Tracy Mann

CSE210

Bro. Parrish

Section A8

Abstraction is basically being able to make your solution generic or conceptually abstract. To break out the pieces of the concept you are working with and allow a user to use the object in an abstract manner without having to know how it is done or what is under the hood. The user only needs to know the API or interface functions to use the object. The only items worked with in the object are related to exactly what the object is in a generic sense. My Task object has a task type and task state to help track the progress of the task at hand. Not all tasks are assigned so my upper-level task object does not have a name assigned or a schedule time assigned.

In my example I have hidden all of my inner working from the end user using the internal keyword. As below:

public class Task : DescribedObject

{

internal static string ObjectNameDisplay { get; } = "task";

internal TaskType TaskType { get; set; } = TaskType.Task;

internal TaskState TaskState { get; set; } = TaskState.Template;

internal String Command { get; set; } = "";

internal List<String> AssignedRoles { get; set; } = new();

internal List<String> RequiredPreRequisiteTasks { get; set; } = new();

internal int PreWaitTimeSeconds { get; set; } = 0;

internal int DurationSeconds { get; set; } = 0;

internal int PostWaitTimeSeconds { get; set; } = 0;

Encapsulation protects the inside, both from the user an yourself, from, issue of the inner data workings. The security in my task object is internal because other object in the whole application need access to get the job done, but the outside user does not need it. This would allow me to make a much better Name object that had more features, with out changing the external interface and the user does not need to know in this example it is just a string, but in a future version it could be an entire object that has the ability to break a name apart and work with its parts for other uses.

public class Name : IName

{

internal NameType Type { get; set; }

internal String Value { get; set; }

public Name()

{

Inheritance is the relationship of the object to each other. Such as child object will inherit (have defined as if they were defined for this class alone) all non-private attributes. In my code, the Task object inherits from the Described Object that inherits from the Named Object. Meaning that the Named object has a Name associated with it, in this case a simple Name object. And the described object inherits Name and add to is a property of a description. Then in the end my task has both a name and a description so it inherits the described and gets all the non-private capabilities of both.

public class DescribedObject : NamedObject

{

internal String Description { get; set; }

Polymorphism is the ability to use the parent class from inheritance to handle derived child classes in common ways, based on the non-private attribute the parent class gave the child classes. Effectively allowed each child to be it own type but change to the needed state when used in a larger operation.

My project uses polymorphism in its many different types of tasks that can be displayed, exported and imported as a group or related classes even though they are different classes.

public class TemplateTask : Task

{

…

internal override void Display(int option = -1) => base.Display(option);

template task just uses the parent as it does not have new data to show.

public class TemplateMitigation : TemplateTask

{

…

internal override void Display(int option = -1)

{

base.Display(option);

if (option >= 0) Console.WriteLine(String.Format("{0} Risk Name: {1}", new string(' ', option.ToString().Length), Risk.Name.Value));

else Console.WriteLine(String.Format("\tRisk Name: {0}", Risk.Name.Value));

}

On the other hand the template mitigation task has an associated risk tied to it so the display adds the view of that data when it is being displayed beside the other tasks.

The great note also for this is that it reduces the code redundancy and number of places to put basically the same routine over and over again allowing for easier modification (single point of contact). As my example above it shows that the basic pieces that are common to task and template mitigation task are handled by task, and mitigation only handles what is new for it alone.