Aaron Erbe

CSE 210 Articulate

Week 14

**Abstraction** is simplifying the code. We do this by utilizing objects and classes. By using classes, we break up the code and focus in on one object which has unique Attributes and Methods. By thinking and developing in this way, the code for an object is clear, simple and highly flexible. It enables us to update our code by only touching what is needed and minimizing the impact on the rest. It also allows us to create multiple ‘objects’ with similar attributes and methods easily which helps keep our main code clean and organized. For example in the Foundation 1 section of my Final project, I created a Video Class to track multiple YouTube videos. Each video has similar attributes (title, author, length). I can create and store multiple videos as their own object within a List using:

//list of video classes

        List<Video> v = new List<Video>();

Now that each object is stored within a list, it makes it very easy to iterate through the list, calling the specific methods within that class. For example, to call the Display method for each object, I would do this:

        foreach (Video v in videos){

            Console.WriteLine();

            v.DisplayVideoInfo();

        }

**Encapsulation** is carefully hiding the details and access to your classes. This makes your code safer by avoiding accidentally manipulating attributes. It also supports flexibility by forcing the classes to be as self-maintained as possible. You build the methods carefully to manipulate it’s own attributes, then what happens ‘under the hood’ is a don’t care to any external code. This means any updates internal to the class will have no impact to external code. I used Encapsulation throughout all of my Foundation assignments. All of the class attributes are private. All methods are also private unless they need to be accessed by external code. To change class attributes, I use public methods to indirectly change them. I prefer to do this within the Constructor as much as possible. For example, in my Foundation 2 assignment I did this:

    public Product(string productName, string productID, float productPrice, int quanity){

        \_productName = productName;

        \_productID = productID;

        \_productPrice = productPrice;

        \_quantity = quanity;

        CalculateTotalProductPrice();

    }

This uses the public constructor to assign all the attributes within the Product class. There is no other way in my code to change these. Additionally, they cannot be accessed directly. A public method is created to indirectly return back the attributes like this:

    public string GetPackingLabel(){

        //used to build final packing label.  this method returns info for each product to be compiled by orders

        return $"{\_quantity}\tof {\_productName}\t-\t{\_productID}\t-\t${\_productPrice:F2}\teach";

    }

**Inheritance** is a powerful tool that allows you to create Child Classes which inherit the Attributes and Methods of it’s Parent Class. This becomes very useful in creating similar but unique classes. All similarities live within the Parent Class but all unique attributes or methods would live in the Child. An example of this is the Foundation 3 assignment which has a Parent Class “Event” with 3 Child Classes “Lecture”, “Reception”, “Outdoor”. They all share the main attributes of title, description, date, time, address. However, each has unique attributes such as Lecture which has Speaker and Capacity. The child will inherit and the Constructor can collect and pass on the attributes to the Parent like in this example:

    public Lecture(string eventType, string speaker, int capacity, string eventTitle, string description, Address address, string date, string time) :base(eventType, eventTitle, description, address, date, time){

        \_speaker = speaker;

        \_capacity = capacity;

    }

This helps keep your code simple and clean. Updates specific to the child do not impact the other children or parent. Flexibility is greatly improved now. In this case, to add another event, you would simply add another Child class with whatever unique attributes and methods are needed.

**Polymorphism** is allowing the child class to override methods to create slightly unique behaviors. This adds another great level of flexibility. For example, you can setup the Parent with a default method but allow the Child to override if needed. This is demonstrated in the Foundation 4 assignment as shown below. The Parent Class “Activity” sets up a default behavior for GetSpeed:

    public virtual double GetSpeed(){

        return (GetDistance() / GetActivityLength())\*60;

    }

The Child Class “Bicycle” doesn’t need to use this default method. It simply needs to return the speed it collected as a unique attribute as follows:

    public override double GetSpeed(){

        return \_speed;

    }

This flexibility enables us to keep the code clean and maintainable. It also makes it very easy to expand and add additional children with expanded or unique methods as needed.