Employee/Manager/Client Example (Ver 5 3/6/2016

Introduction

This document is a supplement to the Final Project Part 1 document. It only contains a basic description of the Employee/Manager/Client example and very brief discussions of the mechanics of .NET classes. For more in depth information, refer to the Project document or the textbook. It is not as extensive as the Final Project Part I document, which should be read first before moving on to this document.

The goal of the Employee/Manager/Client (called EmpMan for short) Project is to illustrate the use of inheritance and databases in modeling the data for a small, simple organization. The Project involves the use of one form and requires validation of all data entered in the form in order to prevent bad data from getting into the database. We have added the use of a serializable file and yet another use of a text file as secondary features of our example.

Modeling the Data

The organization has three data entities: managers, workers and clients. The entities have unique attributes, but some fields are common to two or more entities. A good object model arranges data in the same way as is done in a normalized database. Consider the data entities:



Notice that all three entities have the attributes {Name, Birth Date, ID} in common. To reflect this fact, we factor out these commonalities and place them in a new entity, which we call a Person. The three entities can be related in a tree structure as shown below with Person at the root node:



We also see that both the Manager and Worker have the attribute {Job Title} in common, but Client does not. So we factor out the Job Title attribute and create a fifth entity, Employee. The five entities relate to one another as shown next. We consider the Client,, Manager, and Worker entities as "leaf nodes" in our inheritance hierarchy." These three leaf nodes have no remaining common data.



The Person object hierarchy shown above has five nodes: one root node, one inner node and three leaf nodes. The “is-a” test can be used to check object hierarchies: A Manager is an Employee and an Employee is a Person. Replace “is a(n)” in the previous statements with inherits and Manager inherits Employee and Employee inherits Person. The use of the colon : between the entities Employee and Person creates the inheritance relationship between the Employee and the Person classes. The “<Serializable()>” tag in the first line will force .NET to include functions that allow objects to be saved to an XML-coded file (similar to Java). The “abstract” term disallows the direct instantiation of an Employee. Person is also an abstract class. The use of the keyword abstract in this way means that the programmer can only instantiate objects that are leaf nodes (Manager, Worker and Client).

 // Employee inherits the data and methods in Person

 [Serializable()] public abstract class Employee : Person

 {

 private string HiddenEmployeeJobTitle; // Client Type

 // Default constructor

 public Employee ()

 {

 HiddenEmployeeJobTitle = "";

 } // end Employee default constructor

 . . .

 } // end Employee Class

The remaining tasks are to code and test the class hierarchy. To the five resulting classes, we add a Person List Class. This is done to illustrate the use of such lists with entries that are part of a class hierarchy and to enable the creation of a serializable file made up entirely of entries in the Person List.

Each node in the Person hierarchy includes the following:

* The private data
* Default and parameterized constructors
* Properties to access the private data
* Methods to save data from the form to the list and display data in the list on the form
* An overridden toString function to convert the data in the objects to a string
* A shared (static) Validate function to check that all necessary data is entered correctly on the form

The tree on the next page shows the complete Person class hierarchy. It is rendered automatically by VB .NET 2012 by right clicking on the project in the Solution Explorer and selecting “View Class Diagram”.

The PersonList entity (not shown) is a list of Person objects. Because the Manager, Worker and Client classes directly or indirectly inherit the Person class, the PersonList can contain instances of all three objects. This is called late binding.

The use of the <Serializable()> tag facilitates the conversion of an instantiated object to a file. This is referred to as a persistent object. This object can also be read from a file back to an instance in memory. The Project contains information about persistent objects.



The FormController class (also not shown) contains methods to activate and deactivate parts of the form based on the current object. The figure on the next page shows the form after the “CreateClient” button is clicked. We have attempted to illustrate a form that reflects as well as possible the inheritance hierarchy formed by the 5 classes just described. You are STRONGLY urged to use this same form to save time and energy in designing and building your project.

The Form



Why Inheritance?

The use of inheritance as shown provides us with a tool that more accurately reflects the relationships among the various entities modeled in a software system. It also enables programmers to reduce the propensity for redundancy is our code. Redundant (common) attributes across entities are factored out and placed in a separate entity. In so doing, we also enable the programmer to factor out the methods that operate on these attributes. So both shared attributes and methods need be written only once and accessed at lower levels in the hierarchy as will be illustrated as we implement our Final Project.