

Systems Development: Object Oriented Analysis and Design

(H172 35)

Introducing Inheritance

Employees Walkthrough

Step 4 – Problems with *protected* instance variables

***Problems with*** *protected* ***Instance Variables***

In our example, we declared base-class instance variables as protected so that derived classes could access them. Inheriting protected instance variables enables you to directly access the variables in the derived class *without* invoking the set or get accessors of the corresponding property, thus violating encapsulation. In most cases, it’s better to use private instance variables to encourage proper software engineering. Your code will be easier to maintain, modify and debug.

Using protected instance variables creates several potential problems. First, since the derived-class object can set an inherited variable’s value directly *without* using a property’s set accessor; a derived-class object can assign an *invalid* value to the variable. For example, if we were to declare CommissionEmployee’s instance variable grossSales as protected, a derived-class object (e.g., BasePlusCommissionEmployee) could then assign a *negative* value to grossSales. The second problem with using protected instance variables is that derived-class methods are more likely to be written to *depend* on the base class’s data implementation. In practice, derived classes should depend only on the base-class *services* (i.e. non-private methods and properties) and *not* on the base-class data implementation. With protected instance variables in the base class, we may need to modify all the derived classes of the base class if the base-class implementation changes. For example, if for some reason we were to change the names of instance variables firstName and lastName to first and last, then we would have to do so for all occurrences in which a derived class directly references base-class instance variables firstName and lastName. In such a case, the software is said to be **fragile** or **brittle**, because a small change in the base class can “break” derived-class implementation. You should be able to *change* the base-class *implementation* while still providing the *same services* to the derived classes. Of course, if thebase-class services change, we must re-implement our derived classes.

We are going to re-examine our hierarchy again, this time using the best software engineering practices…

Update class CommissionEmployee to once again declare the instance variables as *private*, bearing in mind we still have the public *properties* for each instance variable (FirstName, LastName, SocialSecurityNumber, GrossSales, CommissionRate).

Update class CommissionEmployee methods Earnings() and ToString() use the class’s *properties* to obtain the values of its instance variables, therefore if we decide to change instance variable names these methods will not require modification – only the bodies of the properties that directly manipulate them will need to change. Localising the effect of changes is good software engineering practice.

Update class BasePlusCommissionEmployee methods Earnings() and ToString() to invoke property BaseSalary’s get accessor to obtain the base salary value, rather than accessing baseSalary directly. If we decide to rename instance variable baseSalary, only the body of property BaseSalary will need to change. See individual details below on how to fully re-implement each of these methods:

BasePlusCommissionEmployee Earnings() – this method overrides class CommissionEmployee’s method. Update this version to obtain the portion of the employees earnings based on commission alone (i.e. previously CommissionRate \* GrossSales), by directly calling CommissionEmployee’s Earnings() method with the expression base.Earnings(), then add the BaseSalary to this to calculate the total earnings of the BasePlusCommissionEmployee i.e. full statement is BaseSalary + base.Earnings();

Note the syntax used to invoke an overridden base class method from a derived class – place the keyword base and the member access (.) operator before the base class method name. This is good software engineering practice – by having BasePlusCommissionEmployee Earnings() invoke CommissionEmployee Earnings(), we avoid duplicating code and reduce code-maintenance problems.

BasePlusCommissionEmployee ToString() – similarly, this method overrides CommissionEmployee’s ToString() method. Update this version to call CommissionEmployees ToString() method by using base.ToString(), then add the code to output the remainder of the object’s string representation i.e. the base salary