ITSS SOFTWARE DEVELOPMENT / SOFTWARE DESIGN AND CONSTRUCTION
Lab 07 - 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 before the next class**: Create a branch named "***release/lab07***" 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 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. STEP 1: 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. For more complicated classes, you may need to have more classes, even a package or subsystem for one boundary class.

System/device boundary classes

In the next lab, we will learn and practice how to evolve the boundary class for Interbank into a subsystem. We can assume this part as a black box of a subsystem as VNPaySubsystem.

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. Remember to use Generalization/Association or other object-oriented techniques to remove redundancies if any.

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

[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.

design, this can also be reused when communicate with other web information systems). Remember to use Generalization/Association or other object-oriented techniques to remove redundancies if any.

2.2. STEP 2: CLASS DESIGN

Please do the class design for the use case “Place order” including “Pay order” following the below sub steps. Write in the document using SDD template for this design.

2.2.1. Define attributes

Sources of possible attributes:

- Domain knowledge
- Requirements
- Glossary
- Domain Model
- Business Model

Attributes are used instead of classes where:

- Only the value of the information, not it's location, is important
- The information is uniquely "owned" by the object to which it belongs; no other objects refer to the information.
- The information is accessed by operations that only get, set, or perform simple transformations on the information; the information has no "real" behavior other than providing its value.

If, on the other hand, the information has complex behavior, or is shared by two or more objects, the information should be modeled as a separate class.

Attributes are domain-dependent. (An object model for a system includes those characteristics that are relevant for the problem domain being modeled.)

Remember, the process is use-case-driven. Thus, all discovered attributes should support at least one use case. For this reason, the attributes that are discovered are affected by what functionality/domain is being modeled.

Attributes the class needs to carry out its operations and the states of the class are identified during the definition of methods. Attributes provide information storage

for the class instance and are often used to represent the state of the class instance. Any information the class itself maintains is done through its attributes.

You may need to add additional attributes to support the implementation and establish any new relationships that the attributes require.

Check to make sure all attributes are needed. Attributes should be justified — it is easy for attributes to be added early in the process and survive long after they are no longer needed due to shortsightedness. Extra attributes, multiplied by thousands or millions of instances, can have a large effect on the performance and storage requirements of the system.

In analysis, it was sufficient to specify the attribute name only, unless the representation was a requirement that was to constrain the designer.

During Design, for each attribute, define the following:

- Its **name**, which should follow the naming conventions of both the implementation language and the project.
- Its **type**, which will be an elementary data type supported by the implementation language.
- Its **default or initial value**, to which it is initialized when new instances of the class are created.
- Its **visibility**, which will take one of the following values:
    - **Public**: The attribute is visible both inside and outside the package containing the class.
    - **Protected**: The attribute is visible only to the class itself, to its subclasses, or to **friends** of the class (language dependent)
    - **Private**: The attribute is visible only to the class itself and to **friends** of the class.

For persistent classes, also include whether the attribute is persistent (the default) or transient. Even though the class itself may be persistent, not all attributes of the class need to be persistent.

Derived attributes and operations indicate a constraint between values. They can be used to describe a dependency between attribute values that must be

maintained by the class. They do not necessarily mean that the attribute value is always calculated from the other attributes.

2.2.2. Define operations

To identify operations on design classes:

- Study the responsibilities of each corresponding analysis class, creating an operation for each responsibility. Use the description of the responsibility as the initial description of the operation.
- Study the Use-Case Realizations in the class participations to see how the operations are used by the Use-Case Realizations. Extend the operations, one Use-Case Realization at a time, refining the operations, their descriptions, return types and parameters. Each Use-Case Realization's requirements, as regards classes, are textually described in the Flow of Events of the Use-Case Realization.
- Study the use-case Special Requirements, to make sure that you do not miss implicit requirements on the operation that might be stated there.

Use-Case Realizations cannot provide enough information to identify all operations. To find the remaining operations, consider the following:

- Is there a way to initialize a new instance of the class, including connecting it to instances of other classes to which it is associated?
- Is there a need to test to see if two instances of the class are equivalent?
- Is there a need to create a copy of a class instance?
- Are any operations required on the class by mechanisms which they use? (Example: a “garbage collection” mechanism may require that an object be able to drop all of its references to all other objects in order for unused resources to be freed.)

Operations should be named to indicate their outcome. For example, getBalance() versus calculateBalance(). One approach for naming operations that get and set properties is to simply name the operation the same name as the property. If there is a parameter, it sets the property; if not, it returns the current value.

You should name operations from the perspective of the client asking for a service to be performed by the class. For example, getBalance() versus

receiveBalance(). The same applies to the operation descriptions. Descriptions should always be written from the operation USER's perspective. What service does the operation provide?

It is best to specify the operations and their parameters using implementation language syntax and semantics. This way the interfaces will already be specified in terms of the implementation language when coding starts.

In addition to the short description of the parameter, be sure to include things like:

- Whether the parameter should be passed by value or by reference, and if by reference, is the parameter changed by the operation? By value means that the actual object is passed. By reference means that a pointer or reference to the object is passed.
- The signature of the operation defines the interface to objects of that class, and the parameters should therefore be designed to promote and define what that interface is. For example, if a parameter should never be changed by the operation, then design it so that it is not possible to change it — if the implementation environment supports that type of design.
- Whether parameters may be optional and/or have default values when no value is provided.
- Whether the parameter has ranges of valid values.

The fewer parameters you have, the better. Fewer parameters help to promote understandability, as well as maintainability. The more parameters clients need to understand, the more tightly coupled the objects are likely to be conceptually.

One of the strengths of OO is that you can manipulate very rich data structures, complete with associated behavior. Rather than pass around individual data fields (for example, StudentID), strive to pass around the actual object (for example, Student). Then the recipient has access to all the properties and behavior of that object.

Operation visibility is the realization of the key object-orientation principle of encapsulation.

- **Public** members are accessible directly by any client.
- **Protected** members are directly accessible only by instances of subclasses.

- **Private** members are directly accessible only by instances of the class to which they are defined.

How do you decide what visibility to use? Look at the Interaction diagrams on which the operation is referenced. If the message is from outside of the object, use public. If it is from a subclass, use protected. If it's from itself, use private. You should define the most restrictive visibility possible that will still accomplish the objectives of the class. Client access should be granted explicitly by the class and not taken forcibly.

Visibility applies to attributes as well as operations. In the UML, you can specify the access clients have to attributes and operations. Export control is specified for attributes and operations by preceding the name of the member with the following symbols:

- + Public
- # Protected
- - Private

The owner scope of an attribute/operation determines whether or not the attribute/operation appears in each instance of the class (instance scoped), or if there is only one instance for all instances of the class (classifier scoped).

Classifier-scoped attributes and operations are denoted by underlining their names. Lack of an underline indicates instance-scoped attributes and operations.

Classifier-scoped attributes are shared among all instances of the classifier type.

In most cases, attributes and operations are instance scoped. However, if there is a need to have a single instance of an operation, say to generate a unique ID among class instances, or to create a class instance, the classifier scope operations can be used.

Classifier-scoped operations can only access classifier-scoped attributes.

2.2.3. Define methods

A method specifies the implementation of an operation. It describes *how* the operation works, not just *what* it does.

The method, if described, should discuss:

- How operations are to be implemented.

- How attributes are to be implemented and used to implement operations.
- How relationships are to be implemented and used to implement operations.

The requirements will naturally vary from case to case. However, the method specifications for a class should always state:

- What is to be done according to the requirements.
- What other objects and operations are to be used.

More specific requirements may concern:

- How parameters are to be implemented.
- Any special algorithms to be used.

In many cases, where the behavior required by the operation is sufficiently defined by the operation name, description and parameters, the methods are implemented directly in the programming language. Where the implementation of an operation requires use of a specific algorithm, or requires more information than is presented in the operation's description, a separate method description is required.

2.2.4. Draw state machine diagram

The state an object resides in is a computational state, and is defined by the stimuli the object can receive and what operations can be performed as a result. An object that can reside in many computational states is state-controlled. For each class exhibiting state-controlled behavior, describe the relations between an object's states and an object's operations.

A state machine is a tool for describing:

- States the object can assume.
- Events that cause an object to transition from state to state.
- Significant activities and actions that occur as a result.

A state machine is a diagram used to show the life history of a given class, the events that cause a transition from one state to another, and the actions that result from a state change. State machines emphasize the event-ordered behavior of a class instance.

The state space of a given class is the enumeration of all the possible states of an object.

- A **state** is a condition in the life of an object. The state of an object determines its response to different events.
- An **event** is a specific occurrence (in time and space) of a stimulus that can trigger a state transition.
- A **transition** is a change from an originating state to a successor state as a result of some stimulus. The successor state could possibly be the originating state. A transition may take place in response to an event, and can be labeled with an event.
- A **guard condition** is a Boolean expression of attribute values that allows a transition only if the condition is true.
- An **action** is an atomic execution that results in a change in state, or the return of a value.

An **activity** is a non-atomic execution within a state machine.

The initial state is the state entered when an object is created

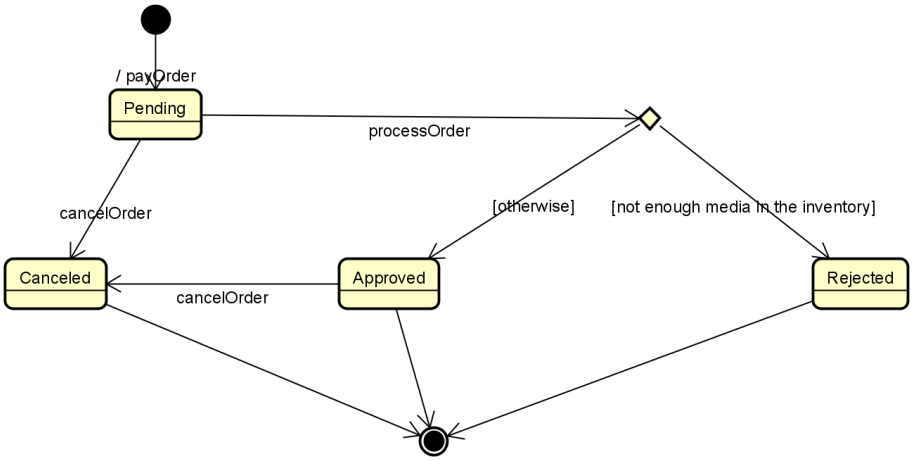
- An initial state is mandatory.
- Only one initial state is permitted.
- The initial state is represented as a solid circle.

A final state indicates the end of life for an object

- A final state is optional.
- More than one final state may exist.
- A final state is indicated by a bull's eye.

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 an example of a state machine diagram for an "Order" object from the moment when a customer places an order until the time an admin processes the order. You need to investigate the problem statement and SRS to make a more comprehensive state machine diagram for this object.



2.3. STEP 3: CLASS DIAGRAM

2.3.1. Define relationships

To find relationships, start studying the links in the Communication diagrams. Links between classes indicate that objects of the two classes need to communicate with one another to perform the use case. Thus, an association or an aggregation is needed between the associated classes.

Reflexive links do not need to be instances of reflexive relationships; an object can send messages to itself. A reflexive relationship is needed when two different objects of the same class need to communicate.

The navigability of the relationship should support the required message direction. In the above example, if navigability was not defined from the Client to the Supplier, then the PerformResponsibility message could not be sent from the Client to the Supplier.

Focus only on associations needed to realize the use cases; do not add associations you think "might" exist unless they are required based on the Interaction diagrams.

Remember to give the associations role names and multiplicities. You can also specify navigability, although this will be refined in Class Design.

Write a brief description of the association to indicate how the association is used, or what relationships the association represents.

Association

Associations represent structural relationships between objects of different classes; they connect instances of two or more classes together for some duration.

You can use associations to show that objects know about other objects. Sometimes, objects must hold references to each other to be able to interact; for example, to send messages to each other. Thus, in some cases, associations may follow from interaction patterns in Sequence diagrams or Communication diagrams.

Most associations are simple (exist between exactly two classes), and are drawn as solid paths connecting pairs of class symbols. Ternary relationships are also possible. Sometimes a class has an association to itself. This does not necessarily mean that an instance of that class has an association to itself; more often, it means that one instance of the class has associations to other instances of the same class.

An association may have a name that is placed on, or adjacent to the association path. The name of the association should reflect the purpose of the relationship and be a verb phrase. The name of an association can be omitted, particularly if role names are used.

Avoid names like "has" and "contains," as they add no information about what the relationships are between the classes.

Each end of an association has a role in relationship to the class on the other end of the association. The role specifies the face that a class presents on each side of the association. A role must have a name, and the role names on opposite sides of the association must be unique. The role name should be a noun indicating the associated object's role in relation to the associating object.

The use of association names and role names is mutually exclusive: one would not use both an association name and a role name. For each association, decide which conveys more information.

The role name is placed next to the end of the association line of the class it describes.

In the case of self-associations, role names are essential to distinguish the purpose for the association.

In the above example, the Professor participates in two separate association relationships, playing a different role in each.

Multiplicity lets you know the lower and the upper bound number of relationships that a given object can have with another object. Many times you do not know what the maximum number of instances may be, and you will use the "*" to specify that this number is unknown.

The most important question that multiplicity answers: Is the association mandatory? A lower bound number that is greater than zero indicates that the relationship is mandatory.

Multiplicity

Multiplicity can be defined as: The number of instances one class relates to one instance of another class.

- For each role, you can specify the multiplicity of its class and how many objects of the class can be associated with one object of the other class.
- Multiplicity is indicated by a text expression on the role. The expression is a comma-separated list of integer ranges.
- It is important to remember that multiplicity is referring to instances of classes (objects) and their relationships.
- Multiplicity must be defined on both ends of the association.

Multiplicity is indicated by a text expression on the role.

- The expression is a comma-separated list of integer ranges.
- A range is indicated by an integer (the lower value), two dots, and an integer (the upper value).
- A single integer is a valid range, and the symbol "*" indicates "many." That is, an asterisk "*" indicates an unlimited number of objects.
- The symbol "*"by itself is equivalent to "0..*" That is, it represents any number including none. This is the default value.
- Optional value has the multiplicity 0..1.

Aggregation

Aggregation is a stronger form of association which is used to model a whole-part relationship between model elements. The whole/aggregate has an aggregation association to the its constituent parts. A hollow diamond is attached to the end of an association path on the side of the aggregate (the whole) to indicate aggregation.

Since aggregation is a special form of association, the use of multiplicity, roles, navigation, and so forth is the same as for association.

Sometimes a class may be aggregated with itself. This does not mean that an instance of that class is composed of itself (that would be silly). Instead, it means that one instance if the class is an aggregate composed of other instances of the same class.

Some situations where aggregation may be appropriate include:

- An object is physically composed of other objects (for example, car being physically composed of an engine and four wheels).
- An object is a logical collection of other objects (for example, a family is a collection of parents and children).
- An object physically contains other objects (for example, an airplane physically contains a pilot).

Aggregation should be used only where the "parts" are incomplete outside the context of the whole. If the classes can have independent identity outside the context provided by other classes, or if they are not parts of some greater whole, then the association relationship should be used.

When in doubt, an association is more appropriate. Aggregations are generally obvious. A good aggregate should be a natural, coherent part of the model. The meaning of aggregates should be simple to understand from the context. Choosing aggregation is only done to help clarify; it is not something that is crucial to the success of the modeling effort.

The use of aggregation versus association is dependent on the application you are developing. For example, if you are modeling a car dealership, the relationship between a car and the doors might be modeled as aggregation, because the car always comes with doors, and doors are never sold separately. However, if you are

modeling a car parts store, you might model the relationship as an association, as the car (the body) might be independent of the doors.

Navigability

The navigability property on a role indicates that it is possible to navigate from an associating class to the target class using the association. This may be implemented in a number of ways: by direct object references, by associative arrays, hash-tables, or any other implementation technique that allows one object to reference another.

- Navigability is indicated by an open arrow placed on the target end of the association line next to the target class (the one being navigated to). The default value of the navigability property is true (associations are bi-directional by default).
- When no arrowheads are shown, the association is assumed to be navigable in both directions.

During Use-Case Analysis, associations (and aggregations) may have been assumed to be bi-directional (that is, communication may occur in both directions).

- During **Class Design**, the association's navigability needs to be refined so that only the required communication gets implemented.
- Navigation is readdressed in **Class Design** because we now understand the responsibilities and collaborations of the classes better than we did in Use-Case Analysis. We also want to refine the relationships between classes.Two-way relationships are more difficult and expensive to implement than one-way relationships. Thus, one of the goals in **Class Design** is the reduction of two-way (bi-directional) relationships into one-way (unidirectional) relationships.
- The need for navigation is revealed by the use cases and scenarios. The navigability defined in the class model must support the message structure designed in the interaction diagrams.
- In some circumstances even if two-way navigation is required, a one-way association may suffice. For example, you can use one-way association if navigation in one of the directions is very infrequent and does not have

stringent performance requirements, or the number of instances of one of the classes is very small.

Dependency

During Analysis, we assumed that all relationships were structural (associations or aggregations). In Design, we must decide what type of communication pathway is required.

A dependency relationship denotes a semantic relationship between model elements, where a change in the supplier may cause a change in the client.

There are four options for creating a communication pathway to a supplier object:

- **Global**: The supplier object is a global object. Is the receiver a "global?" If so, establish a dependency between the sender and receiver classes in a class diagram containing the two classes.
- **Parameter**: The supplier object is a parameter to, or the return class of, an operation in the client object. Is the reference to the receiver passed as a parameter to the operation? If so, establish a dependency between the sender and receiver classes in a Class diagram containing the two classes.
- **Local**: The supplier object is declared locally (that is, created temporarily during execution of an operation). Is the receiver a temporary object created and destroyed during the operation itself? If so, establish a dependency between the sender and receiver classes in a class diagram containing the two classes.
- **Field**: The supplier object is a data member in the client object.

A dependency is a type of communication pathway that is a transient type of relationship.These occur when the visibility is global, parameter, or local.

Look at each association relationship and determine whether it should remain an association or become a dependency. Associations and aggregations are structural relationships (field visibility). Association relationships are realized by variables that exist in the data member section of the class definition. Any other relationships (global, local, and parameter visibility) are dependency relationships.

According to the UML 2 Specification, a link is an instance of an association. Specifically, an association declares a connection (link) between instances of the

associated classifiers (classes). A dependency relationship indicates that the implementation or functioning of one or more elements requires the presence of one or more other elements.

Note that links modeled as parameter, global, or local are transient links. They exist only for a limited duration and in the specific context of the collaboration; in that sense, they are instances of the association role in the collaboration. However, the relationship in a class model (independent of context) should be, as stated above, a dependency. In *The UML Reference Manual,* the definition of a transient link is: "It is possible to model all such links as associations, but then the conditions on the associations must be stated very broadly, and they lose much of their precision in constraining combinations of objects." In this situation, the modeling of a dependency is less important than the modeling of the relationship in the collaboration, because the dependency does not describe the relationship completely, only that it exists.

If you believe that a link in a collaboration is a transient link, it indicates that the context-independent relationship between the classes is a dependency. Dependency is a “summary” relationship — for details you have to consult the behavioral model.

Context is defined as "a view of a set of related modeling elements for a particular purpose."

Generalization

Generalization can be defined as: A specialization/generalization relationship, in which objects of the specialized element (the child) are substitutable for objects of the generalized element (the parent). (*The Unified Modeling Language User Guide*, Booch, 1999.)

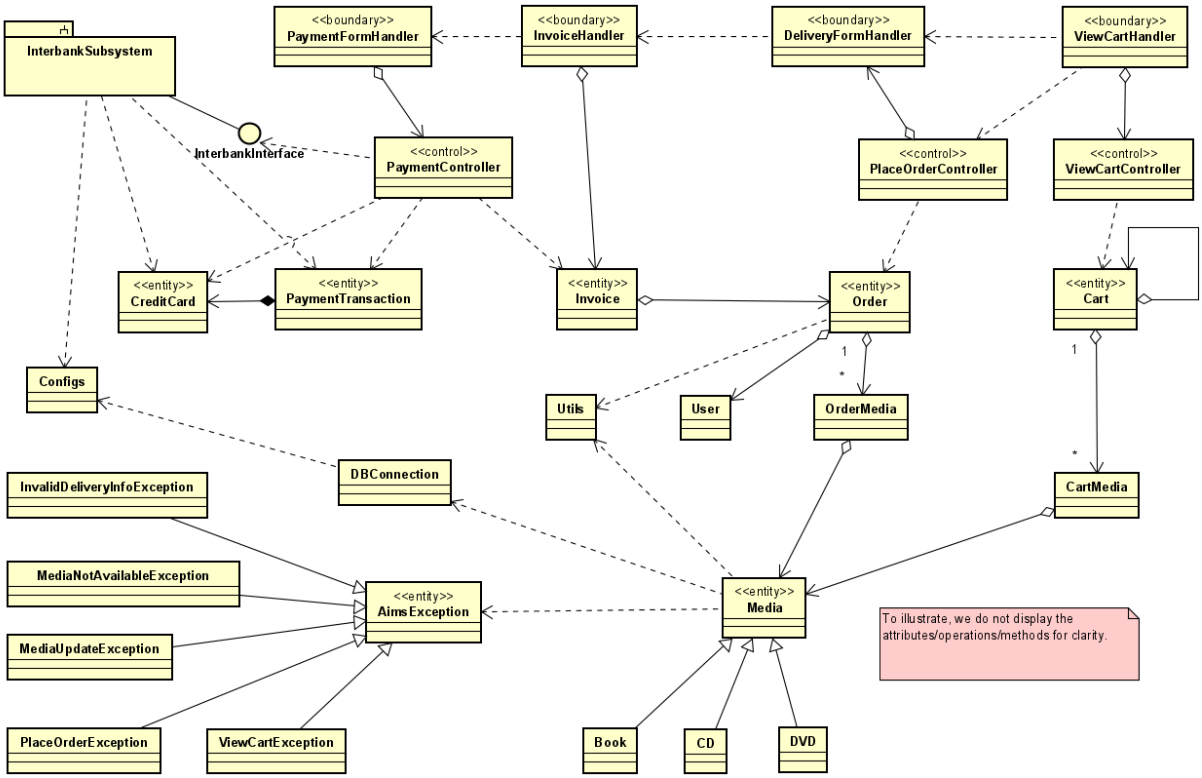
- The subclass may be used where the superclass is used, but not vice versa. The child inherits from the parent.
- Generalization is transitive. You can always test your generalization by applying the “is a kind of” rule. You should always be able to say that your specialized class “is a kind of” the parent class.
- The terms “generalization” and “inheritance” are generally interchangeable, but if you need to distinguish, generalization is the name of the relationship.

Inheritance is the mechanism that the generalization relationship represents/models.

Inheritance can be defined as: The mechanism by which more-specific elements incorporate the structure and behavior of more-general elements. (*The Unified Modeling Language User Guide*, Booch, 1999.)

- Single inheritance: The subclass inherits from only one superclass (has only one parent).
- Multiple inheritance: The subclass inherits from more than one superclass (has multiple parents).

To illustrate, the following figure shows an example of relationships for AIMS among classes. You should put them into corresponding packages.



2.3.2. Define packages

A **package** is a general purpose mechanism for organizing elements into groups. They provide the ability to organize the model under development. A package is represented as a tabbed folder.

Later in this module, we will contrast “vanilla” packages, as defined above, with subsystems.

When identifying classes, you should group them into packages, for organizational and configuration management purposes.

The Design Model can be structured into smaller units to make it easier to understand. By grouping Design Model elements into packages and subsystems, then showing how those groupings relate to one another, it is easier to understand the overall structure of the model.

You might want to partition the Design Model for a number of reasons:

- You can use packages and subsystems as order, configuration, or delivery units when a system is finished.
- Allocation of resources and the competence of different development teams might require that the project be divided among different groups at different sites.
- Subsystems can be used to structure the Design Model in a way that reflects the user types. Many change requirements originate from users; subsystems ensure that changes from a particular user type will affect only the parts of the system that correspond to that user type.
- Subsystems are used to represent the existing products and services that the system uses.

When the boundary classes are distributed to packages, there are two different strategies that can be applied. Which one to choose depends on whether or not the system interfaces are likely to change greatly in the future.

- If it is *likely* that the system interface will be replaced, or undergo considerable changes, the interface should be separated from the rest of the Design Model. When the user interface is changed, only these packages are affected. An example of such a major change is the switch from a line-oriented interface to a window-oriented interface.
- If no major interface changes are planned, changes to the system services should be the guiding principle, rather than changes to the interface. The boundary classes should then be placed together with the entity and control

classes with which they are functionally related. This way, it will be easy to see what boundary classes are affected if a certain entity or control class is changed.

Mandatory boundary classes that are not functionally related to any entity or control classes, should be placed in separate packages, together with boundary classes that belong to the same interface.

If a boundary class is related to an optional service, group it in a separate subsystem with the classes that collaborate to provide the service. The subsystem will map onto an optional component that will be provided when the optional functionality is ordered.

A package should be identified for each group of classes that are functionally related. There are several practical criteria that can be applied when judging if two classes are functionally related. These are, in order of diminishing importance:

- If changes in one class' behavior and/or structure necessitate changes in another class, the two classes are functionally related.
- It is possible to find out if one class is functionally related to another by beginning with a class — for example, an entity class — and examining the impact of it being removed from the system. Any classes that become superfluous as a result of a class removal are somehow connected to the removed class. By superfluous, we mean that the class is only used by the removed class, or is itself dependent upon the removed class.
- Two objects can be functionally related if they interact with a large number of messages, or have an otherwise complicated intercommunication.
- A boundary class can be functionally related to a particular entity class if the function of the boundary class is to present the entity class.
- Two classes can be functionally related if they interact with, or are affected by changes in, the same actor. If two classes do not involve the same actor, they should not lie in the same package. The last rule can, of course, be ignored for more important reasons.
- Two classes can be functionally related if they have relationships between each other (associations, aggregations, and so on). Of course, this criterion cannot be followed mindlessly but can be used when no other criterion is applicable.
- A class can be functionally related to the class that creates instances of it.

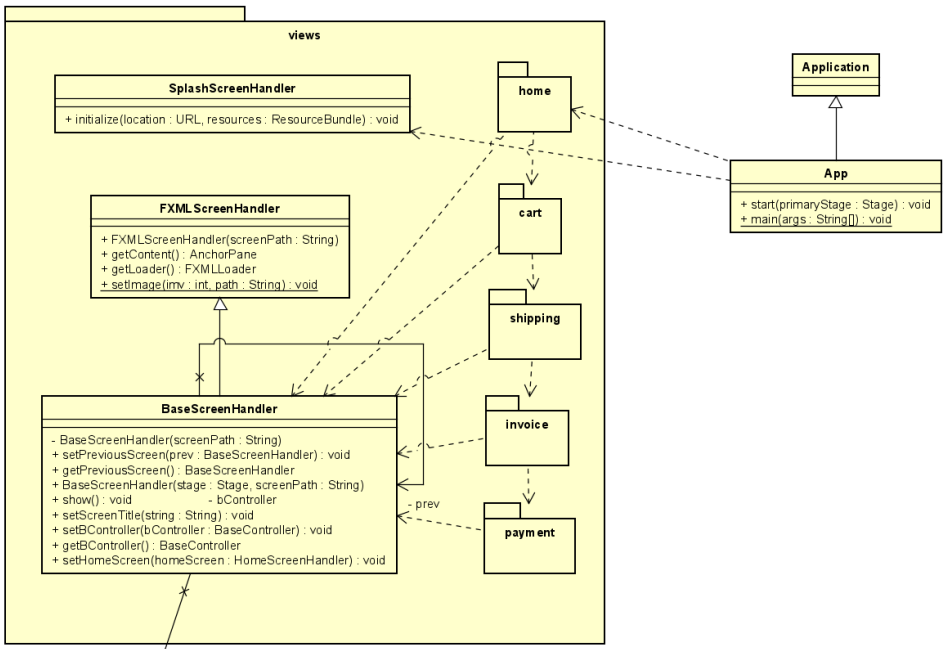
These two criteria determine when two classes should **not** be placed in the same package:

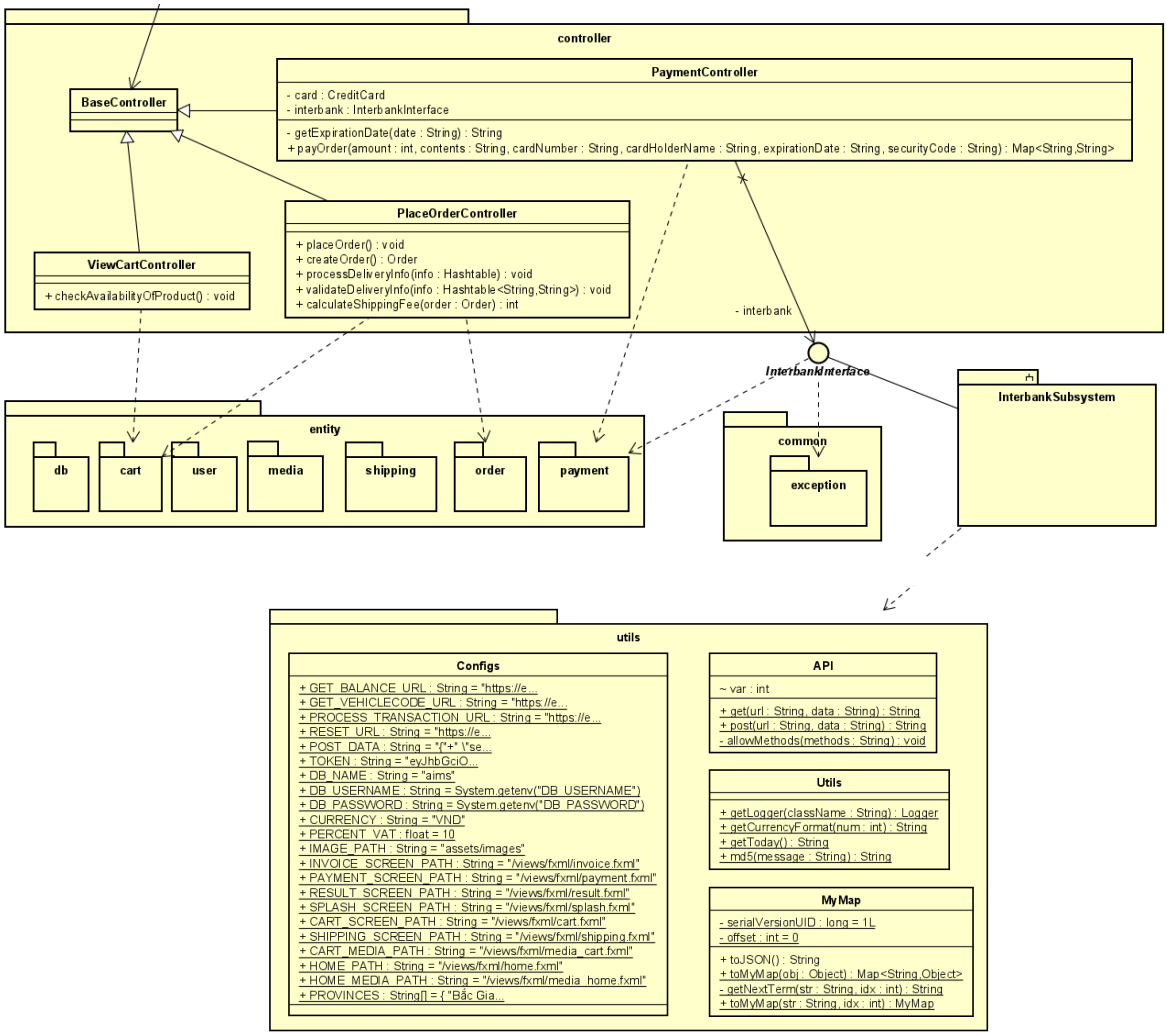
- Two classes that are related to different actors should not be placed in the same package.
- An optional and a mandatory class should not be placed in the same package.

2.3.3. Draw general class diagram

If you can put everything in a detailed class diagram (attribute, operation, relationship), you may not need to draw a general class diagram.

If there are too many classes with detailed properties, you may need to draw a general class diagram, which includes main design classes, packages and subsystems. You don't need to show the detailed members in packages or subsystems that you draw in the general class diagram.





2.3.4. Detailed Class Diagram

You should have a detail class diagram for each part/package/subsystem in your software that is not shown detail in the general class diagram.

3. ASSIGNMENT

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

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.