ITSS SOFTWARE DEVELOPMENT
Lab 06 - Class Design

Lecturer: NGUYEN Thi Thu Trang, trangntt@soict.hust.edu.vn

1. SUBMISSION GUIDELINE

You are required to push all your work to the valid GitHub repository complying with the naming convention:

"<MSTeamName>-<StudentID>.<StudentName>".

For this lab, you have to turn in your work twice before the following deadlines:

- **Right after class**: Push all the work you have done during class time to Github.
- **10 PM the day after the class**: Create a branch named "***release/lab06***" in your GitHub repository and push the full submission for this lab, including in-class tasks and homework assignments, to this branch.

2. IN-CLASS ASSIGNMENT

In this section, we will get familiar with the software detailed design process and try ourselves with class design for the Case Study.

You are asked to work individually for this section, and then put all your file(s) and directories to a directory, namely "DetailedDesign/ClassDesign". After that, push your commit to your individual repository before the announced deadline.

2.1. CREATE INITIAL DESIGN CLASSES

In this step, we try to map the analysis classes and design elements (e.g., class, group of classes, package, subsystem). A design class should have a single well-focused purpose and should do one thing well. We would identify design classes by considering each class in the unified class diagram from architectural design, along with its class stereotype. Note that we have not applied any design patterns[1] in this lab yet.

2.1.1. Design Boundary Classes

User interface (UI) boundary classes

In case of the Case Study, JavaFX 15 works as our tool to develop our UI. Thus, from our architecture viewpoint, each UI boundary class is corresponded to design class(s) handling the events/actions from humans which are captured in corresponding FXML files. Although in JavaFX, the design class(s) is called the "controller" of the FXML files, it does not play as the role of the control class in UML. Consequently, most of those event handlers are quite simple, and hence the mapping here is 1-1.

System/device boundary classes

In the previous lab, we have evolved the boundary class for Interbank into a subsystem. This subsystem, however, has not well-designed: the InterbankSubsystemController is so complex, and a part of the InterbankBoundary might be reusable. There is only one external web information system (i.e., interbank) in the current Case Study; however, the future is an unknown whereas there are a huge number of systems that require communication with REST APIs over HTTP. One of them could be the next external system of our developing AIMS Software while the communication protocols for those systems are similar and consistent. As a result, for the reusability, we need a new class, for example, API, to be responsible for API communication such as HTTP GET and HTTP POST. Besides, we would consider the problem or the controller later.

2.1.2. Design Entity Classes

For the current Problem statement and the 2 use cases "Pay Order" and "Place Order", most of the entity classes in the architectural design are simple and could be 1-1 correspondence to the design classes.

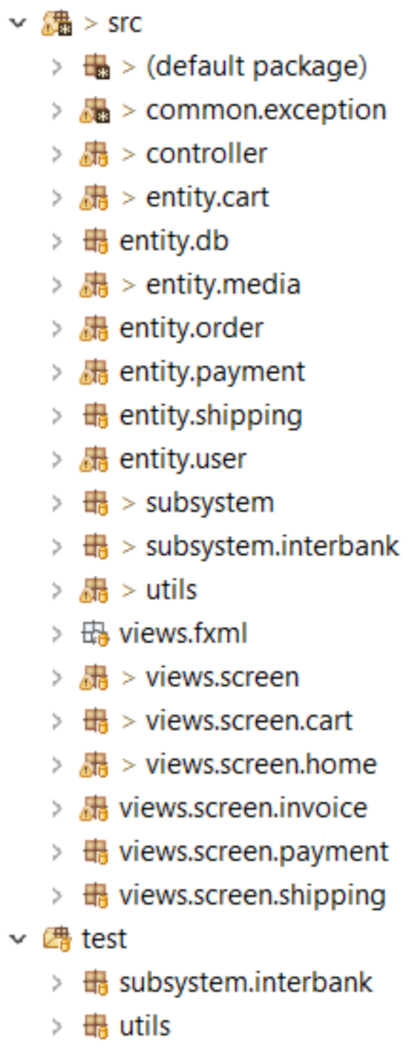
[1] A software design pattern is a general, reusable solution to a commonly occurring problem within a given context in software design. See more at this link.

2.1.3. Design Control Classes

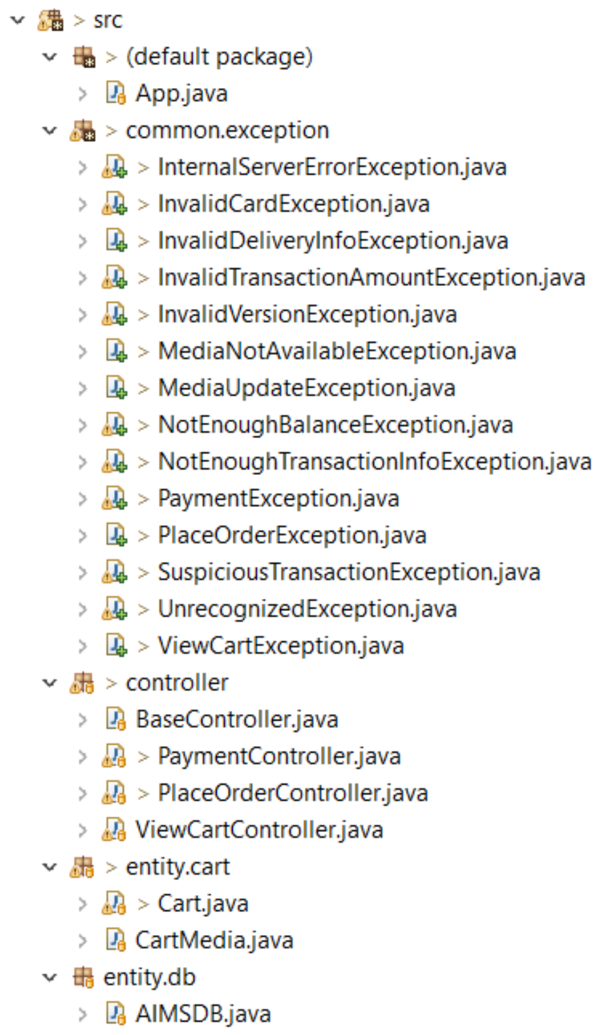
Likewise, most of the control classes in the architectural design are simple and could be one-to-one correspondence to the design classes. However, the InterbankSubsystemController is currently responsible for at least 2 tasks: (1) data flow control and (2) data conversion (e.g., to convert the requests with data to the required format and to handle the response). Thus, we need at least another class (e.g., JSON or MyMap) to take the responsibility of data conversion (based on the design, this can also be reused when communicate with other web information systems).

2.1.4. Group Design classes in Packages

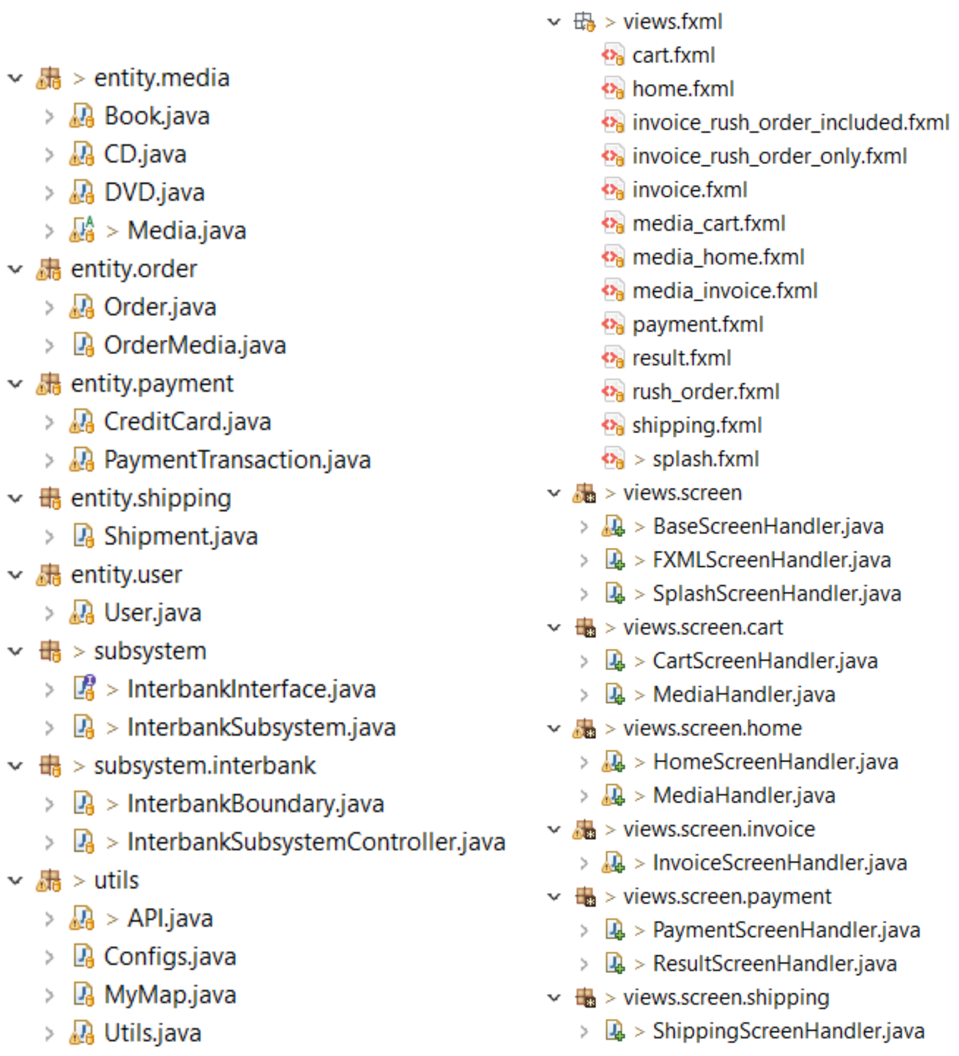
Here is a way of grouping design classes in packages.



```
v > src
    > > (default package)
    > > common.exception
    > > controller
    > > entity.cart
    > entity.db
    > > entity.media
    > entity.order
    > entity.payment
    > entity.shipping
    > entity.user
    > > subsystem
    > > subsystem.interbank
    > > utils
    > views.fxml
    > > views.screen
    > > views.screen.cart
    > > views.screen.home
    > views.screen.invoice
    > views.screen.payment
    > views.screen.shipping
v test
    > subsystem.interbank
    > utils
```



```
v > src
    v > (default package)
        > App.java
    v > common.exception
        > > InternalServerErrorException.java
        > > InvalidCardException.java
        > > InvalidDeliveryInfoException.java
        > > InvalidTransactionAmountException.java
        > > InvalidVersionException.java
        > > MediaNotAvailableException.java
        > > MediaUpdateException.java
        > > NotEnoughBalanceException.java
        > > NotEnoughTransactionInfoException.java
        > > PaymentException.java
        > > PlaceOrderException.java
        > > SuspiciousTransactionException.java
        > > UnrecognizedException.java
        > > ViewCartException.java
    v > controller
        > BaseController.java
        > > PaymentController.java
        > > PlaceOrderController.java
        > ViewCartController.java
    v > entity.cart
        > > Cart.java
        > CartMedia.java
    v entity.db
        > AIMSDB.java
```

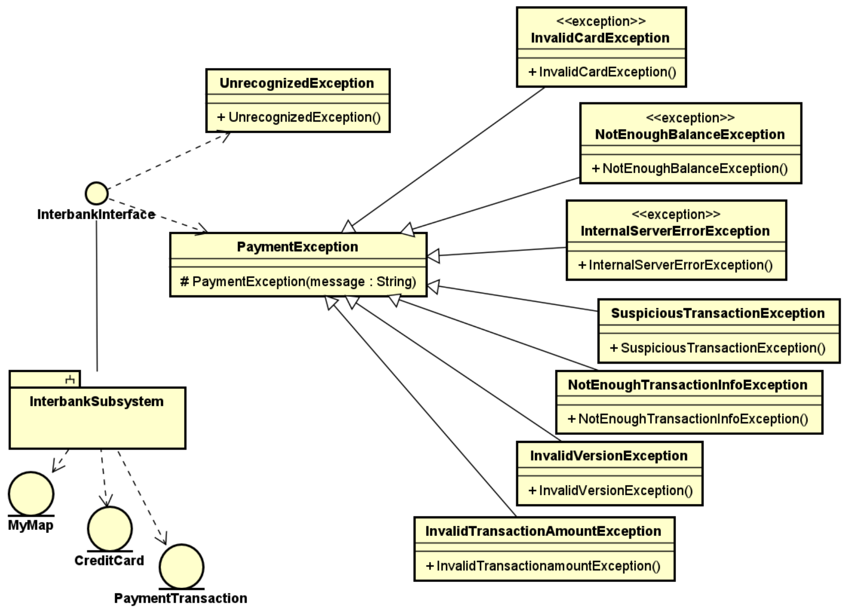


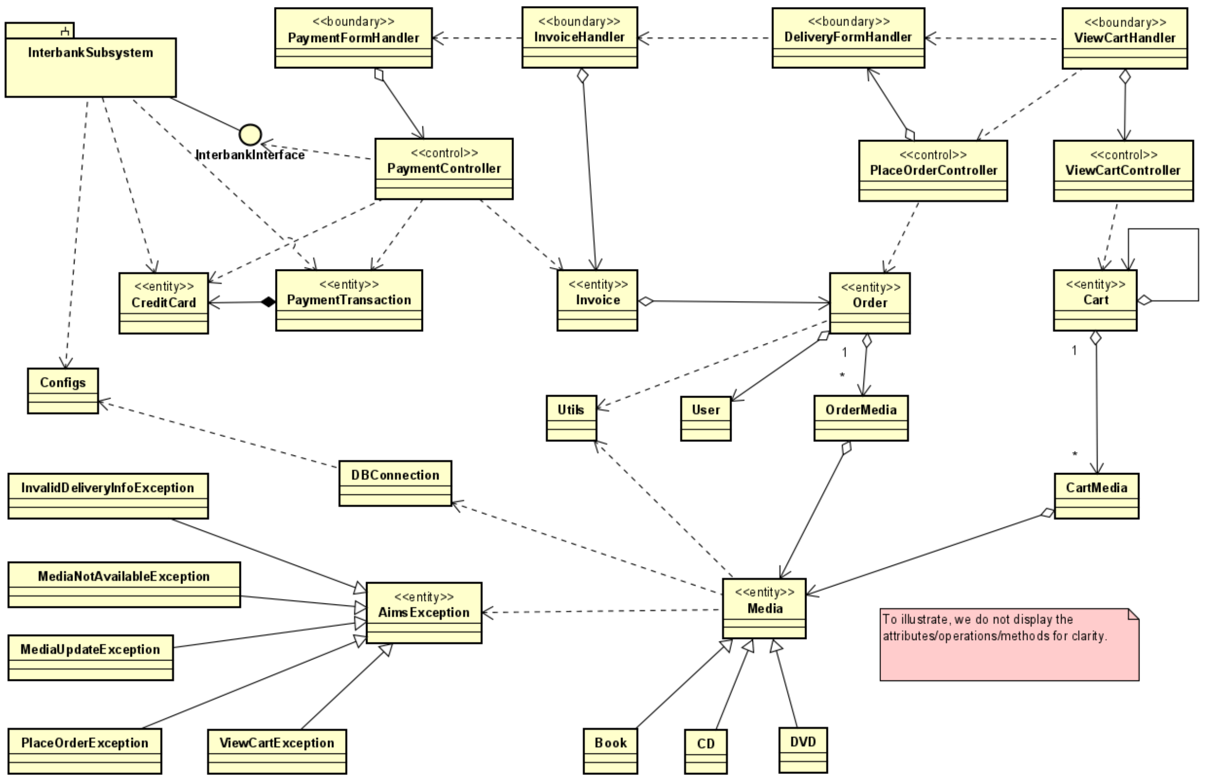
```
v entity.media
    > Book.java
    > CD.java
    > DVD.java
    > > Media.java
v entity.order
    > Order.java
    > OrderMedia.java
v entity.payment
    > CreditCard.java
    > PaymentTransaction.java
v entity.shipping
    > Shipment.java
v entity.user
    > User.java
v > subsystem
    > > InterbankInterface.java
    > > InterbankSubsystem.java
v > subsystem.interbank
    > > InterbankBoundary.java
    > > InterbankSubsystemController.java
v > utils
    > > API.java
    > Configs.java
    > MyMap.java
    > Utils.java
```

```
v > views.fxml
    cart.fxml
    home.fxml
    invoice_rush_order_included.fxml
    invoice_rush_order_only.fxml
    invoice.fxml
    media_cart.fxml
    media_home.fxml
    media_invoice.fxml
    payment.fxml
    result.fxml
    rush_order.fxml
    shipping.fxml
    > splash.fxml
v > views.screen
    > > BaseScreenHandler.java
    > > FXMLScreenHandler.java
    > > SplashScreenHandler.java
v > views.screen.cart
    > > CartScreenHandler.java
    > > MediaHandler.java
v > views.screen.home
    > > HomeScreenHandler.java
    > > MediaHandler.java
v > views.screen.invoice
    > > InvoiceScreenHandler.java
v > views.screen.payment
    > > PaymentScreenHandler.java
    > > ResultScreenHandler.java
v > views.screen.shipping
    > > ShippingScreenHandler.java
```

2.2. DEFINE RELATIONSHIPS BETWEEN CLASSES

To illustrate, the following figures show the relationships between classes. We do not display the attributes and operations/methods of classes since it would be hard for you to see the relationships.

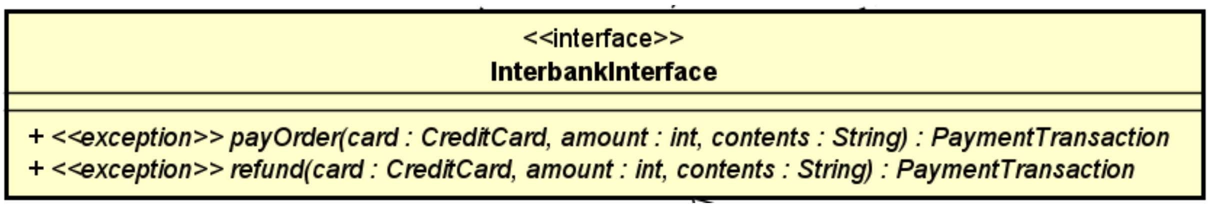

<<exception>>
InvalidCardException
+ InvalidCardException()
UnrecognizedException
+ UnrecognizedException()
<<exception>>
NotEnoughBalanceException
+ NotEnoughBalanceException()
InterbankInterface
<<exception>>
InternalServerErrorException
+ InternalServerErrorException()
PaymentException
# PaymentException(message : String)
SuspiciousTransactionException
+ SuspiciousTransactionException()
NotEnoughTransactionInfoException
+ NotEnoughTransactionInfoException()
InterbankSubsystem
InvalidVersionException
+ InvalidVersionException()
MyMap
CreditCard
InvalidTransactionAmountException
+ InvalidTransactionamountException()
PaymentTransaction



2.3. CLASS DESIGN

In this subsection, we illustrate how to do class design step by step.

2.3.1. Class “InterbankInterface”


<<interface>>
InterbankInterface
+ <<exception>> payOrder(card : CreditCard, amount : int, contents : String) : PaymentTransaction
+ <<exception>> refund(card : CreditCard, amount : int, contents : String) : PaymentTransaction

Attribute

None

Operation

#	*Name*	*Return type*	*Description (purpose)*
1	payOrder	PaymentTransaction	Pay order, and then return the payment transaction
2	refund	PaymentTransaction	Refund, and then return the payment transaction

Parameter:

- card - the credit card used for payment/refund
- amount - the amount to pay/refund
- contents - the transaction contents

Exception:

- PaymentException - if responded with a pre-defined error code
- UnrecognizedException - if responded with an unknown error code or something goes wrong

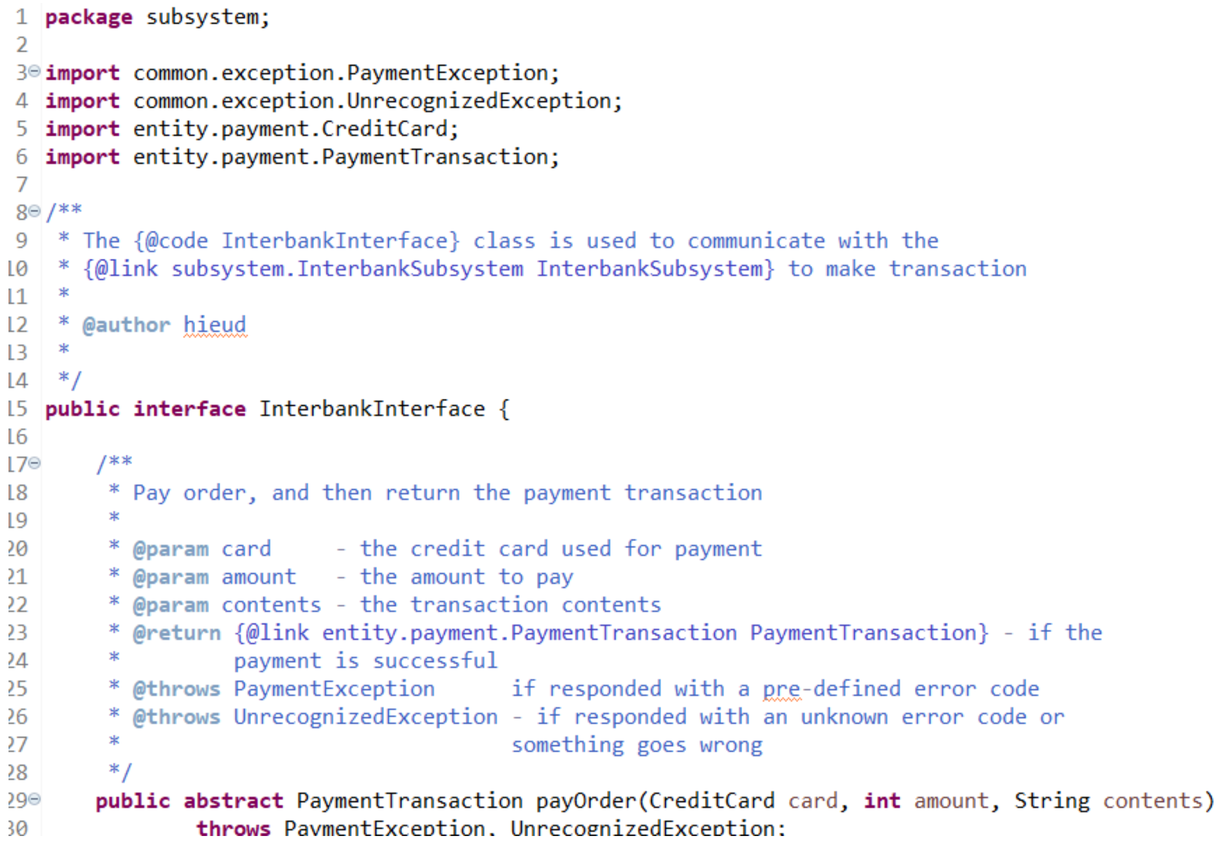
Method

None

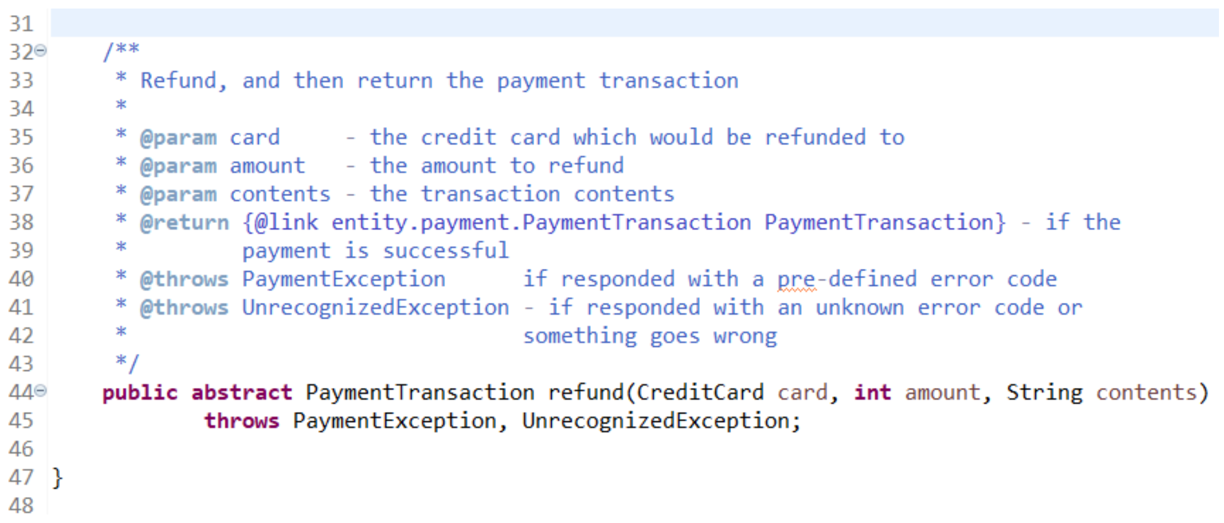
State

None

When you are programming, please always remember to do the programming documentation **immediately, especially public elements**. The programming documentation for this class is illustrated as follows.



```java
package subsystem;

import common.exception.PaymentException;
import common.exception.UnrecognizedException;
import entity.payment.CreditCard;
import entity.payment.PaymentTransaction;

/**
 * The {@code InterbankInterface} class is used to communicate with the
 * {@link subsystem.InterbankSubsystem InterbankSubsystem} to make transaction
 *
 * @author hieud
 *
 */
public interface InterbankInterface {

    /**
     * Pay order, and then return the payment transaction
     *
     * @param card     - the credit card used for payment
     * @param amount   - the amount to pay
     * @param contents - the transaction contents
     * @return {@link entity.payment.PaymentTransaction PaymentTransaction} - if the
     *         payment is successful
     * @throws PaymentException      if responded with a pre-defined error code
     * @throws UnrecognizedException - if responded with an unknown error code or
     *                               something goes wrong
     */
    public abstract PaymentTransaction payOrder(CreditCard card, int amount, String contents)
            throws PaymentException, UnrecognizedException;
```



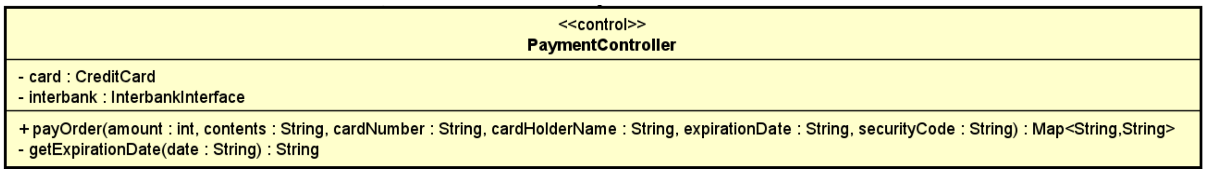
```java

    /**
     * Refund, and then return the payment transaction
     *
     * @param card     - the credit card which would be refunded to
     * @param amount   - the amount to refund
     * @param contents - the transaction contents
     * @return {@link entity.payment.PaymentTransaction PaymentTransaction} - if the
     *         payment is successful
     * @throws PaymentException      if responded with a pre-defined error code
     * @throws UnrecognizedException - if responded with an unknown error code or
     *                               something goes wrong
     */
    public abstract PaymentTransaction refund(CreditCard card, int amount, String contents)
            throws PaymentException, UnrecognizedException;

}

```

Please see the following links for better understanding:

https://users.soe.ucsc.edu/~eaugusti/archive/102-winter16/misc/howToWriteJavaDocs.html

https://www.oracle.com/technical-resources/articles/java/javadoc-tool.html

2.3.2. Class “PaymentController”



<<control>>
PaymentController
- card : CreditCard
- interbank : InterbankInterface
+ payOrder(amount : int, contents : String, cardNumber : String, cardHolderName : String, expirationDate : String, securityCode : String) : Map<String,String>
- getExpirationDate(date : String) : String

Attribute

#	*Name*	*Data type*	*Default value*	*Description*
1	card	CreditCard	NULL	Represent the card used for payment
2	interbank	InterbankInterface	NULL	Represent the Interbank subsystem

Operation

#	*Name*	*Return type*	*Description (purpose)*
1	payOrder	Map<String,String>	Pay order, and then return the result with a message

Parameter:

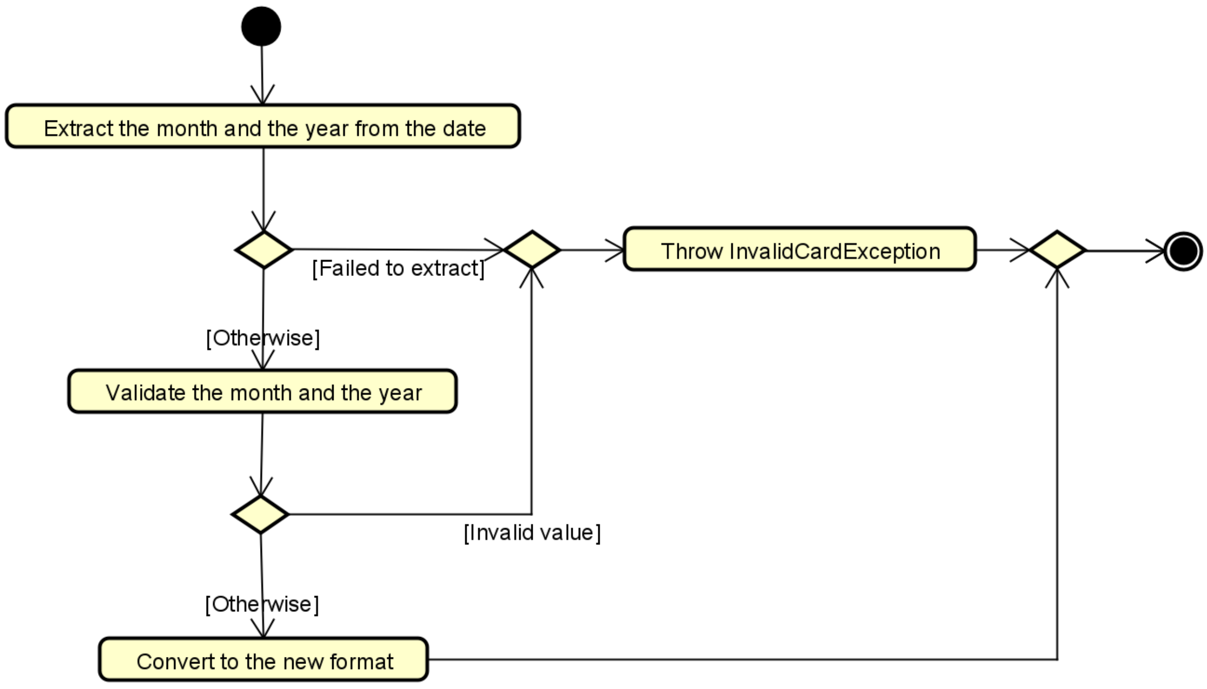
- amount - the amount to pay
- contents - the transaction contents
- cardNumber - the card number
- cardHolderName - the card holder name
- expirationDate - the expiration date in the format "mm/yy"
- securityCode - the cvv/cvc code of the credit card

Exception:

- None

Method

- getExpirationDate: Given the String “date” representing the expiration date in the format "mm/yy", this method convert it into the required format “mmyy.” The algorithm is illustrated as follows.


Extract the month and the year from the date
[Failed to extract]
Throw InvalidCardException
[Otherwise]
Validate the month and the year
[Invalid value]
[Otherwise]
Convert to the new format

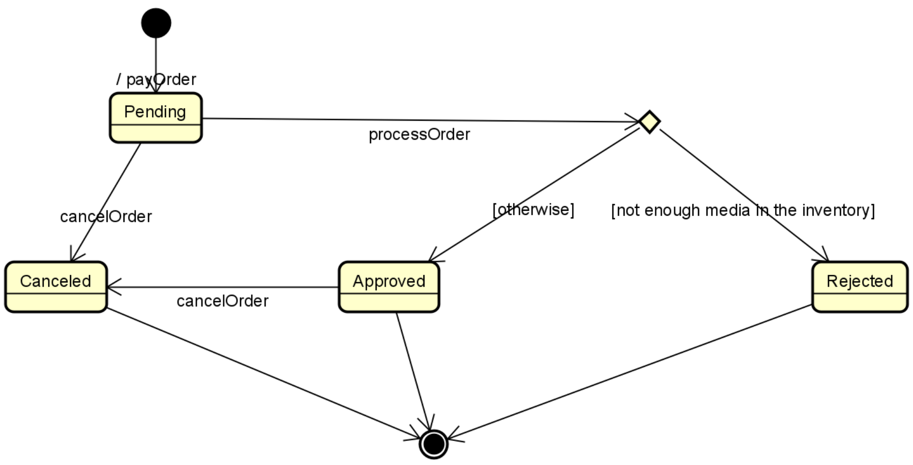
State

None

2.3.3. State Machine Diagram for an "Order" Object

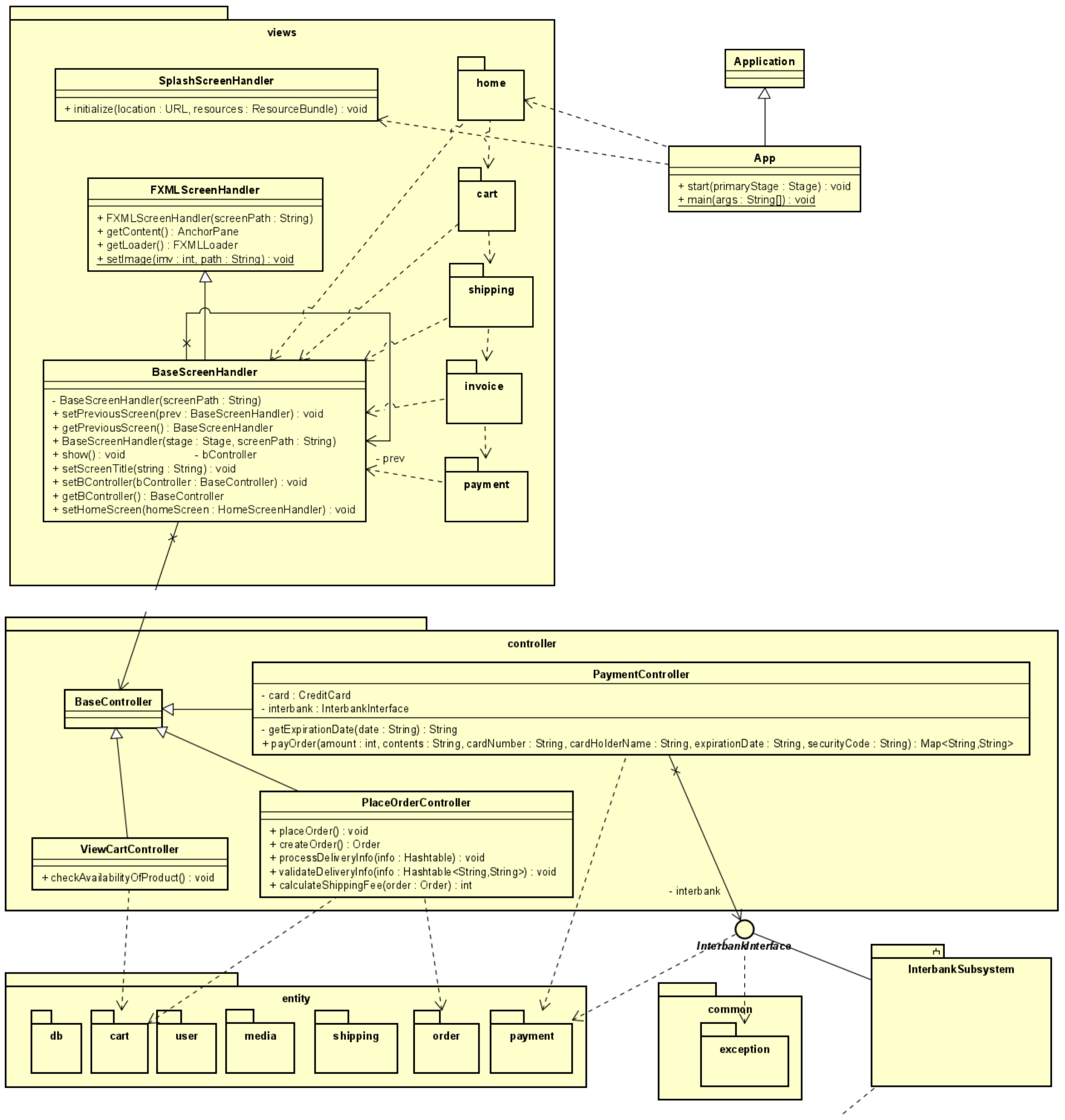
We only need to define states for objects having named states so that we know how an object's state affect its behavior and model its behavior. At first, we find candidate classes having named states, then find all possible states for an object of that class. After that, we model by drawing state machine diagram.

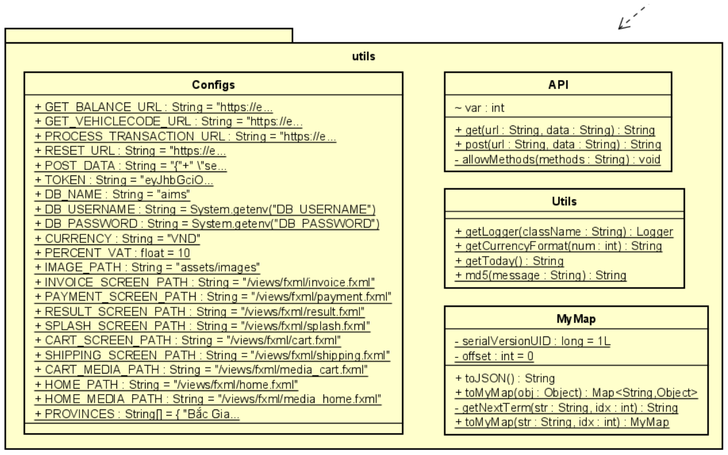
Clearly, "Order" object is one of those having named states. The following figures illustrate the state machine diagram for an "Order" object from the moment when a customer places an order until the time an admin processes the order.



2.4. CLASS DIAGRAM

Finally, we put all the design classes into 1 general class diagram. Note that we shall not show the details of subsystem(s). In case there are too many details, we can either do the same with packages or make class diagrams for each package/subsystem.


views
SplashScreenHandler
+ initialize(location : URL, resources : ResourceBundle) : void
home
cart
shipping
invoice
payment
FXMLScreenHandler
+ FXMLScreenHandler(screenPath : String)
+ getContent() : AnchorPane
+ getLoader() : FXMLLoader
+ setImage(imv : int, path : String) : void
BaseScreenHandler
- BaseScreenHandler(screenPath : String)
+ setPreviousScreen(prev : BaseScreenHandler) : void
+ getPreviousScreen() : BaseScreenHandler
+ BaseScreenHandler(stage : Stage, screenPath : String)
+ show() : void
- bController
+ setScreenTitle(string : String) : void
+ setBController(bController : BaseController) : void
+ getBController() : BaseController
+ setHomeScreen(homeScreen : HomeScreenHandler) : void
- prev
Application
App
+ start(primaryStage : Stage) : void
+ main(args : String[]) : void
controller
BaseController
PaymentController
- card : CreditCard
- interbank : InterbankInterface
- getExpirationDate(date : String) : String
+ payOrder(amount : int, contents : String, cardNumber : String, cardHolderName : String, expirationDate : String, securityCode : String) : Map<String,String>
ViewCartController
+ checkAvailabilityOfProduct() : void
PlaceOrderController
+ placeOrder() : void
+ createOrder() : Order
+ processDeliveryInfo(info : Hashtable) : void
+ validateDeliveryInfo(info : Hashtable<String,String>) : void
+ calculateShippingFee(order : Order) : int
- interbank
InterbankInterface
entity
db
cart
user
media
shipping
order
payment
common
exception
InterbankSubsystem


utils
Configs
+ GET_BALANCE_URL : String = "https://e...
+ GET_VEHICLECODE_URL : String = "https://e...
+ PROCESS_TRANSACTION_URL : String = "https://e...
+ RESET_URL : String = "https://e...
+ POST_DATA : String = "{"+" \"se...
+ TOKEN : String = "eyJhbGciO...
+ DB_NAME : String = "aims"
+ DB_USERNAME : String = System.getenv("DB_USERNAME")
+ DB_PASSWORD : String = System.getenv("DB_PASSWORD")
+ CURRENCY : String = "VND"
+ PERCENT_VAT : float = 10
+ IMAGE_PATH : String = "assets/images"
+ INVOICE_SCREEN_PATH : String = "/views/fxml/invoice.fxml"
+ PAYMENT_SCREEN_PATH : String = "/views/fxml/payment.fxml"
+ RESULT_SCREEN_PATH : String = "/views/fxml/result.fxml"
+ SPLASH_SCREEN_PATH : String = "/views/fxml/splash.fxml"
+ CART_SCREEN_PATH : String = "/views/fxml/cart.fxml"
+ SHIPPING_SCREEN_PATH : String = "/views/fxml/shipping.fxml"
+ CART_MEDIA_PATH : String = "/views/fxml/media_cart.fxml"
+ HOME_PATH : String = "/views/fxml/home.fxml"
+ HOME_MEDIA_PATH : String = "/views/fxml/media_home.fxml"
+ PROVINCES : String[] = { "Bắc Gia...
API
~ var : int
+ get(url : String, data : String) : String
+ post(url : String, data : String) : String
- allowMethods(methods : String) : void
Utils
+ getLogger(className : String) : Logger
+ getCurrencyFormat(num : int) : String
+ getToday() : String
+ md5(message : String) : String
MyMap
- serialVersionUID : long = 1L
- offset : int = 0
+ toJSON() : String
+ toMyMap(obj : Object) : Map<String,Object>
- getNextTerm(str : String, idx : int) : String
+ toMyMap(str : String, idx : int) : MyMap

2.5. CLASS DESIGN FOR "PLACE RUSH ORDER"

In this part, you are asked to design classes for "Place Rush Order" step-by-step by yourself.

When you complete your tasks, please export your work into a PDF file, and then save it inside "DetailedDesign/ClassDesign" directory. Remember to push all your work to your individual repository.