**1. Project Name: MappingModels**

The project appears to implement a data mapping system, where different source models (e.g., DIRS21) and target models (e.g., Google) are dynamically mapped using a clean and structured approach.

**2. Layers in the Architecture:**

1. **Business Logic Layer:**

**Folder: BusinessLogic**

Purpose: Contains core logic for mapping and processing, ensuring separation of concerns between data handling and model representation.

**Key Files:**

**AddressExtensions**.**cs**: Likely contains extension methods for handling address-related logic or transformations.

**JsonProcessor.cs**: Handles JSON serialization/deserialization or transformation logic. And it comes with great function that it Maps the Json Properties with Source Model Class and find out which properties didn’t get much from Model, and Vice versa in the case of Json which Model Properties are not available inside the Json feed.

**ModelMapper.cs**: A critical file likely responsible for implementing mapping logic between source and target models. This file will handle all the Model Mappings, Complex properties implementation. Which Property of one Model should Match to which Property of Another Model. Which Source Model should Match to which Target Model. Since this Class is Partial. And it can have multiple version of Mappings for different Business Models. Infact, if we have to Map Lots of Models then folders can be added and it can have different implementation for different models and their properties.

1. **Model Layer:**

**Folder: Models**

Subfolders

* **Source:** Contains source models (e.g., DIRS21) that represent the input data structure.
* **Target:** Contains target models (e.g., Google) that represent the output data structure.

More Models Folders can be added based on the requirements

**Purpose:** Defines data models for both source and target, ensuring clear separation of concerns between the data structures of the external systems.

**Key Highlights:**

**DIRS21 Subfolder:** Includes model **DIRS21Model** and it’s dependant Models such as Address.cs, Guest.cs, and Tariff.cs, etc..

**Google Subfolder:** Includes model **GoogleModel** and it’s dependant Models such as Address.cs, Room.cs, and Traveler.cs, etc

***Note: Some Dependant Model may have the same Model Classes Name Such as Address in source and Target Models folder. But their Implementation can be different. ForExample Source Model Address have different Implementation and Target Model Address has a different Implementation.***

The parallel organization of models indicates a clear intent to maintain a 1-to-1 mapping capability.

1. **Mapping & Handler Layer:**

File: **MapHandler.cs**

Purpose: Acts as a central controller or orchestrator for the mapping process. It uses ModelMapper.cs and interact with both Source and Target models, and has Logic implemented Which Model should be converted to which Model and which Model Should be Mapped with which Model based on the Starting Point call. It’s Pretty Flexible More Model Routes can be added inside this by using Swtich cases.

File: **CustomExtensionsMethods.cs**

Purpose: In this class we can add Method which adheres to best practices like Single Responsibility Principle (SRP) and DRY (Don’t Repeat Yourself), Whenver needed. It keeps the code modular, readable, and easier to maintain.

1. **Application Entry Point:**

File: **Program.cs**

Purpose: Contains the entry point of the application, initializing configurations, setting up dependencies, and triggering the mapping process. You also have to select your File which you want to process.

**3. Modular and Extensible Design:**

**Folder Separation**: The folder structure segregates business logic (BusinessLogic) and data models (Models) effectively.

**Scalability:** Adding new source or target systems can be done easily by creating respective folders and models.

**Reusability:** Shared logic (e.g., extensions and processors) can be reused across different mapping configurations.

**4. Clean Architecture Principles Followed:**

1. **Separation of Concerns:** Business logic is separate from models and mapping logic.
2. **Modularity:** Individual components (e.g., source and target models) can evolve independently.
3. **Testability:** Core logic and model mappings are isolated, making them easier to test.

How Can this system be Extented for More Models.

Simply we will have to go to Models And whatever we want to add either Source or Target Model we can add them inside into respective folder. Once Models are added then You have to go to ModelMapper .cs class then added configuration there as well.

That which Models needs to map with which. And if both model have complex properties then those also needs to be mentioned and explicitly defined. Just like the sample in the file There are comments are instruction mentioned Just in case of reference.   
Note. For one to One Mapping You don’t have implement the Behavior it will automatically Map.   
For Complex Properties you have to define.

The sample Input which gave me it was quite complex in the DIRS21 data there are Guests which have List of addresses, but in the case of Google Travelers only have 1 Address means it was not collection so, in the one class it’s collection of address and in the 2nd class it’s Just one. So that Mapping gets handled by Model and it’s conversion.

Means Our Program assigns DIRS21’s first address of Guest to Google’s Traveler’s address. As Google Model does not accept collection of addresses.   
So that is the Configuration was Implemented inside the ModelMapper.cs class.   
  
Note: “ModelMapper.cs” is a Partial class. It’s as many as files can be added as Partial class just to keep things clean. For each Model Mapping.

As there are Many Automatically Mapping Libraries are available this program is using Mapperly. All the other Automappers they all have their implementation styles and In my experience some gets complicated when complicated properties you have to define. So I had to pick which implementation for complex mapping was easy and as well as support some automatic Mapping for similar data fields.

This Program also have a Logic Implemented that When it reads JSON it Maps the Json file Properties and sub properties with Source Model and it can give us list of Matched and UnMatched Properties, from Both JSON file and Source Model as well. For example If JSON has an extra property which is not implemented inside the Model, it will notify user, And if Model has any property which is not available in the JSON file it will also notify that. So both properties will be notified on the User’s screen. This can further can go to some File for further reference. So that User doesn’t have to manually Map and find out the missing ones.   
 In the ConvertStrict Method of JasonProcessor Class, There are 2 Variables Named called. unmatchedInJson, unmatchedInModel. We can write their values into some file for cross examination. But the Despite the fact you see Unmatching data will be converted.

**Conclusion:** This Application has potential to Grow and it can work according close to the expections well, where it’s implementation, maintenance is easy.

I have Also implemented one more solution style which is through Reflection no use of any 3rd Partly Library.   
But again that also comes with custom Implementations where Properties are complex and there Naming convention is different just like in this Case. But further more, this can be enhanced through different Mechanism. If you want to see that Solution I can send you that as well.

So, that’s pretty Much it. Thank you for your time on reading! If something is not clear or not proper please feel free to drop me a message.

The 3rd Party Library which was used in the program was Mapperly, More information about them can be found on this link: <https://mapperly.riok.app/docs/intro/>

And their documentation is also available for reference.