05/22/2019

Anthony Roth

**Final Project - Roth’s AudioMedia Player**

**Table of Contents**

1. Application Description………………………………………………………………………………….Page 2
2. Design Patterns………………………………………………………………………………………………Pages 3 - 5
   1. Singleton…………………………………………………………………………………………….Page 3
   2. Factory………………………………………………………………………………………………..Page 4
   3. State……………………………………………………………………………………………………Page 5
3. Design Pattern Alternatives…………………………………………………………………………….Page 6

**Application Description**

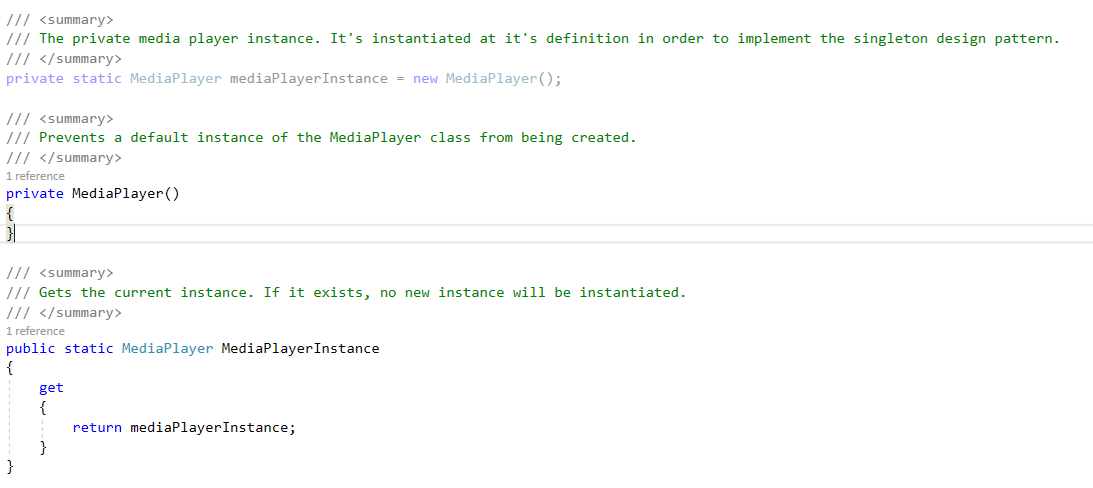
This program is designed to take user input in the form of an audio file, and play it for the user. The user can pause the file, play it, or stop the file from playing entirely. Not to mention, the user can switch the file being played at any time using the browse button. There are two allowable audio file types. These are .wav and .mp3. Filters are set in the program to disable any other file types, avoiding any potential bugs with unacceptable file types. An interface in conjunction with an enum, is utilized to determine the allowable file types.

The MediaFileFactory is then used to create an abstract version of the file in the program, containing the path name, file type, and the actual C# type as properties/setter methods. Using these properties, we can set the file name, and track length, as well as the passed time in the current track. Essentially, this program is a simple audio file player through and through.

**Design Patterns – Singleton**

When taking a look at the MediaPlayer custom class, we can see that the constructer is listed as private, disallowing any instances besides the internally created instance. This internal instance is a private static field meaning it can be accessed but never recreated outside of the class itself. A potential object can retrieve the necessary data utilizing a property that gets said static field but creating a separate instance is still impossible with this implementation.

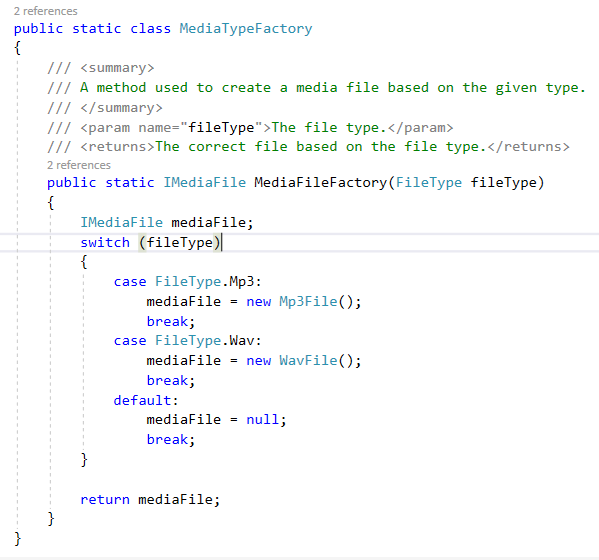
Overall, this design pattern is sufficient for this project considering that only one instance of the MediaPlayer class needs to be present at one time. If two were to exist, then the user could accidentally play two files at the same time, which runs contrary to the program’s inherit functionality. Not to mention, it could potentially cause catastrophic performance issues if left unchecked. A great design pattern as the basis of the class’s creation.

Below is an example of this design pattern implementation:

**Design Pattern – Factory**

Creating a system for object creation diversion is the key to this implementation of the Factory Pattern. When creating the custom file objects, I used a static factory method called MediaFileFactory. In order to determine which file type object to return, it uses a custom enum appropriately named FileType.

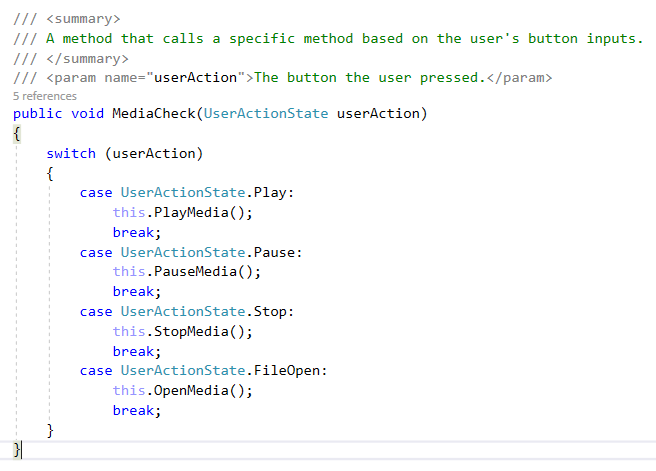
The two values are Mp3, and Wav. Both are ran through a switch case statement to determine which instance the ImediaFile will become. This file type parameter is determined based on a regex match found in a private method used to set the actual/current file object on the MediaPlayer class. This forces the program to check if the file path name has an appropriate extension for the program as well as creating a custom c# object based on said extension.

Below is an example of the factory pattern implementation:

**Design Pattern – State**

Implementing the state pattern was a necessity in order to track the user input’s at runtime. This implementation uses an enum who’s value changes to the current user action. This applies to playing, stopping, pausing, and browsing for files. Depending on the current value of this enum, a different private method will be called inside the MediaCheck method. This is accomplished using a switch case statement that uses said enum.

All of these methods are what allow the media player to function sequentially but still allowing for asynchronous user input. Whilst from a processing standpoint it may not be the most sufficient it allows for more control and flow over your program’s call stack. In a program such as this, we need to be able to switch states on the fly but still give the user great control of their experience.

Below is an example of the state design pattern implementation:

**Design Pattern Alternatives**

When looking at our current design patterns, we can see that the same functionality is still achievable using other design patterns. Take for example, the factory implementation for media files. Instead of using a factory, we could have used an iterator that contains both possible media types. It would iterate on subjects based on the same regex logic, utilizing LINQ’s foreach statement.

Looking at the singleton pattern implementation, it works very well for this program. However, using the strategy pattern, we could have made all the methods available to a set of related algorithms. Each of these could run the same algorithm but with different implementations based on the initial interface.

This could also negate the need for the state pattern, as said behaviors could have resided in a purely user action-based interface, using delegates and asynchronous programming. Another good replacement for the state pattern would be the observer pattern. We could have observer classes that watch the subject (The mediaplayer class) and receive notifications and update the other observers when any changes are made. This way we can change our method calls based on the current update information.