VIETNAM GENERAL CONFEDERATION OF LABOUR
TON DUC THANG UNIVERSITY
FACULTY OF INFORMATION TECHNOLOGY

NGUYEN HUYNH PHUONG TOAN – 521H0486
THAI NGUYEN THANH CONG – 521H0017
TRAN TRUNG KIEN – 519H0306

POS SYSTEM

FINAL REPORT
DESIGN PATTERN

HO CHI MINH CITY, YEAR 2024

VIETNAM GENERAL CONFEDERATION OF LABOUR
TON DUC THANG UNIVERSITY
FACULTY OF INFORMATION TECHNOLOGY

NGUYEN HUYNH PHUONG TOAN – 521H0486
THAI NGUYEN THANH CONG – 521H0017
TRAN TRUNG KIEN – 519H0306

POS SYSTEM

FINAL REPORT
DESIGN PATTERN

Under the supervision of
MSc.Post-Doctoral Fellow. Vu Dinh Hong

HO CHI MINH CITY, YEAR 2024

ACKNOWLEDGMENT

We would like to express our sincere gratitude to MSc.Post-Doctoral Fellow. Vu Dinh Hong, our esteemed instructor, for his invaluable guidance and expertise in the subject. Mr. Hong's dedication and insightful teachings during our classes have provided us with essential knowledge, laying the foundation for a deeper understanding of the subject matter. We are deeply appreciative of the knowledge imparted by Mr. Hong, which has greatly enriched our learning experience. Once again, we extend our heartfelt thanks to you, sir.

Ho Chi Minh City, 19 April 2024

Author

(Signature and full name)

Nguyễn Huỳnh Phương Toàn

Trần Trung Kiên

Thái Nguyễn Thành Công

DECLARATION OF AUTHORSHIP

The undersigned hereby declares that the thesis was carried out by oneself under the guidance and supervision of MSc.Post-Doctoral Fellow. Vu Dinh Hong; and that the work and the results contained in it are original and have not been submitted anywhere for any previous purposes. The data and figures presented in this thesis are for analysis, comments, and evaluations from various resources through one's own work and have been duly acknowledged in the reference part.

In addition, other comments, reviews and data used by other authors, and organizations have been acknowledged, and explicitly cited.

Full responsibility will be taken for any fraud detected in the thesis. Ton Duc Thang University is unrelated to any copyright infringement caused on my work (if any).

Ho Chi Minh City, 19 April 2024

Author

(signature and full name)

Nguyễn Huỳnh Phương Toàn

Trần Trung Kiên

Thái Nguyễn Thành Công

POS SYSTEM

ABSTRACT

In this report, we will present the design patterns applied to the old POS project from the previous course. For each applied pattern, we will present the reason for applying the pattern to the problem to be solved, provide class diagrams and source code applied for this pattern in the project. After applying the above patterns, the system has become easy to understand, easy to manage and easy to upgrade in the future.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABBREVIATIONS

POS	Point of Sale
CSV	Comma-Separated Values
XLSX	Excel Spreadsheet
SMTP	Simple Mail Transfer Protocol

CHAPTER 1. INTRODUCTION

1.1 Overview the project

The current POS system has management subsystems such as product management, employee management, customer management,... To facilitate system maintenance and upgrades when the demand for the system increases. The system needs to apply design patterns.

This project aims to enhance the existing POS system by integrating design patterns to improve its scalability, maintainability, and adaptability. By analyzing the current system architecture and identifying areas for improvement in product, employee, and customer management subsystems, the project will systematically implement selected design patterns to streamline system maintenance and accommodate future upgrades seamlessly. Through this approach, the enhanced POS system will be equipped to meet growing business demands while ensuring efficient and reliable operations.

1.2 Patterns in project

In this project, we applied 11 different design patterns to our website. Including 8 patterns we have learned in class and 3 patterns (Builder, Façade, Chain of Responsibility) that we have learned more about. You can see all applied patterns in Table 1.1 below.

Table 1.1: Patterns in project

Patterns	**Where to apply**
Builder	Apply to creating an *Account* , a complex object.
Strategy	Apply to changing *Payment Methods* at Runtime.
Factory Method	Apply to create specific *Payment Method*.

Patterns	**Where to apply**
Façade	Apply to simplify the complex Dashboard interface
Singleton	Apply to ensure there is only one unique instance for the *EmailSenderService* class throughout the entire program
Command	Apply to separating the email sending request (sender) from the request execution object: *JavaMailSenderImpl*, *MimeMessage* (receiver)
Template Method	Apply to creating templates for deleting a record in the database (*Account*, *Product* and *Customer*)
Decorator	Apply to additional decorations for exporting a csv file
Adapter	Apply to convert the Excel file export interface to match the CSV file export interface
Chain of Responsibility	Apply to sequential processing of steps to authenticate users at login
Observer	Apply to notify users every time a *Product's* information changes

CHAPTER 2. APPLY PATTERNS

2.1 Builder pattern

2.1.1 Reason for applying

Realize that the account object is a complex object with many properties. It is necessary to build many different versions, with the desire to separate the complex object construction process into many simple separate steps, each focusing on a single aspect, to easily change the way of initialization.

In summary, the reason for applying the Builder Pattern here is to achieve the following purposes:

- Ensure the Single Responsibility Principle by isolating complex Account construction code from the business logic of the product.
- Accounts can be constructed step-by-step, construction steps can be deferred, or steps can be run recursively.
- The same construction code can be reused when building various representations of accounts.

2.1.2 Class diagram

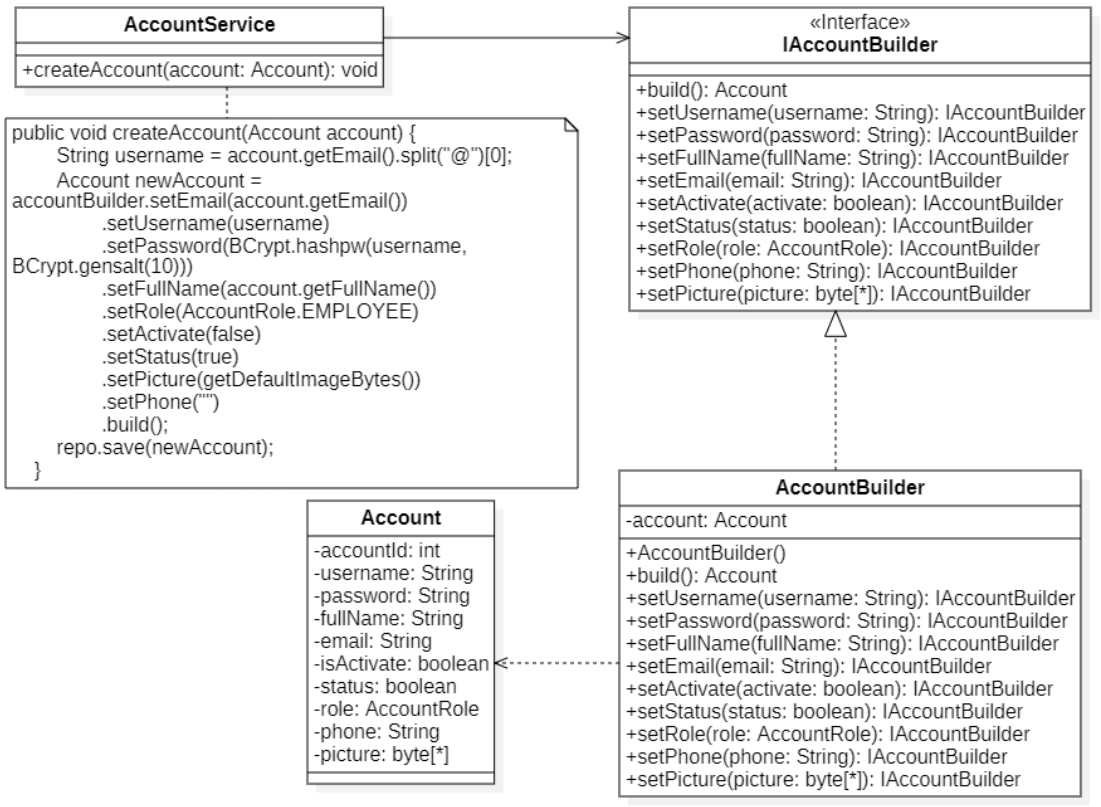


Figure 2.1: Builder pattern class diagram

2.1.3 Code implementation

```java
public interface IAccountBuilder {
    Account build();
    IAccountBuilder setUsername(String username);
    IAccountBuilder setPassword(String password);
    IAccountBuilder setFullName(String fullName);
    IAccountBuilder setEmail(String email);
    IAccountBuilder setActivate(boolean activate);
    IAccountBuilder setStatus(boolean status);
    IAccountBuilder setRole(AccountRole role);
    IAccountBuilder setPhone(String phone);
    IAccountBuilder setPicture(byte[] picture);
}
```

Figure 2.2: Code for IAccountBuilder interface

```java
public class AccountBuilder implements IAccountBuilder {
    private Account account;
    public AccountBuilder() {
        this.account = new Account();
    }
    @Override
    public Account build() {
        return this.account;
    }
    @Override
    public IAccountBuilder setUsername(String username) {
        this.account.setUsername(username);
        return this;
    }
    @Override
    public IAccountBuilder setPassword(String password) {
        this.account.setPassword(password);
        return this;
    }
    @Override
    public IAccountBuilder setFullName(String fullName) {
        this.account.setFullName(fullName);
        return this;
    }
    @Override
    public IAccountBuilder setEmail(String email) {
        this.account.setEmail(email);
        return this;
    }
    @Override
    public IAccountBuilder setActivate(boolean activate) {
        this.account.setActivate(activate);
        return this;
    }
    @Override
    public IAccountBuilder setStatus(boolean status) {
        this.account.setStatus(status);
        return this;
    }
    @Override
    public IAccountBuilder setRole(AccountRole role) {
        this.account.setRole(role);
        return this;
    }
    @Override
    public IAccountBuilder setPhone(String phone) {
        this.account.setPhone(phone);
        return this;
    }
    @Override
    public IAccountBuilder setPicture(byte[] picture) {
        this.account.setPicture(picture);
        return this;
    }
}
```

Figure 2.3: Code for AccountBuilder class

```java
@Entity
@Data
@AllArgsConstructor
@NoArgsConstructor
@ToString
public class Account {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private int accountId;
    private String username;
    private String password;
    private String fullName;
    private String email;
    private boolean isActivate;
    private boolean status;
    @Enumerated(EnumType.STRING)
    private AccountRole role;
    private String phone;
    @Lob
    @Column(columnDefinition = "LONGBLOB")
    private byte[] picture;
}
```

Figure 2.4: Code for Account class

```java
public class AccountService {
    // remaining attributes ...

    @Autowired
    private IAccountBuilder accountBuilder;

    public void createAccount(Account account) {
        String username = account.getEmail().split("@")[0];

        Account newAccount = accountBuilder.setEmail(account.getEmail())
                .setUsername(username)
                .setPassword(BCrypt.hashpw(username, BCrypt.gensalt(10)))
                .setFullName(account.getFullName())
                .setRole(AccountRole.EMPLOYEE)
                .setActivate(false)
                .setStatus(true)
                .setPicture(getDefaultImageBytes())
                .setPhone("")
                .build();
        repo.save(newAccount);
    }

    // remaining methods ...
}
```

Figure 2.5: Code for using AccountBuilder to create a new account

2.2 Strategy pattern

2.2.1 Reason for applying

Initially our POS only supports cash payment methods. Due to increasing customer demand, customers want to have more options for payment methods that the system can support. We have developed and expanded to integrate new payment methods such as payment through VNPay, MoMo, Paypal. Currently, the aim is to make the system easier to maintain and upgrade in the future as well as allowing the system to change the payment method at Runtime.

In summary, the reason for applying the Strategy Pattern here is to achieve the following purposes:

- Algorithms for specific Payment method can be swapped at runtime.

- The implementation details of a Payment method can be isolated from the code that uses it.
- Inheritance can be replaced with composition.
- Ensure the Open/Closed Principle by allowing new payment methods to be introduced without having to change the context.

2.2.2 Class diagram

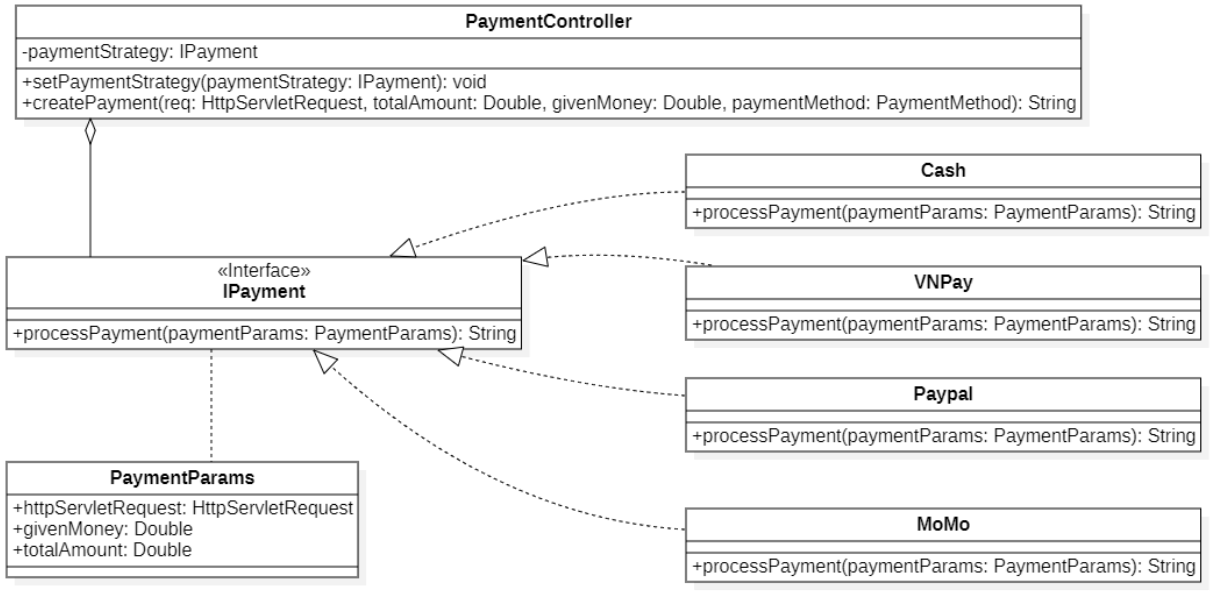


Figure 2.6: Strategy pattern class diagram

2.2.3 Code implementation

```
1  public interface IPayment {
2      public String processPayment(PaymentParams params) throws Exception;
3  }
```

Figure 2.7: Code for Ipayment interface

```java
@Getter
@Setter
public class PaymentParams {
    public HttpServletRequest httpServletRequest;
    public Double givenMoney;
    public Double totalAmount;
}
```

Figure 2.8: Code for PaymentParams class

```java
public class Cash implements IPayment{
    @Override
    public String processPayment(PaymentParams params) {
        return "POS/step3";
    }
}
```

Figure 2.9: Code for Cash class

```java
public class VNPay implements IPayment{
    @Override
    public String processPayment(PaymentParams params) {
        long amount = (long) (params.getTotalAmount()*23000.0*100);
        String vnp_TxnRef = VNPayConfig.getRandomNumber(8);

        Map<String, String> vnp_Params = new HashMap<>();
        vnp_Params.put("vnp_Version", VNPayConfig.vnp_Version);
        vnp_Params.put("vnp_Command", VNPayConfig.vnp_Command);
        vnp_Params.put("vnp_TmnCode", VNPayConfig.vnp_TmnCode);
        vnp_Params.put("vnp_Amount", String.valueOf(amount));
        vnp_Params.put("vnp_CurrCode", "VND");
        vnp_Params.put("vnp_TxnRef", vnp_TxnRef);
        vnp_Params.put("vnp_OrderInfo", "Payment order:" + vnp_TxnRef);
        vnp_Params.put("vnp_OrderType", "other");
        vnp_Params.put("vnp_Locale", "us");
        vnp_Params.put("vnp_ReturnUrl", VNPayConfig.vnp_ReturnUrl);
        vnp_Params.put("vnp_IpAddr", VNPayConfig.getIpAddress(params.getHttpServletRequest()));

        Calendar cld = Calendar.getInstance(TimeZone.getTimeZone("Etc/GMT+7"));
        SimpleDateFormat formatter = new SimpleDateFormat("yyyyMMddHHmmss");
        String vnp_CreateDate = formatter.format(cld.getTime());
        vnp_Params.put("vnp_CreateDate", vnp_CreateDate);

        cld.add(Calendar.MINUTE, 15);
        String vnp_ExpireDate = formatter.format(cld.getTime());
        vnp_Params.put("vnp_ExpireDate", vnp_ExpireDate);

        List fieldNames = new ArrayList(vnp_Params.keySet());
        Collections.sort(fieldNames);
        StringBuilder hashData = new StringBuilder();
        StringBuilder query = new StringBuilder();
        Iterator itr = fieldNames.iterator();
        while (itr.hasNext()) {
            String fieldName = (String) itr.next();
            String fieldValue = (String) vnp_Params.get(fieldName);
            if ((fieldValue != null) && (fieldValue.length() > 0)) {
                try {
                    hashData.append(fieldName);
                    hashData.append('=');
                    hashData.append(URLEncoder.encode(fieldValue, StandardCharsets.US_ASCII.toString()));
                    query.append(URLEncoder.encode(fieldName, StandardCharsets.US_ASCII.toString()));
                    query.append('=');
                    query.append(URLEncoder.encode(fieldValue, StandardCharsets.US_ASCII.toString()));
                    if (itr.hasNext()) {
                        query.append('&');
                        hashData.append('&');
                    }
                } catch (UnsupportedEncodingException e) {
                    throw new RuntimeException(e);
                }
            }
        }
        String queryUrl = query.toString();
        String vnp_SecureHash = VNPayConfig.hmacSHA512(VNPayConfig.secretKey, hashData.toString());
        queryUrl += "&vnp_SecureHash=" + vnp_SecureHash;
        String paymentUrl = VNPayConfig.vnp_PayUrl + "?" + queryUrl;
        return paymentUrl;
    }
}
```

Figure 2.10: Code for VNPay class

```java
public class Paypal implements IPayment{
    @Override
    public String processPayment(PaymentParams params) {
        try {
            Payment payment = new PaypalService()
                        .createPayment(params.getTotalAmount());
            for(Links link: payment.getLinks()) {
                if(link.getRel().equals("approval_url")) {
                    return link.getHref();
                }
            }
        } catch (PayPalRESTException e) {
            e.printStackTrace();
        }
        return "redirect:/";
    }
}
```

Figure 2.11: Code for Paypal class

```java
public class MoMo implements IPayment{
    @Override
    public String processPayment(PaymentParams params){
        LogUtils.init();
        String requestId = String.valueOf(System.currentTimeMillis());
        String orderId = String.valueOf(System.currentTimeMillis());
        Long amount = (Long) (params.getTotalAmount() * 23000); // convert to USD

        String orderInfo = "Pay With MoMo";
        String returnURL = "http://localhost:8888/cart/complete";
        String notifyURL = "http://localhost:8888/cart/step-2";
        Environment environment = Environment.selectEnv("dev");

        PaymentResponse captureWalletMoMoResponse = null;
        try {
            captureWalletMoMoResponse = CreateOrderMoMo
                    .process(environment, orderId, requestId, Long.toString(amount),
                            orderInfo, returnURL, notifyURL, "", RequestType.PAY_WITH_METHOD, Boolean.TRUE);
            return captureWalletMoMoResponse.getPayUrl();
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
}
```

Figure 2.12: Code for MoMo class

```java
@RestController
@RequestMapping("/api/payment")
public class PaymentController {
    private IPayment paymentStrategy;

    public void setPaymentStrategy(IPayment paymentStrategy) {
        this.paymentStrategy = paymentStrategy;
    }

    @GetMapping("/create_payment")
    public String createPayment(
            HttpServletRequest req, Double totalAmount, Double givenMoney, PaymentMethod paymentMethod)
            throws Exception {
        PaymentParams params = new PaymentParams();
        params.setHttpServletRequest(req);
        params.setTotalAmount(totalAmount);
        params.setGivenMoney(givenMoney);

        setPaymentStrategy(new SimplePaymentFactory().createPayment(paymentMethod));
        return paymentStrategy.processPayment(params);
    }
}
```

Figure 2.13: Code for PaymentController

2.3 Factory Method pattern

2.3.1 Reason for applying

After expanding the POS System with many new payment methods, users now have more choices. To facilitate future management and expansion, the need now is to separate the payment method initialization part from the main business logic class. This makes it possible to expand new subclasses without affecting affects existing code, allowing for easier extension than direct initialization, helping to increase the flexibility and scalability of the system.

In summary, the reason for applying the Factory Method Pattern here is to achieve the following purposes:

- Ensure the Single Responsibility Principle: The Payment method creation code can be moved into one place in the program, making the code easier to maintain and upgrade.
- Ensure the Open/Closed Principle: New types of Payment method can be introduced into the program without breaking existing code.

2.3.2 Class diagram

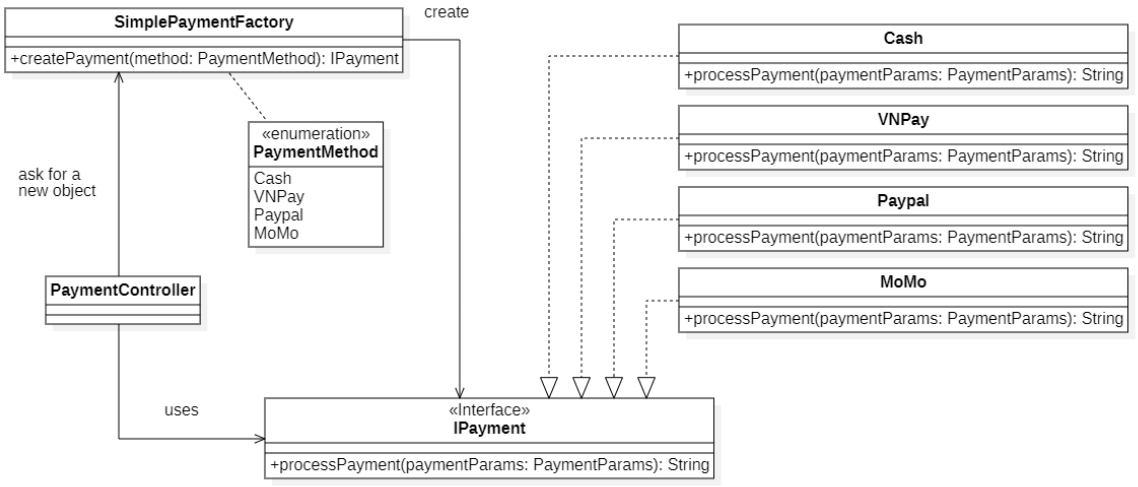
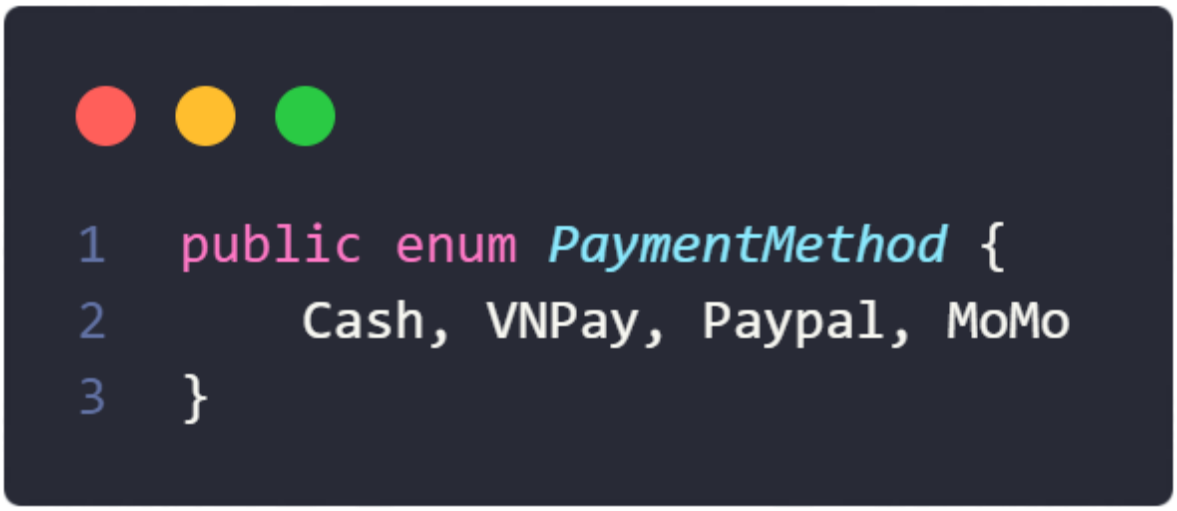


Figure 2.14: Factory Method class diagram

2.3.3 Code implementation

```java
public class SimplePaymentFactory {
    public IPayment createPayment(PaymentMethod method) {
        switch (method) {
            case Cash:
                return new Cash();
            case VNPay:
                return new VNPay();
            case Paypal:
                return new Paypal();
            case MoMo:
                return new MoMo();
            default:
                return null;
        }
    }
}
```

Figure 2.15: Code for SimplePaymentFactory class



```
1   public enum PaymentMethod {
2       Cash, VNPay, Paypal, MoMo
3   }
```

Figure 2.16: Enum type for PaymentMethod

```
@GetMapping("/create_payment")
    public String createPayment(
            HttpServletRequest req, Double totalAmount, Double givenMoney, PaymentMethod paymentMethod)
            throws Exception {

        PaymentParams params = new PaymentParams();
        params.setHttpServletRequest(req);
        params.setTotalAmount(totalAmount);
        params.setGivenMoney(givenMoney);

        // Use SimplePaymentFactory to create specific Payment method
        setPaymentStrategy(new SimplePaymentFactory().createPayment(paymentMethod));

        return paymentStrategy.processPayment(params);
    }
```

Figure 2.17: Code for using SimplePaymentFactory to get object

The Code for Cash, VNPay, Paypal, MoMo classes and intefcade IPayment are the same with Figure 2.9, Figure 2.10, Figure 2.11, Figure 2.12, Figure 2.7

2.4 Façade pattern

2.4.1 Reason for applying

The original POS system had a very complex dashboard page, requiring many subsystems to perform calculations such as *AccountService, ImageService, DashboardService, CustomerService, WebUtils*. The code currently is very difficult to manage because there are too many dependencies on subsytems. With the desire

to help users have a simpler interface to use, the system needs to hide the complex details inside by creating a *DashboardFacade* class to integrate services together.

In summary, the reason for applying the Façade Pattern here is to achieve the following purposes:

- Provides *DashboardFacade* a simple interface to complex subsystems.
- Separate your source code from the complexity of the subsystem to increase independence and mobility, reduce dependence.

2.4.2 Class diagram

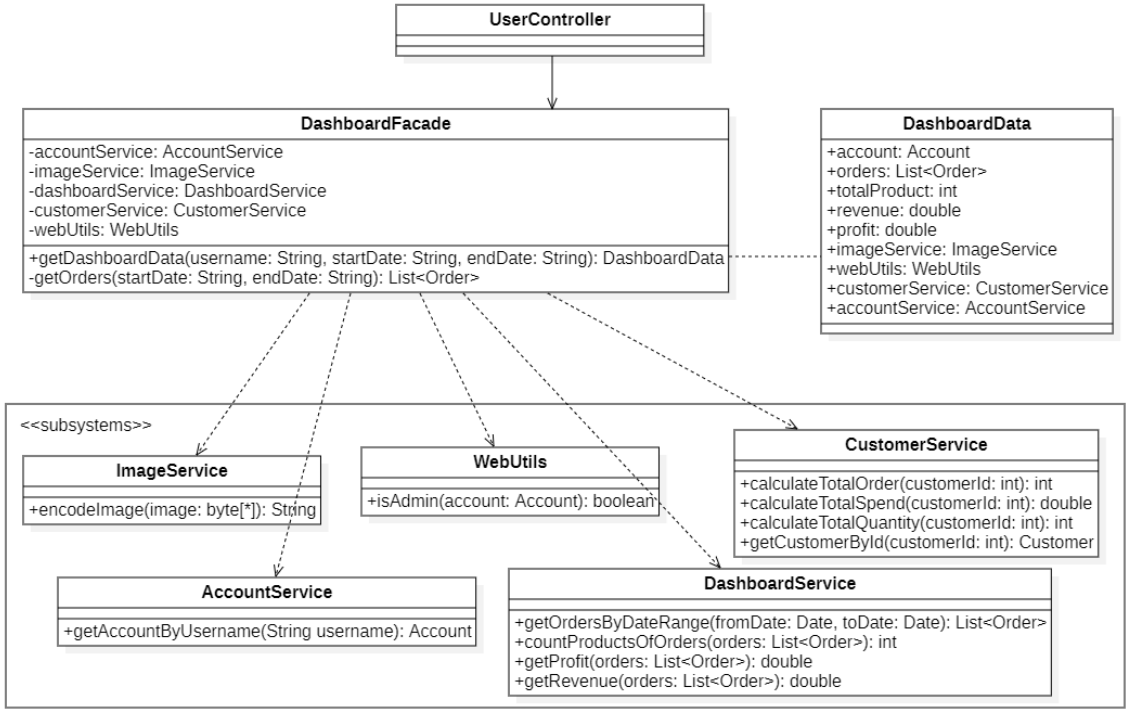


Figure 2.18: Façade pattern class diagram

2.4.3 Code implementation

```java
@Service
public class ImageService {

    public static String encodeImage(byte[] image) {
        return Base64.getEncoder().encodeToString(image);
    }
}
```

Figure 2.19: Code for ImageService class

```java
@Service
@AllArgsConstructor
public class AccountService {
    @Autowired
    private AccountRepository repo;

    public Account getAccountByUsername(String username) {
        return repo.findByUsername(username);
    }
}
```

Figure 2.20: Code for AccountService class

```java
@Component
public class WebUtils {
    public static boolean isAdmin(Account account) {
        return account.getRole() == AccountRole.ADMIN;
    }
}
```

Figure 2.21: Code for WebUtils class

```java
@Service
public class CustomerService {
    @Autowired
    private OrderRepository orderRepository;
    @Autowired
    private OrderDetailRepository orderDetailRepository;
    @Autowired
    private CustomerRepository customerRepository;

    public int calculateTotalOrder(int customerId) {
        return orderRepository.countAllByCustomerId(customerId);
    }

    public double calculateTotalSpend(int customerId) {
        List<Order> orders = orderRepository.findAllByCustomerId(customerId);
        double total = 0;
        for(Order order: orders) {
            total += order.getTotalAmount();
        }
        return  total;
    }

    public int calculateTotalQuantity(int customerId) {
        List<Order> orders = orderRepository.findAllByCustomerId(customerId);
        int total = 0;
        for(Order order: orders) {
            List<OrderDetail> orderDetails = orderDetailRepository.findAllByOrderId(order.getOrderId());
            for(OrderDetail detail: orderDetails) {
                total += detail.getQuantity();
            }
        }
        return  total;
    }

    public Customer getCustomerById(int customerId) {
        return customerRepository.findByCustomerId(customerId);
    }
}
```

Figure 2.22: Code for CustomerService class

```java
@Service
public class DashboardService {
    @Autowired
    OrderRepository orderRepository;
    @Autowired
    OrderDetailRepository orderDetailRepository;
    @Autowired
    ProductRepository productRepository;

    public List<Order> getOrdersByDateRange(Date fromDate, Date toDate) {
        LocalDate today = LocalDate.now();
        Date from, to = null;
        if (isToday(fromDate, toDate)) {
            from = Date.from(today.atStartOfDay(ZoneId.systemDefault()).toInstant());
            to = Date.from(today.plusDays(1).atStartOfDay(ZoneId.systemDefault()).toInstant());
        } else if (isYesterday(fromDate, toDate)) {
            to = Date.from(today.atStartOfDay(ZoneId.systemDefault()).toInstant());
            from = Date.from(today.minusDays(1).atStartOfDay(ZoneId.systemDefault()).toInstant());
        } else if (isLast7Days(fromDate, toDate)) {
            to = Date.from(today.atStartOfDay(ZoneId.systemDefault()).toInstant());
            from = Date.from(today.minusDays(6).atStartOfDay(ZoneId.systemDefault()).toInstant());
        } else if (isThisMonth(fromDate, toDate)) {
            LocalDate firstDayOfMonth = today.withDayOfMonth(1);
            from = Date.from(firstDayOfMonth.atStartOfDay(ZoneId.systemDefault()).toInstant());
            LocalDate firstDayOfNextMonth = today.plusMonths(1).withDayOfMonth(1);
            to = Date.from(firstDayOfNextMonth.atStartOfDay(ZoneId.systemDefault()).toInstant());
        } else {
            return orderRepository.findByOrderDateBetween(fromDate, toDate);
        }
        return orderRepository.findByOrderDateBetween(from, to);
    }

    public double getProfit(List<Order> orders) {
        List<Double>  revenues = getProfitOfOrders(orders);
        double total = 0;
        for(Double revenue: revenues) {
            total += revenue;
        }
        return total;
    }

    public int countProductsOfOrders(List<Order> orders) {
        int count = 0;
        for (Order order: orders) {
            List<OrderDetail> details = orderDetailRepository.findAllByOrderId(order.getOrderId());
            for (OrderDetail detail: details) {
                count += detail.getQuantity();
            }
        }
        return count;
    }

    public double getRevenue(List<Order> orders) {
        double revenue = 0;
        for (Order order: orders) {
            revenue += order.getTotalAmount();
        }
        return revenue;
    }
}
```

Figure 2.23: Code for DashboardService class

```java
@Component
public class DashboardFacade {
    private AccountService accountService;
    @Autowired
    private ImageService imageService;
    @Autowired
    private DashboardService dashboardService;
    @Autowired
    private CustomerService customerService;
    @Autowired
    private WebUtils webUtils;

    public DashboardData getDashboardData(String username, String startDate, String endDate) {
        Account account = accountService.getAccountByUsername(username);
        List<Order> orders = getOrders(startDate, endDate);
        int totalProduct = dashboardService.countProductsOfOrders(orders);
        double revenue = dashboardService.getRevenue(orders);
        double profit = dashboardService.getProfit(orders);

        return new DashboardData(account, orders, totalProduct, revenue, profit,
                imageService, webUtils, customerService, accountService);
    }

    private List<Order> getOrders(String startDate, String endDate) {
        List<Order> orders = new ArrayList<>();
        if (startDate != null && endDate != null) {
            SimpleDateFormat dateFormat = new SimpleDateFormat("MMMM/dd/yyyy");
            try {
                Date parsedStartDate = (!startDate.isEmpty()) ? dateFormat.parse(startDate) : null;
                Date parsedEndDate = (!endDate.isEmpty()) ? dateFormat.parse(endDate) : null;

                orders = dashboardService.getOrdersByDateRange(parsedStartDate, parsedEndDate);
            } catch (ParseException e) {
                e.printStackTrace();
            }
        } else {
            // Today
            LocalDate today = LocalDate.now();
            Date fromDate = Date.from(today.atStartOfDay(ZoneId.systemDefault()).toInstant());
            Date toDate = Date.from(today.plusDays(1).atStartOfDay(ZoneId.systemDefault()).toInstant());

            orders = dashboardService.getOrdersByDateRange(fromDate, toDate);
        }
        return orders;
    }
}

```

Figure 2.24: Code for DashboardFacade class

```java
public class DashboardData {
    public Account account;
    public List<Order> orders;
    public int totalProduct;
    public double revenue;
    public double profit;
    public ImageService imageService;
    public WebUtils webUtils;
    public CustomerService customerService;
    public AccountService accountService;
}
```

Figure 2.25: Code for DashboardData class

```java
@Controller
@RequestMapping("/user")
@AllArgsConstructor
public class UserController {
    @Autowired
    private DashboardFacade dashboardFacade;

    @GetMapping("/dashboard")
    public String dashboard(HttpSession session, Model model,
                            @RequestParam(name = "startDate", required = false) String startDate,
                            @RequestParam(name = "endDate", required = false) String endDate) {

        String username = session.getAttribute("username").toString();
        DashboardData dashboardData = dashboardFacade.getDashboardData(username, startDate, endDate);
        model.addAttribute("dashboardData", dashboardData);

        return "Dashboard/dashboard";
    }
}
```

Figure 2.26: Code for using DashboardFacade in UserController class

2.5 Singleton pattern

2.5.1 Reason for applying

In our system, there is a class that uses the *EmailSenderService* class to perform the service of sending mail to employees' emails. To avoid resource conflicts when using *EmailSenderService*, we decided to apply Singleton pattern to solve that problem.

In summary, the reason for applying the Singleton Pattern here is to achieve the following purposes:

- In the *EmailSenderService* class, when sending emails, resources are used to connect to SMTP. Applying Singleton helps limit unnecessary connections, thus affecting the system's performance.
- Creating a Singleton class helps manage SMTP connection resources effectively. When *EmailSenderService* is needed, we can simply call the getInstance() method to retrieve the pre-initialized objects, rather than creating new ones for each use, which makes management easier.
- When executing the application with multiple threads simultaneously creating *EmailSenderService* objects, resource conflicts may occur. By using Singleton, we ensure that only one object is created and used, thus avoiding resource conflicts.

2.5.2 Class diagram

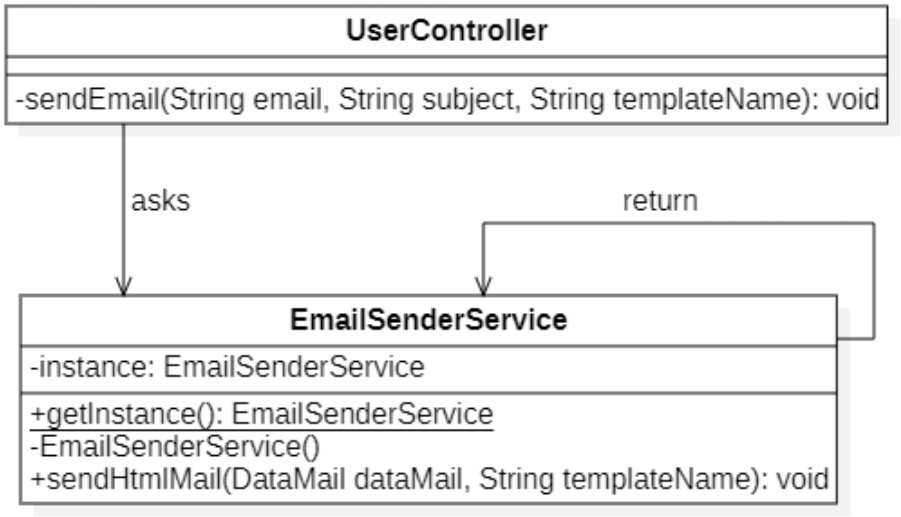


Figure 2.27: Singleton pattern class diagram

2.5.3 Code implementation

```java
@Service
public class EmailSenderService implements MailService{
    private EmailSenderService() {
        this.mailSender = new JavaMailSenderImpl();
        mailSender.setHost("smtp.gmail.com");
        mailSender.setPort(587);
        mailSender.setUsername("gtvprimeofficial@gmail.com");
        mailSender.setPassword("jihfblsarmerterk");

        java.util.Properties props = mailSender.getJavaMailProperties();
        props.put("mail.smtp.auth", true);
        props.put("mail.smtp.starttls.enable", true);
    }

    private static EmailSenderService instance = new EmailSenderService();

    public static EmailSenderService getInstance() {
        return instance;
    }

    private final JavaMailSenderImpl mailSender;

    @Override
    public void sendHtmlMail(DataMail dataMail, String templateName) throws MessagingException {
        MimeMessage message = mailSender.createMimeMessage();

        MimeMessageHelper helper = new MimeMessageHelper(message, true, "utf-8");

        Context context = new Context();
        context.setVariables(dataMail.getProps());

        SpringTemplateEngine templateEngine = getSpringTemplateEngine();
        String html = templateEngine.process(templateName, context);

        helper.setTo(dataMail.getTo());
        helper.setSubject(dataMail.getSubject());
        helper.setText(html, true);

        ICommand emailCommand = new EmailCommand(mailSender, message);
        emailCommand.execute();
    }

    public SpringTemplateEngine getSpringTemplateEngine() {
        SpringTemplateEngine templateEngine = new SpringTemplateEngine();
        ClassLoaderTemplateResolver templateResolver = new ClassLoaderTemplateResolver();

        templateResolver.setPrefix("templates/");
        templateResolver.setSuffix(".html");
        templateResolver.setTemplateMode("HTML");
        templateResolver.setCharacterEncoding("UTF-8");
        templateEngine.setTemplateResolver(templateResolver);

        return templateEngine;
    }
}
```

Figure 2.28: Code for EmailSenderService

```java
private void sendEmail(String email, String subject, String templateName) {
    String defaultUsername = email.split("@")[0];
    try {
        DataMail dataMail = new DataMail();
        dataMail.setTo(email);
        dataMail.setSubject(subject);
        Map<String, Object> props = new HashMap<>();
        props.put("username", defaultUsername);

        String token = jwt.createToken(defaultUsername);
        String confirmationLink = "http://localhost:8888/confirm?token=" + token;
        props.put("link", confirmationLink);
        dataMail.setProps(props);

        //Get Instance of singleton
        EmailSenderService.getInstance().sendHtmlMail(dataMail, templateName);

    } catch (MessagingException exp){
        exp.printStackTrace();
    }

}
```

Figure 2.29: Code for using singleton in UserController

2.6 Command pattern

2.6.1 Reason for applying

Notice that *JavaMailSenderImpl* is an object whose source code cannot be changed, because it is an object of Springboot's Java Mail library. With the desire to separate the part that executes the email request and the part that sends the request. We decided to use the Command pattern to apply it.

In summary, the reason for applying the Command Pattern here is to achieve the following purposes:

- The Command Pattern allows us to separate the functionality of sending emails into a separate object, *EmailCommand*, without directly depending on the *EmailSenderService*. This reduces the complexity of *EmailSenderService* by dividing the sending email functionality into a separate part, making it easier to reuse and maintain.

- Due to the separation between calling and execution, we can easily extend the application by adding new command classes without affecting the existing code. For example, if we want to add another type of notification besides email, we can simply implement a new command class without modifying *EmailSenderService*. By dividing parts of the application into commands, maintaining, and updating the code becomes easier.

2.6.2 Class diagram

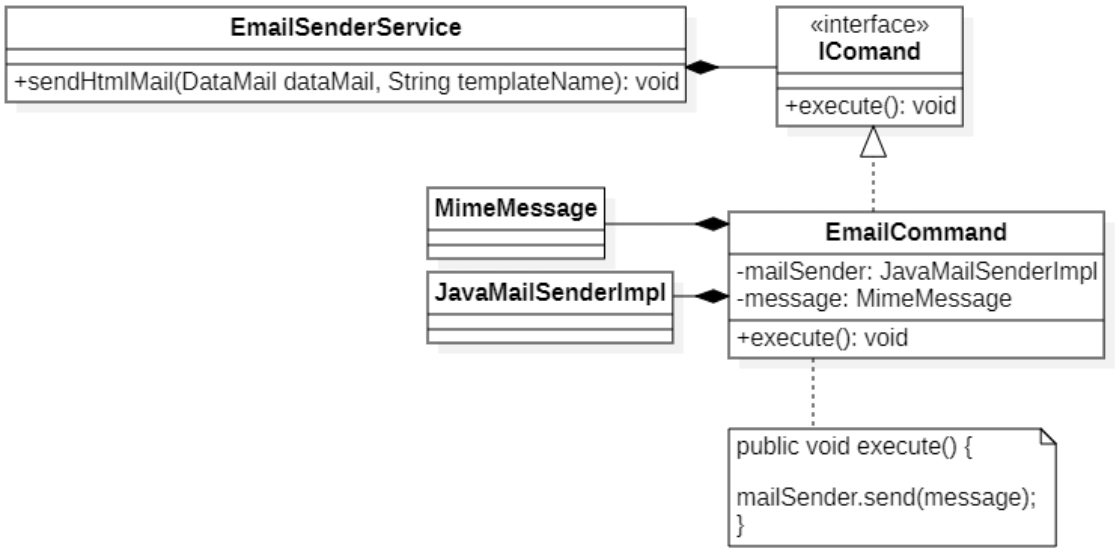


Figure 2.30: Command pattern class diagram

2.6.3 Code implementation

```java
public interface ICommand {
    void execute();
}

public class EmailCommand implements ICommand{
    private final JavaMailSenderImpl mailSender;
    private final MimeMessage message;

    public EmailCommand(JavaMailSenderImpl mailSender, MimeMessage message) {
        this.mailSender = mailSender;
        this.message = message;
    }

    @Override
    public void execute() {
        mailSender.send(message);
    }
}
```

Figure 2.31: Code for Icommand interface and EmailCommand class

```java
public void sendHtmlMail(DataMail dataMail, String templateName) throws MessagingException {
    MimeMessage message = mailSender.createMimeMessage();

    MimeMessageHelper helper = new MimeMessageHelper(message, true, "utf-8");

    Context context = new Context();
    context.setVariables(dataMail.getProps());

    SpringTemplateEngine templateEngine = getSpringTemplateEngine();
    String html = templateEngine.process(templateName, context);

    helper.setTo(dataMail.getTo());
    helper.setSubject(dataMail.getSubject());
    helper.setText(html, true);

    // Apply command pattern
    ICommand emailCommand = new EmailCommand(mailSender, message);
    emailCommand.execute();
}
```

Figure 2.32: Code for EmailSenderService class use command

2.7 Template Method pattern

2.7.1 Reason for applying

Realizing that deleting entities in the System has similar steps, to create a framework for the steps to delete an entity we decided to apply the Template Method pattern and allow specific entity classes to be defined. concrete meaning of abstract steps.

In summary, the reason for applying the Template Method Pattern here is to achieve the following purposes:

- The template method pattern allows me to define the overall structure of an algorithm in an operation on this case the delete operation, while allowing subclasses to define specific steps. This can save time and effort, as I don't have to rewrite the entire algorithm each time you need to use it.
- The template method pattern can make my code more maintainable. This is because the overall structure of the algorithm is centralized in the template class, which makes it easier to understand and modify.
- By using the template method pattern, you can avoid duplicating code in my subclasses. This can make your code more concise and easier to read.
- The template method pattern allows me to define the overall structure of an algorithm, while allowing subclasses to define specific steps. This makes it a flexible pattern that can be used to implement a wide variety of algorithms or funciton.

2.7.2 Class diagram

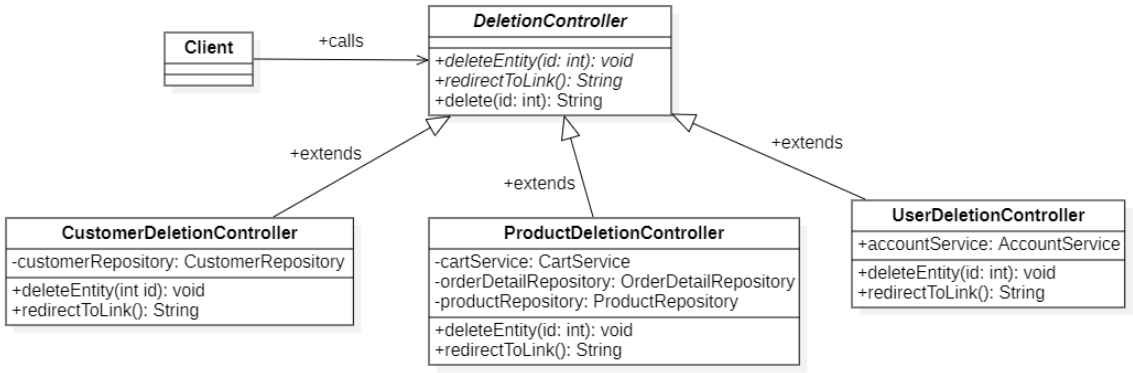


Figure 2.33: Template Method pattern class diagram

2.7.3 Code implementation

```
1  public abstract class DeletionController {
2      public String delete(int id){
3          deleteEntity(id);
4          return redirectToLink();
5      }
6
7      protected abstract void deleteEntity(int id);
8      protected abstract String redirectToLink();
9  }
```

Figure 2.34: Coder for DeletionController abstract class

```java
public class ProductDeletionController extends DeletionController {
    private CartService cartService;
    private OrderDetailRepository orderDetailRepository;
    private ProductRepository productRepository;

    @Override
    protected void deleteEntity(int id) {
        if (!cartService.isInOrder(id) && !orderDetailRepository.existsByProductId(id)) {
            productRepository.deleteById(id);
        }
    }

    @Override
    protected String redirectToLink() {
        return "redirect:/product";
    }
}
```

Figure 2.35: Code for ProductDeletionController class

```java
public class CustomerDeletionController extends DeletionController{
    private CustomerRepository customerRepository;

    @Override
    protected void deleteEntity(int id) {
        customerRepository.deleteById(id);
    }

    @Override
    protected String redirectToLink() {
        return "redirect:/customer";
    }
}
```

Figure 2.36: Code for CustomerDeletionController class

```java
public class UserDeletionController extends DeletionController {
    private AccountService accountService;

    @Override
    protected void deleteEntity(int id) {
        accountService.deleteAccount(id);
    }

    @Override
    protected String redirectToLink() {
        return "redirect:/user";
    }
}
```

Figure 2.37: Code for UserDeletionController class

2.8 Decorator pattern

2.8.1 Reason for applying

During the system development phase, the function of exporting employee lists to csv files was developed. Later, the employee list system tends to get longer and longer, leading to the file size also increasing.

Therefore, with the desire to reduce the download file size, so that the file download process is more optimized. The system decides to compress the csv file before downloading. But changing the code of the existing class *CSVExporter* will violate the Open/Closed Principle, as well as editing the old code is unreasonable because the code is still running stably.

Therefore, the system decided to apply the Decorator pattern to decorate the file compression behavior for exporting the employee list

In summary, the reason for applying the Decorator Pattern here is to achieve the following purposes:

- *CSVExporter's* behavior can be extended without creating a new subclass.

- To expand functionality while still complying with the Open/Closed Principle

2.8.2 Class diagram

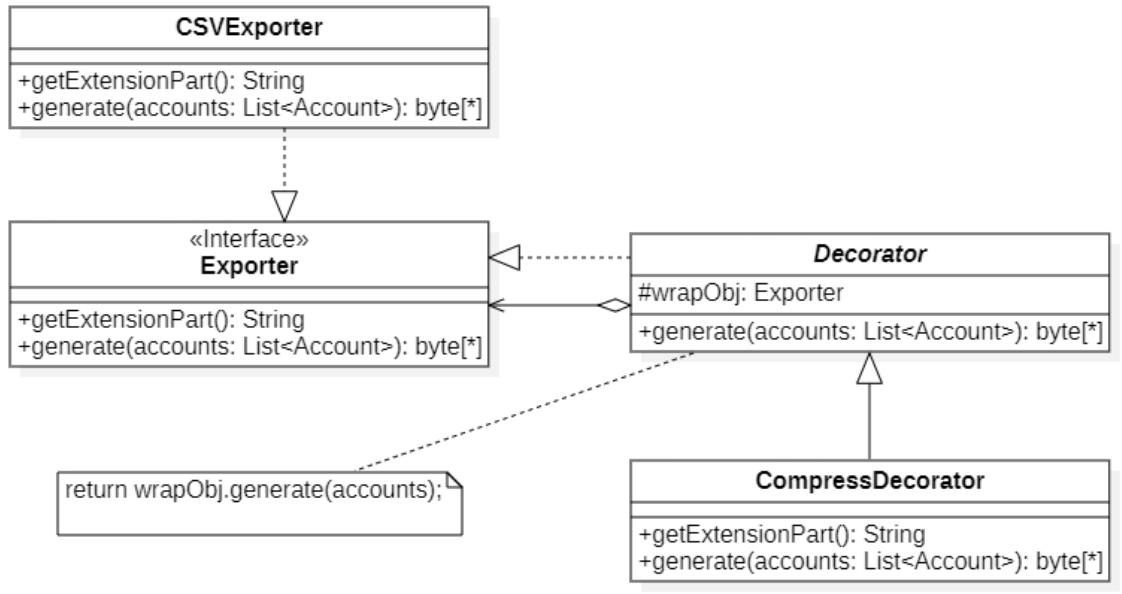


Figure 2.38: Decorator pattern class diagram

2.8.3 Code implementation

```
1  public interface Exporter {
2      String getExtensionPart();
3      byte[] generate(List<Account> accounts);
4  }
```

Figure 2.39: Code for Exporter interface

```java
public class CSVExporter implements Exporter{
    @Override
    public String getExtensionPart() {
        return ".csv";
    }

    public byte[] generate(List<Account> accounts) {
        try (ByteArrayOutputStream baos = new ByteArrayOutputStream();
             Writer writer = new OutputStreamWriter(baos)) {
            CSVWriter csvWriter = new CSVWriter(writer,
                    CSVWriter.DEFAULT_SEPARATOR,
                    CSVWriter.NO_QUOTE_CHARACTER,
                    CSVWriter.DEFAULT_ESCAPE_CHARACTER,
                    CSVWriter.DEFAULT_LINE_END);

            String[] header = {"Full Name", "Email", "Status", "Role", "Phone"};
            csvWriter.writeNext(header);

            // Write CSV data
            for (Account account : accounts) {
                String[] data = {
                        account.getFullName(),
                        account.getEmail(),
                        String.valueOf(account.isStatus()),
                        account.getRole().toString(),
                        account.getPhone()
                };
                csvWriter.writeNext(data);
            }
            csvWriter.flush();
            csvWriter.close();

            return baos.toByteArray();
        } catch (IOException e) {
            e.printStackTrace();
            return null;
        }
    }
}
```

Figure 2.40: Code for CSVExporter class

```java
public abstract class Decorator implements Exporter{
    protected Exporter wrapObj;

    public Decorator(Exporter wrapObj) {
        this.wrapObj = wrapObj;
    }

    @Override
    public byte[] generate(List<Account> accounts) {
        return wrapObj.generate(accounts);
    }
}
```

Figure 2.41: Code for Decorator abstract class

```java
public class CompressDecorator extends Decorator{
    public CompressDecorator(Exporter wrapObj) {
        super(wrapObj);
    }

    @Override
    public String getExtensionPart() {
        return ".zip";
    }

    @Override
    public byte[] generate(List<Account> accounts) {
        byte[] data = super.generate(accounts);

        try (ByteArrayOutputStream baos = new ByteArrayOutputStream();
             ZipOutputStream zipOut = new ZipOutputStream(baos)) {

            zipOut.putNextEntry(new ZipEntry("staff" + wrapObj.getExtensionPart()));

            zipOut.write(data);
            zipOut.closeEntry();
            zipOut.finish();

            return baos.toByteArray();
        } catch (IOException e) {
            e.printStackTrace();
            return data;
        }
    }
}
```

Figure 2.42: Code for CompressDecorator class

2.9 Adapter pattern

2.9.1 Reason for applying

Some time after the system developed the CSV file export feature, due to increased demand. The csv file does not meet some features such as not being able to read information in Vietnamese language well, not supporting images, and not having advanced features such as statistical calculations.

The system decided to expand the XLSX file export options, providing many advanced features, and working well when used with Excel.

On the other hand, exporting csv files still works fine, so completely replacing the option to export csv files is unreasonable. Besides, the csv file export option is

still working well with *CompressDecorator*, so the system decided to use the Adapter pattern to connect two incompatible interfaces without having to modify the old code.

In summary, the reason for applying the Adapter Pattern here is to achieve the following purposes:

- Ensure the Open/Closed Principle: New types of exporters can be introduced into the program without breaking the existing code. Work well with old code and *Decorator*
- Provide users with more file export options. CSV for simple users, XLSX for users with advanced usage needs

2.9.2 Class diagram

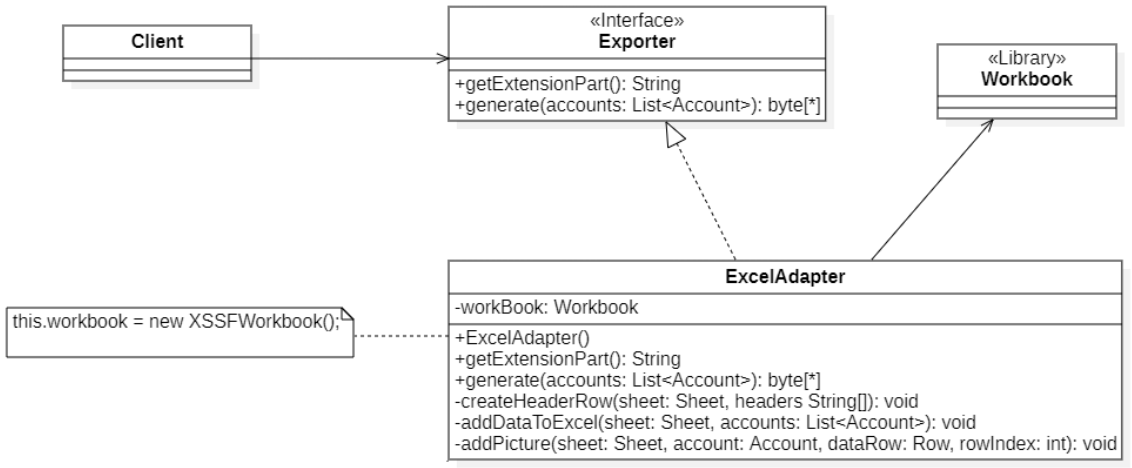


Figure 2.43: Adapter pattern class diagram

2.9.3 Code implementation

```java
public class ExcelAdapter implements Exporter{
    private Workbook workbook;

    public ExcelAdapter() {
        this.workbook = new XSSFWorkbook();
    }

    @Override
    public String getExtensionPart() {
        return ".xlsx";
    }

    @Override
    public byte[] generate(List<Account> accounts) {
        try (ByteArrayOutputStream baos = new ByteArrayOutputStream()) {
            Sheet sheet = workbook.createSheet("Staffs");
            sheet.setColumnWidth(0, 6400);
            String[] headers = {"Avatar", "Full Name", "Email", "Status", "Role", "Phone"};

            createHeaderRow(sheet, headers);
            addDataToExcel(sheet, accounts);

            sheet.setColumnWidth(0, 30 * 256);
            for (int i = 1; i < headers.length; i++) {
                sheet.autoSizeColumn(i);
            }

            workbook.write(baos);
            return baos.toByteArray();
        } catch (IOException e) {
            e.printStackTrace();
            return null;
        }
    }

    // remaining methods
}
```

Figure 2.44: Code for ExcelAdapter class

Code for Expoter interface are the same with Figure 2.39

```java
public class SimpleExporterFactory {
    public static Exporter createExporter(ExportFormat format) {
        switch (format) {
            case CSV:
                return new CSVExporter();
            case XLSX:
                return new ExcelAdapter();
            default:
                return null;
        }
    }
}
public enum ExportFormat {
    CSV, XLSX
}
```

Figure 2.45: SimpleExpoterFactory support to create specific Exporter

```java
@GetMapping("/export")
public ResponseEntity<byte[]> export(ExportFormat format, boolean compress) {
    List<Account> accounts = accountService.getAllAccounts();
    Exporter exporter = SimpleExporterFactory.createExporter(format);

    if (compress) {
        exporter = new CompressDecorator(exporter);
    }

    byte[] data = exporter.generate(accounts);

    HttpHeaders headers = new HttpHeaders();
    headers.setContentType(MediaType.TEXT_PLAIN);
    headers.setContentDispositionFormData("filename", "staffs" + exporter.getExtensionPart());

    return new ResponseEntity<>(data, headers, HttpStatus.OK);
}
```

Figure 2.46: Client use Adapter fit with existed code

2.10 Chain of Responsibility pattern

2.10.1 Reason for applying

The current POS system has a user authentication process at login that is quite complicated, difficult to understand, and difficult to maintain, upgrade, and manage.

Therefore, the system decided to divide the user authentication process into sequential chains, separating each process into a certain source code. Applying the Chain of Responsibility pattern is a good idea at this point.

In summary, the reason for applying the Chain of Responsibility Pattern here is to achieve the following purposes:

- The order of account checking for login can be controlled.
- Ensure the Single Responsibility Principle: Middleware classes that invoke operations can be decoupled from classes that perform operations.
- Ensure the Open/Closed Principle: New Middleware can be introduced into the app without breaking the existing code.

2.10.2 Class diagram

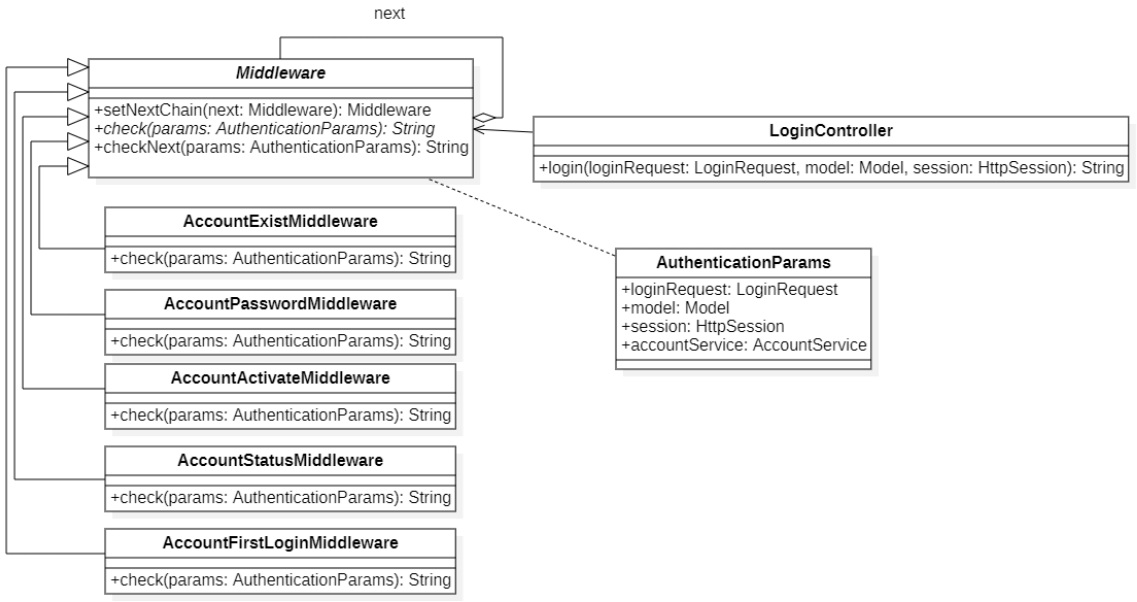


Figure 2.47: Chain of Responsibility pattern class diagram

2.10.3 Code implementation

```java
public abstract class Middleware {
    protected Middleware next;

    public Middleware setNextChain(Middleware next) {
        this.next = next;
        return next;
    }

    public abstract String check(AuthenticationParams params);

    protected String checkNext(AuthenticationParams params) {
        if (next != null) {
            return next.check(params);
        }
        params.session.setAttribute("username", params.loginRequest.getUsername());
        params.session.setAttribute("account", params
            .accountService.getAccountByUsername(params.loginRequest.getUsername()));
        return "redirect:/";
    }
}
```

Figure 2.48: Code for Middleware class

```java
public class AuthenticationParams {
    public LoginRequest loginRequest;
    public Model model;
    public HttpSession session;
    public AccountService accountService;
}
```

Figure 2.49: Code for AuthenticationParams class

```
public class AccountExistMiddleware extends Middleware {
    @Override
    public String check(AuthenticationParams params) {
        Account account = params.accountService.getAccountByUsername(params.loginRequest.getUsername());
        if (account == null) {
            params.model.addAttribute("error", "Your account does not exist, please contact the admin");
            return "/Login";
        } else {
            return checkNext(params);
        }
    }
}
```

Figure 2.50: Code for AccountExistMiddleware class

```
public class AccountPasswordMiddleware extends Middleware {
    @Override
    public String check(AuthenticationParams params) {
        Account account = params.accountService.getAccountByUsername(params.loginRequest.getUsername());
        String hashedPassword = account.getPassword();
        if (!BCrypt.checkpw(params.loginRequest.getPassword(), hashedPassword)) {
            params.model.addAttribute("error", "Your username or password is not correct");
            return "/Login";
        } else {
            return checkNext(params);
        }
    }
}
```

Figure 2.51: Code for AccountPasswordMiddleware class

```
public class AccountActivateMiddleware extends Middleware {
    @Override
    public String check(AuthenticationParams params) {
        Account account = params.accountService.getAccountByUsername(params.loginRequest.getUsername());
        if (!account.isActivate()) {
            params.model.addAttribute("error", "Please activate your account via email");
            return "/Login";
        } else {
            return checkNext(params);
        }
    }
}
```

Figure 2.52: Code for AccountActivateMiddleware class

```
public class AccountStatusMiddleware extends Middleware {
    @Override
    public String check(AuthenticationParams params) {
        Account account = params.accountService.getAccountByUsername(params.loginRequest.getUsername());
        if (!account.isStatus()) {
            params.model.addAttribute("error", "Your account is blocked, please contact the admin");
            return "/Login";
        } else {
            return checkNext(params);
        }
    }
}
```

Figure 2.53: Code for AccountStatusMiddleware class

```
public class AccountFirstLoginMiddleware extends Middleware {
    @Override
    public String check(AuthenticationParams params) {
        if (params.loginRequest.getPassword().equals(params.loginRequest.getUsername())) {
            params.session.setAttribute("usernameWelcome", params.loginRequest.getUsername());
            return "redirect:/welcome";
        } else {
            return checkNext(params);
        }
    }
}
```

Figure 2.54: Code for AccountFirstLoginMiddleware class

```
@PostMapping("/login")
public String login(LoginRequest loginRequest, Model model, HttpSession session)
{
    AuthenticationParams params = new AuthenticationParams(loginRequest, model, session, accountService);
    Middleware authenticationChain = new AccountExistMiddleware();
    authenticationChain.setNextChain(new AccountPasswordMiddleware())
            .setNextChain(new AccountActivateMiddleware())
            .setNextChain(new AccountStatusMiddleware())
            .setNextChain(new AccountFirstLoginMiddleware());

    return authenticationChain.check(params);
}
```

Figure 2.55: Code for client where use middleware

2.11 Observer pattern

2.11.1 Reason for applying

Realizing that *Product* is an object that often must update its selling price and import price, but because there are too many products, if there is any small change in

a specific product, it will be difficult to detect. So, the system decides to apply the Observer pattern. Notify *Users* and *Customers* every time a product's status changes.

In summary, the reason for applying the Observer Pattern here is to achieve the following purposes:

- Add a subscription mechanism to the publisher class so individual objects can subscribe to or unsubscribe from a stream of events coming from that publisher.
- The *Product* doesn't need to know the concrete details of *Account* and *Customer*. This reduces dependencies and makes it easier to add, remove, or change observers dynamically without affecting the Product.
- Can introduce new types of observers that want to react to product changes without having to modify the *Product* class itself. This promotes the Open/Closed.
- The *Product* is responsible for managing its own data and state.
- Observers have the specific responsibility of reacting to changes, promoting better organization of your code.
- A single change in the *Product* can trigger updates in multiple dependent *Accounts* and *Customers*. This is particularly useful for keeping different parts of your system in sync.

2.11.2 Class diagram

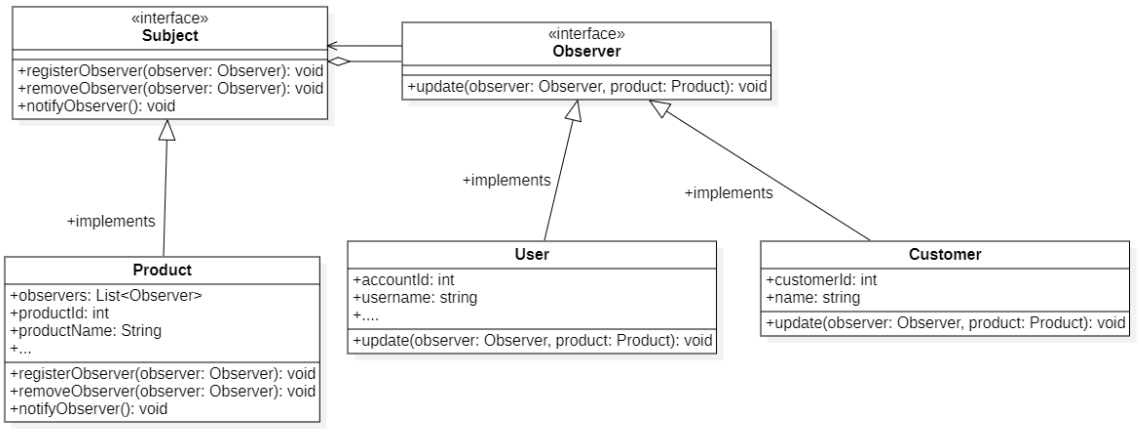


Figure 2.56: Observer pattern class diagram

2.11.3 Code implementation

At this point, there are several reasons to consider the necessity of applying this pattern to the system. Therefore, the system has not yet been applied to the system. Reasons that must be considered include:

- There is no specific form of notification: sending email, sending sms or developing a notification box feature for the website.
- Notify customers about products they are not interested in need to develop new notification features, sign up to receive notifications about favorite products.

REFERENCES

[1] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, “Design Patterns : Elements of Reusable Object-Oriented Software”.

[2] “[O`Reilly. Head First] - Head First Design Patterns - [Freeman].pdf.”

[3] “The Catalog of Design Patterns.” Accessed: Apr. 21, 2024. [Online]. Available: https://refactoring.guru/design-patterns/catalog

[4] PhucVR, “nguyenphuc22/Design-Patterns.” Apr. 08, 2024. Accessed: Apr. 10, 2024. [Online]. Available: https://github.com/nguyenphuc22/Design-Patterns