DVT Elevator Challenge

Your task is to develop a challenging console application in C# that simulates the movement of elevators within a large building, with the aim of optimizing passenger transportation efficiently. The application should adhere to Object-Oriented Programming (OOP) principles to ensure modularity and maintainability.

The console application must include the following key features:

1. Real-Time Elevator Status

   Display the real-time status of each elevator, including its current floor, direction of movement, whether it's in motion or stationary, and the number of passengers it is carrying.

2. Interactive Elevator Control

   Allow users to interact with the elevators through the console application. Users should be able to call an elevator to a specific floor and indicate the number of passengers waiting on each floor.

3. Multiple Floors and Elevators Support

   Design the application to accommodate buildings with multiple floors and multiple elevators. Ensure that elevators can efficiently move between different floors.

4. Efficient Elevator Dispatching

   Implement an algorithm that efficiently directs the nearest available elevator to respond to an elevator request. Minimize wait times for passengers and optimize elevator usage.

5. Passenger Limit Handling

   Consider the maximum passenger limit for each elevator. Prevent the elevator from becoming overloaded and handle scenarios where additional elevators might be required.

6. Consideration for Different Elevator Types

   Although the challenge focuses on passenger elevators, consider the existence of other elevator types, such as high-speed elevators, glass elevators, and freight elevators. Plan the application's architecture to accommodate future extension for these types.

7. Real-Time Operation

   Ensure that the console application operates in real-time, providing immediate responses to user interactions and accurately reflecting elevator movements and status.

Keep in mind that the building has a central computer controlling all elevators, and the application should consider the coordination of multiple elevators' movements.

Your solution should demonstrate an efficient and well-structured approach, showcasing your understanding of OOP principles, algorithm design, and real-time processing. The challenge is designed to test your ability to build a sophisticated elevator simulation capable of handling various scenarios found in a large building with multiple elevator types.

Those reviewing your submission consider the following:

GitHub Submission

- Repo was made public
- .gitignore present
- Code committed using Git commands (not upload)
- Main branch builds
- Meaningful commit messages
- Informative readme present
- Bonus: Tags, branching, automated builds

Comments/Documentation

- Informative readme present
- Code comments in line with MS documentation
- Code comments per generally accepted industry guidelines

Clean Architecture, or similar

Project/s must be structured according to clean architecture or similar. Adopting clean architecture or similar principles in C# code promotes better software design and development practices, leading to more maintainable, robust, and scalable applications. It also enables better alignment with business requirements and simplifies the development process for teams.

SOLID

Single Responsibility Principle (SRP)

Each class or component in the application should have a single responsibility related to elevator movement. For example, you might have separate classes for elevator control, floor management, and passenger handling. This ensures that each class is focused and easier to understand.

Open/Closed Principle (OCP)

Design the elevator system in a way that allows you to extend its behaviour without modifying existing code. For example, if you want to add a new type of elevator or implement additional control strategies, you should be able to do so by adding new classes or methods without changing the existing ones.

Liskov Substitution Principle (LSP)

Ensure that any subclass or derived class you create can be used interchangeably with its parent class. In the context of the elevator simulation, this means that any specific elevator implementation should be able to replace the base elevator class without changing the expected behaviour of the system.

Interface Segregation Principle (ISP)

Design interfaces that are specific to the needs of the client code. Avoid creating large, monolithic interfaces that require the implementation of unnecessary methods. For instance, you might have separate interfaces for elevator control, floor events, and passenger interactions, tailored to each client's requirements.

Dependency Inversion Principle (DIP)

Depend on abstractions rather than concrete implementations. For instance, your elevator control system should depend on high-level interfaces (abstractions) that define elevator behaviour, rather than directly relying on specific elevator classes. This allows for easier swapping of different elevator implementations.

By adhering to these SOLID principles, your elevator simulation application will have a clear and modular structure, making it easier to understand, maintain, and extend in the future. The separation of concerns, flexible design, and decoupling of dependencies will contribute to a more robust and scalable elevator simulation.

Unit Tests

Unit tests play a crucial role in the elevator coding challenge, just as they do in any software development project. Here are some key reasons why unit tests are important in the context of the elevator coding challenge:

- Validation of Elevator Behaviour

  Unit tests allow you to verify that the elevator's core functionality is working correctly. By writing tests for elevator movement, floor button presses, door operations, and other critical aspects, you can ensure that the elevator behaves as expected and meets the requirements.

- Regression Testing

  As you make changes and improvements to the elevator code, unit tests act as a safety net to catch any unintended side effects or regressions. Running unit tests after each modification helps to quickly identify issues and prevent new bugs from being introduced.

- Refactoring Support

  When refactoring or optimizing the codebase, unit tests provide confidence that the behaviour of the elevator remains intact. If all tests pass after refactoring, it indicates that the changes did not break any existing functionality.

- Isolation of Bugs

  Unit tests help isolate bugs and pinpoint the specific areas where problems exist. This reduces the time and effort required to debug and fix issues in the code.

- Code Design and Modularity

  Writing unit tests often requires breaking down the code into smaller, testable units. This promotes better code design, modularity, and separation of concerns, making the codebase more maintainable.

- Documentation and Usage Examples

  Unit tests serve as executable documentation that showcases how different parts of the elevator system should behave. Other developers can use these tests to understand the intended behaviour and usage of various components.

- Collaboration and Teamwork

  When multiple developers are working on the elevator challenge, unit tests provide a common language for understanding and validating each other's code. They act as a form of communication between team members.

- Test-Driven Development (TDD)

Adopting TDD principles in the elevator coding challenge can lead to a more deliberate and systematic approach to development. Writing tests first helps to drive the design and implementation of the elevator system.

- Continuous Integration (CI)

  Unit tests are essential for setting up a continuous integration pipeline. Automated tests can be executed on each code commit, ensuring that the elevator system remains functional as new code is integrated.

- Scalability and Extensibility

  As the elevator system grows and evolves, unit tests provide a foundation for adding new features and extending functionality without introducing regressions.

In summary, unit tests are a critical aspect of the elevator coding challenge because they improve code quality, provide a safety net for changes, enhance maintainability, and ensure the elevator system behaves as expected. Investing time in writing comprehensive and well-structured unit tests pays off in the long run by reducing the likelihood of bugs, improving developer productivity, and increasing the overall reliability of the application.

Logic/Algorithm

- Use Built-in Data Structures and Collections

  Utilize C# built-in collections like *List<T>*, *Dictionary<TKey, TValue>*, and *Queue<T>* to manage elevator states, track floor button presses, and handle passenger queues efficiently.

- Avoid Unnecessary Code Duplication

  Encapsulate common elevator movement and floor handling logic in separate methods to avoid duplicating code across different parts of the application.

- Choose the Right Algorithm for the Task

  For sorting floors or optimizing elevator movement, consider using efficient sorting algorithms like *quicksort* or *mergesort* to minimize processing time.

- Optimize for Readability and Maintainability

  Use descriptive variable and method names to make the code self-explanatory. Add comments where needed to explain complex elevator control strategies or algorithms.

- Handle Exceptions Gracefully

  Use try-catch blocks to handle exceptional cases, such as invalid floor numbers or passenger queues, and provide meaningful error messages.

- Avoid Magic Numbers and Strings

  Define constants or enumerations for elevator states, floor numbers, or passenger-related values, making the code more readable and maintainable.

- Use LINQ for Data Manipulation

  Utilize LINQ for querying and manipulating passenger queues, floor requests, or elevator state management, as it simplifies code and enhances readability.

- Prefer Asynchronous Programming for IO-bound Operations

  For handling multiple elevators or passengers asynchronously, use async and await when performing I/O-bound operations like simulating elevator movement or handling button presses.

- Implement Proper Error Handling

  Gracefully handle exceptions in scenarios like incorrect input or invalid elevator operations. Log errors and provide helpful messages to assist with debugging.

- Use Recursion with Care

  Consider iterative approaches to manage elevator movement and passenger queues, especially for larger elevator systems, to avoid potential stack overflow issues.

- Profile and Optimize Performance

  Profile the elevator system to identify any performance bottlenecks. Optimize critical sections or algorithms that affect elevator movement and passenger handling.

- Unit Test Your Code

  Write unit tests to verify the correctness of the elevator's movement logic, floor handling, and passenger interactions. Automated tests ensure that the elevator system behaves as expected and remains reliable as you make changes.

By applying these industry best practices to the elevator challenge, you can build a well-structured, efficient, and maintainable C# application that accurately mimics elevator movement and provides a seamless experience for passengers.

C# Standards

C# coding standards are a set of guidelines and best practices that define how C# code should be written to ensure consistency, readability, maintainability, and code quality. In the context of the elevator challenge, following C# standards is essential for the following reasons:

Consistency

Consistent code is easier to read and understand. Adhering to C# standards ensures that all developers involved in the project write code in a uniform manner, reducing confusion and making collaboration more efficient.

Readability

Well-formatted and consistent code enhances readability. Elevator simulation code can be complex, and following standards makes it easier for other developers (and your future self) to grasp the code's logic quickly.

Maintainability

C# standards promote modular and well-organized code. This makes it easier to maintain and modify the elevator simulation as the project evolves or new features are added.

Debugging and Troubleshooting

Code that follows C# standards is less prone to errors and easier to debug. Properly named variables, clear comments, and well-structured methods help identify issues faster during the development process.

Code Reviews
Following C# standards simplifies the code review process. Code that adheres to best practices is more likely to pass code reviews smoothly and receive valuable feedback from peers.

Performance
While C# standards are primarily focused on readability and maintainability, following them can indirectly lead to improved performance. Well-organized code allows developers to identify and optimize bottlenecks more effectively.

Codebase Scalability
The elevator simulation project may expand over time. Adhering to C# standards makes the codebase scalable and easily maintainable as the project grows in complexity.

Reduced Technical Debt
Writing code without proper standards can result in technical debt, which may lead to future challenges and maintenance issues. Following standards helps reduce technical debt and ensures a clean, manageable codebase.

Team Collaboration
When multiple developers work on the same project, C# standards foster collaboration by providing a shared understanding of coding conventions and practices.

Best Practices Integration
C# standards often incorporate industry best practices, such as SOLID principles, which have been proven to produce high-quality, robust code. Integrating these practices ensures that the elevator simulation adheres to these design principles.

Remember that C# coding standards may vary based on the organization or development team. Some common standards include naming conventions, indentation rules, usage of comments, guidelines for class and method structure, and best practices for exception handling. By adhering to the agreed-upon standards, your team can build a reliable and maintainable elevator simulation application that meets industry expectations.

Validation
In the context of the elevator challenge, validation in C# is crucial to ensure that the elevator simulation operates correctly, efficiently, and safely. Validation involves checking the input data and user interactions to ensure they meet certain criteria and adhere to expected rules. Here's why validation is important and how it can be applied to the elevator challenge:

Safety
The elevator simulation should prevent any unsafe or invalid operations, such as moving to a non-existent floor, opening doors while the elevator is in motion, or exceeding the elevator's weight capacity. Proper validation can help avoid potentially hazardous situations.

Data Integrity
Validate input data to prevent incorrect or malformed data from being processed. For instance, ensure that floor numbers provided by users are within the valid range, or verify that passenger information is complete and accurate.

User Input

Validate user interactions with the elevator system, such as floor button presses and passenger requests. This ensures that only valid requests are processed, avoiding unnecessary movements or erroneous passenger assignments.

Exception Handling

Implement appropriate exception handling for invalid scenarios. For example, throw exceptions when attempting to move to an invalid floor or when a passenger request cannot be fulfilled due to capacity constraints.

Preventing Inconsistent States

Validate the elevator's state to avoid inconsistencies. For instance, the elevator should not accept floor requests when its doors are open or reject passenger requests if the elevator is already full.

Performance Optimization

Validation can also help optimize the performance of the elevator simulation. By validating inputs and user interactions early in the process, unnecessary calculations or movements can be avoided, leading to more efficient elevator operations.

Error Reporting

Validation helps in providing meaningful error messages or feedback to users when they make invalid requests. Clear and informative error messages improve the user experience and facilitate troubleshooting.

Security

In scenarios where the elevator simulation involves access control (e.g., keycard authentication to access certain floors), validation ensures that only authorized users can make specific requests.

Improving User Experience

For the elevator simulation, consider providing a clear and intuitive user interface that displays elevator status, current floor, and buttons for floor selections. The user interface should be designed to be easy to understand and use, reducing the chance of user errors and frustrations.

Meaningful Feedback on User Actions

When a user interacts with the elevator simulation, provide immediate and meaningful feedback. For example, display messages indicating that their floor selection has been registered, the elevator is arriving, or that their request cannot be fulfilled due to capacity constraints.

Validation of user input to feedback to the user if their command was valid, or what their command is doing is key to a successful application. Consider using packages such as FluentValidation to help.

Error Handling

Error handling in C# is essential for the elevator challenge to create a reliable and user-friendly elevator simulation. Here's why error handling is important in the context of the elevator challenge and some best practices to follow:

Importance of Error Handling in the Elevator Challenge

Safety and User Experience

Proper error handling ensures the safety of passengers and the smooth operation of the elevator. By handling errors gracefully, you can prevent hazardous situations and provide meaningful feedback to users, enhancing their experience.

Robustness
Error handling makes the elevator simulation more robust, allowing it to recover from unexpected events or invalid user inputs. This prevents crashes or system failures during elevator operations.

Debugging and Troubleshooting
Effective error handling assists in debugging and troubleshooting the elevator simulation. When errors occur, relevant information and stack traces can be logged, helping developers identify the root cause and fix issues quickly.

Preventing Unexpected Behaviours
With proper error handling, you can prevent the elevator from entering inconsistent states or executing undefined operations. For example, the elevator should not attempt to move between non-existent floors.

User Feedback
When users interact with the elevator simulation, error handling provides meaningful feedback, guiding them on how to correct their actions. Clear error messages help users understand why their requests cannot be fulfilled.

Best Practices for Error Handling in the Elevator Challenge

Use Try-Catch Blocks
Wrap critical sections of code, such as elevator movement or passenger interactions, with try-catch blocks. This allows you to catch and handle exceptions that may occur during execution.

Catch Specific Exceptions
Catch specific exceptions related to elevator operations, such as *InvalidFloorException* or *CapacityExceededException*. Avoid catching generic exceptions like Exception, as they can hide specific issues.

Provide Clear Error Messages
Include meaningful error messages in exception handling. For instance, if a user selects an invalid floor, display a message like "Invalid floor selection. Please choose a valid floor."

Log Exceptions
Implement logging to record exception details, such as timestamps, error messages, and relevant context information. Log entries help in diagnosing and resolving issues in the elevator simulation.

Fail Gracefully
When unexpected errors occur, the elevator simulation should fail gracefully. Consider returning to a safe state or providing alternative options rather than crashing entirely.

Use Custom Exception Types
Create custom exception classes when necessary to represent specific elevator-related errors. This promotes code readability and allows for targeted exception handling.

Validate User Inputs
Before processing user interactions, validate inputs to ensure they meet specific criteria. Check if floor selections are within the valid range and validate passenger data before assigning them to the elevator.

Handle UI Exceptions
If the elevator simulation has a graphical user interface, catch exceptions related to UI events to prevent application crashes and provide feedback to users.

Unit Test Error Scenarios
Write unit tests to cover error scenarios, ensuring that exception handling and error messages work as expected during testing.

Review and Refine
Regularly review the error handling mechanism and refine it based on user feedback and observed issues. Continuous improvement ensures that the elevator simulation remains robust and user-friendly.

By incorporating these best practices into the elevator challenge, you can create a well-rounded and reliable elevator simulation that handles errors gracefully, enhances user experience, and operates smoothly under various scenarios.

User Feedback
User interaction and providing good system output are essential aspects of any software application, and they play a significant role in creating a positive user experience. In the elevator challenge you are limited to a console application which has limited graphical abilities. This makes it all the more important to feed back to the user what the system requires and what the system is doing.

Here's why these elements are important:

Importance of User Interaction
Engagement and Satisfaction
User interaction makes the application more engaging, encouraging users to actively participate and explore its functionalities. A well-designed user interface and intuitive interaction flow contribute to user satisfaction.

Efficiency and Productivity
Intuitive user interaction streamlines the user's workflow and reduces the time and effort required to accomplish tasks. Users can work more efficiently and be more productive.

User Empowerment
A good user interface empowers users by providing clear and accessible controls, options, and information, giving them a sense of control over the application.

Error Prevention
Intuitive interaction design can help prevent user errors and misunderstandings. It guides users through the application, reducing the likelihood of making mistakes.

Learnability
A well-designed user interface with consistent interaction patterns makes it easier for new users to learn how to use the application quickly.

Accessibility and Inclusivity
User interaction should consider accessibility guidelines to ensure that all users, including those with disabilities, can use the application effectively.

Brand Perception

The quality of user interaction directly influences the user's perception of the application and the brand behind it. A positive experience builds trust and brand loyalty.

Importance of Providing Good System Output:

User Feedback and Guidance

System output, such as messages and notifications, informs users about the status of their actions and provides guidance on how to proceed. Clear and meaningful output reduces confusion and helps users understand what is happening.

Error Handling and Recovery

When errors occur, good system output provides informative error messages that guide users on how to correct their actions or recover from the issue.

Transparency and Accountability

System output should be transparent and informative, especially when the application performs critical operations. Users should be informed about what the system is doing on their behalf.

Real-time Feedback

For interactive tasks or simulations like the elevator challenge, real-time system output informs users about the consequences of their actions and provides immediate feedback on the application's response.

Progress Indication

Long-running tasks or processes benefit from progress indications to let users know that the application is still working and hasn't stalled. Consider feeding back to the user that the elevator is moving to its destination.

Confirmation and Validation

System output can provide confirmation messages to verify that a task has been completed successfully, giving users peace of mind. Imagine stepping into an elevator, have the doors close, and there are no displays or buttons to interact with the elevator, or know what it is doing!

In summary, user interaction and providing good system output are vital for creating a user-friendly and effective software application. A well-designed user interface and informative system output enhance user satisfaction, increase efficiency, prevent errors, and build a positive brand perception. These elements are fundamental to fostering a successful user experience and ensuring the application meets the needs and expectations of its users.