Order processing

Authors: TS + PW

Introduction

Your task is to prepare an order processing system with two modules: `Payments` and `Shipment`. While right now the code is combined in directories, imagine a situation where these modules are separated and developed by different teams.

Payment processing

Customer orders are currently stored in two separate databases `LocalOrdersDB` and `GlobalOrdersDB`. Future changes in their internal structure should not affect the "client" == `Program.cs` in any way. Furthermore, all orders should be processed together by the system, regardless of destination country. `Order` contains information about selected payment methods: `SelectedPayments`. Each `Payment` corresponds to a particular `PaymentMethod` and a maximum amount that can be paid via this method. For now, the system should support three payment methods: `PayPal`, `Invoice`, `CreditCard`. However, it should be designed to allow adding new ones in the future (adding a new payment method shouldn't require modifying existing ones). Payment processing iterates over selected methods and attempts to pay the maximum amount assigned to each one until the total price is paid. The processing order is always the same: `PayPal` -> `Invoice` -> `CreditCard`.

An attempt to pay can be unsuccessful, implemented as follows:

- Every third `Invoice` payment fails.
- Every `PayPal` payment has a 30% probability to fail (Random set with seed `1234`).
- `CreditCard` payments always succeed. If a payment was unsuccessful, it is skipped and processing continues with the next method. Successful payments should be added to `FinalizedPayments` with the amount paid.

During payment processing, the `Status` of an `Order` should be updated accordingly:

- `WaitingForPayment` when it is unpaid,
- `PaymentProcessing` when it is partially paid,
- `ReadyForShipment` when it is fully paid and can be processed via the `Shipment` module.

Design requirements for Payments

- Adding new payment methods should be possible without modifying existing `PaymentMethods` components, except the `PaymentMethod` enum.
- Modifying one payment method should be possible without modifying any others.
- If the full amount has already been paid, any subsequent payment methods shouldn't be processed.
- Existing components provided with the task should be used.

Shipment process

Add a filter that selects all orders whose status is `ReadyForShipment` and implement the shipment process for these orders. Print labels for each order and print labels for parcels created from orders. For now, the system should support two shipment providers: `LocalPost` and `Global`. However, it should be designed to allow adding new ones in the future (adding a new provider shouldn't require modifying existing ones).

ShipmentProvider selection:

- `LocalPost` is used when the destination country is `Polska`.
- `Global` is used for orders sent to other countries.

Tax calculation:

- The database `TaxRatesDB` contains value added tax rates by country.
- Assume that it is provided by a third party, so its internal structure isn't known to ShipmentProvider (and may change in the future).
- Tax is calculated as X percent of `PaidAmount`, where X is the rate obtained from the database.

Labeling and parcelling:

- Each ShipmentProvider has its own label generation logic and groups orders into parcels differently.
- `LocalPost` packs all orders in a single `IParcel`
- `Global` separates orders into multiple `IParcels`, grouping them by `Recipient.Country`

First, register all your orders in the proper provider and for each order print a label appropriate for the provider (`Printer.PrintLabel`). Provide logic for label printing to be reusable in a future implementation. To achieve that, use the `ILabelFormatter` interface. Labels for `LocalPost` don't include destination country, for example:

```
Shipment provider: LocalPost
Janina Osiwiecka
Kamionka 3/34
Lipsko
25-895
```

Labels for `Global` include destination country, for example:

```
Shipment provider: Global
Lea Mathias
9054 Share-wood
Manhattan
90561
USA
```

Next, print the summary for each `IParcel` created by providers that were used during registration. Preferred format:

```
```

```
Shipment provider: Global
TotalPrice:     246,00, TotalTax:       0,00, TotalPriceWithTax:     246,00
--------------------------------------------------------
-------------------------Hawaii-------------------------
 OrderId           Amount                Tax   AmountWithTax
       1           246,00               0,00          246,00
TOTALS:            246,00                             246,00

--------------------------------------------------------
```

Design requirements for Shipments

- ShipmentProvider objects are "heavy", so use lazy initialization: create only when the first order wants to register in the provider.
- To generate summary of a `IParcel`, please use provided `ISummaryFormatter` interface with default implementation `SummaryFormatter`.
- Adding new shipment providers, e.g. for specific countries, shouldn't require changes in the "client" == `Program.cs`.
- Adding new shipment providers shouldn't require changes in existing shipment providers.
- Changing the implementation of the tax rates database shouldn't require any changes to shipment providers.

Remarks

- Both modules should have diagnostic printings (`Console.WriteLine`). See example in `output.txt`.