Willie Chew
James Douthit
Sricharan Reddy Varra

CSCI 4448
Project 0
9/2/20

1. Provide definitions for each of the following terms. Remember to cite sources, even if the source is the textbook. You may want to look for OO books (including the textbook) on the CU Libraries link to O'Reilly-Safari e-books, under the Sciences tab here: https://libguides.colorado.edu/strategies/ebooks, please note that Wikipedia is not usable as a primary source.

- **abstraction**
- **encapsulation**
- **polymorphism**
- **cohesion**
- **coupling**
- **identity**

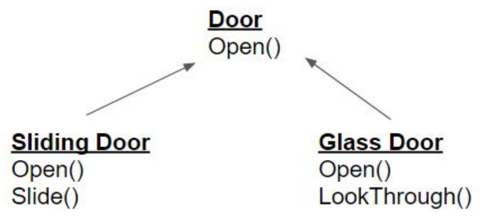
Besides the definition, discuss how the term applies to the object-oriented notion of a class. Provide examples of both good and bad uses of these terms in the design of a class or a set of classes (can be code, psuedo-code, a text, or a graphic example).

Abstraction :
According to the Lecture Slides 4(OO Paradigm), abstraction is a set of concepts that some entity provides you in order for you to achieve a task or solve a problem. In other words, abstraction describes the act of representing a complex entity by the concepts that it offers you to solve a problem.
A class for an integer object can be abstracted as an object that holds a value equivalent to a mathematical integer. This abstracted representation of the integer does not consider the bits that represent the integer on the systems level.
A bad implementation of an integer would represent the object as the bits used by the system to represent it. It would be harder with this implementation to use the integer object as the bits would need to be repeatedly analyzed every time it's value is needed.

Encapsulation:
According to the Lecture Slides 4(OO Paradigm), encapsulation is a set of language-level mechanisms or design techniques that hide implementation details of various classes or modules. In other words, encapsulation is the process of simplifying complex systems by hiding away unnecessary details that refer to the implementation of a class.
The integer class described above can be used with mathematical operations such as add or multiply. Such operations are achieved through various bitwise operations. Information of the implementation of such bitwise operations do not contribute to the higher level task of adding or multiplying integers so we can hide the bitwise operations with encapsulation.

A bad implementation of an integer would involve complex bitwise operations to be considered when mathematical operations are being done. Such an implementation would exponentially increase the development time as bit operations increase significantly for complex math equations.

Polymorphism:
According to the Lecture Slides 4(OO Paradigm) slide 49, polymorphism describes the treatment of various similar objects as if they were instances of an abstract class but get the behavior that is required for their specific subclass.
An example of polymorphism would be an implementation of objects called Glass Door and Sliding Door which inherits features from an object called Door.

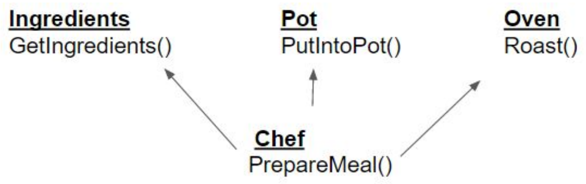


Both the Glass Door and Sliding Door have the open() function that they inherit from Door. While both objects behave like Door's, each have specific functions that are unique to their specific subclasses.
A bad implementation of the Door scenario is illustrated below.

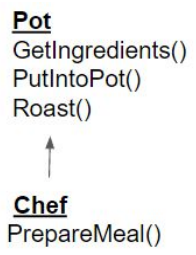
Sliding Door
Slide()
Glass Door
Open()
LookThrough()

Sliding Door and Glass Door cannot be interchangeable since they are not implemented with similar functions even if they are similar in functionality.

Cohesion:
According to the Lecture Slides 4(OO Paradigm), cohesion in computer science refers to how closely the operations in a routine are related. Merriam-Webster defines cohesion as the act or state of sticking together tightly. Cohesion in our Object Oriented context refers to how well the classes and functions we create work together.
Good cohesion is illustrated below.

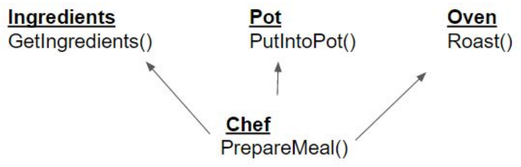


Changes to any component can be done without affecting any of the other components.
Bad cohesion is illustrated below.

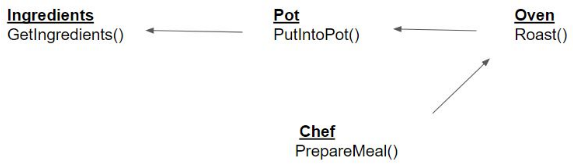


If the chef is unable to produce cooked food, it would be hard to debug which component failed if the oven object has all of the relevant routines.

Coupling:
According to the Lecture Slides 4(OO Paradigm), coupling refers to the strength of a connection between two routines.
Good coupling is illustrated below.



The components above exert weak coupling as they are not dependent on each other to complete their respective tasks.
Bad coupling is illustrated below.



The components above exert strong coupling as they are dependent on each other to complete the overall tasks. If one component fails, there will be no result.

Identity:
According to the Lecture Slides 5(OO Fundamentals) slide 32, it was stated that: we do not want to create a class for objects that do not have a unique identity in our problem domain. The statement conveys that identity, in our Object Oriented context, refers to the features that make two objects of the same class unique. Two car objects could be identical in implementation but be instantiated with different starting variables.
<image>
An example of the identity property failing would be if all car objects did not have identity and all changes made to a car object were made to all car instances. In that scenario, there

would be no point in having multiple instances of the car object if they all share the same features.

2. A company has asked us to design a customer loyalty system. The system will allow customers to join or leave, track the money customers who participate spend each month, allow mass e-mailing of sale information to customers, and provide individual customers with discount coupons based on their monthly spend of certain amounts with the company. Using a level of abstraction, develop a design for this system using the functional decomposition approach (similar to that shown in slide 11 of lecture 4). You can assume the existence of a database that contains all the information you need for your system. For your answer, first describe the functional decomposition approach, discuss what assumptions you are making concerning this problem (any information that was not directly provided to you), and then present your design (in a text description, pseudo-code, or graphically).

The functional decomposition approach abides by the fundamental idea of breaking down large problems into smaller steps where it's easy to solve the problem in a few steps. The assumptions that will be made for the functional decomposition approach are:

- That we need a central "main program".
- The database is a relational database (i.e. SQL, AWS Redshift).
    - It houses a easy to query structure of customers
    - Expenditures of each purchase for each customer with the date associated
    - Also contains all the products with their original price and current price (with a discount for example)
- Assume that the general sales can be decided / sent at execution time (not hard-coded in)
- The individual discounts are executed by a Time-Series algorithm that uses monthly data (last *n* months) to determine a loyalty member's individual discount coupon.
    - Discount coupons are monthly

The functional design of this loyalty system takes into account the granular parts. These individual parts include:

- The ability to join the program
- The ability to leave the program
- The tracking of customers expenditure at their company
- Mass-emailing customers about sales
- Provide individual customers with coupons based on how much they spend
    - Track expenditure

    - Send coupon via email

The central main program will contain a call to a function that adds a user to the program. Their information will be added to the database. When an individual requests to leave the program their user information will be deleted in the database. This would be a branching path. If and only if the user requests to leave the program their information gets deleted. A completely different script will house the tracking ability, adding information to the database as needed for purchases made by customers. A separate script would handle the mass emails of sales by aggregating all the unique users in the database and sending the email out then when the main program is told to do so. From the main program, each month the discount coupon script is executed that takes into account all of the purchases loyalty members have made since they entered the program, and sends them an email.

3. Now develop a design for the same customer loyalty system described in question 2 using the object-oriented approach, keeping in mind the points discussed on slides 30-32 of lecture 4. Identify the classes you would include in your design and their responsibilities. (As before, you can assume the existence of a database and that you will be able to create objects based on the information stored in that database.) Then, identify what objects you would instantiate and in what order and how they would work together and communicate with each other to fulfill the responsibilities associated with the customer loyalty system.

Classes to Include:

- Customer: Houses a list of transactions, expenditures, and user information for a particular customer.
- Transaction: Contains purchases which are from the database. Transactions can have multiple products within them, so they are therefore a list.
- Product: A class that can hold arbitrary product information, such as name, various manufacturer and seller codes, and categories.
- Mass Email: A class that takes in a list of customers and an email. It sends the mail to the list of customers. The email can have codes, and more in it.
- Coupon Aggregation: Aggregates similar loyalty members based on their spending over *n* months.

Delegation is the main design principle in play here. In terms of the structure, the Customer class is like a small dataset specific to a single loyalty member. It holds the transactions, expenditures and user information, such as names, billing/ shipping address. The

Transaction class contains information about a single transaction made by a customer; which contains a multitude of products purchased over that single transaction. A product is a class that gets its information from the database such as the product name, and various product ids. Overall the structure for a given customer is:

- Customer *i*:
    - List of Transactions:
        - Transaction #1
        - …
        - Transaction #*j*
            - Product #1
            - …
            - Product #*k*

Next the Mass Email object is able to take a list of customers, and an email which is sent. Coupons that are sent to the loyalty members are aggregated by the Coupon Aggregation class which determines the list of customers to give coupons once it parses the database of orders by unique loyalty members. The Coupon Aggregation object will have an overloaded method of Mass Email, which sends the coupons to those customers. The order of instantiation is, initialize the customers, then initialize their transactions with various products by selecting the correct rows from the database. When it is time to send a mass email, the Mass Email class gets instantiated, and when it is time to send coupons, both the Mass Email and Coupon Aggregation classes get instantiated, with Coupon Aggregation sending data to Mass Email.

4. In lecture 5, we present "design by contract" in OOAD as an implicit contract. What is the contract? What good happens if we follow the contract? What bad happens if we break it? How could we make the contract explicit? (Cite any sources supporting your answers.)

The contract is the developer's choice to respect the public methods and existing setups when writing new code so as to not corrupt or endanger the viability of the public methods. Importantly, thinking of the designations in this way allows for simplified public methods and code that is very easily maintained. Because the public methods and their reliant distinctions can be trusted to work as intended, the code is easy to review and doesn't need to account for many exceptions or unforeseen cases written later. The bad thing that will happen by breaking the contract is that some methods may not function properly or at all anymore and will need to be complicated with exceptions and catches that weren't there before. This contract can be made explicit by using a language allowing the developers to codify their rules when writing the original classes and subclasses, or by using a package allowing something similar in your chosen language.

Source: Lecture 5