ITSS SOFTWARE DEVELOPMENT/ SOFTWARE DESIGN AND CONSTRUCTION
Lab 08 - System Interface 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 before the next class**: Create a branch named "***release/lab08***" in your GitHub repository and push the full submission for this lab, including in-class tasks and homework assignments, to this branch. Remember to export your diagrams to PNG files and push them with .astah files to GitHub.

2. IN-CLASS ASSIGNMENT

In this section, we will get familiar with the software detailed design process and start with interface 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/InterfaceDesign". After that, push your commit to your individual repository before the announced deadline.

2.1. SYSTEM INTERFACE DESIGN

When the analysis class is complex, such that it appears to embody behaviors that cannot be the responsibility of a single class acting alone, or the responsibilities may need to be reused, the analysis class should be refined into a subsystem. This is a decision based largely on conjecture guided by experience. The actual representation may take a few iterations to stabilize.

As discussed earlier, the use of a subsystem allows the interface to be defined and stabilized, while leaving the design details of the interface implementation to remain hidden while its definition evolves.

The decision to make something a subsystem is often driven by the knowledge and experience of the architect. Since it tends to have a strong effect on the partitioning of the solution space, the decision needs to be made in the context of the whole model. It is the result of more detailed design knowledge, as well as the imposition of constraints imposed by the implementation environment.

When an analysis class is evolved into a subsystem, the responsibilities that were allocated to the "superman" analysis class are then allocated to the subsystem and an associated interface (that is, they are used to define the interface operations). The details of how that subsystem actually implements the responsibilities (that is, the interface operations) is deferred until Subsystem Design.

A subsystem is a model element that has the semantics of a package, such that it can contain other model elements, and a class, such that it has behavior. A subsystem realizes one or more interfaces, which define the behavior it can perform.

A subsystem can be represented as a UML package (that is, a tabbed folder) with the «subsystem» stereotype.

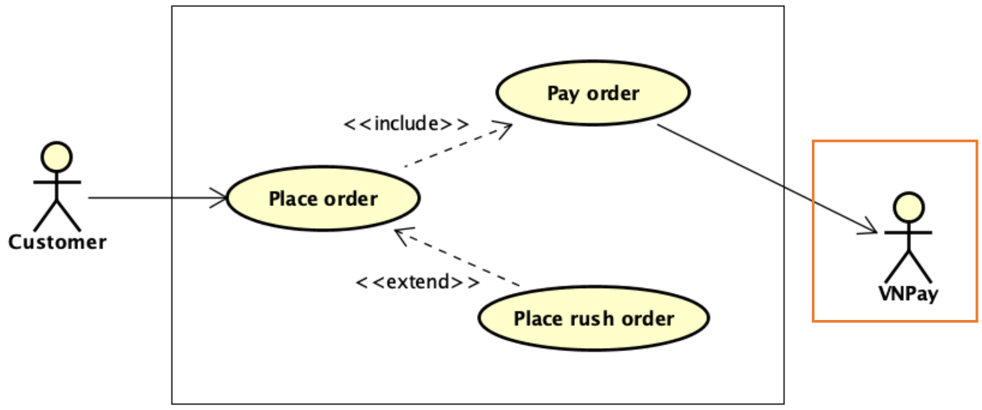
An interface is a model element that defines a set of behaviors (a set of operations) offered by a classifier model element (specifically, a class, subsystem, or component). The relationship between interfaces and classifiers (subsystems) is not always one-to-one. An interface can be realized by multiple classifiers, and a classifier can realize multiple interfaces.

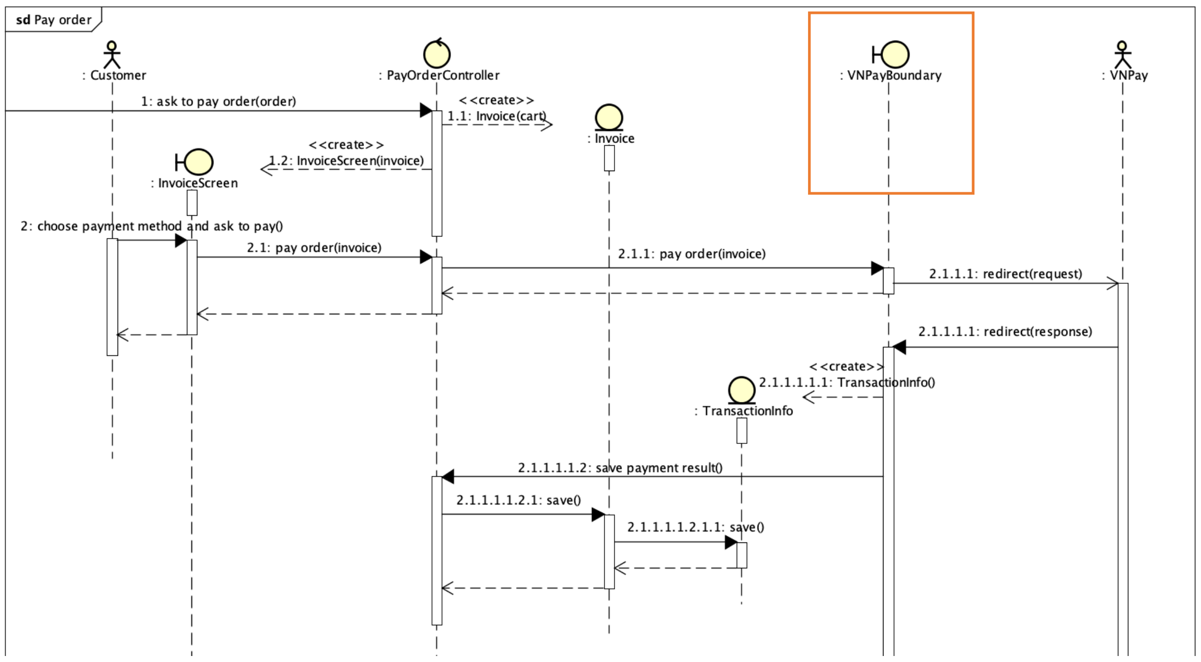
Realization is a semantic relationship between two classifiers. One classifier serves as the contract that the other classifier agrees to carry out.

The realization relationship can be modeled as a dashed line with a hollow arrowhead pointing at the contract classifier (canonical form), or when combined with an interface, as a "ball" (elided form). Thus, in the above example, the two interface/subsystem pairs with the relation between them are synonymous.

Interfaces are a natural evolution from the public classes of a package (described on the previous slide) to abstractions outside the subsystem. Interfaces are pulled out of the subsystem like a kind of antenna, through which the subsystem can receive signals. All classes inside the subsystem are then private and not accessible from the outside.

For AIMS, consider again a part of use case diagram related to place orders. VNPay is a system actor. Therefore, we need to design for the boundary class of this actor, i.e. VNPayBoundary class.




sd Pay order
: Customer
: PayOrderController
: VNPayBoundary
: VNPay
1: ask to pay order(order)
<<create>>
1.1: Invoice(cart)
: Invoice
<<create>>
1.2: InvoiceScreen(invoice)
: InvoiceScreen
2: choose payment method and ask to pay()
2.1: pay order(invoice)
2.1.1: pay order(invoice)
2.1.1.1: redirect(request)
2.1.1.1.1: redirect(response)
<<create>>
2.1.1.1.1.1: TransactionInfo()
: TransactionInfo
2.1.1.1.1.2: save payment result()
2.1.1.1.1.2.1: save()
2.1.1.1.1.2.1.1: save()

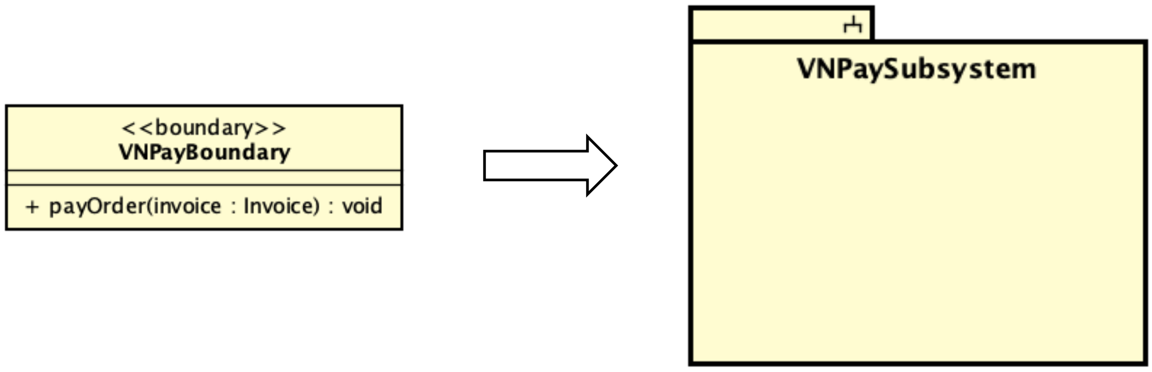
2.2. STEP 1: IDENTIFY SUBSYSTEMS

A subsystem encapsulates its implementation behind one (or more) interfaces. Interfaces isolate the rest of the architecture from the details of the implementation. Operations defined for the interface are implemented by one or more elements contained within the subsystem.

An interface is a pure specification. Interfaces provide the “family of behavior” that a classifier that implements the interface must support. Interfaces are separate things that have separate life spans from the elements that realize them.This separation of interface and implementation exemplifies the OO concepts of modularity and encapsulation, as well as polymorphism. Note: Interfaces are not abstract classes. Abstract classes allow you to provide default behavior for some or all of their methods. Interfaces provide no default behavior.

As mentioned earlier, an interface can be realized by one or more subsystems. Any two subsystems that realize the same interfaces can be substituted for one another. The benefit of this is that, unlike a package, the contents and internal behaviors of a subsystem can change with complete freedom, so long as the subsystem's interfaces remain constant.

As can be seen, the VNPayBoundary class in the analysis class diagram provides complex services related to the communication between AIMS Software and the Interbank while representing an independent capability with clear interface. Hence, we evolve the VNPayBoundary from an analysis class into a subsystem.



2.3. STEP 2: IDENTIFY SUBSYSTEM INTERFACE

Once the subsystems are identified, their interfaces need to be identified.

- **Identify candidate interfaces**. Organize the subsystem responsibilities into groups of cohesive, related responsibilities. These groupings define the initial, first-cut set of interfaces for the subsystem. To start with, identify an operation for each responsibility, complete with parameters and return values.

- **Look for similarities between interfaces**. Look for similar names, similar responsibilities, and similar operations. Extract common operations into a new interface. Be sure to look at existing interfaces as well, re-using them where possible.
- **Define interface dependencies**. Add dependency relationships from the interface to all classes and/or interfaces that appear in the interface operation signatures.
- **Map the interfaces to subsystems**. Create realization associations from the subsystem to the interface(s) it realizes.
- **Define the behavior specified by the interfaces**. If the operations on the interface must be invoked in a particular order, define a state machine that illustrates the publicly visible (or inferred) states that any design element that realizes the interface must support.
- **Package the interfaces**. Interfaces can be managed and controlled independently of the subsystems themselves. Partition the interfaces according to their responsibilities.

Then, you need to make the interface specification, including:

- **Interface name**: Name the interface to reflect the role it plays in the system. The name should be short — one-to-two words. It is not necessary to include the word "interface" in the name; it is implied by the type of model element (that is, interface).
- **Interface description**: The description should convey the responsibilities of the interface. The description should be several sentences long, up to a short paragraph. The description should not simply restate the name of the interface. Instead, it should illuminate the role the interface plays in the system.
- **Operation definition**: Each interface should provide a unique and well-defined set of operations. Operation names should reflect the result of the operation. When an operation sets or gets information, including “set” or “get” in the name of the operation is redundant. Give the operation the same name as the property of the model element that is being set or retrieved. Example: name() returns the name of the object; name(aString) sets the name of the object to aString.

- **Operation description:** Each operation should have a description what the operation does, including any key algorithms, and what value it returns. Name the parameters of the operation to indicate what information is being passed to the operation. Identify the type of the parameter.
- **Interface documentation**: The behavior defined by the interface is specified as a set of operations.

This activity should following steps in the Class Design lab (previous lab).

2.3.1. Interface description

Based on the responsibilities of the VNPaySubsystem, we can design the interface IPayment following the step in the Class design lab.

2.3.2. Operation definition

Client is the class (or superclass of) that send the message payOrder() to the subsystem. Following parameters are general for IPayment, not specific for VNPay:

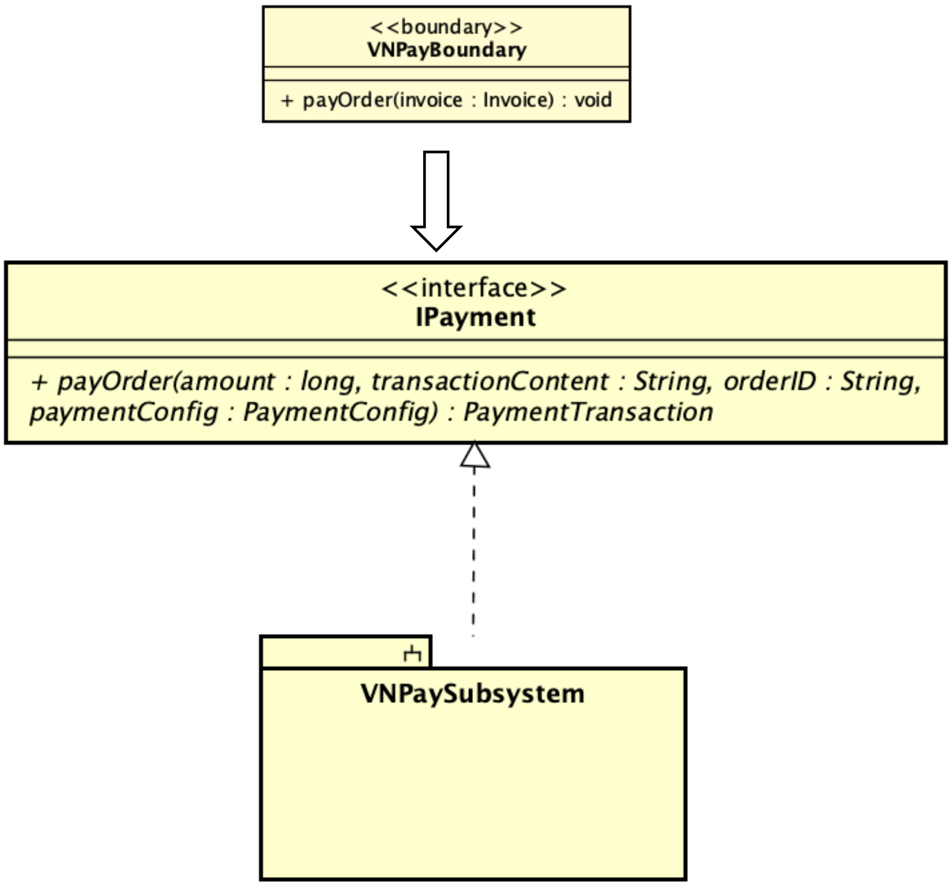
- **amount: double**
- ~~bankCode: String~~
- createdDate: Date
- currency: String
- ipAddress: String
- locale: String
- **transactionContent: String**
- returnURL: String
- expiredDate: DateTime
- **orderID: String**

public payOrder(amount: long, transactionContent: String, orderID: String, paymentConfig: PaymentConfig): PaymentTransaction.

PaymentTransaction includes the following information:

- transactionNo: String
- amount: double
- bankCode: String
- bankTransactionNo: String
- cardType: String
- payDate: DateTime
- transactionContent: String

- orderID: String



<<boundary>>
VNPayBoundary
+ payOrder(invoice : Invoice) : void
<<interface>>
IPayment
+ payOrder(amount : long, transactionContent : String, orderID : String, paymentConfig : PaymentConfig) : PaymentTransaction
VNPaySubsystem

When the transaction is not successful, how can you return the response code for the PlaceOrderController? Is it included in the PaymentTransaction?

VNSubsystem send an error message to the PlaceOrderController? But how?

- Java has a mechanism to handle error by exceptions

→ payOrder throws exceptions?

payOrder() throws BlockedException, NotRegisterOnlinePaymentException, WrongAuthenException …

What happens when VNPay or other payment channels add or remove/change error codes?

→ Generalize different types of exceptions by a superclass: PaymentException

payOrder() throws PaymentException.

2.3.3. Operation description

This activity should following the step of operation design in the Class Design lab.

2.4. STEP 3: SUBSYSTEM DESIGN

We need to design for subsystem to realize the interface. If there is only one class, e.g. InterbankSubsystemController, it will be very complex, which includes many responsibilities. 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.

The last step of System Interface Design is to design for the identified subsystem.

Each subsystem should be as independent as possible from other parts of the system. It should be possible to evolve different parts of the system independently from other parts. This minimizes the impact of changes and eases maintenance efforts.

It should be possible to replace any part of the system with a new part, provided the new part supports the same interfaces. In order to ensure that subsystems are replaceable elements in the model, the following conditions are necessary:

- A subsystem should not expose any of its contents. No element contained by a subsystem should have "public" visibility. No element outside the subsystem should depend on the existence of a particular element inside the subsystem.
- A subsystem should depend only on the interfaces of other model elements, so that it is not directly dependent on any specific model elements outside the subsystem. The exceptions are cases where a number of subsystems share a set of class definitions in common, in which case those subsystems "import" the contents of the packages that contain the common classes. This should be done only with packages in lower layers in the architecture, and only to ensure that common definitions of classes that must pass between subsystems are consistently defined.

- All dependencies on a subsystem should be dependencies on the subsystem interfaces. Clients of a subsystem are dependent on the subsystem interface(s), not on elements within the subsystem. In that way, the subsystem can be replaced by any other subsystem that realizes the same interfaces.

These are the major steps involved in the **Subsystem Design** activity:

- We first must take the responsibilities allocated to the subsystems and further allocate those responsibilities to the subsystem elements.
- Once the subsystem elements have been identified, the internal structure of the subsystems (a.k.a. subsystem element relationships) must be documented.
- Once you know how the subsystem will implement its responsibilities, you need to document the interfaces upon which the subsystem is dependent.
- Finally, we will discuss the kinds of things you should look for when reviewing the results of **Subsystem Design**.

2.4.1. Distribute subsystem behaviors to subsystem elements

You first must take the responsibilities allocated to the subsystems and further allocate those responsibilities to the subsystem elements.

The purpose of this step is to:

- Specify the internal behaviors of the subsystem
- Identify new classes or subsystems needed to satisfy subsystem behavioral requirements.

In designing the internals of the subsystem interactions, you will need to take into consideration the use-case details; the architectural framework, including defined subsystems, their interfaces and dependencies; and the design and implementation mechanisms selected for the system.

Up to this point, you have created interaction diagrams in terms of design elements (that is, design classes and subsystems). In this activity, you will describe the "local" interactions within a subsystem to clarify its internal design.

The external behaviors of the subsystem are defined by the interfaces it realizes. When a subsystem realizes an interface, it makes a commitment to support each and every operation defined by the interface.

The operation may be in turn realized by:

- An operation on a class contained by the subsystem; this operation may require collaboration with other classes or subsystems.
- An operation on an interface realized by a contained subsystem.

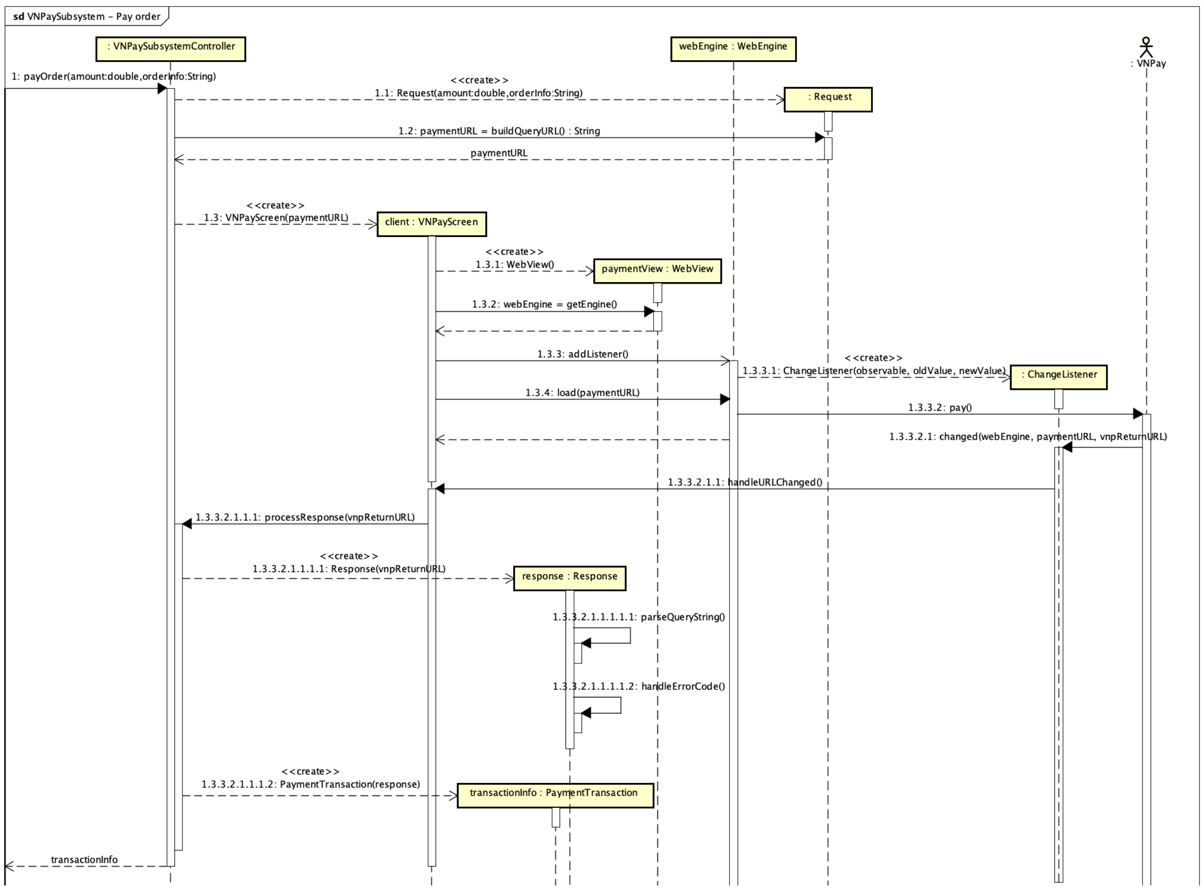
The collaborations of model elements within the subsystem should be documented using sequence diagrams that show how the subsystem behavior is realized. Each operation on an interface realized by the subsystem should have one or more documenting sequence diagrams. This diagram is owned by the subsystem, and is used to design the *internal* behavior of that subsystem.

For each interface operation, identify the classes (or, where the required behavior is complex, a contained subsystem) within the current subsystem that are needed to perform the operation. Create new classes/subsystems where existing classes/subsystems cannot provide the required behavior (but try to reuse first).

Creation of new classes and subsystems should force reconsideration of subsystem content and boundary. Be careful to avoid having effectively the same class in two different subsystems. Existence of such a class implies that the subsystem boundaries may not be well-drawn. Periodically revisit Identify Design Elements to re-balance subsystem responsibilities.

The collaborations of model elements within the subsystem should be documented using interaction diagrams that show how the subsystem behavior is realized. Each operation on an interface realized by the subsystem should have one or more documenting interaction diagrams. This "internal" interaction diagram shows exactly what classes provide the interface, what needs to happen internally to provide the subsystem’s functionality, and which classes send messages out from the subsystem. These diagrams are owned by the subsystem, and are used to design the internal behavior of the subsystem. The diagrams are essential for subsystems with complex internal designs. They also enable the subsystem behavior to be easily understood, rendering it reusable across contexts.

The following is a sample of the sequence diagram of AIMS. You should investigate the VNPay Sandbox document to complete this diagram if necessary, then update the interaction diagrams of the “Pay order” use case in design level, with the update of subsystem instead of analysis classes.


sd VNPaySubsystem – Pay order
: VNPaySubsystemController
webEngine : WebEngine
: VNPay
1: payOrder(amount:double,orderInfo:String)
<<create>>
1.1: Request(amount:double,orderInfo:String)
: Request
1.2: paymentURL = buildQueryURL() : String
paymentURL
<<create>>
1.3: VNPayScreen(paymentURL)
client : VNPayScreen
<<create>>
1.3.1: WebView()
paymentView : WebView
1.3.2: webEngine = getEngine()
1.3.3: addListener()
<<create>>
1.3.3.1: ChangeListener(observable, oldValue, newValue)
: ChangeListener
1.3.4: load(paymentURL)
1.3.3.2: pay()
1.3.3.2.1: changed(webEngine, paymentURL, vnpReturnURL)
1.3.3.2.1.1: handleURLChanged()
1.3.3.2.1.1.1: processResponse(vnpReturnURL)
<<create>>
1.3.3.2.1.1.1.1: Response(vnpReturnURL)
response : Response
1.3.3.2.1.1.1.1.1: parseQueryString()
1.3.3.2.1.1.1.1.2: handleErrorCode()
<<create>>
1.3.3.2.1.1.1.2: PaymentTransaction(response)
transactionInfo : PaymentTransaction
transactionInfo

Please draw the class diagram for this subsystem.

2.4.2. Document subsystem elements

Do the steps of class design lab for all subsystem elements including:

- **Interface/Class name**: Name the interface to reflect the role it plays in the system. The name should be short — one-to-two words. It is not necessary to include the word "interface" in the name; it is implied by the type of model element (that is, interface).
- **Interface/Class description**: The description should convey the responsibilities of the interface. The description should be several sentences long, up to a short paragraph. The description should not simply restate the name of the interface. Instead, it should illuminate the role the interface plays in the system.
- **Attribute/Operation definition**: Each interface should provide a unique and well-defined set of operations. Operation names should reflect the result of the operation. When an operation sets or gets information, including "set" or "get" in the name of the operation is redundant. Give the

operation the same name as the property of the model element that is being set or retrieved. Example: name() returns the name of the object; name(aString) sets the name of the object to aString.

- **Operation/Method description:** Each operation should have a description what the operation does, including any key algorithms, and what value it returns. Name the parameters of the operation to indicate what information is being passed to the operation. Identify the type of the parameter.

a. zDescribe subsystem dependencies

When a subsystem element uses some behavior of an element contained by another subsystem or package, a dependency on the external element is needed.

If the element on which the subsystem is dependent is within a subsystem, the dependency should be on the subsystem interface, not on the subsystem itself or on any element in the subsystem. This allows the design elements to be substituted for one another as long as they offer the same behavior. It also gives the designer total freedom in designing the internal behavior of the subsystem, as long as it provides the correct external behavior. If a model element directly references a model element in another subsystem, the designer is no longer free to remove that model element or redistribute the behavior of that model element to other elements. As a result, the system is more brittle.

If the element the subsystem element is dependent on is within a package, the dependency should be on the package itself. Ideally, a subsystem should only depend on the interfaces of other model elements for the reasons stated above. The exception is where a number of subsystems share a set of common class definitions, in which case those subsystems “import” the contents of the packages containing the common classes. This should be done only with packages in lower layers in the architecture to ensure that common class definitions are defined consistently. The disadvantage is that the subsystem cannot be reused independent of the depended-on package.

The following dependency diagram is an example for VNPay subsystem. The subsystem depends on the exception package, which includes exceptions raising during the payment process. It may not include all necessary dependencies, you have to find all for VNPay subsystem.

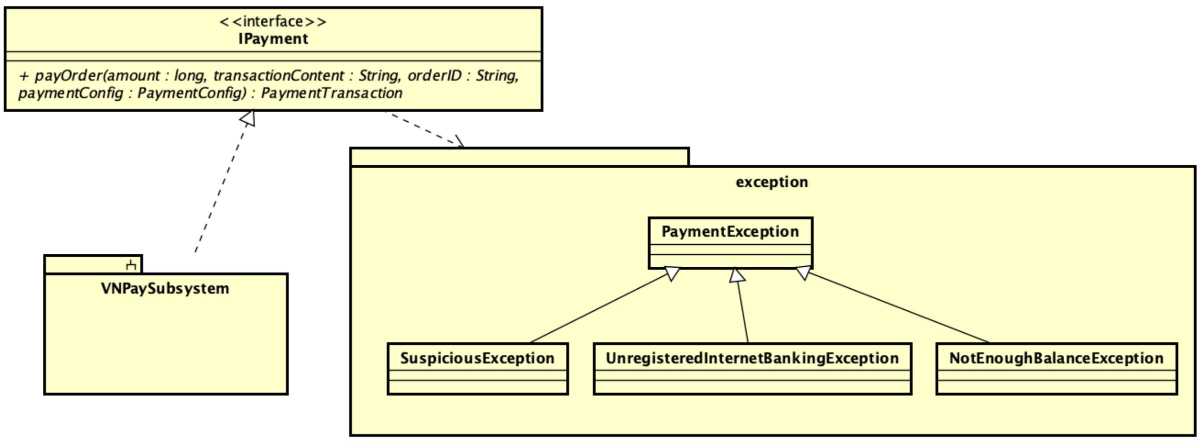
public class PayOrderBoundary {

```
    PayOrderController controller
        = new PayOrderController(new VNPaySubsystem());
    …
    controller.setPaymentMethod(new PaypalSubsystem());
    controller.pay();
}

public class PayOrderController extends Client {
    IPayment payment;

    public PayOrderController(IPayment payment){
        this.payment = payment;
    }

    pay(…){
        payment.payOrder(…,…);
    }
}
```


<<interface>>
IPayment
+ payOrder(amount : long, transactionContent : String, orderID : String, paymentConfig : PaymentConfig) : PaymentTransaction
VNPaySubsystem
exception
PaymentException
SuspiciousException
UnregisteredInternetBankingException
NotEnoughBalanceException

3. ASSIGNMENT

You are asked to read carefully the above guidelines in each step, and complete all necessary requirement for the VNPay subsystem.

When you finish the task of this part, please export your work to PNG files. Then put your work (.astah) and the exported file in the directory "DetailedDesign/InterfaceDesign".