Tracy Mann

CSE210

Bro. Parrish

Section A8

Polymorphism is the ability or process of making something have many possibilities to change into based on context. This allows related objects to have methods with the same name but complete that method with specific instructions for that specific object or type. This strengthens the other principles allowing the how to for a specific object that inherits a parent’s method to be encapsulated in that child object while leaving the name of the method generic for all of the related types from the parent class. So now you can work on a collection of similar objects typed by the parent type, call the same method for each of the different types in the collection and based on the “context” or exact type of object being called against, the code executed will be specific for the type.

This continues to help to simplify the design of the object. Allowing the common code to be defined in the parent object and made specific to a child object only if needed. The child may have descriptors that the parent may not have access to, but an overridden method in the child object does. Allowing the child to adjust how it implements that method given the private information held inside the child object. This reenforces encapsulation inside inheritance. One example I remember that demonstrates polymorphism is if you were to define an object called dragon, used to represent the magical intelligent serpentine beast of lore. The dragon would have descriptors of body, tail, wings, and color. The methods could include fly, bite, tail attack, claw attack, and breathe attack. A dragon is not a specific type of dragon, instead it is an architype(abstract) of dragon that all other dragons are based on, so this object itself cannot exist, only the specific types that inherit dragon can. A specific dragon type would include but not be limited to a red dragon, white dragon, and black dragon. One example of polymorphism comes into play in this case with the breath attack, as each specific type of dragon has its own type of dragon breathe attack. A red dragon breathes flame, white breathes ice and a black breathes acid. The method breathe attack cannot be defined at the dragon object, because we do not know the details of a specific dragon type to know how to do a breath attack but the dragon object does know there is a breath attack and defines it as an abstract or undefined method. The specifics are detailed in the individual definition of a specific dragon type. All dragons do breathe attacks but context dictates what kind.

In the assignment, I used polymorphism for the eternal quest program to determine the completion state of a goal, to display the goal, to implement the report event method correctly for each type of goal, and to display the request points value message for each goal. In the case of the display the request message the method is actually able to be defined in the architype, because the basic parent class has all the details needed to complete this method as follows:

internal virtual void DisplayRequestPointValue()

{

  Console.WriteLine(Configuration.Dictionary["RequestPointValueMessage"]);

}

This method is declared as virtual to allow a specific child class to override this behavior if this method needs to be defined different based on the circumstances of the context of the child class. This used the Request Point Value Message to be displayed for the generic goal type meaning any that do not override its behavior as follows for the eternal goal object.

internal override void DisplayRequestPointValue()

{

Console.WriteLine(Configuration.Dictionary["RequestRepeatPointValueMessage"]);

}

Here we override the parents’ method to change its behavior and instead of the original message to be displayed it shows the Request Repeat Point Value Message instead.

The other three methods do not have enough information in the parent goal object to even build a default definition. In this case we define them as having to be defined in each child class by declaring them as abstract and giving no implementation of the method.

internal abstract void DisplayGoal(int index = -1);

internal abstract Boolean IsCompleted();

internal abstract int Report();

In my case I also had a JSON Goal class that inherited the same methods to my JSON Eternal Goal, so to make a common base for both the Eternal Goal and the JSON Eternal Goal, I defined an internal static method to be used by both to keep the behavior the same in the related classes for the override.

internal static void DISPLAY\_GOAL(EternalGoal goal, Configuration configuration, int index = -1)

{

if (index >= 0) Console.WriteLine(String.Format((String)configuration.Dictionary["SimpleGoalIndexedDisplayFormat"], index, (Char)configuration.Dictionary["IncompleteSymbol"], goal.Name, goal.Description));

else Console.WriteLine(String.Format((String)configuration.Dictionary["SimpleGoalNonIndexedDisplayFormat"], (Char)configuration.Dictionary["IncompleteSymbol"], goal.Name, goal.Description));

}

internal override void DisplayGoal(int index = -1)

{

DISPLAY\_GOAL(this, Configuration, index);

}

In the JSON Eternal Goal as follows:

internal override void DisplayGoal(int index = -1)

{

EternalGoal.DISPLAY\_GOAL((EternalGoal)(Goal)(JSONGoal)this, Configuration, index);

}

For an eternal goal the abstract display goal method had only an optional index and adds nothing more to choose to display the goal information between and indexed line or non-indexed line. One of the specific items in this definition is that both of the lines are to always display the goal with an incomplete symbol, instead of having any ability to show the goal as complete.

The same functions in the simple goal show that with other detail held only in the simple goal the behavior needs to change.

internal static void DISPLAY\_GOAL(SimpleGoal goal, Configuration configuration, int index = -1)

{

Char check = (Char)configuration.Dictionary["IncompleteSymbol"];

     if (goal.IsCompleted()) check = (Char)configuration.Dictionary["CompleteSymbol"];

     if (index >= 0) Console.WriteLine(String.Format((String)configuration.Dictionary["SimpleGoalIndexedDisplayFormat"], index, check, goal.Name, goal.Description));

     else Console.WriteLine(String.Format((String)configuration.Dictionary["SimpleGoalNonIndexedDisplayFormat"], check, goal.Name, goal.Description));

}

internal override void DisplayGoal(int index = -1)

{

     DISPLAY\_GOAL(this, Configuration, index);

}

In the JSON Simple Goal as follows:

internal override void DisplayGoal(int index = -1)

{

SimpleGoal.DISPLAY\_GOAL((SimpleGoal)(Goal)(JSONGoal)this, Configuration, index);

}

The JSON Simple Goal stays common to the Simple goal using the static function. That static function is used to define the abstract, differently than the Eternal Goals. In this case we use the Is Completed method of goal to determine which symbol to use in the formatted line display.

In the checklist goal we define this yet again different as follows:

internal static void DISPLAY\_GOAL(ChecklistGoal goal, Configuration configuration, int index = -1)

{

Char check = (Char)configuration.Dictionary["IncompleteSymbol"];

     if (goal.IsCompleted()) check = (Char)configuration.Dictionary["CompleteSymbol"];

     if (index >= 0) Console.WriteLine(String.Format((String)configuration.Dictionary["ChecklistGoalIndexedDisplayFormat"], index, check, goal.Name, goal.Description, goal.NumberOfTimes, goal.TargetNumberOfTimes));

else Console.WriteLine(String.Format((String)configuration.Dictionary["ChecklistGoalNonIndexedDisplayFormat"], check, goal.Name, goal.Description, goal.NumberOfTimes, goal.TargetNumberOfTimes));

}

internal override void DisplayGoal(int index = -1)

{

DISPLAY\_GOAL(this, Configuration, index);

}

In the JSON Checklist Goal as follows:

internal override void DisplayGoal(int index = -1)

{

ChecklistGoal.DISPLAY\_GOAL((ChecklistGoal)(Goal)(JSONGoal)this, Configuration, index);

}

In this example the Is Complete Method is also used, but we also output additional information into the formatted output for the number of times and targeted number of times the goal is to be reported to be considered complete.

Because Goal has abstract methods in it it must be defined as abstract as follows:

public abstract class Goal

{

…

}

public abstract class JSONGoal : Goal

{

…

}

Also, because JSON Goal does not implement or define those same abstract goals, it too much remain abstract.

What does all this do for us? In my list goals methods of my Goals class as follows:

internal void ListCurrent()

{

ForEach((goal) => {

      if (!goal.IsCompleted()) goal.DisplayGoal();

     });

}

internal void ListAll()

{

ListCurrent();

ForEach((goal) => {

      if (goal.IsCompleted()) goal.DisplayGoal();

});

}

Because my goals class is defined as a List of Goal as follows:

internal class Goals : List<Goal>

{

…

}

I will be allowed to loop through the list of Goals and call the display method for each, not having to know what type of goal they are and each goal based on the context of what type of goal it is will implement the correct way to display the details of that goal. Thus polymorphing the goal from one to the other based on what is needed at the time.