COS 20007: Object Oriented Programming

Hurdle Task 2: Semester Test

Show Wai Yan/105293041

1. Describe the principle of polymorphism and how it was used in Task 1.

Polymorphism lets different classes share a common interface, with each providing its own method implementation. In my Task 1 code, I used polymorphism through the abstract Thing class and List<Thing> collections.

In Thing.cs, I defined Thing with abstract methods Size() and Print() :

```
public abstract class Thing
{
    public abstract int Size();
    public abstract void Print();
}
```

The abstract keyword means Thing cannot be instantiated, and any class inheriting from it (like File and Folder) must provide concrete implementations for Size() and Print(). This enforces a contract, ensuring all derived classes have these methods, which is key to **polymorphism**.

In FileSystem.cs and Folder.cs , I used List<Thing> to store both File and Folder objects:

```
private List<Thing> _contents;
```

In Folder.cs, _contents is also a List. By typing _contents as List, I can store both File and Folder objects in the same list, treating them as Thing instances. In Program.cs, I add mixed types to the FileSystem's _contents

In Program.cs , I can add mixed types to FileSystem 's _contents , for example:

```
FileSystem midTest = new FileSystem();
midTest.Add(new File($"{myStudentId}-00", "txt", new Random().Next(1000, 10000)));
midTest.Add(new Folder("Test1"));
```

When PrintContents() calls item.Print() , polymorphism ensures the correct Print() runs based on whether item is a File or Folder at run-times.. Using List<Thing> lets me treat all objects as Thing , while abstract methods guarantee consistent interfaces, making the code flexible and extensible.

2. Consider the FileSystem and Folder classes from the updated design in Task 1. Do we need both of these classes? Explain why or why not.

In my Task 1 code, both the FileSystem and Folder classes serve distinct purposes.

The FileSystem class, defined in `FileSystem.cs`, represents the top-level container for the file system. It holds a `List<Thing>` to store files and folders and provides a method to print the entire structure.

The Folder class, in `Folder.cs`, represents a directory that can contain files and other folders, also using a List for its contents. It calculates the total size of its contents and provides a detailed `Print()` method.

Do we need both classes? I believe both are necessary for the following reasons.

Distinct Responsibilities: FileSystem acts as the root of the file system, providing a high-level view of all top-level items (files and folders). Its `PrintContents()` method gives a broad overview, starting with "This File System contains:". In contrast, Folder represents nested directories, with a more detailed `Print()` that counts files and folders and reports their total size. Merging them would blur this goal, making it harder to differentiate the root from subdirectories.

Hierarchical Structure: The code models a real file system where folders can be nested (e.g., Test2 contains Test2Child in `Program.cs`). Folder inherits from Thing, allowing it to be stored in FileSystem's or another Folder's `_contents` list, supporting recursive nesting. FileSystem doesn't need to inherit from `Thing` since it's not part of another container, so combining them would complicate this hierarchy.

Extensibility: Keeping FileSystem separate allows for potential future features, like file system-wide operations (e.g., searching or managing multiple roots, and cleaning), without mixing with folder-specific logic. Folder's design focuses on managing its contents.

Could they be merged? Theoretically, I could make FileSystem a special case of Folder by having it inherit from `Thing` and use the same logic. However, this would add unnecessary complexity, as FileSystem doesn't need a `Size()` method or to be nested in another container. It would also make the code less intuitive, as the root file system isn't a folder.

In conclusion, I think both classes are needed because they serve distinct roles where FileSystem as the root container and Folder as a nestable directory, and this concept is enhancing clarity and supporting the hierarchical design.

3. What is wrong with the class name Thing? Suggest a better name for the class, and explain the reasoning behind your answer.

The name Thing is too generic, lacking specificity about what it represents. In the context of my code, Thing is an abstract base class for File and Folder, which are components of a file system. A name like Thing doesn't indicate that it's related to file system entities, making the code less clear for others (or myself) to understand at once. Good class names should be descriptive, following naming conventions that reflect purpose or behavior, especially for an abstract class that defines a shared interface.

Suggested Name: FileSystemItem

I propose renaming Thing to FileSystemItem. This name clearly indicates that the class represents an item in a file system, such as a file or folder. It's specific enough to convey the context while remaining broad enough to encompass both File and Folder, which inherit from it.

Reasoning Behind FileSystemItem

1. **Clarity and Context**: FileSystemItem make explicitly and immediately clear that this class is the base for file system components. In Program.cs, I create files and folders (e.g., new File(...) and new Folder("Test1")), and FileSystemItem reflects their shared role as items in FileSystem's or Folder's _contents list.
2. **Polymorphic Usage**: Since Thing defines abstract methods Size() and Print() (used by File and Folder), FileSystemItem suggests a common clearity name for any file system's items, aligning with polymorphism.
3. **C#'s Naming Conventions**: In C#, class names should be nouns that describe the entity. So, FileSystemItem is a clear, descriptive noun, unlike Thing, which is overly abstract and could apply to any context.

In summary, Thing is too vague, reducing code clarity. Renaming it to FileSystemItem makes the code more descriptive, aligns with the file system context, and enhances maintainability, especially when reading Program.cs or other classes that rely on this abstraction.

4. Define the principle of abstraction, and explain how you would use it to design a class to represent the FileSystem and Folder classes

Abstraction is the principle of hiding complex implementation details and exposing only the essential features of an object through a simplified interface. It allows me to focus on what an object does rather than how it does it, making the code easier to understand and maintain.

Applying Abstraction to Design a Class for FileSystem and Folder:
In my Task 1 code, FileSystem and Folder share similar functionality (both store a List<Thing> and manage contents), but I need to consider whether a single class could abstract their common behavior. To apply abstraction, I would design a class that captures their shared essence while hiding implementation details.

Proposed Design: SystemContainer Class
I would create a class called SystemComponent to represent the common functionality of FileSystem and Folder. This class would define the core operations (storing and managing items) and leave specific behaviors (like printing or hierarchy) to derived classes. Besides that, folder and file system are the core component of Operating System, so they are SystemComponent,which define is-a relation.

Here's how I would design it:

```
namespace Task1
{
    public class SystemComponent
    {
        protected List<Thing> _contents;
           private string _name;



        protected SystemComponent(string name)
        {
            _contents = new List<Thing>();
```

```
            _name = name;
        }

        public void Add(Thing toAdd)
        {
            _contents.Add(toAdd);
        }

        // some functionalities's implementation that are hided to user

           public void remove(Thing toRemove) {}

           public void CheakHealth() {}

           public int Size() // to know OS level storage and file storage

           public string Name
        {
                        get{return _name;}
        }
    }
}
```

Above code is only demonstration purpose.

FileSystem and Folder inherits the _contents list and Add() method, the abstraction hides the list management details. They both need remove and checking health functionalities for remove harmful items and unnecessary items from system and folder.

Note: Using class does not mean that it will support automatically abstraction concept of OOP. The reason why I used class to explain this is that for the scenario of sometimes we need Virtual Machine in our computer, and when we create, that VM software create a folder that behave like a File System of Virtual OS in that Virtual Environment. The folder is still a folder to our own OS, however, the Virtual OS make that folder as File System and operate like a OS's root folder. In our case, our component's inheritance class will create what the software needed (i.e. actual FileSystem or just folder). So, whatever the inheritance instance of class – SystemComponent are that they can both make the functionalities that defined the SystemComponent and do not need to know the actual implementation to the user of FileSystem and Folder class. So, user of the each class, don't need to know how each functionalities work, however, only need to know what functionalities work.

How Abstraction is Applied:

- **Simplified Interface**: SystemComponent exposes only essential operations (Add and Remove), hiding the complexity of list management. Users of FileSystem or Folder don't need to know how items are stored, only that they can add items and Checking Health.
- **Common Functionality**: The _contents list and Add() method are centralized in SystemComponent, reducing code duplication between FileSystem and Folder (both had identical Add methods and _contents fields in the original code).

By abstracting shared behavior into SystemComponent, I simplify the design, eliminate redundancy, and make it easier to extend (e.g., adding new container types). Unlike merging FileSystem and Folder (as discussed in Question 2), this approach keeps their distinct roles (root vs. nestable directory) while leveraging abstraction to streamline common functionality.

5. Which Pass (and Credit) tasks you have submitted to Canvas utilize both the principles of polymorphism and abstraction? You can list two examples

Example 1: ShapeDrawer in 4.1P and 5.3C

Abstraction: I hid implementation details by creating a Shape class that exposes only the essential interfaces (Draw(), IsAt(), etc.) while concealing the specific drawing mechanics of each shape. Users of my Shape objects don't need to know how a circle or rectangle is drawn internally.

Polymorphism: I implemented different shape classes (MyRectangle, MyCircle, MyLine) that can all be treated uniformly through the Shape interface. In my Drawing.Draw() method, I loop through shapes and call their Draw() method without needing to know their specific types.

```
foreach (Shape s in _shapes)
{
    s.Draw();
}
```

Example 2: SwinAdventure in 5.2P_Iteration3 and 6.1P_Iteration4

Abstraction: I created a system where game entities expose only relevant behaviors (Locate(), FullDescription) while hiding their internal implementation. For example, users of my GameObject classes don't need to know how items are stored in bags or how identifiers are matched.

Polymorphism: I implemented the IHaveInventory interface in multiple classes (Player, and Bag), each with their own implementation of the Locate method:

```
public interface IHaveInventory
{
    GameObject? Locate(string id);
}
```

In my commands, I can work with any object implementing this interface without knowing its specific type.

```
IHaveInventory? container = FetchContainer(containerId, p);
if (container != null)
{
    GameObject? thing = container.Locate(thingId);
    // Use the located object...
}
```

This allows me to handle different container types (player inventories, and bags) through a common interface, demonstrating polymorphism in action.