Software design COMP.SE.110 Group Project Design Document

Java the Hutt

1 Contents

2 High level description

2.1 Project Overview

The project is divided into three main modules: UI, Service, and Data. The architecture follows a modular design, which promotes clear separation of concerns and ensures scalability and maintainability.

2.1.1 Third-party components and libraries used

The project uses two APIs: International Monetary Fund's DataMapper API, v1 for retrieving GDP data by year for countries and The World Bank's Indicators API, v2 for retrieving the percentage of employment in the agricultural sector for countries.

2.2 Modules

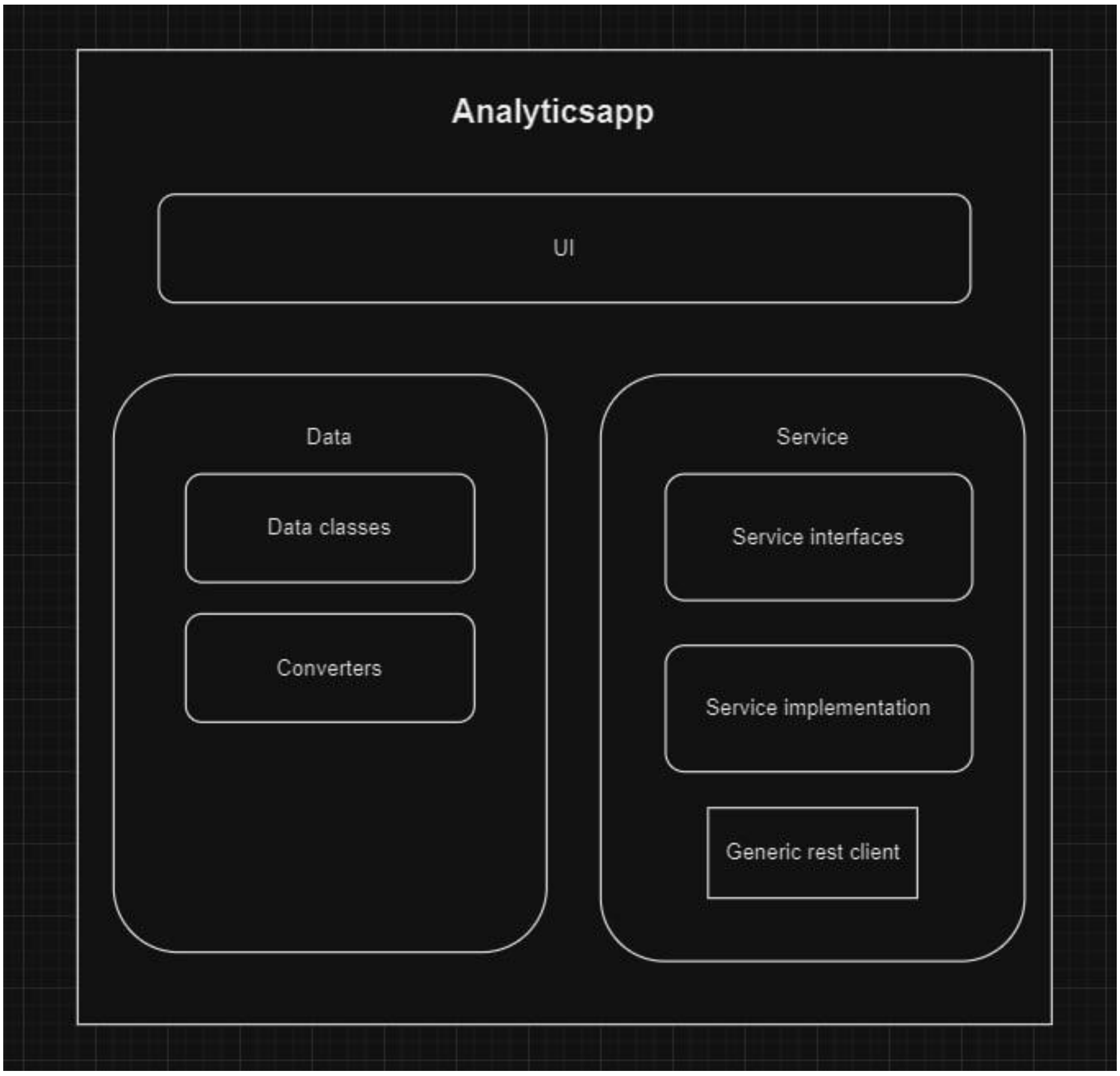
UI Module: The user interface layer is responsible for managing user interactions. It takes user input to initiate service calls and then displays the processed data in a chart. The chart uses the LineChartModule to show the data with two differently scaled y-axes, representing the data efficiently within a single visualization. If data is missing or the user provides invalid input, error messages are displayed under the chart.

Service Module: The service layer acts as an intermediary between the UI and the data handling logic. Upon receiving a request from the UI, it calls the appropriate API and forwards the response to the Data module for processing. While the forwarding of the API response to the Data module in the actual code is made from the UI module, it is basically equivalent to the control flow described in this document. Uses the factory pattern for creating service objects AgricultureService and GDPService, both implementing the ApiService interface. The `Service` class is responsible for making the necessary HTTP requests.

Data Module: This module handles the conversion and management of data. It processes API responses and converts them into appropriate data objects, which are formed from the data returned by the service module. This module has base classes for `GDP` and Agricultural Employement by Country (AgrEmplByCountry), and converters for converting JSONobjects to these two class objects.

2.3 Architecture

Overall architecture of the application.



Control Flow:

1. The UI calls the desired service.
2. The Service makes an API call and passes the result to the Data module.
3. The Data module processes the API data and creates data objects.
4. The processed data objects are returned to the UI through the Service.

This architecture enables parallel development, allowing multiple team members to work concurrently on the UI, Service, and Data modules. The design adheres to the principles of object-oriented programming (OOP) and leverages Java design patterns such as Converter pattern which in the Gang of four design patterns is known as ` Adapter pattern`. As the project progressses, code components will have more dependency injection, abstractions and inheritance used.

The core principle behind the design (although naming and some conventions modified) Model-View-Controller (MVC), also MVP pattern was used to design the chose approach pattern to ensure a clean separation between the data model, business logic, and user interface.

For the class diagram see Annex A.

2.4 Testing

Testing for the different modules is implemented with JUnit Jupiter v5.11.3. The testing is done for the following categories in their corresponding test files:

- Converters
  - AgrEmplByCountryConverterTest.java
  - GDPConverterTest.java
- Service
  - ServiceTest.java
  - AgricultureServiceTest.java
  - GDPServiceTest.java

The two converters convert the corresponding JSON to the correct data transfer objects so that they're easier to handle programmatically. Testing for the converters is done with mock data JSON so that it does not rely on the API fetching functionality of Service, and the test cases remain separated.

Service tests check if the API calls return data which is of the right format and expected values. Actual API calls are made with example year and country code parameters. Cases are also checked where the code should return an error, such as when a non-existent country code is used. I/O is also checked such that the country list is able to be read from file and has the correct content.

The tests can be run in various way, here are two examples:

1. Using Visual Studio Code and its 'Test Runner for Java' extension. Run the tests from the 'Testing' side panel on the left.
1. Go to the project in a command window and type 'mvn test'.

As of December 2024, all tests pass.

2.5 API usage

The project uses two external APIs to retrieve data:

2.5.1 International Monetary Fund: DataMapper API, v1

The IMF DataMapper API provides various economic data for countries or regions. In this project, it is used to retrieve GDP data by year for different countries.

Documentation: https://www.imf.org/external/datamapper/api/help

Example API call:

https://www.imf.org/external/datamapper/api/v1/indicator/NGDPD/country/fin

Example return format (years 2000-2003, values in billions of USD):

```
{
 2000: 126.075,
 2001: 129.534,
 2002: 140.305,
 2003: 171.609,
}
```

2.5.2 The World Bank: Indicators API, v2

The World Bank Indicators API v2 offers over 16 000 indicators from over 45 databases with data from up to 50 years ago. The themes are mostly in economy-relates matters, including international debt statistics, world development indicators, and subnational poverty.

In this project the API is used to gain access to the percentage of employment being in the agricultural sector in a country.

Documentation:

https://datahelpdesk.worldbank.org/knowledgebase/topics/125589-developer-information

Example API call:

https://api.worldbank.org/v2/country/fin/indicator/SL.AGR.EMPL.ZS?date=2000:2003&format=json

Example return format (relevant fields are “date” and “value”):

```
[
 {
  indicator: {id: SL.AGR.EMPL.ZS, value: <id explanation>},
  country: {id: <country iso2code> , value: <country name>},
  countryiso3code: <country iso3 code>,
  date: <year (string)>,
  value: <agriculture % of total employment (float)> || null,
  unit: <empty string>,
  obs_status: <empty string>,
  decimal: 0
 }
]
```

2.5.3 Countries

Country data is sourced from https://github.com/stefangabos/world_countries and formatted with custom code. Updates were made to some country names to maintain consistency.

3 Self-evaluation

Our initial design for the application architecture was to implement three modules: UI, Service and Data, which are also described earlier in this document. The chosen architecture has supported implementation mainly because it has made simultaneous developing problem-free. This is due to the work rarely overlapping between modules. Also, the design ensures that each module has a single responsibility.

We were able to stick to the original design very well. We did not need to make any notable changes to the design, although the "inner" designs of some of the modules have been modified slightly along the way. This was done to follow some design pattern, for example. Implementation of the base features has been quite clear during the whole process: what needs to be done, where, when, etc. To address how the design corresponds to quality, we have planned the application to be relatively easily tested. Also, the design follows separation of concerns.

4 Use of AI

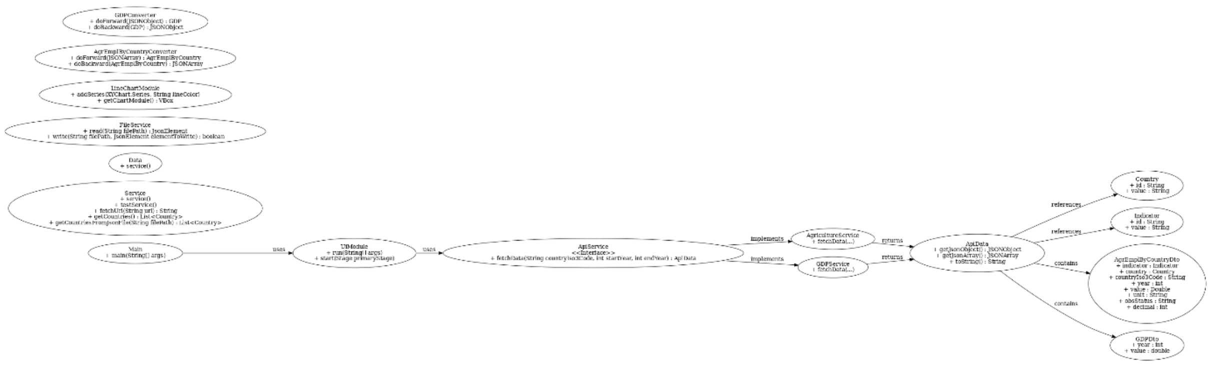
Our use of AI has primarily focused on GitHub Copilot to assist and speed up our programming process. GitHub Copilot was effective in providing correct solutions for simple problems and errors. However, for more complex structures and advanced tasks, it struggled to offer accurate or usable answers.

When we were throwing ideas around in the beginning, we used ChatGPT's ideas as a source for the idea we ended up with. Additionally, ChatGPT was used to review and improve the English grammar in this documentation.

The class diagram is completely drawn by ChatGPT with the project code files as prompt. It seemed to correctly recognize the relations between classes and interfaces but it tends to draw the picture in a very non-readable way, for example the picture is very wide.

5 Annex

A.


GDPConverter
+ doForward(JSONObject) : GDP
+ doBackward(GDP) : JSONObject
AgrEmplByCountryConverter
+ doForward(JSONArray) : AgrEmplByCountry
+ doBackward(AgrEmplByCountry) : JSONArray
LineChartModule
+ addSeries(XYChart.Series, String lineColor)
+ getChartModule() : VBox
FileService
+ read(String filePath) : JsonElement
+ write(String filePath, JsonElement elementToWrite) : boolean
Data
+ service()
Service
+ service()
+ testService()
+ fetchUrl(String url) : String
+ getCountries() : List<Country>
+ getCountriesFromJsonFile(String filePath) : List<Country>
Main
+ main(String[] args)
uses
UIModule
+ run(String[] args)
+ start(Stage primaryStage)
uses
ApiService
<<Interface>>
+ fetchData(String countryIso3Code, int startYear, int endYear) : ApiData
implements
AgricultureService
+ fetchData(...)
implements
GDPService
+ fetchData(...)
returns
returns
ApiData
+ getJsonObject() : JSONObject
+ getJsonArray() : JSONArray
+ toString() : String
references
Country
+ id : String
+ value : String
references
Indicator
+ id : String
+ value : String
contains
AgrEmplByCountryDto
+ indicator : Indicator
+ country : Country
+ countryIso3Code : String
+ year : int
+ value : Double
+ unit : String
+ obsStatus : String
+ decimal : int
contains
GDPDto
+ year : int
+ value : double