Modernizing Legacy Software

Ferry Booking System

Version 1.0

11/19/2022

N8 Swalley

Luka Popovic

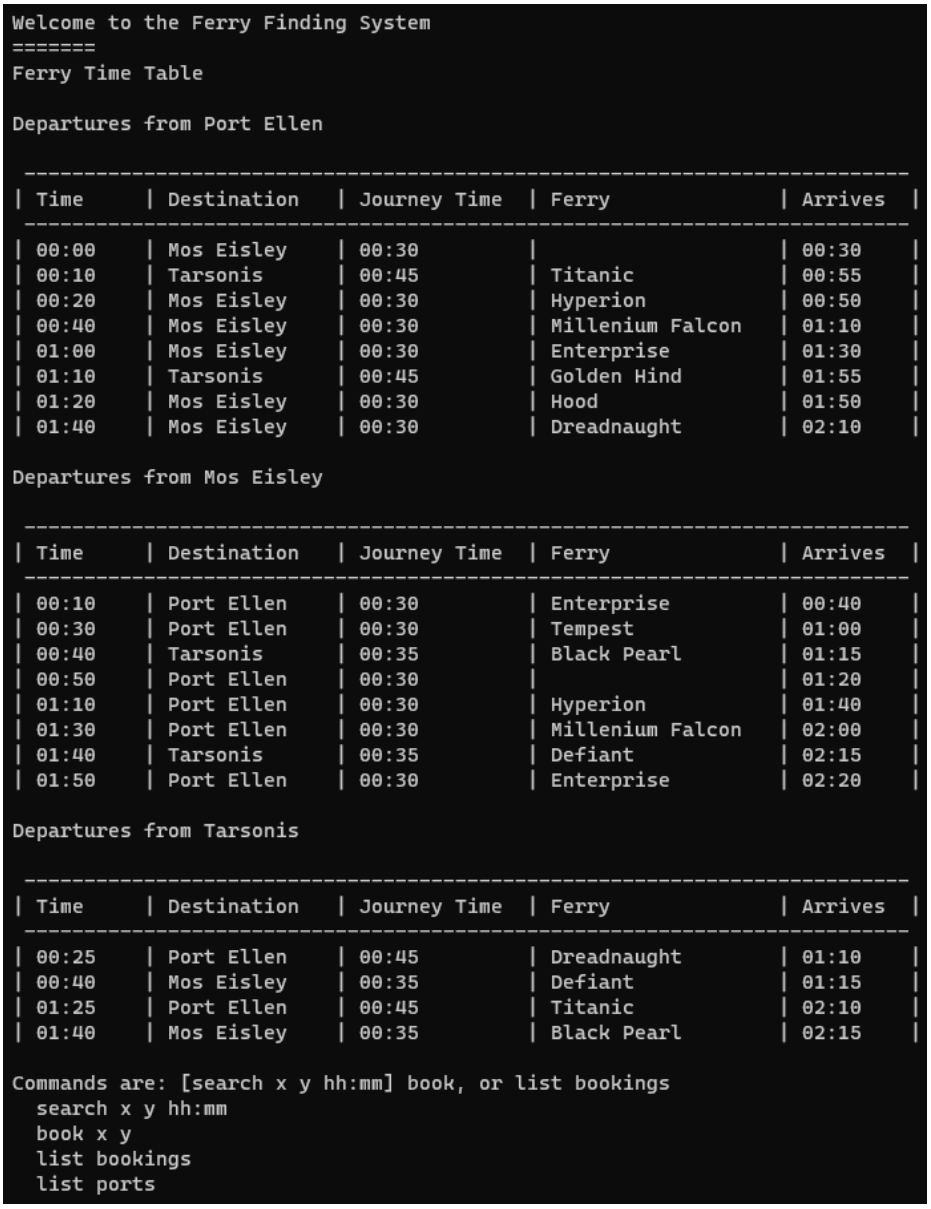
Table of Contents

Version History

Version Number	Date Modified	Author	Description of Change
1.0	11/19/2022	N8 Swalley	Creation

Original Application

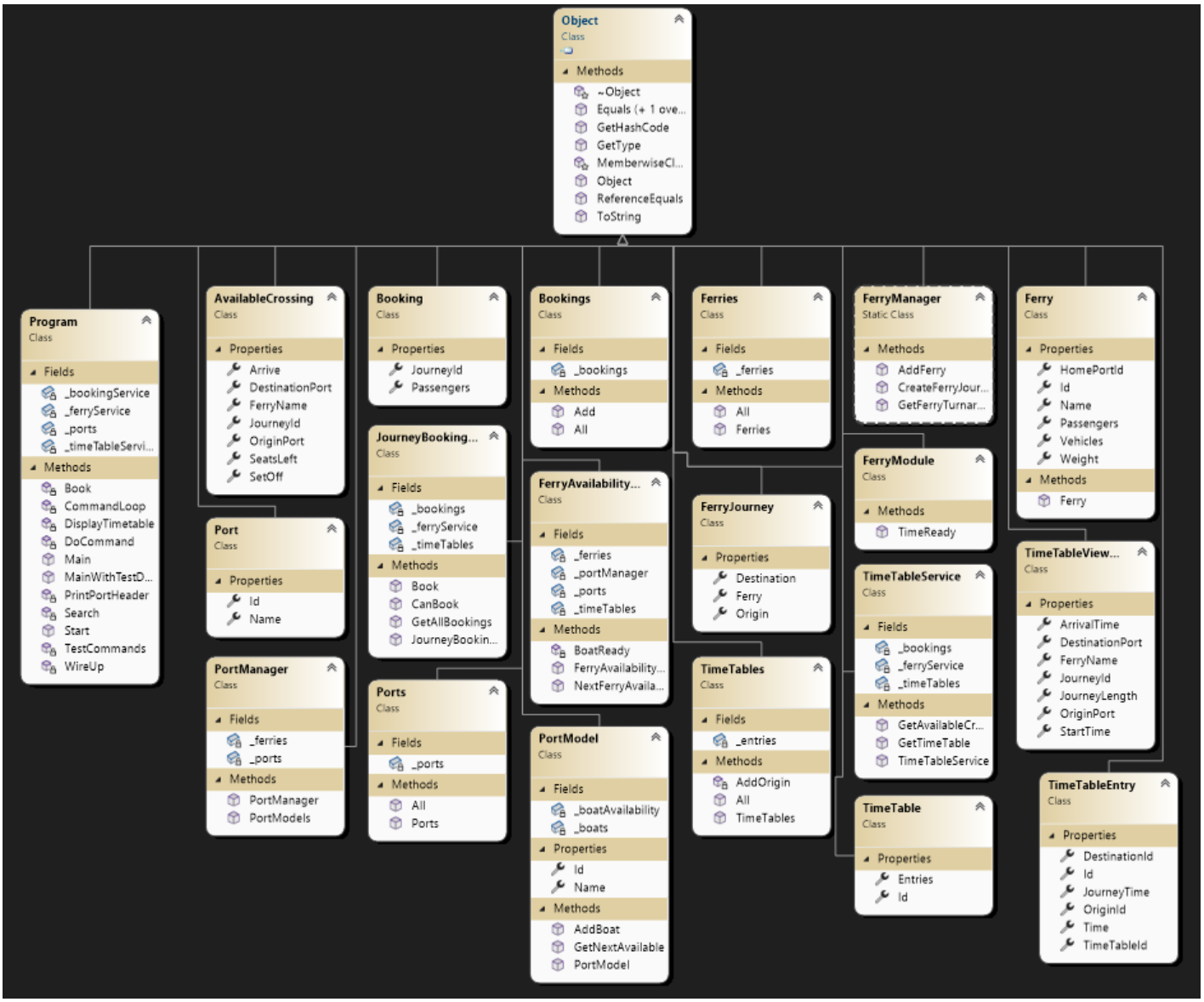
The **Ferry Finding System** is a legacy system application that was designed to allow users to search from a list of available ferries and book a journey. The screenshot below demonstrates the end-user experience. The user has the ability to perform several different commands in the ferry booking system. The user can quickly search the table of available ferries, book journeys, list the ports, and list bookings. This legacy system meets the needs it was originally designed for, but doesn't allow for much growth. This report will outline the modernization process of the ferry booking legacy system in C#.



```
Welcome to the Ferry Finding System
=======
Ferry Time Table

Departures from Port Ellen

 ------------------------------------------------------------------------
| Time     | Destination   | Journey Time  | Ferry             | Arrives  |
 ------------------------------------------------------------------------
| 00:00    | Mos Eisley    | 00:30         |                   | 00:30    |
| 00:10    | Tarsonis      | 00:45         | Titanic           | 00:55    |
| 00:20    | Mos Eisley    | 00:30         | Hyperion          | 00:50    |
| 00:40    | Mos Eisley    | 00:30         | Millenium Falcon  | 01:10    |
| 01:00    | Mos Eisley    | 00:30         | Enterprise        | 01:30    |
| 01:10    | Tarsonis      | 00:45         | Golden Hind       | 01:55    |
| 01:20    | Mos Eisley    | 00:30         | Hood              | 01:50    |
| 01:40    | Mos Eisley    | 00:30         | Dreadnaught       | 02:10    |

Departures from Mos Eisley

 ------------------------------------------------------------------------
| Time     | Destination   | Journey Time  | Ferry             | Arrives  |
 ------------------------------------------------------------------------
| 00:10    | Port Ellen    | 00:30         | Enterprise        | 00:40    |
| 00:30    | Port Ellen    | 00:30         | Tempest           | 01:00    |
| 00:40    | Tarsonis      | 00:35         | Black Pearl       | 01:15    |
| 00:50    | Port Ellen    | 00:30         |                   | 01:20    |
| 01:10    | Port Ellen    | 00:30         | Hyperion          | 01:40    |
| 01:30    | Port Ellen    | 00:30         | Millenium Falcon  | 02:00    |
| 01:40    | Tarsonis      | 00:35         | Defiant           | 02:15    |
| 01:50    | Port Ellen    | 00:30         | Enterprise        | 02:20    |

Departures from Tarsonis

 ------------------------------------------------------------------------
| Time     | Destination   | Journey Time  | Ferry             | Arrives  |
 ------------------------------------------------------------------------
| 00:25    | Port Ellen    | 00:45         | Dreadnaught       | 01:10    |
| 00:40    | Mos Eisley    | 00:35         | Defiant           | 01:15    |
| 01:25    | Port Ellen    | 00:45         | Titanic           | 02:10    |
| 01:40    | Mos Eisley    | 00:35         | Black Pearl       | 02:15    |

Commands are: [search x y hh:mm] book, or list bookings
  search x y hh:mm
  book x y
  list bookings
  list ports
```

Class Diagram

The class diagram pictured below demonstrates the design of the original ferry booking system. This application is lacking several important features of object oriented programming including abstraction, encapsulation, inheritance, and polymorphism. It fails to make use of any structural, behavioral, or creational software design patterns often implemented in industry. This system is riddled with spaghetti code and is difficult to trace and test efficiently because it was not created with the intention to do so.



Initial Testing

Before any changes were made to the software, intricate testing of the system was documented below to help us better understand the needs of the client and user. The commands listed below are the original test cases created to test how the functions are actually implemented within the code.

```
1 reference | 0 changes | 0 authors, 0 changes
private static void TestCommands()
{
    DoCommand("help");
    DoCommand("list ports");
    DoCommand("search 2 3 00:00");
    DoCommand("search 2 3 00:00");
    DoCommand("book 10 2");
    DoCommand("search 2 3 00:00");
    DoCommand("book 10 10");
    DoCommand("book 10 1");
    DoCommand("search 1 2 01:00");
    DoCommand("book 4 2");
    DoCommand("book 6 8");
    DoCommand("search 1 2 01:00");
    DoCommand("search 1 3 01:00");
    DoCommand("search 1 3 01:30");
    DoCommand("book 5 16");
    DoCommand("book 16 16");
    DoCommand("search 1 3 00:00");
    DoCommand("list bookings");
}
```

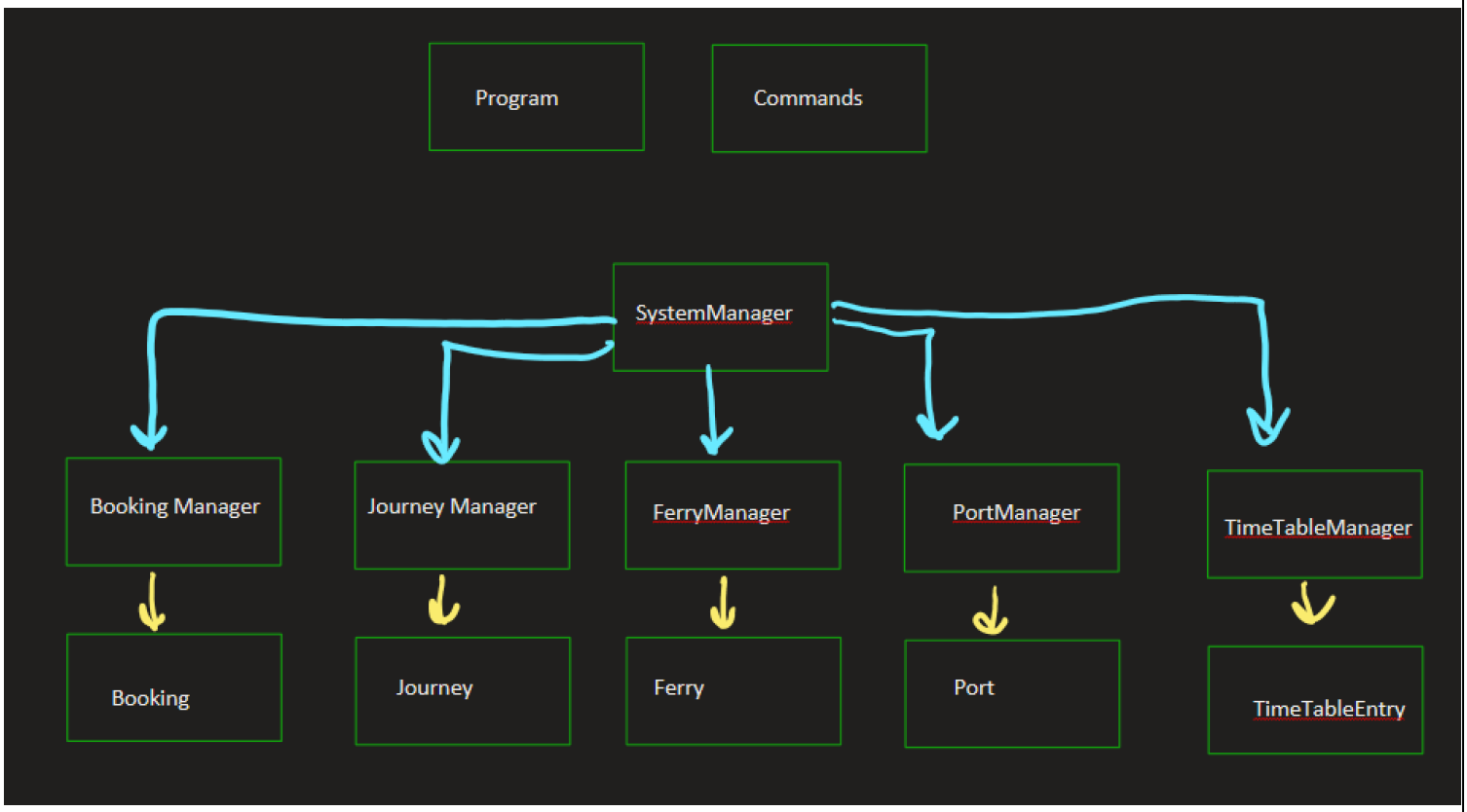
These test commands successfully test each operation within the program. The user is able to call the 'help' command to display the instructions on the screen. The user is able to call the 'list ports' command to list the ports in the system. The user can call the 'search' command to search the system for trips leaving port x and arriving to port y after the giving time. The user can call the 'book' command to book a trip to port x with y number of passengers. Finally, the user can call the 'list bookings' command to list all of the user's booked trips.

This is a very simple version of unit testing that successfully tests each function within a particular class, but the original author of this code did not account for future testing or improvements.

Preliminary Design

For the purpose of improving overall design and allow for more efficient testing in the future, we implemented a second version of the program called 'FerryLegacy.Tests'. This will allow us to make use of the built-in unit test feature in Visual Studio, efficiently and safely test functions across the entire system, and keep a well-documented report of version history. This implementation will be very beneficial for future programmers and testers of this application.

In addition, we want to clean up the system by removing any and all unnecessary classes. For example, there was both a Booking and Bookings class which don't use any level of inheritance or polymorphism. There were countless other poorly coded classes and methods like this throughout the entire program. We will later outline the full changes to the system, but for now, we need to generate a preliminary class diagram that we want to model our improved system after. We discovered a powerful design pattern from a previous management system which essentially broke the problem down into managers, submanagers, and jobs. After tracing through the application and understanding the functions within each class, we developed a model that we hope to implement in the system. It essentially breaks our system up into specific tasks, which are monitored and carried out with the help of an overseeing manager class. Finally, one master class will oversee the submanagers which in turn, helps them carry out their designated tasks. Here is a simplified model:



As you can already tell, this model provides a very well-organized version of the program. It provided us with a roadmap of how we want to restructure the classes and combine similar functions. The SystemManager is intended to oversee the entire system and handle each submanager class. Each submanager class is responsible for handling its specified tasks and operations within the system.

Improving System Design

We made several improvements to the overall system design and consolidated many classes and functions. Specifically, we tried our best to combine similar behaviors in the program and improve the foundation. For example, the original application had broken up bookings, ferries, and corresponding journeys in a rather complicated manner. Classes like “AvailableCrossing”, “FerryJourney”, “FerryAvailabilityService”, and “JourneyBookingService” made it very difficult to trace and understand the different parts of the whole. Instead, we decided to rename confusing classes and implement a better solution for booking a journey on a ferry. Here is a specific list of each class we either removed, renamed, or reimplemented in the system:

- AvailableCrossing
- Bookings
- Ferries
- FerryAvailabiltyService
- FerryJourney
- FerryModule
- JoruneyBookingService
- TimeTable classes

After in-depth analysis and testing, these classes were determined to be the best areas of improvement for the system. Comments were added in each class to outline the edits we made, but here is a more detailed overview of the different classes we either added or improved in the system:

- SystemManager: Initializes and manages all submanager classes. Provides access to getters and setters of class objects (ex: getJourney). This was originally handled in the Program class, but we separated the Program class to handle commands and submanager classes separately.
- Commands: Handles all user commands that were previously called in Program class. We used this class to edit the start up commands, fix the bug with the display table, and add new commands to the system. In addition, we added several exception handlers to account for errors with running the commands.
- Program: We broke this class up into the SystemManager class and the Commands class. The new purpose of the program class is so act as the main function which initializes our system manager class and begins reading in commands from the console.
- Booking: Implemented a journey class object, added attributes such as weight and vehicles for new features
- BookingManager: Consolidated bookings and booking class, combined JourneyBookingService class functionality with BookingManger. Added new parameters for booking a journey, instead of being based on only the number of seats left, we added vehiclesLeft and weightLeft as well. Made effective use of SystemManager class for getting access to data.
- Ferry: Added Journey class object instead of relying solely on matching IDs. Also found that the ID was no longer being used in the program and thus served as dead code. We commented out the ID for that reason.
- FerryManager: Reimplemented methods to use Ferry and Journey objects as their parameters instead of the list of ports and timetable entries. Used json to read data from file and convert to list, created method to return list of all ferries

- Journey: Essentially changed the name of AvailableCrossing to Joruney.cs for consistency reasons. Also combined attributes from FerryJourney class. In addition, we cleaned up the attributes by using class objects and better variable names. Finally, we added weight as an attribute.
- JourneyManager: This is essentially took over for the FerryAvailabilityService and TimeTableService classes. JourneyManager made use of the SystemManager class by creating a list of possible journeys. We added methods to return a specific journey, list all the journey's and available journey's. This class is very similar to the FerryManger and BookingManager classes, as it rightfully should be.
- Port: We added two list attributes, one for ferries and one for TimeTable entries.
- PortManager: Combined functionality from ports class with current PortManager class. In addition, this class handled all functions relating to ports. This class is used to assign ferries to their ports, assign a time table slot to a port, list all the ports, find the next available ferry, and move ferries to a new port. This class is modeled very closely to the other submanager classes.

Fixing Bugs

In this first round of testing, I found several bugs within the program that need to be fixed before any further improvements can be made. The most obvious bug that was first found was that the timetable is missing a ferry name in the first timetable slot. To fix this bug, I relied on the Saff Squeeze method. The Saff Squeeze is a systematic technique for deleting both test code and non-test code from a failing test until the test and code are small enough to understand. Its simple way of homing in on a bug without relying on the debugger in Visual Studio. All testing was completed within the FerryLegacy.Tests solution.

Once we homed in on the bug, we were able to find a better way to display the timetable. Here, we were able to effectively use our previously implemented SystemManager class. This manager class provides access to all the ports and all the journeys, which made it much easier to display the timetable than before. After reimplementing this method and testing other functions in the commands class, we resolved the bug. In turn, we were able to display the correct timetable. This is yet another example of how a well-designed system can improve testing and debugging practices within the application.

Adding Features

Although many of the features we added were briefly outlined already, this section will provide a detailed overview of the many features we implemented in the system. One feature that we implemented was the ability for the user to book a journey based on the number of vehicles and weight, because ferries in real life transport more than just people. We saw this as a very crucial improvement that could offer new functionality to the application. To accomplish this, we added additional attributes (vehicle, weight) to the Booking, Ferry, and Journey classes. Next, we updated the methods in our BookingManager and JourneyManager classes and any other corresponding classes. Finally, the last thing to do was update our commands and methods. We had to update the list bookings method in order for the console to print the full details of the journey.

```
Book:
  book a b c d
       where a - journey id
       where b - number of passengers
       where c - number of vehicles
       where d - total vehicle(s) weight in tons
```

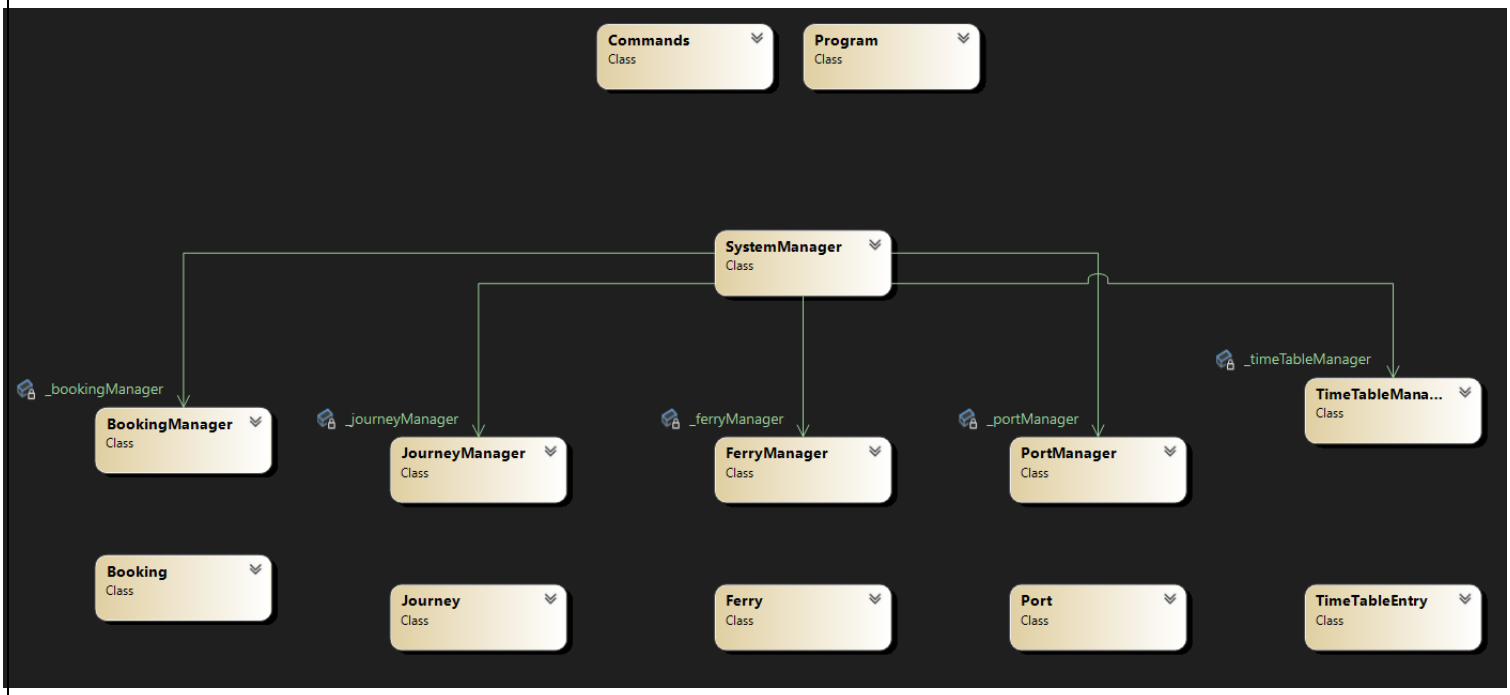
Some other features we added to the system gave the user the ability to clear the console, display the timetable and exit the program. In addition, we wanted to add a fun command to hopefully increase user engagement and improve the overall end-user experience. If a user is traveling alone, we want to give them an option to book a random trip. Here is our implementation:

```
// Random Trip Command - Books a random trip for the user
1 reference | 0 changes | 0 authors, 0 changes
private static void BookRandomCommand()
{
    try
    {
        // Picks random int less than 20
        Random rnd = new Random();
        int randID = rnd.Next(20);

        // books an available journey for one passenger and no vehicles
        bool booked = SystemManager.Book(randID, 1, 0, 0);

        if (booked)
            Console.WriteLine("Booked");
        else
            Console.WriteLine("Cannot book that journey");
    }
    catch (Exception)
    {
        BookingError();
    }
}
```

This creates a fun and interesting solution for indecisive clients. Now, users can type “random trip” and a random journey will be booked for one single passenger. Overall, these newly added features provide very useful functionality in the program and are clear improvements within the system.

Report Summary

This legacy system started out as a cluster of spaghetti code. Over the course of this semester, our team of skilled software engineers relied on learned programming techniques and software design principles to collaborate and modernize this application. In the end, our team was able to successfully test and document the intricacies of the original application, develop a useful framework and class diagram, improve the overall system design, add new features, fix bugs, and develop a plan for the future. Here is the class diagram of our finished, modernized legacy system:



Future Directions

If a team wishes to continue developing this program, they will want to consider focusing on these improvements. Internally, this program could be more optimized in terms of data structures. More specifically, a new process for adding ports and ferries could be implemented. Also, adding more commands to allow the user to access more information about the trip could benefit the functionality of the application. Externally, the program could add features like the pricing of each ferry as well as a function that can find the cheapest trip. This could help the user when deciding where to travel. The Interface is very simple and does its job but can also be improved like anything else. Making the options clearer and more highlighted will help the readability and intuitiveness of the application. Finally, enhancements to the end-user experience and improvements to the user interface would greatly improve this application.