*객체지향 설계와 패턴*  
**LAB ASSIGNMENT #8: 프록시, 책임체인, 플라이웨이트 패턴**

## Weight 10% 마감 4월 28일(수)

**목적:** 프록시, 책임체인, 플라이웨이트 패턴

**프록시 패턴:**

문제 #1: 프록시 디자인 패턴은 안전하고 비용이 많이 드는 다른 객체에 대한 액세스를 제어하는 ​​데 사용된다. 예를 들어 미성년자의 인터넷 사용이다. 미성년자인 학생은 제한된 사이트에 액세스할 수 없다. 미성년자가 제한된 사이트를 로드하려고 하면 사이트가 제한되어 있다는 다른 페이지가 로드된다.

이것이 바로 프록시 패턴이 하는 일이다. 사용자에게 권한이 부여되면 사이트가 열리고(또는 보안 개체에 대한 액세스 권한이 부여됨)그렇지 않으면 오류 메시지가 표시된다.

이 패턴을 아래의 실제 라이브 프로젝트 예제에서 이해해 보자.

전자 상거래 Android 응용 프로그램에서 작업하고 있는데 사용자가 로그인(인증된 액세스)하거나 아직 로그인하지 않은 경우에도 제품을 볼 수 있도록 지원했다(예: 로그인 한 사용자가 가지고 있는 권한이 없는 게스트 사용자이다).

게스트 사용자는 로그인 할 필요가 없으며 제품을 보고 결제할 수 있다. 그러나 사용자가 로그인했는지 여부를 확인하여 로그인 여부, 사용자 유형에 따라 UI가 일부 변경된다. 예를 들어 게스트인 경우 구매(체크아웃)할 때 로그인 페이지가 표시되고 로그인하게 하는 경우다. 다시 로그인하지 않고 진행하기로 선택하면 배송에 대한 세부 정보를 입력할 수 있는 세부 정보 페이지가 표시된다.

결국 사용자가 권한을 부여 받았는지 또는 게스트인지에 따라 다양한 유형의 액세스가 제공되는 문제이다. 프록시 디자인 패턴은 객체에 대한 액세스를 제어하는 이런 문제 유형을 처리하기에 적합합니다.

사용자에 대하여 다음과 같이 설정하였다고 하자.

interface User {

void checkOut();

}

class AuthorizedUser implements User {

@Override

public void checkOut() {

System.out.print("checking out...");

}

}

class GuestUser implements User {

@Override

public void checkOut() {

System.out.print("Showing Registration Page");

}

}

사용자가 로그인한 경우에는 "체크아웃 중 ..."이라는 메시지가 표시되고 그렇지 않으면 "등록 페이지 표시" 메시지가 표시된다. 즉, 로그인 한 사용자의 경우 정상적인 체크아웃 기능이 발생하고 그렇지 않으면 체크 아웃 하기전에 게스트 사용자에 대한 등록 페이지가 로드되도록 프록시 패턴을 이용하여 UserProxy 인터페이스와 UserProxyHandler 클래스를 구현하라. 구동하는 메인 프로그램은 다음과 같다.

class MainClass {

public static void main(String[] args) {

boolean isUserLoggedIn= UserManagerService.isUserLoggedIn();

User user=new UserProxyHandler(isUserLoggedIn).getUser();

user.checkOut();

}

}

**책임 체인 패턴**

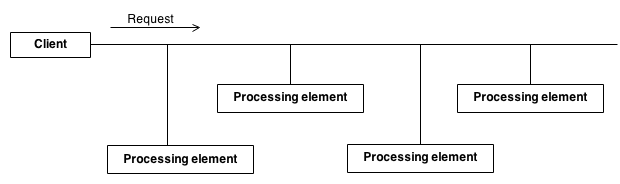
문제 #2:

(1) 책임 체인 패턴이 무엇인가? 어떤 상황에 사용해야 하는 것인가?

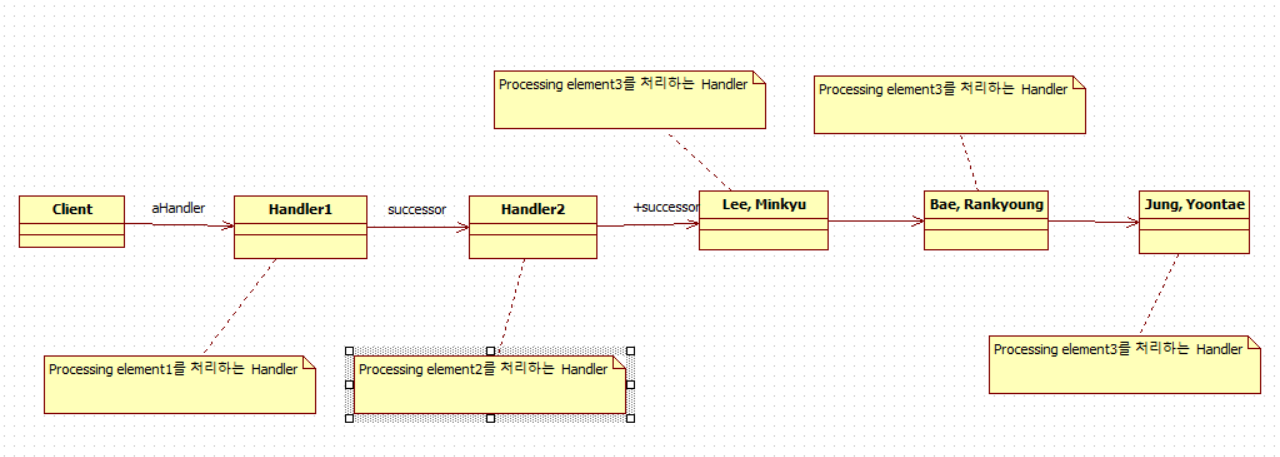
|  |  |
| --- | --- |
|  |  |

어떤 요청이 그 요청을 담당하는 객체에 들어오면 각각의 요청에 대해서 특정한 객체가 담당하는 것이 일반적이지만 객체를 연결리스트와 같은 사슬 방식으로 연결한 후에 요청을 수행하지 못하는 객체라면 다음 객체에 넘기며 책임을 넘기는 형태의 패턴을 뜻한다. 여러 객체 중에서 요청을 처리할 수 있는 객체가 무엇인지 사전에 알 수 없을 때 사용한다. 즉, 클라이언트는 요청을 일단 보내고 요청을 처리할 수 있는 객체가 요청을 처리할 때까지 책임을 넘기는 것이다.

(2) 미래 변경이 예측되는 개수의 핸들러(처리 요소) 즉 "노드" 객체와 처리해야 하는 요청 스트림이 있다. 하드와이어링 된 핸들러와의 관계나 우선 순위, 요청과 핸들러 사이의 매핑없이 요청을 효율적으로 처리해야 한다고 하자.



클라이언트가 각 핸들러에게 처리를 요청하면 파이프 라인으로 구성된 책임 체인의 추상으로 전달되고 책임체인 내부에 있는 처리 요소들이 처리하도록 캡슐화 하라. 즉 클라이언트가 파이프 라인 입구에서 요청을 "시작하고 나가도록" 블록다이어그램을 수정하여 그려라.



(3) 다음 코드를 실행한 후 그 결과를 인쇄하라. 이 코드에 책임체인 패턴이 어떻게 적용되었는지 이해하고 설명하라.

interface Image {

String process();

}

class IR implements Image {

public String process() {

return "IR";

}

}

class LS implements Image {

public String process() {

return "LS";

}

}

class Processor {

private static final Random RANDOM = new Random();

private static int nextID = 1;

private int id = nextID++;

public boolean execute(Image img) {

if (RANDOM.nextInt(2) != 0) {

System.out.println(" Processor " + id + " is busy");

return false;

}

System.out.println("Processor " + id + " - " + img.process());

return true;

}

}

public class ChainDemo {

public static void main( String[] args ) {

Image[] inputImages = {new IR(), new IR(), new LS(), new IR(), new LS(), new LS()};

Processor[] processors = {new Processor(), new Processor(), new Processor()};

for (int i=0, j; i < inputImages.length; i++) {

System.out.println("Operation #" + (i + 1) + ":");

j = 0;

while (!processors[j].execute(inputImages[i])) {

j = (j + 1) % processors.length;

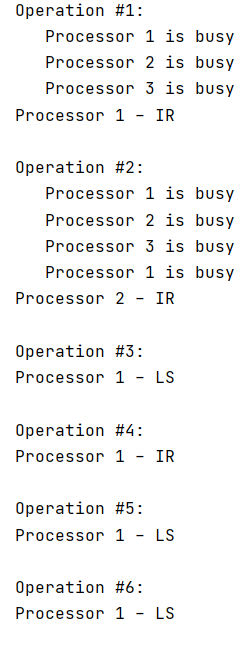
}

System.out.println();

}

}

}



Operation은 Image객체를 프로세스하는 단계이다. 예를 들어Operation #1은 inputImage[0]을 처리한다. Image객체의 처리는 processor의 execute에 의해 수행되는데, 현재의processor가 busy할 시(busy 여부는 RANDOM 변수에 의해 결정됨) 다음 processor가 이를 담당하게 된다. 즉, 현재 processor가 Image를 execute할 수 있으면(책임을 수행할 수 있으면) 자기자신이 수행하고, 할 수 없으면 다음 processor로 넘긴다. 이 코드에서 책임의 체인은 while(!processors[j].execute(inputImages[i])) 블록에 의해 수행된다. 처리할 수 있으면(현재 processor가 책임을 수행할 수 있으면) 수행했다고 현재의 inputImage가 무엇인지 리턴하지만(Processor 1 – IR), 처리할 수 없으면 처리할 수 있을때까지 계속 다음 Processor로 넘긴다(Processor 1 is busy, Processor 2 is busy, Processor 3 is busy … ).

**플라이웨이트 패턴**

문제 #3:

(1) 플라이웨이트 패턴의 각 요소를 나열하고 그 역할을 설명하라.

- FlyWeight

1. (만일 있다면)extrinsic data를 넘겨받아, 연산을 수행

2. 공유에 사용할 클래스들의 인터페이스를 정의

- ConcreteFlyweight

1. 실제 공유될 객체를 표현

2. FlyWeight 인터페이스를 구현하고 intrinsic data를 보관

- UnSharedConcreteFlyweight

1. 공유될 순 없지만, 공유되는 객체들과 같이 취급되는 객체들을 표현

- FlyWeightFactory

1. FlyWeight 객체를 생성하고 관리

2. 공유해주는 역할을 함

- Client

1. (필요하다면extrinsic data를 보관하거나 계산한다

(2) javax.servlet.FilterChain 의 Java API를 조사하고 존재하는 ~~플라이웨이트~~ 책임 체인 디자인 패턴을 식별하여 설명하라.

javax.servlet의 FilterChain 인터페이스는 리소스에 대한 필터링된 요청의 호출체인을 볼 수 있다. Filter는 FiterChain을 이용해서 체인안의 다음 Filter를 호출한다. 만약, 호출 필터가 체인의 마지막 필터인 경우, 체인 끝의 리소스를 호출한다.

FilterChain 인터페이스에 선언된 메소드는 doFilter(request, response)메소드 하나뿐인데, 이 메소드는 체인 안의 호출되어야할 다음 Filter를 가져온다. 만약 호출필터가 체인의 마지막 필터인 경우, 체인 끝의 리소스가 호출된다.

(3) FilterChain API를 이용하는 다음 코드를 읽고 어떤 프로그램인지 자세히 설명하라.

예제 #1:

request와 response에 대한 HTTP 서블릿 객체를 생성하고, FilterChain 객체 filterChain를 생성한다. filter.doFilter(request, response, filterChain)에서 request와 response에 대한 처리를 하게 된다. doFilter메소드를 호출하면 앞서 filterChain 객체를 생성할 때 만든 체인 안에서 request, response에 대한 처리를 수행하고 마지막 필터에선 리소스를 호출한다. assert부분은 response가 잘 처리되었는지 확인하는 테스팅부분이다

@Test

public void filterSendRedirect() throws Exception {

final MockHttpServletRequest request = new MockHttpServletRequest("GET", "/hotels");

MockHttpServletResponse response = new MockHttpServletResponse();

final byte[] responseBody = "Hello World".getBytes("UTF-8");

FilterChain filterChain = (filterRequest, filterResponse) -> {

assertEquals("Invalid request passed", request, filterRequest);

response.setContentLength(100);

FileCopyUtils.copy(responseBody, filterResponse.getOutputStream());

((HttpServletResponse) filterResponse).sendRedirect("http://www.google.com");

};

filter.doFilter(request, response, filterChain);

assertEquals("Invalid status", 302, response.getStatus());

assertNull("Invalid ETag header", response.getHeader("ETag"));

assertEquals("Invalid Content-Length header", 100, response.getContentLength());

assertArrayEquals("Invalid content", responseBody, response.getContentAsByteArray());

assertEquals("Invalid redirect URL", "http://www.google.com", response.getRedirectedUrl());

}

.

예제 #2:

오버라이드된 doFiter메소드는 request의 CharacterEncoding을 설정(select or set)하고, filterChain 객체의 doFilter메소드를 호출한다. doFilter메소드를 호출하면 파라미터의 chain 객체의 체인 안에서 request, response에 대한 처리를 수행하고 마지막 필터에선 리소스를 호출한다

/\*\*

\* Select and set (if specified) the character encoding to be used to

\* interpret request parameters for this request.

\*

\* @param request The servlet request we are processing

\* @param response The servlet response we are creating

\* @param chain The filter chain we are processing

\*

\* @exception IOException if an input/output error occurs

\* @exception ServletException if a servlet error occurs

\*/

@Override

public void doFilter(ServletRequest request, ServletResponse response,

FilterChain chain)

throws IOException, ServletException {

// Conditionally select and set the character encoding to be used

if (ignore || (request.getCharacterEncoding() == null)) {

String characterEncoding = selectEncoding(request);

if (characterEncoding != null) {

request.setCharacterEncoding(characterEncoding);

}

}

// Pass control on to the next filter

chain.doFilter(request, response);

}

예제 #3:

파라미터로 들어온 property가 허용되었다면 파라미터의 FilterChain객체 chain안에서 체인 안에서 request, response에 대한 처리를 수행하고 마지막 필터에선 리소스를 호출한다.

property가 허용되지 않았다면, response가 HttpServeltResponse인지 검사하고, Logger의 Debug가 enable하다면 Logger에서 sm.getString, getRequestURI, property에 대해 디버그를 수행한다. 그 후, denyStatus를 에러로 전송한다. 만약 response가 HttpServeletResponse가 아니라면 response가 Http가 아닌 경우의 에러를 전송한다

/\*\*

\* Perform the filtering that has been configured for this Filter, matching

\* against the specified request property.

\*

\* @param property The request property on which to filter

\* @param request The servlet request to be processed

\* @param response The servlet response to be processed

\* @param chain The filter chain

\*

\* @exception IOException if an input/output error occurs

\* @exception ServletException if a servlet error occurs

\*/

protected void process(String property, ServletRequest request,

ServletResponse response, FilterChain chain)

throws IOException, ServletException {

if (isAllowed(property)) {

chain.doFilter(request, response);

} else {

if (response instanceof HttpServletResponse) {

if (getLogger().isDebugEnabled()) {

getLogger().debug(sm.getString("requestFilter.deny",

((HttpServletRequest) request).getRequestURI(), property));

}

((HttpServletResponse) response).sendError(denyStatus);

} else {

sendErrorWhenNotHttp(response);

}

}

}

**제출 방법**

1. 문제 1, 2, 3은 word 파일에 답을 적어 파일을 제출한다.
2. 4월 28일 실습 시간 종료 20분 전 eClass에 업로드 하여야 한다.