**Task 2 [REFACTORING]:**

***Decision***: Multi-layered architecture is selected.

***Reason***:

Responsibilities are separated, by this way bug fixing and testing become easier.

Layers:

* 1. Entities: Object models are kept.
  2. Core: Core implementations are done, such as API repository. This layer can be separated and used by all projects in company.
  3. DataAccess: Data is served.
  4. Business: Business rules and validations are executed.
  5. API: API requests and responses are handled.

***Decision***: Some design patterns are implemented.

***Reason***:

1- Repository pattern

* 1. Repository pattern is implemented for consuming the 3rd party APIs. Although they have different functionalities, they all have same methods to use their b- functions.
  2. By implementing one repository, we can apply this to all APIs. If a change is required on this implementation, it will be applied to all instances.
  3. Incase on of the APIs has different methods than others, it can be implemented in separately on DataAccess layer.
  4. Although Product and Product Types are handled in ProductData api, I separated two APIs in DataAccess layer. Because they can be separated as microservices in the future. By this implementation, it is enough to change API base URLs.
  5. Entity framework based repository is also added.

2- Builder pattern

* 1. There are some steps (product type and price related information) to build insurance model.
  2. In the future, there may be new logics to build insurance model.
  3. I wanted to keep this steps separated between each other’s. By this way, code becomes easier to read and debug.

3- Factory pattern

* 1. There are different conditions (Price based, product type based) to determine insurance value.
  2. Instead of using if-else blocks, I create related insurance rule depending on the conditions.
  3. By this way, adding or updating new rule becomes easier and safer. In addition, it becomes more readable, maintainable and reusable.
  4. While insurance rules are using for individual product calculations, insurance order rules are used for order-based calculations.

***Decision***: API models are changed.

***Reason***:

* 1. I don't want to send/receive DTO models directly.
  2. I created separate response and request models for API methods.
  3. By this way, I hide my DTO models from user.
  4. In addition, I only process required fields and increase the response time.

**Task 3 [FEATURE 1]:**

***Assumptions***:

* 1. List of OrderProduct objects(Product id and quantity) will be provided in order as input.
  2. TotalInsuranceAmount and list of OrderProductDetail objects are send as response.
  3. New endpoint is required for this request.

***Actions***:

* 1. Insurance order model and insurance order manager are created to calculated insurance amount of order.
  2. In API, api/insurance/order endpoint is created.

**Task 4 [FEATURE 2]:**

***Assumptions***:

1. This request should be added to insurance order calculation.

***Actions***:

1. Insurance order rules are added to business layer.
2. Insurance order rule factory is created.
3. Factory is used in insurance order manager.

**Task 5 [FEATURE 3]:**

***Assumptions***:

1. New endpoint is expected to add.
2. Uploaded surcharge rate is expected to be added to insurance calculation.
3. Given surcharge rate is accepted as a percentage value. For example, if use give rate as 10, it is calculated as 10%.

***Actions***:

1. GetById, GetByProductTypeId, Post, Put, Delete methods are added to endpoint.
2. Surcharge rate is applied to insurance calculation.
3. This endpoint is also reachable at swagger.
4. DB connection string is configurable at appsettings.json file.

