Realisation: CircusTrain

By: Simeon Markov

Institution: Fontys UAS

Course: IN-SDE4

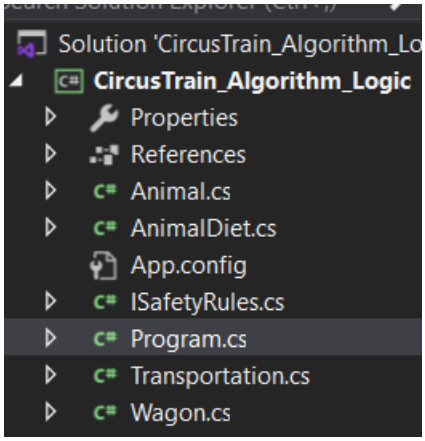
Date: 2026-02-28

Introduction

This document goes over the structuring and implementation of the CircusTrain algorithm and provides the code documentation of the program.

Project structure

The project is a standard console app, where all classes are under a single project. Each class has its own responsibility:



Code documentation

For the inline commenting, MS C# conventions for commenting are kept.

Program.cs: Entry point of the program. The input is handled there.

```
class Program
{
    static void Main(string[] args)
    {

        Animal dog = new Animal();

        dog.Diet = "Carnivore";
        dog.AnimalSize = "Large";

        Animal catphish = new Animal();

        catphish.Diet = "Carnivore";
```

Figure 1: Program class part01

In the **Main()** method all the necessary objects are initialize (Animal objects in Figure 1). After the object is initialized its attributes are called (Diet, AnimalSize) and values are assigned.

```
    List<Animal> listOfAnimals = new List<Animal>();
    listOfAnimals.Add(dog);
    listOfAnimals.Add(cat);
    listOfAnimals.Add(turtle);
    listOfAnimals.Add(bird);
    listOfAnimals.Add(catphish);

    Wagon wg = new Wagon(listOfAnimals);
    Transportation.Distribute(wg);

}
```

Figure 2: Program class part02

After we have all the animals, a list is created to store those animals and then it is passed to a wagon, after which an static class **Transportation** with its method for **Distribution()** is called as it takes wagon as a parameter.

Animal.cs: Contains all the attributes and methods for animals (parent class).

```
// A parent class denoting common attributes/behaiviours across child clases
class Animal
{
    private string diet;
    private string animalSize;

    protected internal Dictionary<string, int> size = new Dictionary<string, int>
    {
        { "Small", 1 },
        { "Medium", 3 },
        { "Large",  5}
    };
```

Figure 3: Animal class part01

Fields diet and animalSize are private and are accessible only through public properties. The points per size are stored in dictionary for key-value pair retrieval.

```
public string Diet
{
    get
    {
        return diet;
    }
    set
    {
        // Enum approach
        if (!Enum.TryParse<AnimalDiet>(value, true, out _))
        {
            throw new ArgumentException("Invalid diet");
        }

        /* Array approach
        if (!arrAnimalDiet.Contains(diet))
        {
            throw new ArgumentException("Invalid diet");
        }*/

        diet = value;
```

Figure 4: Animal class part02

The public property Diet gets the diet (private field) and the under the setter it is set to accept value from incoming input as before that a validation check is performed via the **Enum.Parse()** method. That method checks If the given value is found in the AnimalDiet enumeration, ignoring case (case-insensitive,) if its not found then throw an exception with a readable message.

Wagon.cs: Used for storing/preserving the animals:

```
// Used for storing/preserving the animals
class Wagon : ISafetyRule
{
    public const int MAX_Capacity = 10;
    protected internal List<Animal> animals;

    private readonly HashSet<string> possibleSets = new HashSet<string>()
    {
        "Carnivore Small|Herbivore Medium",
        "Carnivore Small|Herbivore Large",
        "Carnivore Medium|Herbivore Large",
        "Herbivore Small|Herbivore Medium",
        "Herbivore Small|Herbivore Small",
        "Herbivore Medium|Herbivore Medium",
        "Herbivore Large|Herbivore Large",
        "Herbivore Small|Herbivore Large",
        "Herbivore Medium|Herbivore Large"
    };
```

Figure 5: Wagon class part01

Each wagon has its own max capacity, which is 10 points. It accepts a list of animals and the possible pairing sets are stored in hashset, which is later used for symmetric pairing. The class implements the abstract method for the Interface **ISafetyRule**.

```
    public Wagon(List<Animal> listOfAnimal)
    {
        animals = listOfAnimal;
    }

    public bool CheckSafety()
    {
        if (animals.Count < 2)
        {
            return false;
        }

        for (int i = 0; i < animals.Count; i++)
        {
            for (int j = i + 1; j < animals.Count; j++)
            {
                CanPair(animals[i], animals[j]);
            }
        }

        return true;
```

Figure 6: Wagon class part02

After the wagon's list accepts all the animals passed into it, in the **CheckSafety()** method that list is being iterated for checking all the possible pair of animals that could fit in the

wagon. For the pairing the **CanPair()** is responsible, in short what it does is it takes minimum two animals for a pair and then looks into the hashset to check if that pair is included there if not pass. The method also sorts by diet and if the diets happen to be the same it compares by animal sizes, this is done to for the symmetric pairing.

Transportation.cs: Transportation logic for the animals. The logic behind it is that a wagon accepts animals until it overloads and the last animal to overload the wagon is moved to new wagon:

```
static class Transportation
{
    public static List<Wagon> Distribute(Wagon initialWagon)
    {
        initialWagon.CheckSafety();
        List<Wagon> allWagons = new List<Wagon>();

        Wagon currentWagon = new Wagon();
        allWagons.Add(currentWagon);

        Console.WriteLine("Distribution Starting...");
```

Figure 7: Transportation class part01

A list of all used wagons is created for tracking how much wagons will be need to transport given number of animals. And another wagon object is created for tracking the current wagon (overloading tracking).

```
foreach (var animal in initialWagon.animals)
{
    int animalSizeScore = animal.size[animal.AnimalSize];

    int currentLoad = currentWagon.CalculateCurrentSize();


    if (animal.Diet.Equals("Carnivore") && animal.AnimalSize.Equals("Lar
    {
        Wagon dedicatedWagon = new Wagon();
        dedicatedWagon.animals.Add(animal);
        allWagons.Add(dedicatedWagon);
        Console.WriteLine("Carnivore Large is moved to seperate wagon");

        continue;
    }
```

Figure 8: Transportation class part02

The initial wagon is being iterated and a variable for each animal size score is initialized for overload checking. The first condition is that if an animal is carnivore and large it is moved to a new single wagon.

```
        // Check for overload
        if (currentLoad + animalSizeScore > Wagon.MAX_Capacity)
        {
            Console.WriteLine($"[Wagon {allWagons.Count}] is full (Current:
            Console.WriteLine($"Adding animal: {animal.Diet} {animal.AnimalS

            // Create new wagon
            currentWagon = new Wagon();
            allWagons.Add(currentWagon);
        }

        currentWagon.animals.Add(animal);
        Console.WriteLine($"Placed animal ({animal.Diet} {animal.AnimalSize}
    }

    Console.WriteLine("\nDistribution Completed");
    Console.WriteLine($"Total Wagons used: {allWagons.Count}");

    return allWagons;
```

Figure 9: Transportation class part03

If the current wagon overloads, print message how much is loaded the current wagon and then create a new one. After there are no animals left a message is printed to the console indicating the process is over and how many wagons were used.

Conclusion

The algorithmic problem is solved and now the program successfully distributes given number of animals into wagons and print indicating messages about what type of animals are together in a wagon, how much space do they take and the total amount of wagons required for the animals transportation.