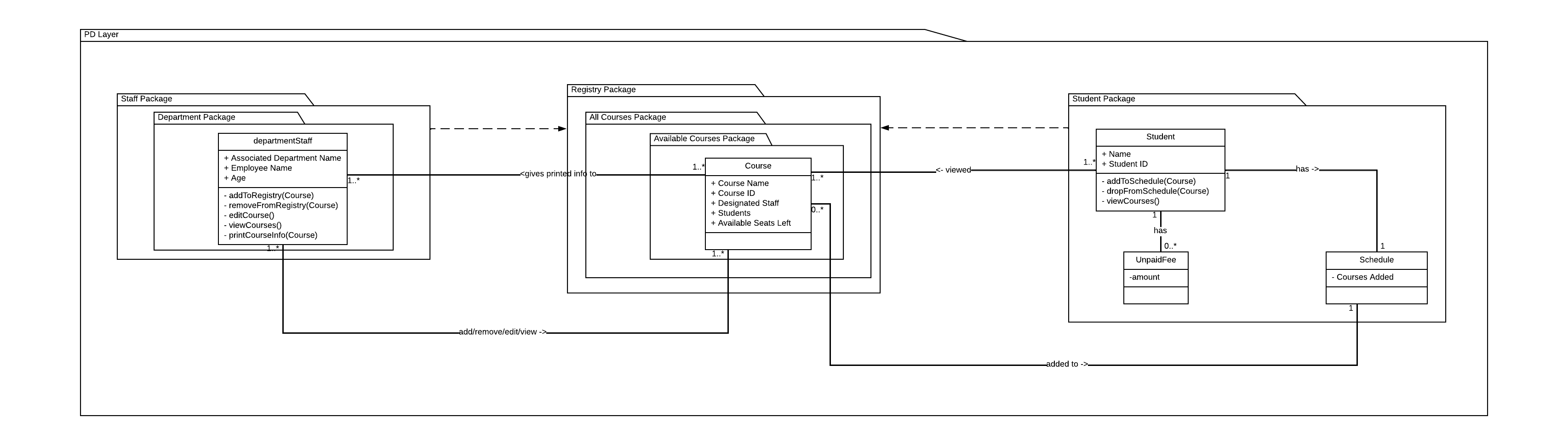
1. .
   1. Regarding the verification and validation walkthrough of the university registration system models we created in Assignments 2 and 3, we have only made models relating to functional models (activity diagrams, use case diagrams, and use case descriptions) and behavioral models (communication diagrams, sequence diagrams, and behavioral state machines).

First, the sequence and communication diagrams must be associated with use cases from the use case diagram and use case descriptions. There are more behavioral models than there are listed use cases in the use case diagram so adjustments must be made to the use case diagram to include the additional use cases from the behavioral model.

Second, actors on the sequence diagrams and the communication diagrams must be associated to the actors mentioned in the use case diagram and use case descriptions. This is true for the behavioral models since they include at least one of the two actors mentioned in the use case diagram.

Third, the messages of the sequence and communication diagrams, and transitions on the behavioral state machine must be related to the events portrayed on the activity diagrams and in the use case descriptions. Even though there is a variation in the order of how the events take place in the activity diagrams and the sequence/communication diagrams, the events/activities of the functional models are related to the messages and transition of the behavioral models.

Fourth, all complex objects represented in the object node of an activity diagram must have a behavioral state machine to represent the object’s lifecycle. Although there is a behavioral state machine to represent the object of a Course mentioned during the activity diagrams, the behavioral state machine could be more detailed to include each possible state that a Course object could experience rather than categorizing multiple states as one final state.



1. The second class diagram is better than the first one. While the first one does describe that the Employee class and the Name-Address class are a part of the Person class, it does not specify the relationship for why they are a part of the Person class. In the second class diagram, we can see that the Name-Address class has an aggregation relationship with the Person class showing that it is a logical part of the Person class. We can also see why the Employee class was connected to the Name-Address class in the first class diagram. Since the Name-Address class is tied to the Person class, the Employee class does not need to have a relationship with the Name-Address class. Also with the inclusion of multiplicities, we can see that for each object of the Name-Address class, there is either zero or multiple objects from the Person class tied to each one. The second class diagram provides a more detailed look at the relationships between the classes in the diagram.
2. The Chain of Responsibility design pattern is used to delegate big tasks to other handlers, with each serving the specific purpose to perform a smaller task. A simple example would be an ATM giving out bills in 100s, 50s, 20s, 10s, 5s, and 1s. When the client requests to withdraw a specific amount of money, the ATM will go through a process once approved to point to a handler of that type of bill from the listed handlers and recursively repeat the cycle until the amount has been given. For instance, if a client were to withdraw $147, the ATM will start with the $100 handler, the handler will see that it’s possible to give $100 once, passes over to the next handler, and repeats the cycle after the last handler passes. Text

   Description automatically generated

The Observer design pattern is used to have one class store the data for other classes to see and update when the class that stores the data updates. A simple example provided below is if a class that stores the number 0 changes that number to a 1 or 2, all the other classes that use that number will update as well. Text

Description automatically generatedText

Description automatically generated