이어그램과 코드는 다음과 같다.




```

1 package org.jsoup.parser;
2
3 public class TreeBuilderFactory {
4     public static XmlTreeBuilder createXmlTreeBuilder() {
5         return new XmlTreeBuilder();
6     }
7     public static HtmlTreeBuilder createHtmlTreeBuilder() {
8         return new HtmlTreeBuilder();
9     }
10 }

```

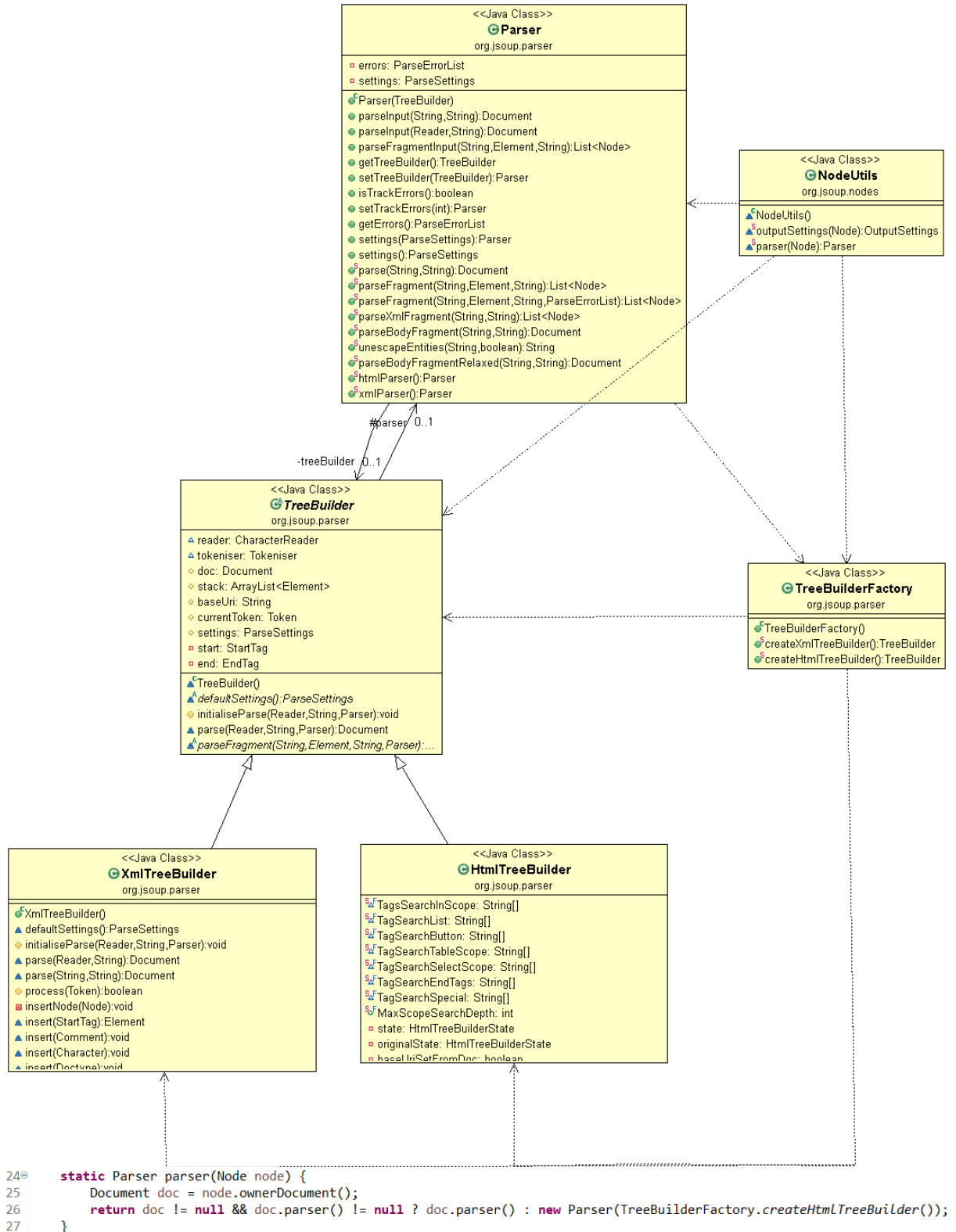
위와 같이 TreeBuilder에 대한 Simple Factory 클래스를 정의하였다.

```

    public static Document parse(String html, String baseUrl) {
        TreeBuilder treeBuilder = TreeBuilderFactory.createHtmlTreeBuilder();
        return treeBuilder.parse(new StringReader(html), baseUrl, new Parser(treeBuilder));
    }

```

또한, 기존의 Parser 클래스에서 concrete TreeBuilder클래스를 생성할 때, Simple factory 클래스를 통해서만 생성하도록 변경하였다.



Node Package에서 Parser를 생성할 때 호출하는 코드도 다음과 같이 변경하였다.

위와 같이 생성에 대한 부분을 변경함으로써, Product(TreeBuilder)를 사용하는 client인 Parser 및 NodeUtils클래스는 concrete Product(HtmlTreeBuilder, XmlTreeBuilder)를 직접적으로 알고 있지 않아도 문제없이 잘 동작할 수 있었다.

변경과정에서 발생했던 문제는 다음과 같다.

```
150 public static List<Node> parseXmlFragment(String fragmentXml, String baseUri) {
151     XmlTreeBuilder treeBuilder = new XmlTreeBuilder();
152     return treeBuilder.parseFragment(fragmentXml, baseUri, new Parser(treeBuilder));
153 }
136 public static List<Node> parseFragment(String fragmentHtml, Element context, String baseUri, ParseErrorList errorList) {
137     HtmlTreeBuilder treeBuilder = new HtmlTreeBuilder();
138     Parser parser = new Parser(treeBuilder);
139     parser.errors = errorList;
140     return treeBuilder.parseFragment(fragmentHtml, context, baseUri, parser);
141 }
```

기존에 Parser 클래스내에 위치한 parseFragment 메서드의 경우 XML 혹은 HTML의 문서 타입에 따라 TreeBuilder 메서드에 대한 상반된 호출 인자를 갖고 있었다.

```
51 abstract List<Node> parseFragment(String inputFragment, Element context, String baseUri, Parser parser);
```

Abstract TreeBuilder 클래스에 정의된 parseFragment 메서드

```
82 List<Node> parseFragment(String inputFragment, Element context, String baseUri, Parser parser) {
83     // context may be null
84     state = HtmlTreeBuilderState.Initial;
85     initialiseParse(new StringReader(inputFragment), baseUri, parser);
86     contextElement = context;
87     fragmentParsing = true;
88     Element root = null;
89
90     if (context != null) {
91         if (context.ownerDocument() != null) // quirks setup:
92             doc.quirksMode(context.ownerDocument().quirksMode());
93
94         // initialise the tokeniser state:
95         String contextTag = context.tagName();
96         if (StringUtil.in(contextTag, "title", "textarea"))
97             tokeniser.transition(TokeniserState.RCDATA);
98         else if (StringUtil.in(contextTag, "iframe", "noembed", "noframes", "style", "xmp"))
99             tokeniser.transition(TokeniserState.RawText);
100         else if (contextTag.equals("script"))
101             tokeniser.transition(TokeniserState.ScriptData);
102         else if (contextTag.equals("noscript"))
103             tokeniser.transition(TokeniserState.Data); // if scripting enabled, rawtext
104         else if (contextTag.equals("plaintext"))
105             tokeniser.transition(TokeniserState.Data);
106         else
107             tokeniser.transition(TokeniserState.Data); // default
108
109         root = new Element(Tag.valueOf("html", settings), baseUri);
110         doc.appendChild(root);
111     }
```

HtmlTreeBuilder 클래스에 정의된 parseFragment 메서드

```
144 List<Node> parseFragment(String inputFragment, String baseUri, Parser parser) {
145     initialiseParse(new StringReader(inputFragment), baseUri, parser);
146     runParser();
147     return doc.childNodes();
148 }
149
150 @Override
151 List<Node> parseFragment(String inputFragment, Element context, String baseUri, Parser parser) {
152     return parseFragment(inputFragment, baseUri, parser);
153 }
```

xmlTreeBuilder 클래스에 정의된 parseFragment 메서드

확인해 본 결과, HtmlTreeBuilder 클래스에 정의된 parseFramgent 메서드의 경우에는 내부적으로 인자값인 Element가 null일 때를 메서드 내에서 처리하지만, xml parser의 경우 인자로 받은 Element를 사용하지 않고 별도의 오버로딩된 parseFragment 메서드를 통해 로직을 수행한다.

Xml에 오버로딩을 통해 추가적으로 정의된 parseFramgement(String, String, Parser)의 경우 TreeBuilder 클래스에 별도로 정의되지 않아서, 호환되지 않는 문제점이 있었다.

```

51     abstract List<Node> parseFragment(String inputFragment, Element context, String baseUri, Parser parser);
52     abstract List<Node> parseFragment(String inputFragment, String baseUri, Parser parser);

```

Abstract TreeBuilder 클래스에 추가한 parseFragment 메서드

```

82     List<Node> parseFragment(String inputFragment, String baseUri, Parser parser)
83     {
84         return parseFragment(inputFragment, null, baseUri, parser);
85     }

```

HtmlTreeBuilder 클래스에 추가한 parseFragment 메서드

```

122     public static List<Node> parseFragment(String fragmentHtml, Element context, String baseUri) {
123         TreeBuilder treeBuilder = TreeBuilderFactory.createHtmlTreeBuilder();
124         return treeBuilder.parseFragment(fragmentHtml, context, baseUri, new Parser(treeBuilder));
125     }

```

위와 같이 부모 클래스에 상응한 API를 정의함으로써, Parser에서는 문서타입과 상관없이 polymorphism을 통한 parseFragment 메서드 호출을 가능하게 하였다. 이로써, Parser 클래스에서는 concrete TreeBuilder 클래스에 대한 정보 없이도 올바르게 수행될 수 있게 되었다.

4. 확장기능

■ 개요

- 우리 팀은 Jsoup을 모르는 사람도 손쉽게 그때그때 문서를 가져와 사용 할 수 있는 Interface를 만들어 보았다. 기존에 Jsoup을 이용해 문서를 가져와 Parsing 하려면 Jsoup.Connect(address).get 을 이용해 문서를 가져와야 했고 얻어진 Document에 대해 필요한 함수를 찾아 입력해 주어야 했다. 우리는 이런 방법뿐만 아니라 프로그램이 한번 실행되면 그때그때 필요한 주소를 입력하고 문서를 가져 올 수 있는 UI 프로그램을 작성하였다.
- 자세하게는 Document를 가져와서 Class명이나 ID 로 Select를 할 수 있고 얻어진 Element에 대해 부모, 자식, 형제 Element를 간단한 입력 명령어로 손쉽게 얻을 수 있는 기능을 가진 인터페이스 이다.

■ 확장기능 소개

- 우리의 확장 기능은 Singleton객체 이자 Façade 객체인 UserInterface 객체를 통해 이용 할 수 있다.

```
public class MainClass {

    public static void main(String[] args) {
        UserInterface userInterface;

        /* Singleton 객체인 UserInterface 객체 가져옴 */
        userInterface = UserInterface.getUserInterface();

        /* 시작! */
        userInterface.run();

    }

}
```

- Singleton 객체인 UserInterface를 불러와 run 함수를 통해 모든 일들을 수행 하게 된다.
- UserInterface를 실행하면 Document를 가져오고 싶은 주소, 하고자 하는 행동 을 선택 할 수 있다.

```
*** Jsoup Interface ***
주소를 입력 해 주세요 : http://127.0.0.1:8000

http://127.0.0.1:8000 를 가져오는중....
문서 가져오기 완료.

*** 무엇을 할까요? ***
1. Search by ClassName
2. Search by Id
3. 다른 주소 입력
0. 끝내기
1
찾고자 하는 Class 이름을 입력 해 주세요 : 3rd

1개의 3rd을(를) 찾았습니다.
<div class="3rd" id="3rd">
  <div class="4th" id="4th">
  </div>
</div>
( 1 / 1 )

1. NEXT
2. PREVIOUS
3. 결정
```

- 맨 처음에 Element를 찾을 때는 같은 속성 이름을 가진 클래스가 여러 개 존재 할 수 있으니 여러 개중에 하나를 선택 할 수 있게 하였다. 만약 사용자가 하나의 Element를 선택 한다면 그 Element로부터 부모, 자식, 형제 Element들을 찾을 수 있게 된다.

➤ UP(부모 찾기), DOWN(자식 찾기), NEXT(형제 찾기), PREVIOUS(형제 찾기)

```

무슨 행동을 할까요?
1. UP!
2. DOWN!
3. NEXT!
4. PREVIOUS!
5. 그만두기
2

( 1 / 1 )
<div class="4th" id="4th">
</div>

무슨 행동을 할까요?
1. UP!
2. DOWN!
3. NEXT!
4. PREVIOUS!
5. 그만두기
1
<div class="3rd" id="3rd">
  <div class="4th" id="4th">
  </div>
</div>

```

(3rd 클래스에서 - > 자식 - > 부모를 탐색 한 결과)

- 5. 그만두기를 누른다면 주소를 다시 입력하거나 끝내는 등의 행동을 할 수 있다.

```

무슨 행동을 할까요?
1. UP!
2. DOWN!
3. NEXT!
4. PREVIOUS!
5. 그만두기
5
*** 무엇을 할까요? ***
1. Search by ClassName
2. Search by Id
3. 다른 주소 입력
0. 끝내기

```

■ 적용된 디자인 패턴

- Singleton Pattern - 확장 기능을 위한 프로그램의 생명 주기를 담당하는 UserInterface 객체를 Singleton 으로 작성하였다.

```

/* Singleton */
private volatile static UserInterface userInterface = null;

public static UserInterface getUserInterface() {
    if (userInterface == null) {
        synchronized(UserInterface.class) {
            if(userInterface == null) {
                userInterface = new UserInterface();
            }
        }
    }
    return userInterface;
}

```

- Façade Pattern – Main 함수 내에 userInterface의 'run' method를 한번 호출 함으로써 모든 확장기능을 이용 할 수 있도록 Façade Pattern을 적용하여 작성 하였다.

```
/* 시작! */
userInterface.run();
```

- Strategy Pattern – 우리 프로그램은 runtime에 사용자의 입력을 받아 Element를 탐색 하는 행위를 한다. Element를 기준으로 runtime에 behavior(algorithm)가 바뀌는 상황 이므로 그때 그때 필요한 알고리즘을 삽입하기 위하여 Strategy Pattern을 이용하였 다.

- Traverse 라는 Interface를 만들었다.

```
public interface Traverse {
    public Element traverse(Element els);
}
```

- Traverse 인터페이스를 구현하는 여러 알고리즘 들이 있다.

```
public class TraverseNext implements Traverse {
    public Element traverse(Element e) {
        if (e.nextElementSibling() != null) {
            e = e.nextElementSibling();
            System.out.println("\n( " + Integer.toString(e.elementSiblingIndex()+1) + " / "
                + Integer.toString(e.elementSiblingSize(e)) + " )");
            return e;
        } else {
            System.err.println("\n마지막 원소 입니다.");
            return e;
        }
    }
}
```

org.jsoup.userInterface

- Traverse.java
- TraverseController.java
- TraverseDown.java
- TraverseNext.java
- TraversePrevious.java
- TraverseUp.java

- TraverseController에서 SetTraverseBehavior를 통해 runtime에 알고리즘을 바꿀 수 있다.

```
void setTraverseBehavior(Traverse tb) {
    this.traverseBehavior = tb;
}
```

```

/* Strategy Pattern */
switch(run_code) {
case 1:
    setTraverseBehavior(new TraverseUp());
    break;
case 2:
    setTraverseBehavior(new TraverseDown());
    break;
case 3:
    setTraverseBehavior(new TraverseNext());
    break;
case 4:
    setTraverseBehavior(new TraversePrevious());
    break;
}
e = traverseBehavior.traverse(e);

```

Traverse 는 결국 Element 에 대한 연산 과정을 지원해주는 알고리즘들의 인터페이스로써 현재는 Up, Down Next, Previous 의 네 가지 동작이 있지만 향후 Element 에 관한 추가적인 연산을 지원하는 확장의 가능성을 열어 두어 알고리즘들을 Encapsulate 하여 runtime 에 바로 교체 할 수 있는 Strategy Pattern 을 적용하기로 하였다.

5. 테스트 수행 내역

- 테스트는 JAVA의 Unit Test 라이브러리인 Junit을 이용 하였다.
- 확장 기능인 Traverse Interface의 TraverseUP, TraverseDown, TraverseNext, TraversePrevious 클래스들의 traverse함수에 대한 테스트를 진행 하였다.
- 테스트는 jsoup의 공식 홈페이지인 jsoup.org 의 Document를 가져와서 진행 하였다.

The screenshot shows the jsoup.org homepage. The main content area lists several features:

- scrape and parse HTML from a URL, file, or string
- find and extract data, using DOM traversal or CSS selectors
- manipulate the HTML elements, attributes, and text
- clean user-submitted content against a safe white-list, to prevent XSS attacks
- output tidy HTML

 Below this, it states: "jsoup is designed to deal with all varieties of HTML found in the wild; from pristine and validating, to invalid tag-soup; jsoup will create a sensible parse tree."

On the right side, there is a sidebar with a list of links:

3. Parsing a body fragment
4. Load a Document from a URL
5. Load a Document from a File
6. Use DOM methods to navigate a document
7. Use selector-syntax to find elements
8. Extract attributes, text, and HTML from elements
9. Working with URLs

At the bottom of the sidebar, under the heading "Extracting data", there is a section titled "Example" which says: "Fetch the Wikipedia homepage, parse it to a DOM, and select the headlines from the...".

On the far right, the "Elements" panel of a web browser is visible, showing the DOM tree structure of the page. The root is <html>, which contains <head> and <body>. The <body> has a class of "n1-home" and contains a <div class="wrap"> which contains a <div class="header"> and a <div class="nav-sections">. The <div class="nav-sections"> contains a with several elements, each with a class like "n1-home", "n1-news", "n1-bugs", "n1-discussion", "n1-download", "n1-api", "n1-cookbook", and "n1-try".


```

public class TraverseTest {

    private static String address;
    private static Document doc;

    private Traverse tb;

    static {
        address = "https://jsoup.org/";
        try {
            doc = Jsoup.connect(address).get();
        } catch (Exception e) {}
    }

    public void TraverseUpTest() {}

    public void TraverseDownTest() {}

    public void TraverseNextTest() {}

    public void TraversePreviousTest() {}

}

```

- TraverseUP에 대한 테스트

```

▼<body class="n1-home">
  ▼<div class="wrap">
    ▼<div class="header">
      ▼<div class="nav-sections">
        ▼<ul>

```

```

@Test
public void TraverseUpTest() {
    Elements els = doc.select(".nav-sections");
    Element e = els.get(0);
    tb = new TraverseUp();
    assertEquals(doc.select(".header").get(0), tb.traverse(e));
    e = tb.traverse(e);
    assertEquals(doc.select(".wrap").get(0), tb.traverse(e));
    e = tb.traverse(e);
    assertEquals(doc.select(".n1-home").get(0), tb.traverse(e));
}

```

- nav-sections 클래스를 가져와 부모 노드를 가져오면서 테스트를 진행하였고 성공적으로 수행 하였다.

- TraverseNext에 대한 테스트

```

▼<ul>
  ▶<li class="n1-home">...</li>
  ▶<li class="n1-news">...</li>
  ▶<li class="n1-bugs">...</li>
  ▶<li class="n1-discussion">...</li>
  ▶<li class="n1-download">...</li>
  ▶<li class="n1-api">...</li>
  ▶<li class="n1-cookbook">...</li>
  ▶<li class="n1-try">...</li>
</ul>

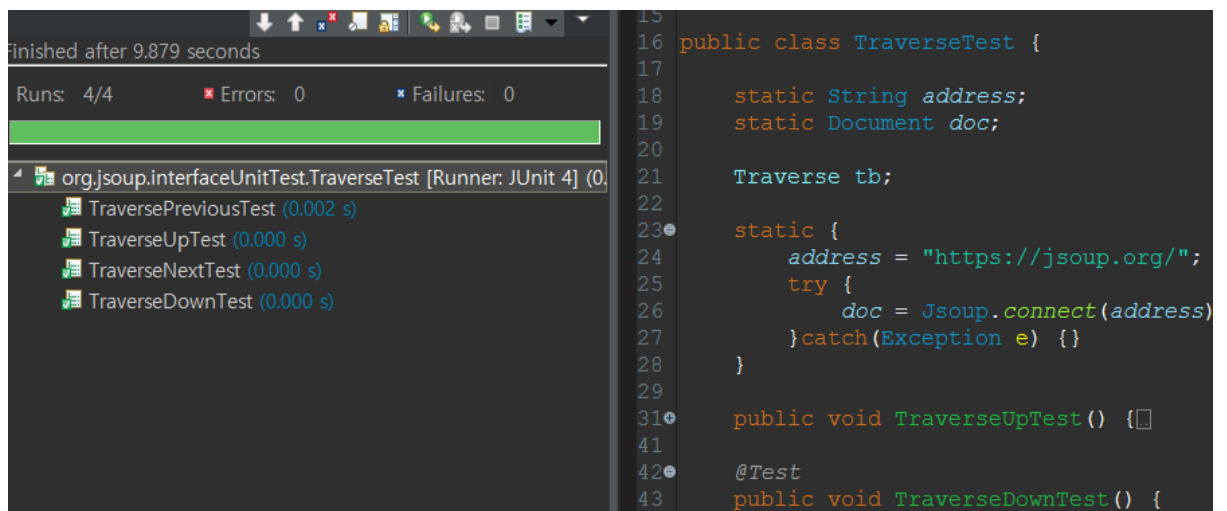
```

```

@Test
public void TraverseNextTest() {
    Elements els = doc.select(".n1-discussion");
    Element e = els.get(0);
    tb = new TraverseNext();
    assertEquals(doc.select(".n1-download").get(0), tb.traverse(e));
    e = tb.traverse(e);
    assertEquals(doc.select(".n1-api").get(0), tb.traverse(e));
    e = tb.traverse(e);
    assertEquals(doc.select(".n1-cookbook").get(0), tb.traverse(e));
    e = tb.traverse(e);
    assertEquals(doc.select(".n1-try").get(0), tb.traverse(e));
    // "n1-try"가 가장 마지막 클래스 이기 때문에 traverse를 한 번 더 해도 "n1-try"를 반환
    e = tb.traverse(e);
    assertEquals(doc.select(".n1-try").get(0), tb.traverse(e));
}

```

- 리스트 중 n1-discussion 클래스를 가져와 Next Sibling을 가져오면서 테스트를 진행해 보았다. n1-try 클래스가 가장 마지막에 있는 클래스 이기 때문에 n1-try 에서 next를 진행 한 경우 다시 n1-try가 반환 되는 것을 확인하였고 테스트가 성공적으로 수행됨을 확인 하였다.
- TraverseDown과 TraversePrevious도 같은 방법으로 테스트를 진행 하였고 아래와 같이 모든 테스트를 통과 하였음을 확인 하였다.



The screenshot shows an IDE with two panels. The left panel displays the test results for the `org.jsoup.interfaceUnitTest.TraverseTest` class, showing four tests passed: `TraversePreviousTest` (0.002 s), `TraverseUpTest` (0.000 s), `TraverseNextTest` (0.000 s), and `TraverseDownTest` (0.000 s). The right panel shows the source code for the `TraverseTest` class, which includes static variables `address` and `doc`, a `Traverse` instance `tb`, and methods for `TraverseUpTest` and `TraverseDownTest`.

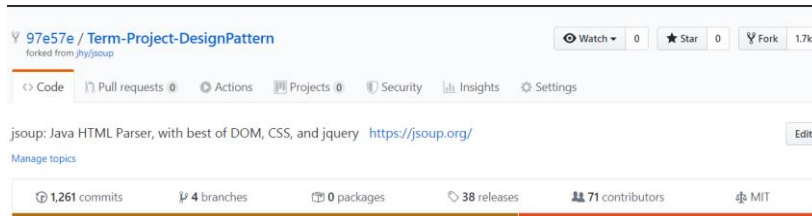
```

15
16 public class TraverseTest {
17
18     static String address;
19     static Document doc;
20
21     Traverse tb;
22
23     static {
24         address = "https://jsoup.org/";
25         try {
26             doc = Jsoup.connect(address)
27             }catch(Exception e) {}
28         }
29
30
31     public void TraverseUpTest() {
32
33     }
34
35     @Test
36     public void TraverseDownTest() {
37
38     }
39
40 }

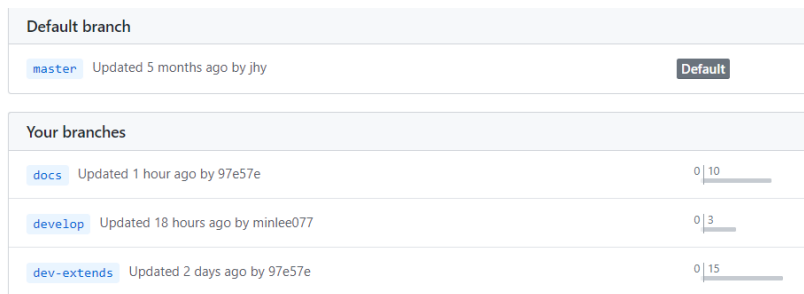
```

6. GitHub 프로젝트 활용 요약

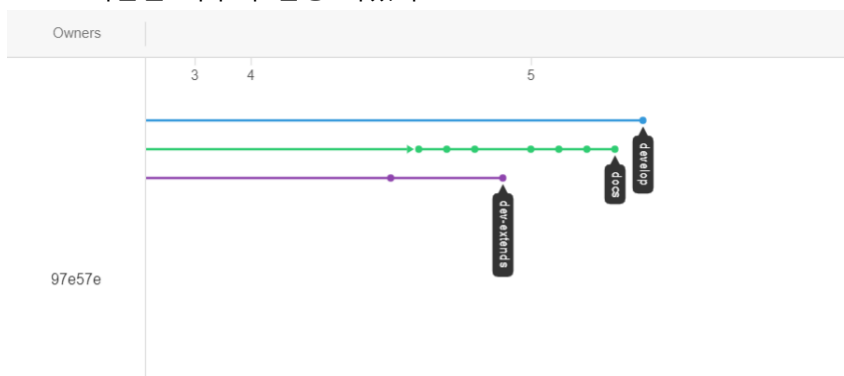
1. 프로젝트는 GITHUB에 올라와 있는 Jsoup 원본 repository (<https://github.com/jhy/jsoup>)를 fork 해서 진행하였다.



2. Develop, dev-extends, docs branch를 생성하여 각각 설계 개선(Develop), 확장 기능 구현(dev-extends), 문서 정리(docs)를 진행 하였다.



3. 우리 팀은 3인으로 진행하여
 - 서재훈 : Jsoup에 적용된 패턴 분석 및 문서화
 - 이정민 : 설계 개선점 찾기 및 설계 개선
 - 이정원 : 추가 기능 구현 및 테스트로 역할을 나누어 진행 하였다.



(진행중인 브랜치)

4. 확장 기능을 구현 하고 설계 개선을 완료 한 뒤 Branch를 합쳐 테스트를 진행 하였다. 이후 master branch로 merge시켜 최종 버전을 올려두었다.

- Develop & dev-extends branch 병합



- 모든 branch 병합

