HTTP 웹 서버 리팩토링 실습

—

실습 환경

- https://github.com/slipp/web-application-server 저장소를 자신의 계정으로 fork한다.
- Fork한 저장소를 https://youtu.be/xid_GG8kL_w 동영상 참고해 로컬 개발 환경을 구축한다.
- web-application-server 저장소의 was-step1-bad-version 브랜치로 변경한다.
    - 브랜치를 변경하는 방법은 https://youtu.be/VeTjDYI7UVs 동영상을 참고한다.

웹 서버 시작 및 테스트

- webserver.WebServer 는 사용자의 요청을 받아 RequestHandler 에 작업을 위임하는 클래스이다.
- 사용자 요청에 대한 모든 처리는 RequestHandler 클래스의 run() 메서드가 담당한다.
- WebServer를 실행한 후 브라우저에서 http://localhost:8080으로 접속해 질문/답변 게시판 서비스 화면이 나타나는지 확인한다.

요구사항

지금까지 단계를 구현하고 보니 소스 코드의 복잡도가 많이 증가했다.

소스 코드 리팩토링을 통해 복잡도를 낮춰보자.

Bad Smell을 찾는 것은 쉽지 않은 작업이다.
리팩토링할 부분을 찾기 힘든 사람은 다음으로 제시하는 1단계 힌트를 참고해 리팩토링을 진행해 볼 것을 추천한다.
만약 혼자 힘으로 리팩토링할 부분을 찾은 사람은 먼저 도움 없이 리팩토링을 진행한다.

리팩토링 1단계 힌트

—

힌트

- RequestHandler 클래스의 책임을 분리한다.
  - RequestHandler 클래스는 많은 책임을 가지고 있다. 객체 지향 설계 원칙 중 “단일 책임의 원칙”에 따라RequestHandler 클래스가 가지고 있는 책임을 찾아 각 책임을 새로운 클래스를 만들어 분리한다.

힌트

- 클라이언트 요청 데이터를 처리하는 로직을 별도의 클래스로 분리한다.(HttpRequest)
- 클라이언트 응답 데이터를 처리하는 로직을 별도의 클래스로 분리한다.(HttpResponse)
- 다형성을 활용해 클라이언트 요청 URL에 대한 분기 처리를 제거한다.

리팩토링 2단계 힌트

—

요구사항

클라이언트 요청 데이터를 처리하는 로직을 별도의 클래스로 분리한다.(HttpRequest)

힌트

- 클라이언트 요청 데이터를 담고 있는 InputStream을 생성자로 받아 HTTP 메소드, URL, 헤더, 본문을 분리하는 작업을 한다.
- 헤더는 Map<String, String>에 저장해 관리하고 getHeader("필드 이름") 메소드를 통해 접근 가능하도록 구현한다.
- GET과 POST 메소드에 따라 전달되는 인자를 Map<String, String>에 저장해 관리하고 getParameter("인자 이름") 메소드를 통해 접근 가능하도록 구현한다.
- RequestHandler가 새로 추가한 HttpRequest를 사용하도록 리팩토링한다.

힌트

GET 테스트 데이터(src/test/resources 에 생성)

```
GET /user/create?userId=javajigi&password=password&name=JaeSung HTTP/1.1
Host: localhost:8080
Connection: keep-alive
Accept: */*
```

빈 공백 문자열

```
public class HttpRequestTest {
    private String testDirectory = "./src/test/resources/";


    @Test
    public void request_GET() throws Exception {
        InputStream in = new FileInputStream(new File(testDirectory + "Http_GET.txt"));
        HttpRequest request = new HttpRequest(in);

        assertEquals("GET", request.getMethod());
        assertEquals("/user/create", request.getPath());
        assertEquals("keep-alive", request.getHeader("Connection"));
        assertEquals("javajigi", request.getParameter("userId"));
    }
}
```

힌트

POST 테스트 데이터(src/test/resources 에 생성)

```
POST /user/create HTTP/1.1
Host: localhost:8080
Connection: keep-alive
Content-Length: 46
Content-Type: application/x-www-form-urlencoded
Accept: */*

userId=javajigi&password=password&name=JaeSung
```

```
public class HttpRequestTest {
    private String testDirectory = "./src/test/resources/";

    @Test
    public void request_POST() throws Exception {
        InputStream in = new FileInputStream(new File(testDirectory + "Http_POST.txt"));
        HttpRequest request = new HttpRequest(in);

        assertEquals("POST", request.getMethod());
        assertEquals("/user/create", request.getPath());
        assertEquals("keep-alive", request.getHeader("Connection"));
        assertEquals("javajigi", request.getParameter("userId"));
    }
}
```

요구사항

클라이언트 응답 데이터를 처리하는 로직을 별도의 클래스로 분리한다.(HttpResponse)

힌트

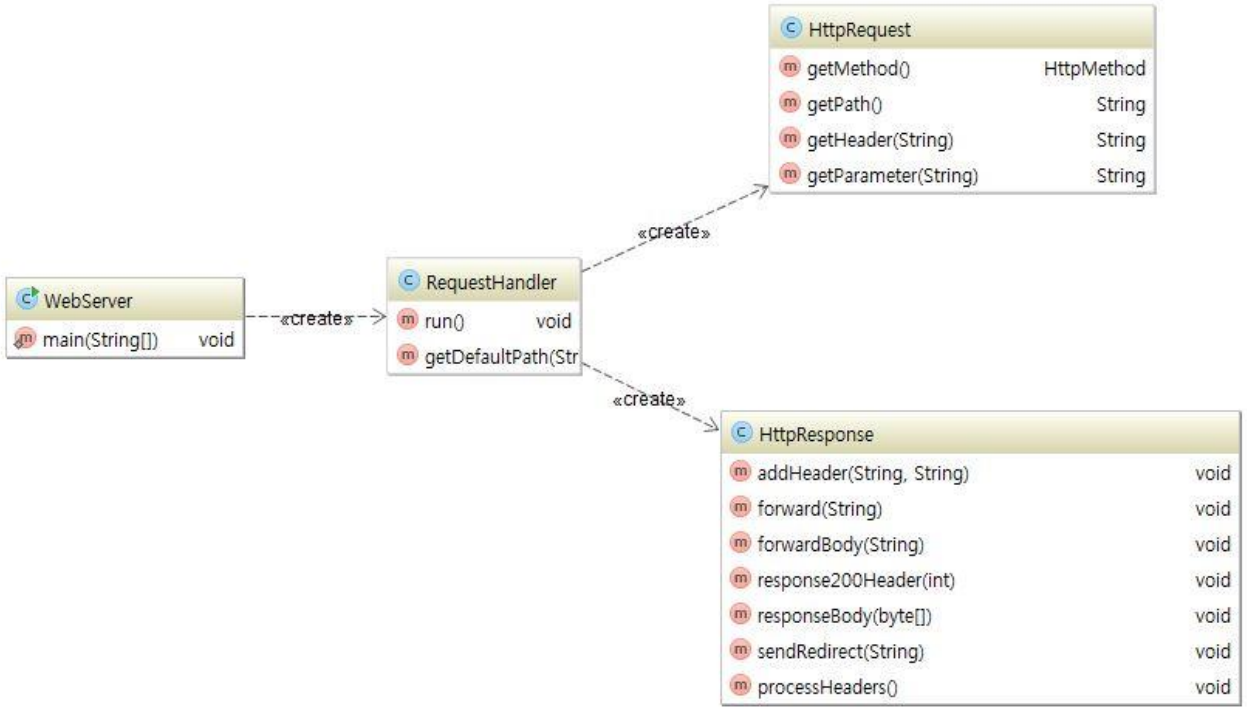
- RequestHandler 클래스를 보면 응답 데이터 처리를 위한 많은 중복이 있다. 이 중복을 제거해 본다.
- 응답 헤더 정보를 Map<String, String>으로 관리한다.
- 응답을 보낼 때 HTML, CSS, 자바스크립트 파일을 직접 읽어 응답으로 보내는 메소드는 forward(), 다른 URL로 리다이렉트하는 메소드는 sendRedirect() 메소드를 나누어 구현한다.
- RequestHandler가 새로 추가한 HttpResponse를 사용하도록 리팩토링한다.

힌트

```
public class HttpResponseTest {
    private String testDirectory = "./src/test/resources/";

    @Test
    public void responseForward() throws Exception {
        // Http_Forward.txt 결과는 응답 body에 index.html이 포함되어 있어야 한다.
        HttpResponse response = new HttpResponse(createOutputStream("Http_Forward.txt"));
        response.forward("/index.html");
    }

    @Test
    public void responseRedirect() throws Exception {
        // Http_Redirect.txt 결과는 응답 headere에 Location 정보가 /index.html로 포함되어 있어야
한다.
        HttpResponse response = new HttpResponse(createOutputStream("Http_Redirect.txt"));
        response.sendRedirect("/index.html");
    }

    @Test
    public void responseCookies() throws Exception {
        // Http_Cookie.txt 결과는 응답 header에 Set-Cookie 값으로 logined=true 값이 포함되어 있
어야 한다.
        HttpResponse response = new HttpResponse(createOutputStream("Http_Cookie.txt"));
        response.addHeader("Set-Cookie", "logined=true");
        response.sendRedirect("/index.html");
    }

    private OutputStream createOutputStream(String filename) throws FileNotFoundException {
        return new FileOutputStream(new File(testDirectory + filename));
    }
}
```

힌트

지금까지 과정으로 리팩토링했을 때의 클래스 다이어그램



요구사항

다형성을 활용해 클라이언트 요청 URL에 대한 분기 처리를 제거한다.

힌트

- 각 요청과 응답에 대한 처리를 담당하는 부분을 추상화해 인터페이스로 만든다. 인터페이스는 다음과 같이 구현할 수 있다.

```
public interface Controller {
    void service(HttpRequest request, HttpResponse response);
}
```

- 각 분기문을 Controller 인터페이스를 구현하는(implements) 클래스를 만들어 분리한다.
- 이렇게 생성한 Controller 구현체를 Map<String, Controller>에 저장한다. Map의 key에 해당하는 String은 요청 URL, value에 해당하는 Controller는 Controller 구현체이다.
- 클라이언트 요청 URL에 해당하는 Controller를 찾아 service() 메소드를 호출한다.
- Controller 인터페이스를 구현하는 AbstractController 추상클래스를 추가해 중복을 제거하고, service() 메소드에서 GET과 POST HTTP 메소드에 따라 doGet(), doPost() 메소드를 호출하도록 한다.

힌트

지금까지 과정으로 리팩토링했을 때의 클래스 다이어그램