Sahib Bajwa
107553096
2 September 2020

1. Provide definitions for each of the following terms:
    - Abstraction: Fields or concepts of an entity that describes its possible functions (Lecture 4). This term applies to the notion of a class due to us wanting to know how a function/class operates or what it is supposedly supposed to do. A good use of this would be if you want to use or replicate a class/function, it would be simple with good abstraction. A bad use of this would be making a function/class purposefully difficult to understand and replicate, which would cause confusion and difficulty.
    - Encapsulation: Hides how something is developed or how something works from an outside operator (Lecture 4). This term applies to the notion of a class due to us possibly not wanting out techniques used in design to be altered by an outside function or operator. If there is a function that should never be changed from the outside, we would want it to be unmanageable from the outside (without the use of a specific function or class created to do that operation). A good use of this would be not allowing a function to change a variable in a function without calling a function that is specifically designed to make sure the variable is changed correctly. A bad use of this would be encapsulating a class where a variable needs to be changed and not including a function/class to change the variable, making it unchangeable.
    - Polymorphism: “Being able to refer to different derivations of a class in the same way, but getting the behavior appropriate to the derived class being referred to” (Shalloway/Trott, Lecture 4). This term applies to the notion of a class because a class can take on many attributes, but making sure that the class returns the specific thing you want is important. A good use of this would be allowing a function/class to return two different fields but only using the correct one that you require. A bad use of this would be only returning one field from the class and then creating an entire new class to retrieve the other field that could have been taken from the first class.

- Cohesion: How well a function or operation is at completing its task (Lecture 4). This term applies to the notion of a class because it is the idea of a class should complete a certain task and that's why the class is created in the first place. A good use of this would be allowing a class to focus on completing on one task and calling that class when we want that task returned. A bad use of this would be a class trying to complete multiple tasks even though we would not want all the different tasks returned together in a practical situation.
- Coupling: "The strength of a connection between two routines" (Lecture 4). This term applies to the notion of a class because if there are two classes with strong coupling, changing one class will alter the other. A good use of this would be two classes that share a variable that needs to always be updated with each call of either class. A bad use of this would be two classes that share a variable that is not supposed to be altered by both classes and updated with each call of either class.
- Identity: Each object has an identity that is used to refer to it for operations. A unique identifier (Lecture 5). This term applies to the notion of a class because classes need to be called so they require a unique identifier (this is also the same with variables and functions within classes). A good example of this is calling a specific variable when there are many variables that might have similar values or names. A bad example of this would be having no identifiers and calling the wrong variable due to having no distinct way of identifying variables.

2. Develop a design for this system using the functional decomposition approach:
    - The functional decomposition approach is breaking down a task into its most basic functions or steps so that we can see the many small steps taken to complete the larger task.
    - Some assumptions I am making:
        i. Joining automatically enrolls the customer in the loyalty program, mass emails, and discount codes.
        ii. You need to connect to the loyalty system in order to check if a customer is part of or has left the loyalty program (like connecting to database on slide 11)

iii. Leaving automatically removes the customer from the loyalty program, mass emails, and discount codes.

- Connect to Loyalty Program database
- Loop through list of registered customers
- Detect money spent by each user each month
- Email sales information and individual discount coupons based on monthly spending
- Disconnect from Loyalty Program database

3. Develop a design for the same customer loyalty system described in question 2 using the object-oriented approach:
    - Classes I would include in my design and their responsibilities:
        i. connectTo – This class would connect to the Loyalty Program database
        ii. loopThrough – This class would loop through the list of customers in the database
        iii. returnData – This class would return the relevant monthly spending and sales information for a customer
        iv. returnCoup – This class would return the individual coupon for the customer based on monthly spending
        v. emailInfo – This class would send an email to the customer containing sales information and individual discount coupons based on monthly spending
        vi. disconnectFrom – This class would disconnect from the Loyalty Program database
    - I would instantiate the objects in the order above
    - returnData would work with loopThrough in order to return the relevant customer data for the customer that has come up in the loop in loopThrough
    - returnCoup would return the coupon that is personalized for the customer based on their monthly spending
    - emailInfo would use returnData and reternCoup to send the personalized email to the customer that has come up within loopThrough

4. In lecture 5, we present “design by contract” in OOAD as an implicit contract.
    - The contract is what the class and/or its public methods promise to complete/keep (Lecture 5).
    - If we follow the contract, then all the tasks we set to be completed before creating the class and its public methods will be taken care of/completed.
    - If we break the contract, we run into the problem of the class not completing all tasks we wanted it to complete in the first place.
    - We can make the contract explicit by forcing the classes/public methods to have forced compliance. This can be seen “where the specification for a routine can be explicitly written in code” (Lecture 5, https://fantom.org/doc/docLang/TypeSystem#:~:text=The%20explicit%20contract%20specifies%20what,cannot%20do%20with%20a%20File%20.).