OOD dev guidelines for homeworks

Disclaimer

All Programming tasks in Java (or any other OO language) should consist of 3 main parts, and their order should be kept, as follows:

1. **Object Oriented Analysis:** during this phase the Programmer tries to understand the problem at hand. He/she dissects the problem into objects, actions and relationships. Once the objects and their business (domain/responsibility) is clear, validation rules can be set up that can be turned into tests.
2. **Object Oriented Design:** during this phase the relationships between objects is very precisely defined and helpful diagrams are sketched up that describe the system.
3. **Object Oriented Programming:** during this phase the programmer actually starts writing code, applying all the rules that have been identified during the OOA and OOD phases. It is very common for the programmer to realize that the initial understanding of the problem and the chosen design might not be appropriate, and requires tweaking. Thus some additional OOA and OOD phases might be required.

Every programmer will do all of these 3, at the same time, intermingled, while writing the code itself, **however** before he/she starts writing any code a thought-process that is focused on OOA and OOD should preside any other actions.

Below you can find a Plan that will cover the basic steps of OOA (1 through 5 and 7) and OOD (5 and 6) that can be applied on simple tasks.

NOTE: this is not a recipe / action-plan that applies to any and all software design tasks.

Plan
Once the exercise / requirements are read and understood try to go through the following steps:

1. Identify the **entities**/objects of the problem -> these are usually the **nouns** that stand out.
2. Identify the **actions** / behaviors that can happen in the system -> these are usually the **verbs** that stand out.

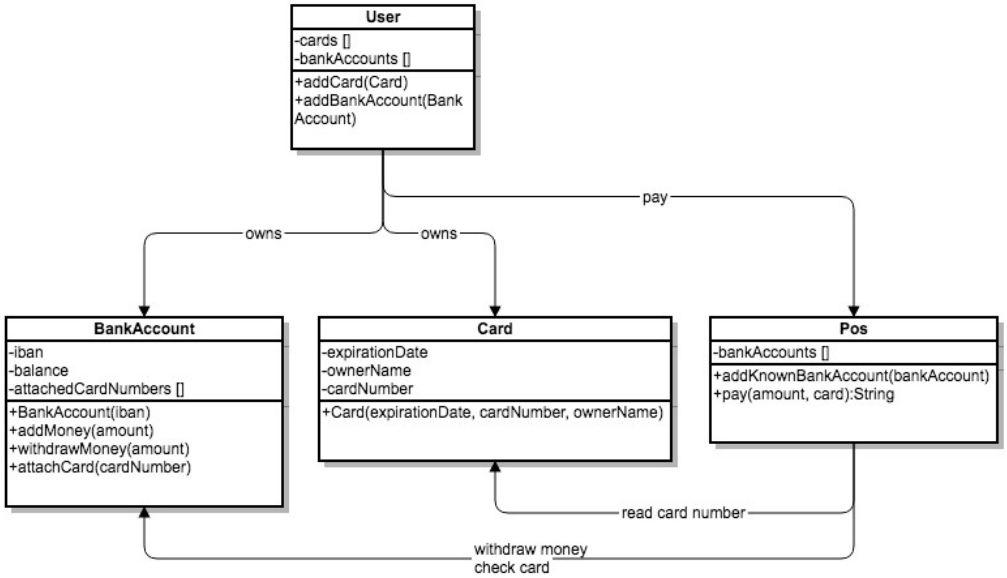
3. Identify the **exceptional flows** of the system.
4. Identify **relationships** between the entities that are stated in the requirements.
5. Once we have identified the **entities**, **actions**, **exceptional flows** and **relationships** we can start thinking about OO Design -> we can start sketching up Classes with relationships between them. Try asking and answering the following questions for each entity:
    - Should this entity be a class, or can it be just a simple class member (i.e a primitive or standard object, like String)?
    - Do I have **is-a** relationships between classes? This means that we can either use **inheritance** or extract **interfaces** between these classes
    - Do we have data that needs to be **encapsulated**?
    - Are there relationships between classes that should be modeled as **aggregation** or **composition**?
6. Once we answer the questions we can start to draw up a class diagram (on paper or in special UML Tools, like gliffy or astah).
7. Based on all of the above we can identify test-cases that can validate that the system works correctly. Based on these we can write Unit and Integration Tests. Try to formulate the statements around the <given><when><then> keywords.

Example
Bank payments

A user can have multiple bank accounts.
A user can have cards emitted to some of the bank accounts.
When the user pays with the card at a POS he receives a receipt if the transaction with the bank account was successful. Otherwise the POS prints an error receipt. Errors occur if the POS does not know about the bank account of the card, if the card is expired or if there is not enough money in the account.

1. Identify the **entities**/objects of the problem -> these are usually the **nouns** that stand out
    - Identified entities: *User, BankAccount, Card, Pos, Receipt*
    - In this step we just identified the entities. We did not create any relationships between them, nor did we attach any actions/behaviors to them. This will follow next.
    - Also, any other entities that are needed will be done during object oriented design process.
2. Identify the **actions** / behaviors that can happen in the system -> these are usually the **verbs** that stand out
    - Identified actions/behaviors: *pay, transact, print receipt, print error receipt.*
3. Identify the **exceptional flows** of the system.
    - Payment will not work if the card is expired
    - Payment will not work if there is not enough money in the account
    - Payment will not work if the Pos can't find the bank account

4. Identify **relationships** between the entities that are stated in the requirements:
    - *User* has *BankAccounts*
    - *User* has *Cards*
    - *Cards* are emitted to some of the *BankAccounts*
    - *User* interacts with *Pos* via a pay action
    - *Pos* prints *Receipts* for both successful payments and failed payments
    - *Pos* interacts with *BankAccounts* via transactions
5. Once we have identified the **entities**, **actions**, **exceptional flows** and **relationships** we can start thinking about OO Design -> we can start sketching up Classes with relationships between them. Try asking and answering the following questions for each entity:
    - Should this entity be a class, or can it be just a simple class member (i.e a primitive or standard object, like String)?
        - User needs to be a class, because it is a complex object that owns other objects (cards, bank accounts)
        - Card needs to be a class, because bank cards have a lot of information that they own and maintain (card nr, card owner name, expiration date, etc). We need card-nr to link the card to a bank-account.
        - Bank Accounts needs to be a class, because they have a lot of information that they own and maintain (balance, iban, card-numbers attached to the account)
        - Pos needs to be a class because it maintains relationships with bank accounts (initiates a transaction) and exposes behavior (i.e payments, receipt printing)
        - Receipts can be simple Strings, because for the sake of the example we just need them to show some text.
    - Do I have **is-a** relationships between classes? This means that we can either use **inheritance** or extract **interfaces** between these classes
        - There aren't any in this system.
    - Do we have data that needs to be **encapsulated**?
        - The communication/transaction between the Pos and the Bank Account can be encapsulated in the Pos. As a real life person when you make a payment you don't see the Pos talking to the bank, and how it does it.
    - Are there relationships between classes that should be modeled as **aggregation** or **composition**?
        - All relationships between the major entities should be aggregation (i.e passing of params via constructor or methods).
        - Users aggregate cards and bank account
        - Cards compose the card-nr, card owner name and expiration date.
        - Bank accounts compose the IBAN, and aggregate balance and the card-numbers of the cards attached to them.
        - Pos aggregates bank accounts.
6. Once we answer the questions we can start to draw up a class diagram (on paper or in special UML Tools, like gliffy or astah).



7. Based on all of the above we can identify test-cases that can validate that the system works correctly. Based on these we can write Unit and Integration Tests. Try to formulate the statements around the <given><when><then> keywords.
    - Integration tests:
    - <given> If user makes payment at pos <when> and has enough money <then> money is withdrawn from the bank account and a receipt is printed with the payed sum.
    - <given> If user makes payment at pos <when> and does not have enough money <then> money is not withdrawn and an error receipt is printed.
    - <given> If user makes payment at pos <when> and the card is expired <then> money is not withdrawn and an error receipt is printed.
    - <given> If user makes payment at pos <when> and the pos does not know about the bank account <then> withdrawn and an error receipt is printed.
    - Unit tests: you need to write tests for the methods of the classes, where they make sense (i.e set/get methods don't need testing if they don't do anything special)

N.B:
Composition vs Aggregation:

A "owns" B = Composition : B has no meaning or purpose in the system without A
A "uses" B = Aggregation : B exists independently (conceptually) from A

e.g: A Company is an aggregation of People. A Company is a composition of Accounts. When a Company ceases to do business its Accounts cease to exist but its People continue to exist.