서비스 서버 구축 방법론

[5팀] - 3차 과제

제출일	2024, 04, 21	**팀장**	김재형
과목	서비스서버구축방법론	팀원	권혁원
팀명	5팀 : 초록빛	**팀원**	정그린

개요

각 팀에서 선택한 서비스의 일부가 될 기능에 대해 동작 시키는 모듈 코드 작성하되, 디자인 패턴을 적용하여 구현. 어떤 의도에서의 작성인지, 왜 해당 디자인패턴으로 적용했는지 설명이 되어있어야 한다.

저희가 구상한 애플리케이션은 해외 여행을 계획하는 이들에게 비행기 티켓 예매부터 숙소예약, 음식점 추천, 여행지 추천까지 편리하게 제공하는 비대면 스마트 패키지 여행 가이드 애플리케이션입니다. 간편한 회원가입과 로그인을 제공하여 사용자들이 서비스를 빠르게 시작할 수 있습니다. 여행 상품 탐색 기능을 통해 사용자들은 직관적이고 편리한 검색으로 원하는 여행 상품을 쉽게 찾을 수 있으며, 상세한 정보를 확인할 수 있습니다. 예약 및 결제 프로세스는 간편하며, 여행 중에도 실시간으로 여행 정보를 제공하여 편리한 이용이 가능합니다. 여행 후기 및 평가를 통해 다양한 의견을 공유할 수 있고, 고객 지원 및 문의 서비스는 빠르고 신속한 처리를 통해 사용자들의 편의를 도모합니다. 또한, 개인화된 서비스와 편의성을 통해 사용자들에게 최적화된 경험을 제공합니다.

저희가 설계한 애플리케이션의 서비스를 모듈로 작성할 부분은 리뷰 및 평가 서비스, 실시간 상담 서비스, 항공/협력 업체 예매/예약 관리 서비스 총 3 분야로, 선정 이유는 애플리케이션을 담당하는 핵심 서비스들 위주로 선택하게 되었습니다.

참고 사이트

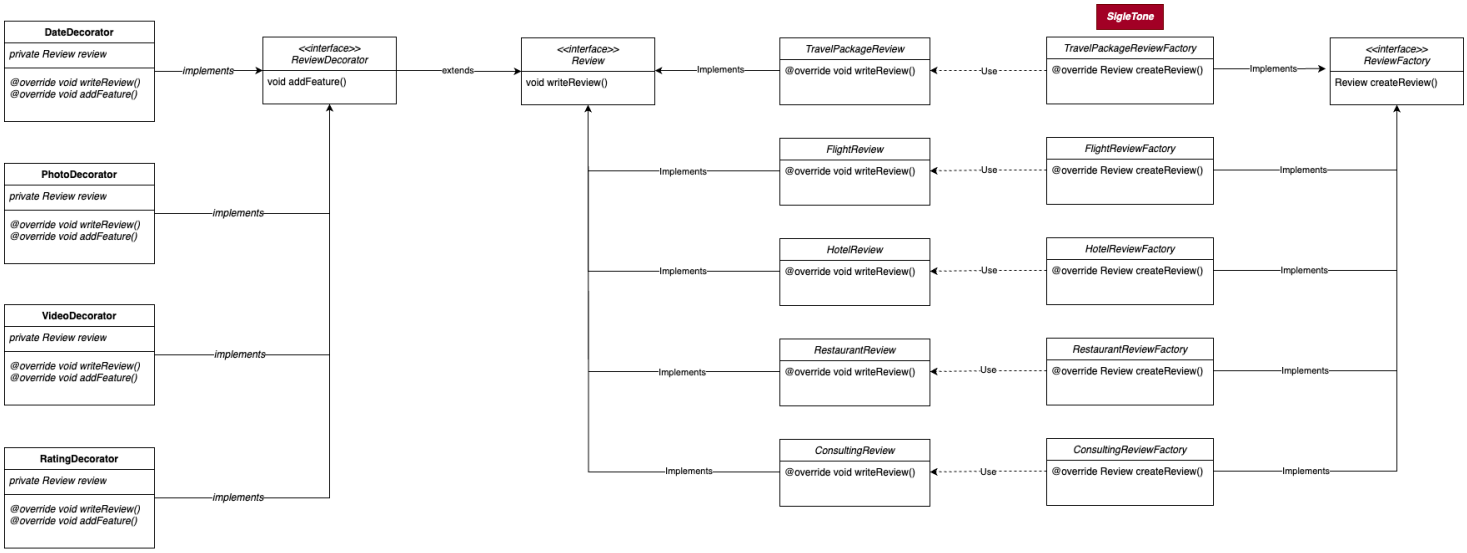
https://github.com/JNUGreenLight/JourneyBuddy

리뷰 및 평가 서비스

패턴 선택 근거

- 생성 디자인 패턴: `팩토리 패턴`, `싱글톤 패턴`
- 선택 이유: 팩토리 메서드 패턴은 객체 생성을 서브 클래스에게 위임하여 객체 생성 과정을 캡슐화하는 패턴입니다. 저희 'JourneyBuddy'에서의 리뷰 및 평가 서비스는 여러 유형의 리뷰나 평가를 생성해야 합니다. 팩토리 패턴을 활용하면 서브 클래스에서 실제로 어떤 유형의 리뷰나 평가를 생성할 지 결정할 수 있으며, 이는 유연성과 확장성을 제공하기 때문에 저희 서비스의 리뷰 및 평가에서 적합하다고 판단하였습니다. 또한 팩토리 클래스들을 Spring boot에서 자바 빈(Bean)으로 관리함으로써 싱글톤 패턴또한 적용하여 불필요한 메모리 누수를 방지하도록 설계하였습니다.
- 구조 디자인 패턴: `데코레이터 패턴`
- 선택 이유: 데코레이터 패턴은 객체의 추가적인 기능을 동적으로 확장할 수 있는 패턴입니다. 여행 리뷰 및 평가 서비스에서는 리뷰 및 평가 객체에 다양한 속성을(사진, 평점, 영상...) 추가해야 합니다. 데코레이터 패턴을 사용하면 이러한 요소들을 유연하게 확장할 수 있기 때문에 기존의 리뷰나 평가 객체를 수정하지 않고도 새로운 기능을 추가할 수 있으므로 코드의 변경을 최소화할 수 있다고 판단하여 데코레이터 패턴을 추가하였습니다.

클래스 다이어그램



파일별 모듈 코드

```java
//Review.java
package greenlight.jejunu.journeybuddy.review.domain;

public interface Review {
    void writeReview();
}
```

```java
//TravelReview.java
package greenlight.jejunu.journeybuddy.review.domain;

public class TravelReview implements Review{
    @Override
    public void writeReview(){
        System.out.println("여행 리뷰 작성");
    }
}
```

```java
//FlightReview.java
package greenlight.jejunu.journeybuddy.review.domain;

public class FlightReview implements Review {
    @Override
    public void writeReview() {
        System.out.println("항공 리뷰 작성");
    }
}
```

```java
//HotelReview.java
package greenlight.jejunu.journeybuddy.review.domain;

public class HotelReview implements Review {
    @Override
    public void writeReview() {
        System.out.println("숙박 리뷰 작성");
    }
}
```

```java
//RestaurantReview.java
package greenlight.jejunu.journeybuddy.review.domain;

public class RestaurantReview implements Review {
    @Override
    public void writeReview() {
        System.out.println("음식점 리뷰 작성");
    }
}
```

```java
//ConsultingReview.java
package greenlight.jejunu.journeybuddy.review.domain;

public class ConsultingReview implements Review {
    @Override
    public void writeReview() {
        System.out.println("상담 리뷰 작성");
    }
}
```

```java
//ReviewFactory.java
package greenlight.jejunu.journeybuddy.review.factory;

import greenlight.jejunu.journeybuddy.review.domain.Review;
import org.springframework.stereotype.Component;

@Component
public interface ReviewFactory {
    Review createReview();
}```

```java
//TravelReviewFactory.java
package greenlight.jejunu.journeybuddy.review.factory;

import greenlight.jejunu.journeybuddy.review.domain.Review;
import
greenlight.jejunu.journeybuddy.review.domain.TravelReview;
```
```

```
public class TravelReviewFactory implements ReviewFactory {
    @Override
    public Review createReview() {
        return new TravelReview();
    }
}
```

```
//FlightReviewFactory.java
package greenlight.jejunu.journeybuddy.review.factory;

import
greenlight.jejunu.journeybuddy.review.domain.FlightReview;
import greenlight.jejunu.journeybuddy.review.domain.Review;

public class FlightReviewFactory implements ReviewFactory {
    @Override
    public Review createReview() {
        return new FlightReview();
    }
}
```

```
//HotelReviewFactory.java
package greenlight.jejunu.journeybuddy.review.factory;

import
greenlight.jejunu.journeybuddy.review.domain.HotelReview;
import greenlight.jejunu.journeybuddy.review.domain.Review;

public class HotelReviewFactory implements ReviewFactory {
    @Override
    public Review createReview() {
        return new HotelReview();
    }
}
```

```
//RestaurantReviewFactory.java
package greenlight.jejunu.journeybuddy.review.factory;
```

```java
import
greenlight.jejunu.journeybuddy.review.domain.RestaurantReview;
import greenlight.jejunu.journeybuddy.review.domain.Review;

public class RestaurantReviewFactory implements ReviewFactory {
    @Override
    public Review createReview() {
        return new RestaurantReview();
    }
}
```

```java
//ConsultingReviewFactory.java
package greenlight.jejunu.journeybuddy.review.factory;

import
greenlight.jejunu.journeybuddy.review.domain.ConsultingReview;
import greenlight.jejunu.journeybuddy.review.domain.Review;


public class ConsultingReviewFactory implements ReviewFactory {
    @Override
    public Review createReview() {
        return new ConsultingReview();
    }
}
```

```java
//ReviewDecorator.java
package greenlight.jejunu.journeybuddy.review.decorator;

import greenlight.jejunu.journeybuddy.review.domain.Review;
import org.springframework.stereotype.Component;

@Component
public interface ReviewDecorator extends Review {
    Review addFeature(Review review);
}
```

```java
//DateDecorator.java
package greenlight.jejunu.journeybuddy.review.decorator;

import greenlight.jejunu.journeybuddy.review.domain.Review;

public class DateDecorator implements ReviewDecorator {
    private Review review;

    public DateDecorator(Review review) {
        this.review = review;
    }

    @Override
    public void writeReview() {
        review.writeReview();
    }

    @Override
    public Review addFeature(Review review) {
        System.out.println("날짜 선택 기능 추가");
        this.review = review;
        return review;
    }
}
```

```java
//PhotoDecorator.java
package greenlight.jejunu.journeybuddy.review.decorator;

import greenlight.jejunu.journeybuddy.review.domain.Review;

public class PhotoDecorator implements ReviewDecorator {
    private Review review;

    public PhotoDecorator(Review review) {
        this.review = review;
    }

    @Override
    public void writeReview() {
```

```
        review.writeReview();
    }

    @Override
    public Review addFeature(Review review) {
        System.out.println("사진 첨부 기능 추가");
        this.review = review;
        return review;
    }
}
```

```
//VideoDecorator.java
package greenlight.jejunu.journeybuddy.review.decorator;

import greenlight.jejunu.journeybuddy.review.domain.Review;

public class VideoDecorator implements ReviewDecorator {
    private Review review;

    public VideoDecorator(Review review) {
        this.review = review;
    }

    @Override
    public void writeReview() {
        review.writeReview();
    }

    @Override
    public Review addFeature(Review review) {
        System.out.println("영상 첨부 기능 추가");
        this.review = review;
        return review;
    }
}
```

```
//RatingDecorator.java
package greenlight.jejunu.journeybuddy.review.decorator;
```

```java
import greenlight.jejunu.journeybuddy.review.domain.Review;

public class RatingDecorator implements ReviewDecorator {
    private Review review;

    public RatingDecorator(Review review) {
        this.review = review;
    }

    @Override
    public void writeReview() {
        review.writeReview();
    }

    @Override
    public Review addFeature(Review review) {
        System.out.println("평점 부여 기능 추가");
        this.review = review;
        return review;
    }
}
```

```java
//ReviewService.java -> 패턴이 적용된 코드 사용예시
package greenlight.jejunu.journeybuddy.review;

import
greenlight.jejunu.journeybuddy.review.decorator.ReviewDecorator
;
import greenlight.jejunu.journeybuddy.review.domain.Review;
import
greenlight.jejunu.journeybuddy.review.factory.ReviewFactory;
import org.springframework.stereotype.Service;

@Service
public class ReviewService{
    private Review review;

    public ReviewService(ReviewFactory rf, ReviewDecorator rd){
        this.review = rf.createReview();
        this.review = rd.addFeature(review);
```

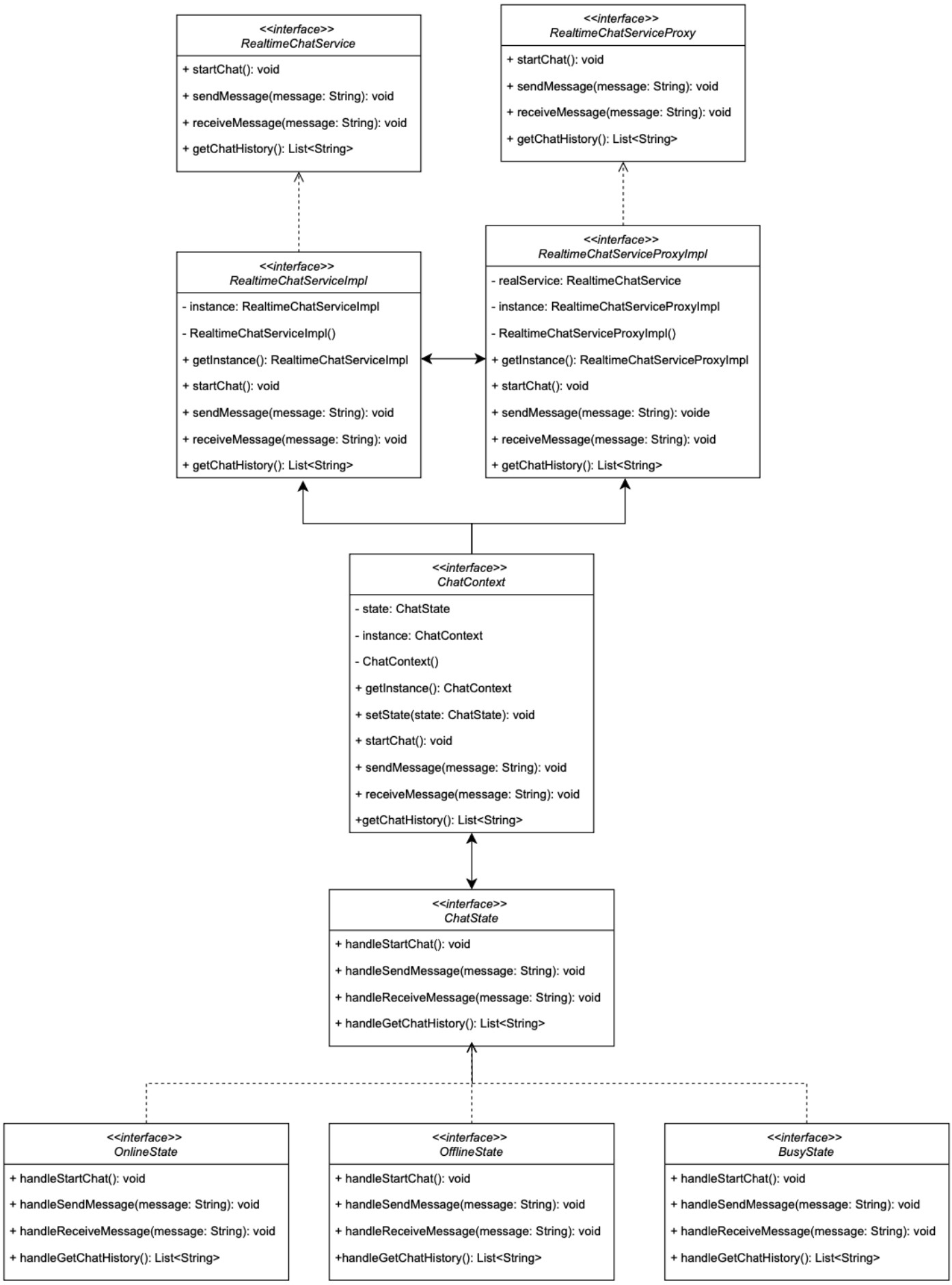
```
    }

    public void writeReviewWithPhoto(){
        review.writeReview();
    }
}
```

실시간 상담 서비스

패턴 선택 근거

- 생성 디자인 패턴: `싱글톤 패턴`
- 선택 이유: 여행 서비스에서 실시간 상담 서비스는 여러 사용자가 동시에 접근할 수 있지만, 별도의 인스턴스를 생성하는 것은 자원 낭비입니다. 따라서 싱글톤 패턴을 사용하여 하나의 인스턴스만을 유지함으로써 자원을 효율적으로 활용하려합니다. 이렇게 하면 어디서든 해당 클래스의 인스턴스에 접근할 수 있으며, 여행 서비스의 다양한 모듈이나 컴포넌트에서 편리하게 실시간 상담 서비스에 접근할 수 있습니다. 또한, 상태의 일관성을 유지하기 위해 싱글톤 패턴을 사용하여 여러 곳에서 동시에 상태를 변경하는 것을 방지하여 안정성과 신뢰성을 높일 수 있습니다.
- 구조 디자인 패턴: `프록시 패턴`
- 선택 이유: 서비스에서는 실시간 상담이라는 중요한 기능을 제공하고 있습니다. 이 기능은 여러 사용자가 동시에 접근하고 요청할 수 있으므로 성능 및 보안 측면에서 중요합니다. 프록시 패턴을 사용하면 실제 상담 서비스에 직접 접근하는 대신 중간에 프록시를 두어 요청을 처리할 수 있습니다. 이를 통해 클라이언트의 요청을 검증하고 보안을 강화할 수 있습니다. 또한 프록시를 사용하여 요청을 캐싱하거나 조작하여 서비스의 성능을 향상시킬 수 있습니다.
- 행동 디자인 패턴: `상태 패턴`
- 선택 이유: 실시간 상담 서비스에서는 사용자의 상태에 따라 동작이 달라집니다. 사용자가 온라인이면 메시지를 전송하고, 오프라인이면 저장해야 합니다. 상태 패턴은 이를 효과적으로 관리할 수 있어 선택하였습니다 . 각 상태를 클래스로 캡슐화하여 상태 전이와 관련된 로직을 내부에 구현하여 코드 수정을 최소화하고 새로운 상태를 쉽게 추가하거나 변경할 수 있습니다.

클래스 다이어그램



파일별 모듈 코드

```
// 싱글톤
public class RealtimeChatService {
```

```
    private static RealtimeChatService instance;
    private List<String> chatHistory;

    private RealtimeChatService() {
        chatHistory = new ArrayList<>();
    }

    public static RealtimeChatService getInstance() {
        if (instance == null) {
            instance = new RealtimeChatService();
        }
        return instance;
    }

    public void startChat() {
        // 채팅 시작
    }

    public void sendMessage(String message) {
        // 메시지 전송
    }

    public void receiveMessage(String message) {
        // 메시지 수신
    }

    public List<String> getChatHistory() {
        return chatHistory;
    }
}
```

```
//싱글톤
public class ChatController {
    private RealtimeChatService chatService;

    public ChatController() {
        chatService = RealtimeChatService.getInstance();
    }

    public void startChat() {
```

```
        chatService.startChat();
    }

    public void sendMessage(String message) {
        chatService.sendMessage(message);
    }

    public void receiveMessage(String message) {
        chatService.receiveMessage(message);
    }

    public void displayChatHistory() {
        // 채팅 기록 표시
    }
}
```

```
//싱글톤
public class RealtimeChatGUI {
    private ChatController chatController;

    public RealtimeChatGUI() {
        chatController = new ChatController();
    }

    public void displayChatWindow() {
        // 채팅 창 표시
    }

    public void displayChatHistory() {
        chatController.displayChatHistory();
    }

    public void sendMessage(String message) {
        chatController.sendMessage(message);
    }

    public void receiveMessage(String message) {
        chatController.receiveMessage(message);
```

```
    }
}
```

```
// 프록시
public interface RealtimeChatService {
    void startChat();
    void sendMessage(String message);
    void receiveMessage(String message);
    List<String> getChatHistory();
}
```

```
// 프록시
public class RealtimeChatServiceImpl implements
RealtimeChatService {
    @Override
    public void startChat() {
        // 채팅 시작 로직
    }

    @Override
    public void sendMessage(String message) {
        // 메시지 전송 로직
    }

    @Override
    public void receiveMessage(String message) {
        // 메시지 수신 로직
    }

    @Override
    public List<String> getChatHistory() {
        // 채팅 기록 반환 로직
        return null;
    }
}
```

```
//프록시
public interface RealtimeChatServiceProxy {
```

```
    void startChat();
    void sendMessage(String message);
    void receiveMessage(String message);
    List<String> getChatHistory();
}
```

```
//프록시
public class RealtimeChatServiceProxyImpl implements
RealtimeChatServiceProxy {
    private RealtimeChatService realService;

    @Override
    public void startChat() {
        // 프록시를 통한 채팅 시작 로직
    }

    @Override
    public void sendMessage(String message) {
        // 프록시를 통한 메시지 전송 로직
    }

    @Override
    public void receiveMessage(String message) {
        // 프록시를 통한 메시지 수신 로직
    }

    @Override
    public List<String> getChatHistory() {
        // 프록시를 통한 채팅 기록 반환 로직
        return null;
    }
}
```

```
//프록시
public class ChatController {
    private RealtimeChatServiceProxy chatService;

    public ChatController(RealtimeChatServiceProxy chatService)
{
```

```
        this.chatService = chatService;
    }

    public void startChat() {
        chatService.startChat();
    }

    public void sendMessage(String message) {
        chatService.sendMessage(message);
    }

    public void receiveMessage(String message) {
        chatService.receiveMessage(message);
    }

    public void displayChatHistory() {
        chatService.getChatHistory();
    }
}
```

```
//프록시
public class RealtimeChatGUI {
    private ChatController chatController;

    public RealtimeChatGUI(ChatController chatController) {
        this.chatController = chatController;
    }

    public void displayChatWindow() {
        // 채팅 창 표시 로직
    }

    public void displayChatHistory() {
        chatController.displayChatHistory();
    }

    public void sendMessage(String message) {
        chatController.sendMessage(message);
    }
```

```
    public void receiveMessage(String message) {
        chatController.receiveMessage(message);
    }
}
```

```
//상태 패턴
public interface ChatState {
    void handleStartChat();
    void handleSendMessage(String message);
    void handleReceiveMessage(String message);
    List<String> handleGetChatHistory();
}
```

```
//상태 패턴
public class OnlineState implements ChatState {
    @Override
    public void handleStartChat() {
        // 온라인 상태에서의 채팅 시작 동작
    }

    @Override
    public void handleSendMessage(String message) {
        // 온라인 상태에서의 메시지 전송 동작
    }

    @Override
    public void handleReceiveMessage(String message) {
        // 온라인 상태에서의 메시지 수신 동작
    }

    @Override
    public List<String> handleGetChatHistory() {
        // 온라인 상태에서의 채팅 기록 조회 동작
        return null;
    }
}
```

```java
//상태 패턴
public class OfflineState implements ChatState {
    @Override
    public void handleStartChat() {
        // 오프라인 상태에서의 채팅 시작 동작
    }

    @Override
    public void handleSendMessage(String message) {
        // 오프라인 상태에서의 메시지 전송 동작
        // (오프라인 상태에서는 메시지를 저장하여 나중에 전송)
    }

    @Override
    public void handleReceiveMessage(String message) {
        // 오프라인 상태에서의 메시지 수신 동작
        // (오프라인 상태에서는 메시지를 저장)
    }

    @Override
    public List<String> handleGetChatHistory() {
        // 오프라인 상태에서의 채팅 기록 조회 동작
        // (오프라인 상태에서는 저장된 채팅 기록 반환)
        return null;
    }
}
```

```java
//상태 패턴
public class BusyState implements ChatState {
    @Override
    public void handleStartChat() {
        // 바쁜 상태에서의 채팅 시작 동작
        // (바쁜 상태에서는 채팅을 시작할 수 없음)
    }

    @Override
    public void handleSendMessage(String message) {
        // 바쁜 상태에서의 메시지 전송 동작
```

```
        // (바쁜 상태에서는 메시지를 저장하여 나중에 전송)
    }

    @Override
    public void handleReceiveMessage(String message) {
        // 바쁜 상태에서의 메시지 수신 동작
        // (바쁜 상태에서는 메시지를 저장)
    }

    @Override
    public List<String> handleGetChatHistory() {
        // 바쁜 상태에서의 채팅 기록 조회 동작
        // (바쁜 상태에서는 저장된 채팅 기록 반환)
        return null;
    }
}
```

```
//상태 패턴
public class ChatContext {
    private ChatState state;

    public void setState(ChatState state) {
        this.state = state;
    }

    public void startChat() {
        state.handleStartChat();
    }

    public void sendMessage(String message) {
        state.handleSendMessage(message);
    }

    public void receiveMessage(String message) {
        state.handleReceiveMessage(message);
    }

    public List<String> getChatHistory() {
        return state.handleGetChatHistory();
```

```
    }
}
```

싱글톤 패턴 구조 설명

1. RealtimeChatService : 싱글톤 패턴을 적용한 클래스로, 실시간 채팅 서비스를 제공합니다. 메시지의 송수신과 채팅 기록 관리 등의 기능을 포함합니다.
2. ChatController : 채팅 서비스를 제어하는 클래스입니다. 실제로 사용자가 채팅을 시작하거나 메시지를 보내고 받는 등의 작업을 합니다. RealtimeChatService와 상호작용하여 채팅 기능을 수행합니다.
3. RealtimeChatGUI : 채팅 서비스를 시각적으로 표현하는 GUI 클래스입니다. 사용자 인터페이스를 관리하고 사용자가 채팅창을 표시하고 메시지를 보내고 받는 등의 작업을 합니다. 사용자와 ChatController를 연결하여 실제로 채팅 서비스를 사용할 수 있도록 합니다.

프록시 패턴 구조 설명

1. RealtimeChatservice : 실제 채팅 서비스를 정의하는 인터페이스입니다. 클라이언트가 사용할 수 있는 메서드를 포함하고 있습니다.
2. RealtimeChatServiceImpl : 실제 채팅 서비스를 구현하는 클래스입니다. RealtimeChatService 인터페이스를 구현하여 각 메서드를 실제로 구현합니다.
3. RealtimeChatServiceProxy : 프록시 패턴을 위한 인터페이스인 RealtimeChatServiceProxy를 정의합니다. 이 인터페이스는 실제 채팅 서비스와 동일한 메서드를 가지고 있습니다.
4. RealtimeChatServiceProxyImpl : 프록시 서비스를 구현하는 클래스입니다. 실제 채팅 서비스의 인스턴스를 가지고 있으며, 클라이언트 요청을 중개하여 실제 서비스에 전달합니다.
5. ChatController : 클라이언트와 프록시 서비스 간의 상호작용을 관리하는 컨트롤러 클래스입니다. 실제 서비스에 직접 접근하는 대신 프록시를 통해 통신합니다.
6. RealtimeChatGUI : 채팅 서비스를 시각적으로 표현하는 GUI 클래스입니다. 사용자 인터페이스를 관리하고 사용자가 채팅을 시각적으로 사용할 수 있습니다. 사용자와 ChatController를 연결하여 실제로 채팅 서비스를 사용할 수 있도록 설계하였습니다.

상태 패턴 구조 설명

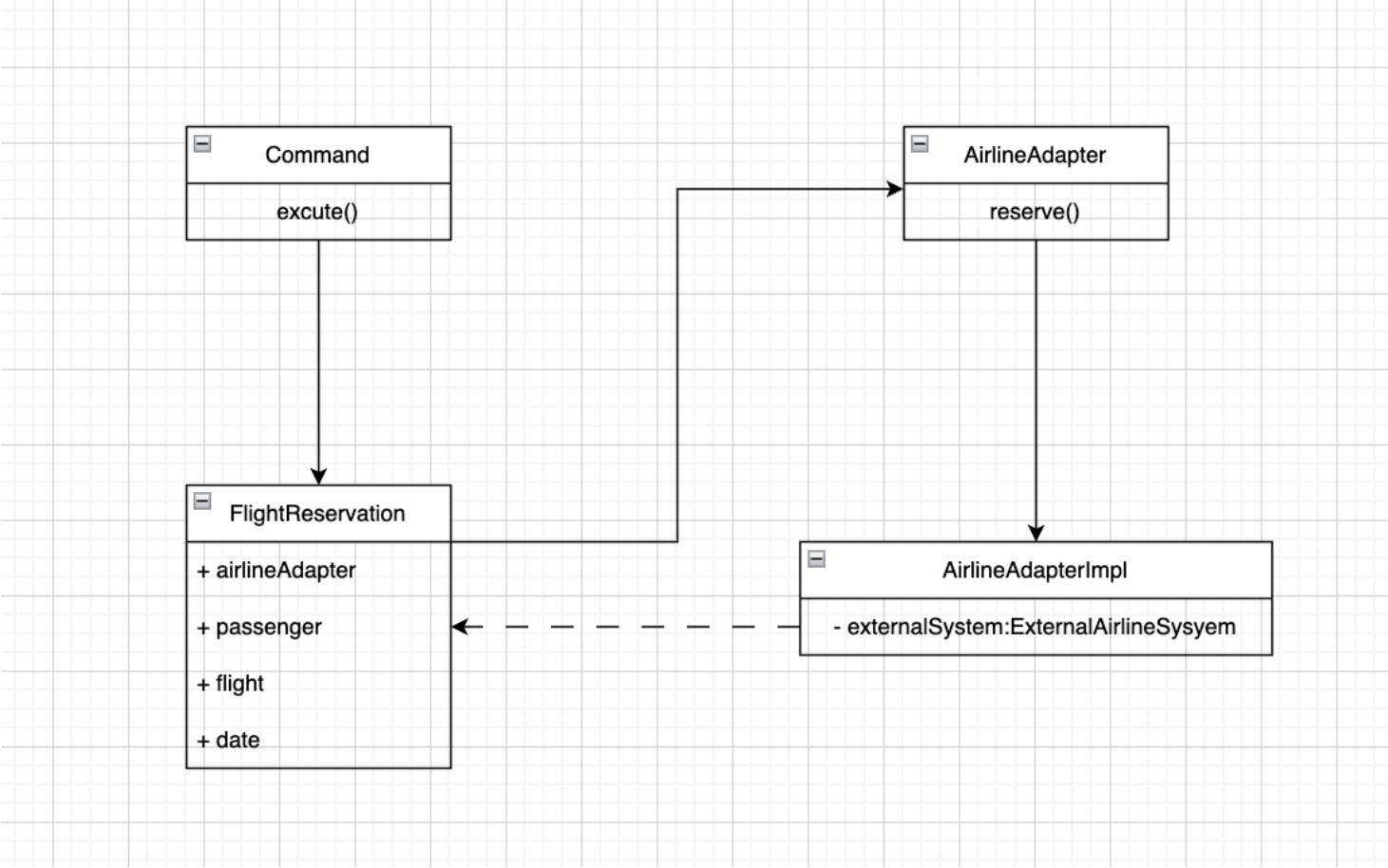
1. ChatState : 채팅 서비스의 상태를 정의하는 인터페이스입니다. 각 상태 클래스는 본 인터페이스를 구현하여 상태별 동작을 정의합니다.
2. Onlinestate: 온라인 상태를 나타내는 클래스입니다. ChatState 인터페이스를 구현하여 각 상태별 동작을 정의합니다.
3. OfflineState : 오프라인 상태를 나타내는 클래스입니다. ChatState 인터페이스를 구현하여 각 상태별 동작을 정의합니다.
4. BusyState : 바쁜 상태를 나타내는 클래스입니다. ChatState 인터페이스를 구현하여 각 상태별 동작을 정의합니다.
5. ChatContext : 채팅 서비스의 상태를 관리하는 클래스입니다. ChatState 인터페이스를 통해 상태별로 동작을 수행합니다.

항공/협력 업체 예매/예약 관리 서비스

패턴 선택 근거

- 구조 디자인 패턴: `어댑터 패턴`
- 선택 이유: 항공/협력 업체 예약/예약 관리 서비스를 구현할 때, 외부 항공사나 협력 업체의 예약 시스템과의 통합이 필요합니다. 그러나 외부 시스템과 서비스의 내부 구조가 호환되지 않을 수 있습니다. 어댑터 패턴을 사용하면 이러한 호환성 문제를 해결할 수 있습니다. 어댑터 클래스를 통해 외부 시스템과의 인터페이스를 서비스의 내부 구조에 맞게 변환하여 사용할 수 있습니다. 예약/예약 관리 서비스 내부에서는 일관된 인터페이스를 사용하여 외부 시스템과 통신할 수 있을 것으로 예상됩니다.
- 행위 디자인 패턴: `커맨더 패턴`
- 선택 이유: 사용자의 예약 생성 및 변경 요청을 객체로 캡슐화하고, 이를 나중에 실행하거나 취소할 수 있는 구조가 필요합니다. 커맨드 패턴을 사용하면 이러한 요청을 객체로 표현하여 관리할 수 있음으로 선택하게 되었습니다. 각각의 예약 요청은 커맨드 객체로 표현되어 예약 서비스 내부에서 실행되거나 취소될 수 있는 구조가 가능합니다. 예약 관리 서비스의 유연성을 높이고, 새로운 예약 관리 기능을 추가하거나 기존 기능을 변경할 때 코드를 수정하는 데 도움이 될 수 있을 것으로 예상하고 추가하였습니다.

클래스 다이어그램



파일별 모듈 코드

```
//외부 항공사 예약 시스템과의 통합을 위한 어댑터 인터페이스
public interface AirlineAdapter {
void reserve(String passenger, String flight, Date date);
}
```

```
//외부 항공사 예약 시스템과의 통합을 위한 어댑터 구현
public class AirlineAdapterImpl implements AirlineAdapter {
        private ExternalAirlineSystem externalSystem;

        public AirlineAdapterImpl(ExternalAirlineSystem
externalSystem) {
        this.externalSystem = externalSystem;
        }

        @Override
        public void reserve(String passenger, String flight,
Date date) {
        externalSystem.book(passenger, flight, date);
```

```
        }
}
```

```
// 예약 커맨드 인터페이스 정의
public interface ReservationCommand {
        void execute();
        }
```

```
// 예약 생성 커맨드
public class CreateReservationCommand implements
ReservationCommand {
    private ReservationService reservationService;
    private Reservation reservation;

    public CreateReservationCommand(ReservationService
reservationService, Reservation reservation) {
        this.reservationService = reservationService;
        this.reservation = reservation;
    }

    @Override
    public void execute() {
        reservationService.createReservation(reservation);
    }
}

// 예약 취소 커맨드
public class CancelReservationCommand implements
ReservationCommand {
    private ReservationService reservationService;
    private String reservationId;

    public CancelReservationCommand(ReservationService
reservationService, String reservationId) {
        this.reservationService = reservationService;
        this.reservationId = reservationId;
    }

    @Override
```

```
    public void execute() {
        reservationService.cancelReservation(reservationId);
    }
}
```

```
// 예약 서비스 커맨드 실행자 : Invoker
@Component
public class ReservationServiceInvoker {
    private Queue<ReservationCommand> commandQueue = new
LinkedList<>();

    // 커맨드를 큐에 추가
    public void addToQueue(ReservationCommand command) {
        commandQueue.add(command);
    }

    // 큐에서 커맨드를 꺼내 실행
    public void processCommands() {
        while (!commandQueue.isEmpty()) {
            ReservationCommand command = commandQueue.poll();
            command.execute();
        }
    }
}
```

```
// 커맨드를 사용하여 클라이언트 예약 생성 및 취소
@Service
public class ReservationClientService {
    @Autowired
    private ReservationServiceInvoker invoker;

    @Autowired
    private ReservationService reservationService;

    public void makeReservation(Reservation reservation) {
        ReservationCommand createCommand = new
CreateReservationCommand(reservationService, reservation);
```

```
        invoker.addToQueue(createCommand);
        invoker.processCommands();
    }

    public void cancelReservation(String reservationId) {
        ReservationCommand cancelCommand = new
CancelReservationCommand(reservationService, reservationId);
        invoker.addToQueue(cancelCommand);
        invoker.processCommands();
    }
}
```

어댑터 패턴 구조 설명

1. 타겟 인터페이스 (Target Interface):
   - `AirlineReservationAdapter` 인터페이스가 타겟 인터페이스 역할을 합니다. 이 인터페이스는 클라이언트가 사용할 예약 서비스의 메서드를 정의합니다. 여기서는 `reserveFlight` 메서드가 이에 해당합니다.
2. 어댑터 클래스 (Adapter Class):
   - `AirlineReservationAdapterImpl` 클래스가 어댑터 클래스 역할을 합니다. 이 클래스는 `AirlineReservationAdapter` 인터페이스를 구현하면서, 외부 항공사 예약 시스템과의 연동을 담당합니다.
   - 생성자를 통해 외부 시스템과의 연동을 위한 `ExternalAirlineReservationSystem` 객체를 받습니다.
3. 어댑티 (Adaptee):
   - `ExternalAirlineReservationSystem` 클래스가 어댑티 역할을 합니다. 이 클래스는 외부 항공사의 예약 시스템과 연동하여 항공편을 예약하는 기능을 제공합니다.
   - 어댑터 클래스의 `reserveFlight` 메서드에서는 실제로 어댑티 객체의 `bookFlight` 메서드를 호출하여 예약을 완료합니다.

커맨더 패턴 구조 설명

1. 예약 커맨드 인터페이스 정의:
   - `ReservationCommand`: 예약 관련 작업을 추상화한 인터페이스입니다. 이 인터페이스를 구현하는 클래스들은 예약 생성 또는 취소와 같은 특정 작업을 수행합니다.

2. 예약 생성 커맨드:
    - `CreateReservationCommand`: 예약 생성 작업을 수행하는 커맨드 클래스입니다. `ReservationService`를 통해 실제로 예약을 생성합니다.
3. 예약 취소 커맨드:
    - `CancelReservationCommand`: 예약 취소 작업을 수행하는 커맨드 클래스입니다. `ReservationService`를 통해 실제로 예약을 취소합니다.
4. 예약 서비스 커맨드 실행자(Invoker):
    - `ReservationServiceInvoker`: 커맨드를 관리하고 실행하는 역할을 수행하는 클래스입니다. 큐에 추가된 커맨드들을 꺼내어 실행합니다
5. 예약 클라이언트 서비스:
    - `ReservationClientService`: 예약 클라이언트 서비스 클래스입니다. 사용자가 예약 생성 또는 취소 요청을 하면 해당 작업을 커맨드로 변환하여 큐에 추가하고, `ReservationServiceInvoker`를 통해 실행합니다.