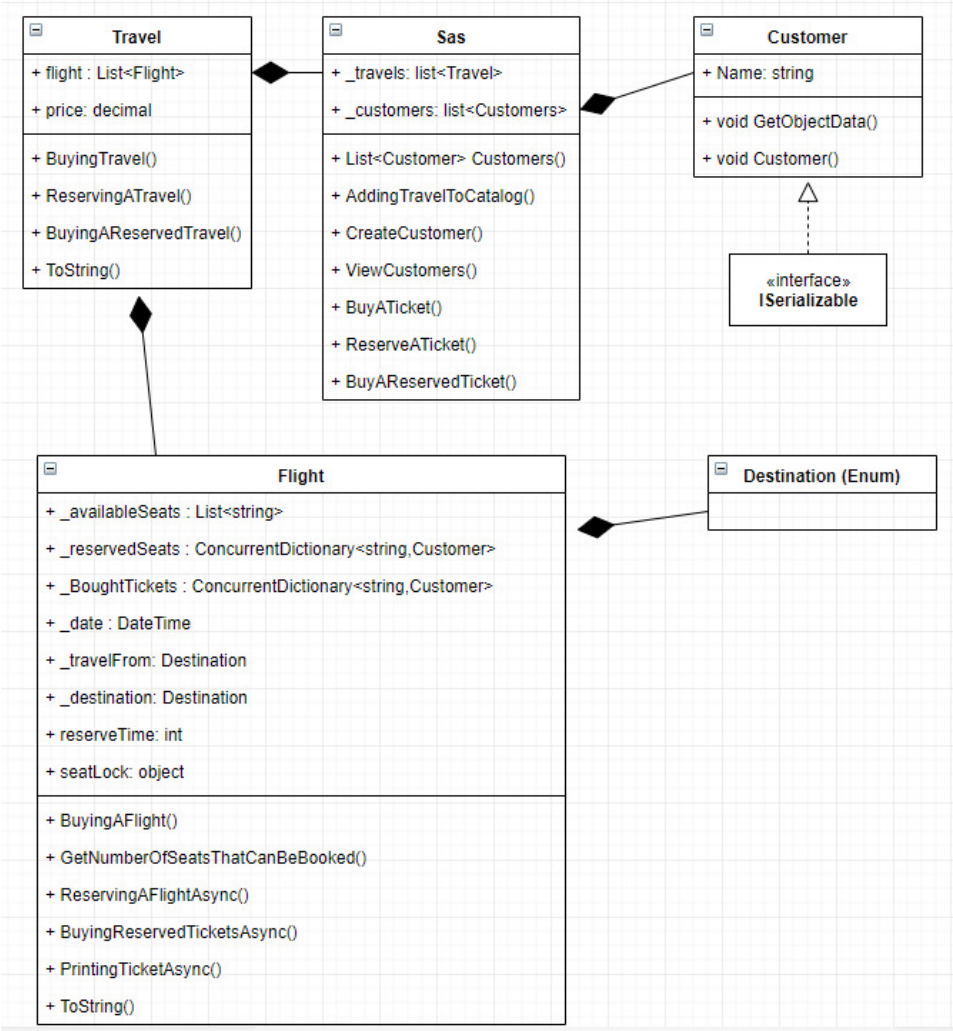
Obligatorisk opgave 3

Løsningen på denne opgave er lavet via consollen, hvor der er lavet et opbygning som nedenstående UML.



Der er lavet en base Sas klasse, der holder indformation om kunder og de tilgængelige rejser som Sas tilbyder. Den har ikke noget særligt over sig andet end den har en CreateCustomer metode som tager en til flere kunder. Det har jeg valgt, da jeg har lavet kunder serializable og derved indlæser en liste af kunder via XML og den ene metode kan få placeret alle de nye kunder på Sas kundeliste, for at holde disse, da der ikke er koblet en database på.

Før program.cs virker kræver det således at man placerer xml filen i på d:\ eller, at man retter stien til.

En rejse består af en til flere enkelt rejser og en tur retur med transit ville altså holde 4 rejser. For at få et lidt pænere print, har jeg overskrevet ToString og jeg har brugt string builder på flight til at samle data omkring de enkelte sæder man køber når man booker en rejse.

Den klasse der er allermest kød på er flight klassen, da det er her, hvor der er et begrænset antal pladser tilgængelig på et fly og hvor disse bliver reserveret og købt. Der er lavet kommentarer i filen, så der fortælles lidt mere i detaljer hvordan de enkelte metoder virker, men grundtanken er at man starter med liste af flysæder som bliver deklareret når en flight klasse bliver initialiseret – I mit eksempel har jeg 12 pladser på mine fly.

Metoder der skal hente data fra denne liste skal forbi en lås, således at der kun er en tråd, der kan fjerne sæder fra listen, så man ikke risikere at der dobbeltbookes (med vilje). Den samme lås bliver brugt i begge metoder, da det er den samme liste der beskyttes – så det er altså kun én af de to metoder (reservering eller køb) der kan laves.

Der er herefter to Concurrent Dictionaries, hvilket jeg har valgt da de er thread safe og gør det muligt for trådede uafhængig af hinanden at reserver/købe et sæde eller købe et reserveret sæde. Dette er valgt da key i dictonary er den sæde string, der er lås omkring – ergo kan to metoder ikke reserver samme sæde samtidigt. Men flere tråde kan tilgå og placere flere forskellige sæder samtidig.

Den mest simple af de 3 primære metoder er køb af en plads, som kan ses nedenstående.

```
public void BuyingAFlight(int numberOfSeats, Customer customer)
{
        // _availableSeats are not threadsafe, so in order to make sure no 2 threads book same seat
        // a lock is placed before accessing the array
        lock (seatLock)
        {
                int seatsToReserve = GetNumberOfSeatsThatCanBeBooked(numberOfSeats);
                StringBuilder seatInfoStringBuilder = new StringBuilder("The following seats was bought: ");

                for (int i = 0; i < seatsToReserve; i++)
                {
                        string seat = _availableSeats[_availableSeats.Count-1];
                        _availableSeats.Remove(seat);
                        _boughtTickets.TryAdd(seat, customer);
                        seatInfoStringBuilder.Append(seat + " ");
                }
                // Removes a whitespace
                seatInfoStringBuilder.Remove(seatInfoStringBuilder.Length - 1, 1);
                // Prints string builder
                Console.WriteLine(seatInfoStringBuilder);

        }

private int GetNumberOfSeatsThatCanBeBooked(int numberOfSeats)
{
        Boolean IsSeatsAvailable = (_availableSeats.Count >= numberOfSeats);
        var seatsToReserve = (IsSeatsAvailable) ? numberOfSeats : _availableSeats.Count;
        return seatsToReserve;
}
```

Den finder først ud af om de ønskede billetter er til rådighed og køber dem, hvis de er – hvis ikke køber den, dem der er tilrådighed. Dette er en antagelse, da man ligeså vel kunne have returneret med besked om at de ikke var tilgængelige.

Et køb sker ved at man finder det sidste sæde der er tilgængeligt (fylder flyet op fra slutningen af). Så fjerner man det fra _availableSeats og prøver at tilføje det til købte billetter. Her kunne man lave noget defensive programming i forhold til om ”try” ikke lykkes – men da vi er inde i låsen, burde det ikke kunne ske og har valgt at lade være.

Så tilføjer vi info om, hvilket sæde der er købt til en stringbuilder, som udskrives når loop afsluttes.

Når vi så kigger på metoden, der reserver pladser, så starter den ud på lidt samme måde på nær, at der laves en liste til at tage imod de reserverede pladser.

Pladserne placeres i en reserved dictonary fremfor bought og er ellers ens – men herefter kommer der en vente tid. Når ventetiden er overstået går den ind og kigger i de købte billetter om de reserverede sæder er her – hvis de er der, er de blevet købt, og der skal ikke ske mere (kunne evt. fjerne sæder fra

den reserveret liste, men tænkt, at det var en måde at se hvor mange sæder bliver reserveret inden køb).

Hvis sæderne ikke er købt fjernes de nu fra den reserverede liste og bliver placeret retur i den available liste.

```
public async void ReservingAFlight(int numberOfSeats, Customer customer)
{
        List<string> _seatsReserved = new List<string>();

        // Using same lock as in buying as both method try to acces _availableSeats
        lock (seatLock)
        {
                int seatsToReserve = GetNumberOfSeatsThatCanBeBooked(numberOfSeats);
                StringBuilder seatInfoStringBuilder = new StringBuilder("The following seats was reserved: ");

                for (int i = 0; i < seatsToReserve; i++)
                {
                        string seat = _availableSeats[_availableSeats.Count - 1];
                        _availableSeats.Remove(seat);
                        _seatsReserved.Add(seat);
                        _reservedSeats.TryAdd(seat, customer);
                        seatInfoStringBuilder.Append(seat + " ");
                }

                // Removes a whitespace
                seatInfoStringBuilder.Remove(seatInfoStringBuilder.Length - 1, 1);
                // Prints string builder
                Console.WriteLine(seatInfoStringBuilder);
        }

        // Above is same as in bought above - but here i wait for a set wait time
        await Task.Delay(reserveTime);

        // When wait is over bought tickets is viewed to see if it have the reserved seat
        // if not the seat was not bought and it is returned to the available list.
        foreach (var seat in _seatsReserved)
        {
                if (!_boughtTickets.ContainsKey(seat))
                {
                        // Not storing the out parameter as i don't have a use for it here.
                        _reservedSeats.TryRemove(seat, out customer);
                        _availableSeats.Add(seat);
                }
        }
}
```

Den sidste af de tre store metoder er nedenfor og den går ind og ser om der findes pladser i den reserverede dictionary, hvor kunden stemmer overens med den kunde som prøver at købe dem. Hvis der er sæder der stemmer, prøver man at fjerne dem fra den reserverede dictionary – hvis det er vellykket tilføjer man det samme sæde til dictionary over købte billetter.

Før det giver mening at lave metoden til en asymmetrisk metode, så skal der gerne være noget der tager tid – så jeg leger at man nu skal udskrive billetterne, hvilket tager tid. Og awaiter denne opgave. Jeg har her lavet en lidt fjollet arkitektur, hvor man laver forbindelse til streamwriter i hvert loop, fremfor at vente tilbage efter, men det er lavet for sjov, så man kan se det som en tung opgave.

For netop at komme denne tunge opgave til hjælp er der implementeret en try delay mønster på udskrivningen, da det ofte vil have flere tråde der prøvede at få adgang til steamwriter tekstfilen før den sidste var blevet færdig, så her looper man lige tilbage efter en ventetid nogle gange, hvis det sker.

```
public async Task BuyingReservedTicketsAsync(int numberOfSeats, Customer customer)
{
        StringBuilder boughtReservedSeatStringBuilder = new StringBuilder("The following seats was bought: ");

        // Creates a counter to make sure it is known how many seat was bought
        int count = 0;

        // Iterate over the reserved seats dictionary
        foreach (KeyValuePair<string, Customer> seat in _reservedSeats)
        {
                // if the count is equal to the number of required seats i stop looking for more
                if (count == numberOfSeats)
                {
                        break;
                }

                // If the value in the dictionay equals the customer who reserved the item
                // and now wanna buy i try to remove it.
                // if succesfull i add it to the boughtticket dictionary and increase count.
                if (seat.Value == customer)
                {
                        if (_reservedSeats.TryRemove(seat.Key, out customer))
                        {
                                _boughtTickets.TryAdd(seat.Key, customer);
                                boughtReservedSeatStringBuilder.Append(seat.Key + " ");
                                try
                                {
                                        await PrintingTicketAsync(seat);
                                }
                                catch(Exception ex)
                                {
                                        Console.WriteLine("Exception Message: " + ex.Message);
                                }
                                count++;
                        }
                }
        }

        Console.WriteLine(boughtReservedSeatStringBuilder);
```

```
private static async Task PrintingTicketAsync(KeyValuePair<string, Customer> seat)
{
        int numberOfRetries = 3;
        int DelayOnRetry = 1000;

        for (int i = 0; i < numberOfRetries; i++)
        {
                try
                {
                        using (StreamWriter streamWriter = new StreamWriter("Tickets.txt", true))
                        {
                                await Task.Delay(5000);
                                await streamWriter.WriteAsync("-------Printing ticket for seat " + seat.Key);
                                return;
                        }
                }
                catch (IOException e) when (i <= numberOfRetries)
                {
                        Thread.Sleep(DelayOnRetry);
                }
        }

}
```

Endeligt er der lavet unit tests, hvor de tre krav er blevet testet, samt nogle ekstra scenarioer:

1. En kunde skal kunne reservere med henblik på køb af flybillet inden for et givet tidsinterval
2. Simulere en kundes køb der er succesfuldt
3. Simulere en kundes køb der ikke er succesfuldt