Name: Armant Touche

Class: CS202 Programming Systems

Instructor: Karla Fant

Term Paper

Intro

Going into CS202 Programming Systems, I was hyped by the rumors that regarded this course. Both negative and positive rumors persisted in my mind going into the Summer term. I faintly heard about Object Oriented Programming (OOP) principles but was only able to say, "Everything is an object." That's it. On top of that, we were to intertwine data structures with others. The challenge was on. This paper will go over how one of my previous program successfully and unsuccessfully met OOP principles. Inheritance, polymorphism, encapsulation, and abstraction will be concepts I will go over in relation to my program. There will be analogies and some references but the paper will be free-form in nature.

You Are Your Parent and MORE

The of point of class inheritance is to maximize re-usability of code and provide a way to package methods into a concise library for others to use. "Class inheritance, lets you derive new classes from old ones, with the derived class inheriting the properties, including the methods, of the old class, called a base class (Prata, 2012, p. 708)." Quoted from *C++ primer plus*, inheriting from parents make you your parents and MORE. The MORE part describes the difference in methods and data fields between parents and their children. For program number five, the parent in my single-inheritance hierarchy was the Food class. Food class acted as "the" glue which is another way describe an interface or what *C++ primer plus* calls an *Abstract base class* (Prata, 2012, p. 749). There were some methods that acted as getter's and setter's but the class acted as an Abstract data type, which if fine for this case. The Food was the base class for three derived classes and since Food's child had no other parent, this hi-

erarchy constituted as a single-inheritance relationship. Single-inheritance is the preferred design because with multiple-inheritance, problems like having same-name methods in multiple base classes becomes a problem (Prata, 2012, p. 809). The classes I choose to derive from Food were: (i) Burger (ii) Pizza (iii) Drink. Each of these classes' read-in method were dynamically bound to Food's read-in method since each derive classes' had different implementation for their respective read-in methods. Each derive class shared a common method but had different implementations which constitutes Food as an interface or the "glue" for the hierarchy.

The biggest, missed subtlety in this class is the Object Oriented Design (OOD). I am used to rushing to code my program but as I continue my computer science journey, I am learning that more positive outcomes come to fruition when more effort is inputted into the design phase. That and also, the lab manuals but that is another topic. Approaching a project with good discipline in the design phase provides a more systematic approach to implementation. Literally, I felt I completed my programs faster when I spent more time designing because the nouns (objects) had established relationship. The "has a" and "is a" relationships are important to have drawn out using the Unified Modeling Language (UML) and Class/Responsibilities/Collaborators (CRC) (Prata, 2012, p. 1208). The Class UML diagramming technique was the real *secret sauce* to efficiently implementing a project. *C++ primer plus* calls the situation in which a programs aims to solve is called the *problem domain* which is more apt, in my own opinion, to highlighting what needs to be implemented in order to solve problems from the domain (Prata, 2012, p. 1207). The Food class has a very limited domain to solve from but the class aims to read-in order information and display orders. I felt confident that I displayed good OOD characteristics because I met the goal for having an common *interface* that is equipped to solve problems from the food-ordering system domain.

Many Forms

Polymorphism is another neat characteristic in C++. Being able to called an *overload method* based on run-time circumstances is very handy. "The *polymorphism* means having many forms (Prata, 2012, p. 412)." "Many forms" is an important distinction to make when *overloading methods*. Going back to program number five, the read-in method was overridden but still is consider polymorphism. The read-in took three different forms in each of the derived classes. Since *overriding* require same signature and return-type versus *overloading*, which just requires same method identifier. When doing the OOD, acknowledging the problem that at run-time, different derive classes need different read-in methods I choose to dynamically bind the read-in method. *Polymorphism* doesn't relate to OOP but when choosing between *overloading* and *overriding* it is important to make clear distinction when implementing polymorphic methods in a inheritance hierarchy.

Representing A Collection of Information with an Interface

Going back to the *problem domain*, thinking of an inheritance hierarchy as user-defined type is useful in abstracting a collection of information as a single entity. The Food class in program number five abstracts the information that can be used to describe a food order. In the base class, the derived classes shared a quantity and delivery (yes or no). Following the shared characteristics, each derived classes had their own information but being able to have a inheritance hierarchy abstracted multiple pieces of information into a single entity called Food. The *class* keyword act as the vehicle for abstracting any collection of information from any related domain into a user-defined type (Prata, 2012, p. 508). The canonical "Bank Account" example used to teach Abstraction is another example of Abstracting banking relating information and methods into one neat package. Just like the aforementioned example, the Food class abstracted food-order information and methods into a single, neat package for usage.

Don't Touch Your What's Not Yours

Now when it comes to the client using an interface, not all the information being abstracted should be accessed by the user. This is where *encapsulation* comes into play to prevent unauthorized access. *Data hiding* and placing class method definition into a separate file are forms of *encapsulation* (Prata, 2012, p. 512). In program number five, I unfortunately didn't adhere to the separation of the prototypes and implementation but I did adhere to having protected data members. I didn't understand how definitions were to be separated into different files when coding in Java. Having data members protected or private prevents the client program from having access to per say a pointer that points to a table of users stored in an AVL tree. Classes that possess good *encapsulation* enforces strong control practices in relation to possibly sensitive data. The same goes for protected or private methods, having the methods encapsulated can prevent the client or user from using methods they aren't authorized to use. To invoke these protected or private methods, placing method call in a wrapper method is a way to allow the client to use a protected or private method. In order to display a single order, the recursive method will access to a private member. This is where a wrapper function comes into play. The encapsulated method will be invoked inside the wrapper method. Encapsulation is another crucial aspect of OOP and it creates a structure within an abstract class type.

Outro

All in all, inheritance, polymorphism, abstraction, and encapsulation are the big concepts to learn when going through CS202. The OOP principles are hard to learn at first but through practices and good OOD, anyone can learn these principles. In program five, I felt I was able to demonstrate these principles well. The inheritance hierarchy of food items demonstrates inheritance. The dynamically bound functions in the derived classes displays how polymorphism works. Being able to create a Food as neat package that entails what goes into a food order is an example of abstraction. Then being able to create structure within the class and creating protected and private meth-

ods displays how encapsulation works. I feel very satisfied to made it through this sequence. I feel I am ready to go onto other subjects offered at Portland State University. Thanks for teaching these courses and providing a good learning environment. Take care!

References

Prata, S. (2012). *C++ primer plus* (6th ed). Retrieved from *Library Genesis*. Upper Saddle River, NJ: Addison-Wesley.