**PR Feedback on Code**

**General Feedback and Code Smells**

* CalculatePrice method is too long, break long methods into sub routines to make code more readable
* Use Constants.cs file to store values that are references in a central location. “Night” is used directly as text, it should be a constant and referenced
* Create a layered application and remove business logic and data access from the same class.
* Add logging and Exception handling
* Add unit testing

**Specific Feedback Based on SOLID Principles**

1. **\*\*Single Responsibility Principle (SRP)\*\***:

- Each class should have a single responsibility.

- Separate concerns related to pricing logic, input handling, and data storage.

The CalculatePrice Class does not have a single responsibility, it directly performs database transactions by querying the base price table to get the price of the tickets. The price Calculator class should focus on doing just calculations and not fetching data. Fetching data should be delegated to a Repository class, which can also be served by a service class acting abstractly to handle request to the Repository class. In effect, the application should be broken down into layers, each layer containing classes that work together to achieve a single responsibility. There should be an Infrastructure layer with classes that handle things such as database connections and data retrieval. There should be a presentation layer that handles serving and presenting the results of operations to the user. There should be a business layer that is responsible for the logic and business rules involved in calculating ticket prices. In the refactored implementation, a business, infrastructure and presentation layer have been created as separate projects, classes have been created in each layer with single responsibilities based on their layers. The repository pattern is used to handle Database access alone, the business layer has classes that handle the business rules for ticket pricing, the presentation layer uses swagger and a web page to interact with the user and display the results of operations.

2. **\*\*Open/Closed Principle (OCP)\*\***:

- Design the system to be open for extension but closed for modification.

- Accommodate new pricing rules or pass types without altering existing code.

Any time a rule needs to be added or modified, the existing code in the Calculate Price method will need to be changed. This will lead to regression issues, as changes break existing code or introduce the need to retest existing code. In the refactored code, the decorator pattern is used to adhere to the open close principle. Anytime a new rule needs to be introduced, a new decorator can be created to extend the existing behavior of the application. In the refactored approach, there is a base rule engine that does the basic ticket calculations, the rule can be decorated with new rules to add additional functionality without the need to change the structure or code of the existing rules. The rules are also configuration driven and use the factory pattern to create and apply new rules, this provides a way to keep the code closed to modification but open to extension via the addition of new decorators to add rules to the existing rules engine.

3. **\*\*Liskov Substitution Principle (LSP)\*\***:

- Derived classes (e.g., child, adult, senior, student) should be substitutable for their base class (generic ticket).

There is no need to have child, adult, senior and student tickets that inherit from a base ticket class because these are just attributes of the type of ticket and don’t change the core functionality of a ticket so I did not introduce this. However, with the Decorator pattern there is a perfect implementation of the substitution principle because there are two decorators: Age and Holiday that can be used interchangeably or together to decorate the rules engine with rules that may be applied in calculating the ticket price based on age or holiday. In the future, new decorators can be added to apply new rules and they can all be used interchangeably or together because they implement the same abstract classes and interfaces correctly. Also, an Interface is create for each of the two repositories : IPricingRepository and IHolidayReposity. These repositories are used to access the pricing and holiday information from a database. The concrete implementation of these repositories can be substituted for any class that implements their interface, this adheres to the LSP principle.

4. **\*\*Interface Segregation Principle (ISP)\*\***:

- Interfaces should be specific to the needs of the implementing classes.

- Avoid fat interfaces with unnecessary methods.

In the refactored solution, two different interfaces are used for the repositories, one for the repository that stores Holiday data and one for the one that stores price data. In reality, all the data could belong to the same repository and one repository could be used for both, but to illustrate the ISP implementation I created two different interfaces and assumed the data came from two different databases. By having different repository interfaces for each conceptual set of data, you allow the developer to interchange each repository with a different implementation in the future, and you can do this independently as opposed to the scenario where there is just one interface that abstracts all repositories.

5. **\*\*Dependency Inversion Principle\*\***:

- Depend on abstractions (interfaces or abstract classes) rather than concrete implementations.

- Use dependency injection to inject dependencies.

To keep the application dynamic, allowing for the services that handle calculation, presentation and data access to be swapped out during runtime, dependency injection is used in the refactored solution. The repositories and services are registered at start up using dependency injection and are lazy loaded when they are needed using constructor dependency injection. This allows just-in-time and efficient use of repositories and services when they are needed.

**Design Rational and Explanation of Refactored Solution**

The refactored solution uses a Domain Driven Onion Architecture to make the code more maintainable and readable. [Understanding the Power of ASP.NET Core Onion Architecture: A Comprehensive Guide | by Anuj Bhor | Medium](https://anujbhor.medium.com/understanding-the-power-of-asp-net-core-onion-architecture-a-comprehensive-guide-e3ed0092b04b) There is a Domain layer that handles business logic, an Infrastructure layer that handles database access and a Rest/Http layer to handles presenting results back to the user. The code is writing with strong adherence to Solid Principles: A decorator pattern is used to create a rules engine which achieves the following:

* Open to extension close to modification by allowing new rules to be added as decorators to exiting rule engine without modifying existing rules
* Single responsibility of each decorator to apply a very specific rule
* Each rule can be applied at runtime and interchangeable in accordance with LSP

Factory class is created to generate the rules based on configuration that can be changed or updated. Rules can be used to decorate rules engine:

* Single Responsibility is maintained because Factory class is focused only on creating and decorating rules. This is not delegated to any other class

Repository Pattern is used to access data and separate data access from Business logic, these are injected into the application using constructor dependency injection when needed

* The repository pattern uses a separate repository for each repository which follows the Interface segregation principle
* Uses dependency injection or inversion of control to inject repositories when they are needed