1. **Task1. Implement trip insurance payment calculation.**

Travel agency needs a program to calculate trip insurance payment in local currency.

Program will accept one parameter: touristName and make a call to database to load trip cost.

There is interface to load trips from db:

```
public interface ITripRepository
{
    TripDetails LoadTrip(string touristName);
}

public class TripDetails
{
    public string TouristName { get; set; }

    public decimal FlyCost { get; set; }

    public decimal AccomodationCost { get; set; }

    public decimal ExcursionCost { get; set; }
}
```

FlyCost – total cost to fly to a destination.

AccomodationCost – total cost for hotels and meal.

ExcursionCost – total cost for all excursions.

There is a formula to calculate insurance payment:

Payment = Constants.A * Rate * FlyCost

+ Constants.B * Rate * AccomodationCost

+ Constants.C * Rate * ExcursionCost;

Rate – is local currency rate. It should be loaded from http service via interface.

```
public interface ICurrencyService
{
    decimal LoadCurrencyRate();
}
```

Implement InsurancePaymentCalculator:

```
public class InsurancePaymentCalculator
{
    private ICurrencyService currencyService;
    private ITripRepository tripRepository;

    public InsurancePaymentCalculator(
        ICurrencyService currencyService,
        ITripRepository tripRepository)
    {
        this.currencyService = currencyService;
        this.tripRepository = tripRepository;
```

```
        }

        public decimal CalculatePayment(string touristName)
        {
            throw new NotImplementedException();
        }
    }
```

In unit tests CalculatorFactory is used.

2. **Task2. Implement trip insurance payment calculation with ability cache.**

After developing PaymentCalculator, it turned out that CurrencyService, which makes http call to Currency Rate Service is very slow. Call to the database from TripRepository is also very slow.

Implement ability to cache payment for each tourist. If cache does not have payment for tourist, it should be computed and put to cache. If there is a payment in cache, it should be used.

There should be ability to switch between simple PaymentCalculator and PaymentCalculator that uses cache. In order to do this, implement Factory:

```
public interface ICalculatorFactory
{
    ICalculator CreateCalculator();

    ICalculator CreateCachedCalculator();
}
```

CreateCalculator – Creates basic calculator which will always calculates insurance payment.

`CreateCachedCalculator` – Creates calculator with caching.

It is recommended to use Decorator pattern. No code duplication is allowed.

3. **Task3. Implement trip insurance payment calculation with logging and rounding**

Implement two additional methods in CalculatorFactory:

```
public interface ICalculatorFactory
{
    ...
    ICalculator CreateLoggingCalculator();

    ICalculator CreateRoundingCalculator();
}
```

`CreateLoggingCalculator() – should return a calculator with logging. At the very beginning, logger should write message “Start” and an the end after calculation, logger should write a message “End”.`

`Here is interface for logging:`

```
public interface ILogger
```

```
    {
        void Log(string message);
    }
```

```
CreateRoundingCalculator() – should return a calculator, which rounds payment so that we
do not have cents. For example:
      34.56$ is rounded to 35$
      34.50$ is rounded to 35$
      34.23$ is rounded to 34$
```

4. **Task4. Implement trip insurance payment calculation with dynamic features.**

Now, we need to have an ability to create a calculator with a different set of capabilities, For example,

CachedLoggingPaymentCalculator – it calculates, caches and logs the result.

RoundingCachedPaymentCalculator – it calculates, caches and rounds the result.

There can be a lot of different combinations of calculator responsibilities.

Implement a method in CalculatorFactory, which will assign responsibilities dynamically.