Query data and construc ontology by handling OOP concepts

Nguyen Viet Hung 20205157 Dang Trong Luat 20205187
Vu Chi Thanh 20200597 Nguyen Hoang Huy 20205183

July 11, 2022

Abstract
Report of the OOP project of group 6.

1 Job assignment

Name	Student id	Task	Percentage of contribution
Dang Trong Luat	20205187	Integrate data into GUI Adding dynamic data manipulation	25%
Vu Chi Thanh	20200597	Query data Transfer data into object	33%
Nguyen Viet Hung	20205157	Design and implement GUI	21%
Nguyen Hoang Huy	20205183	Create SPARQL query design	21%

2 Class dependency diagram

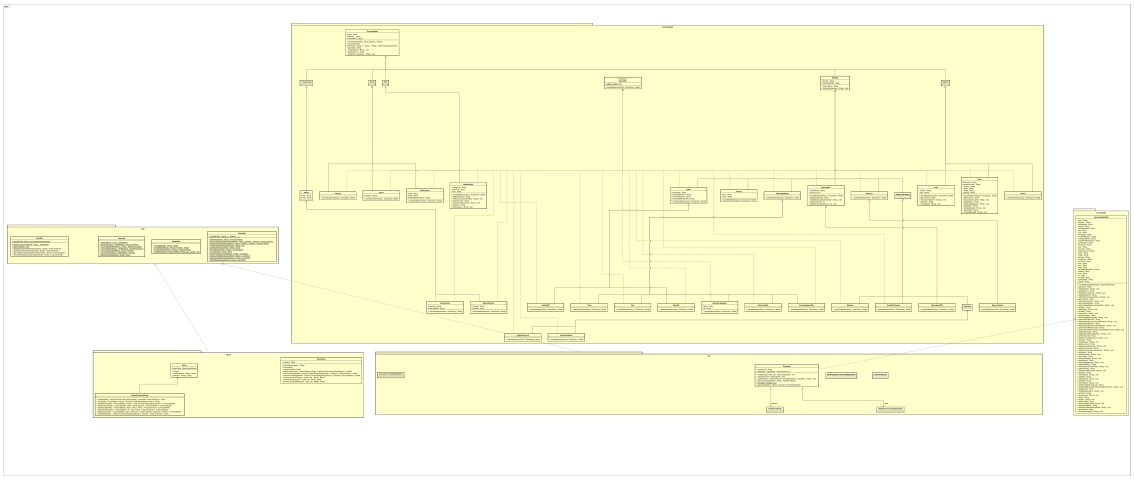


Figure 1: VN Tourism class dependency diagram

For better resolution you can visit this link: class dependency diagram to download svg file.

3 Explanation

3.1 Source code explanation

Source code has 2 main concepts: data display through GUI and store data under turtle format.

3.1.1 Data display through GUI

This is main page of our GUI for this project. By running Main.java in package src.main.java, the program will auto generate query text for fetching appropriate data in dbpedia.org and display a GUI window for user.

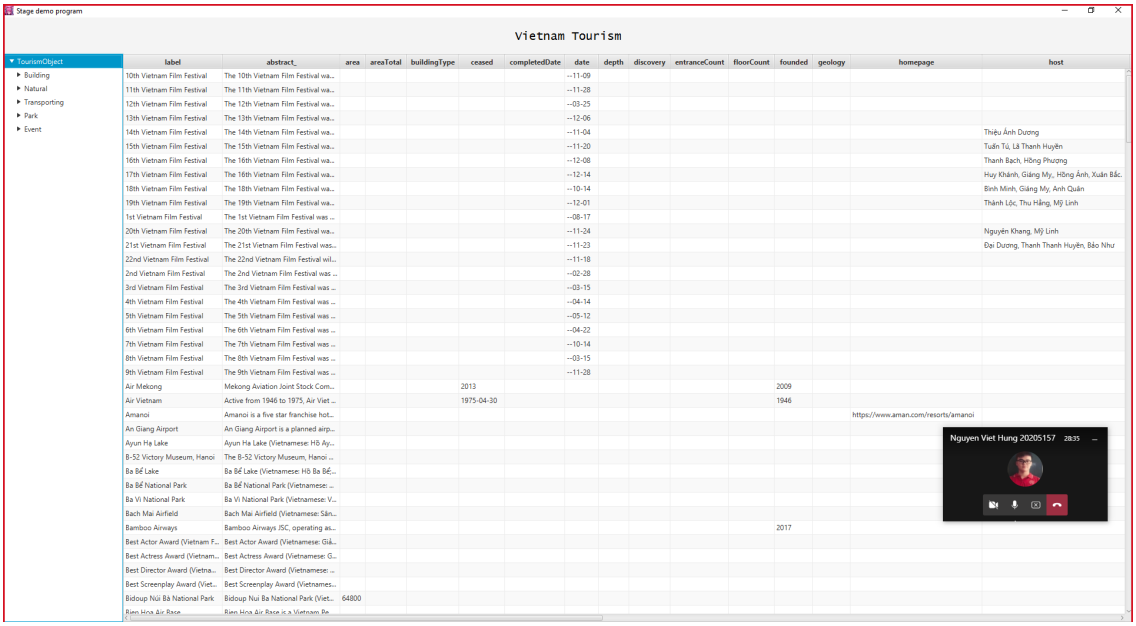
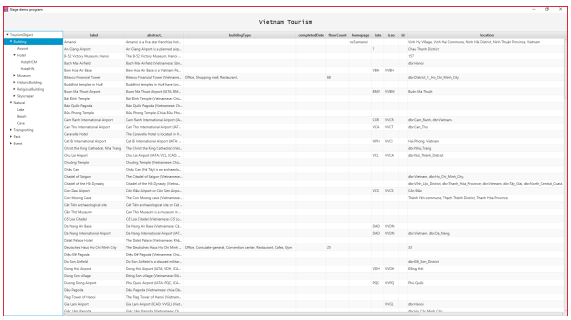
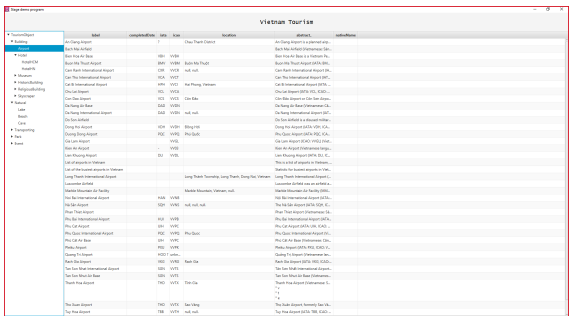


Figure 2: Homepage for displaying all data

By using navigation bar data can be seen in each concept.



(a) All sub categories in VN TOURISM



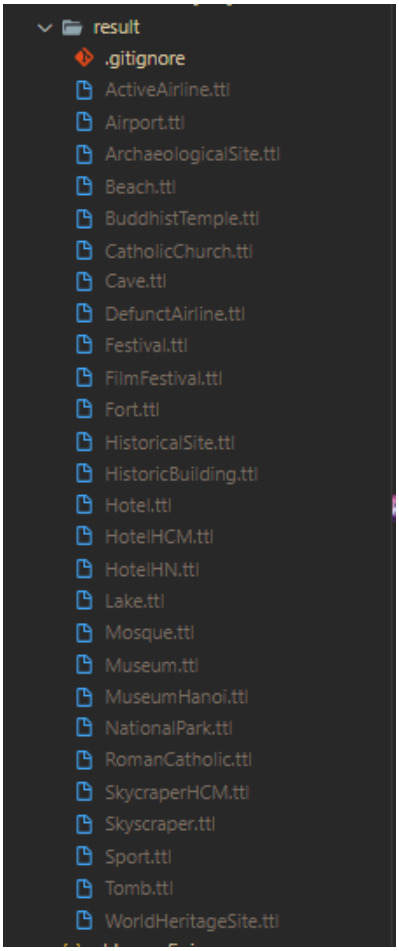
(b) Data in a sub category

3.1.2 Store data

The program will also store data, which can be accessed at src.main.resources.result, under turtle format.

3.1.3 Approach

The classes in **tourismobject package** have inherent properties. Root class is **TourismObject**, its subclasses are **Building**, **Natural**, etc., and each of these class will have their subclasses.
We will create and execute query for each classes that implement **Queryable interface** in **tourismobject package**, then store the result in TURTLE format with the corresponding class's name.
GUI will analyze request from user, read corresponding .ttl file(s), save it as JSON file in cache folder so that if user request the same resource, the program can get data from JSON files without process .ttl files again



(a) All .ttl files which be queried

(b) Example turtle format data

Figure 4: Store data

3.1.4 In-depth package analyze

- Package **sparql**
  This package contains **Query**, **CreateQuery**, **ExecQuery**
  - **Query**: have some generic method and fields for all type of query, such as Select, Construct, etc.
  - **CreateConstructQuery**: inherited from responsible for creating Construct Query object
  - **ExecQuery**: responsible for executing query with a given endpoint
- Package **utils** This package contains some utility classes: **API4GUI**, **ClassUtils**, **JsonUtils**, **ModelUtils**
  - **API4GUI**: supplies some functions for GUI
  - **ClassUtils**: supplies class-related functions
  - **JsonUtils**: supplies JSON-related functions
  - **ModelUtils**: supplies Jena Model-related functions
- Package **tourismobject** This package contains many tourism-related objects
- Package **gui** and package **tourismdata** This package contains GUI's implementation

4 OOP techniques explanation

4.1 Method overloading

We implement many method overloading with an aim to expand code easier in the feature. For example:
public ExecQuery()
public ExecQuery(String endpoint)

4.2 Class organization

We organize classes into package, such as **package sparql** only have classes that responsible for sparql query

4.3 Inheritance

We implement many inheritance relationship to improve re-usability and better class organization. For example, we create class **Query** containing some generic functions and fields, so that in the future if we want to create a **CreateSelectQuery**, we will just need to extends **Query** to avoid code duplication

4.4 Polymorphism

We implement both **Method polymorphism** and **Object polymorphism**.

1. **Method polymorphism**
   This technique is the same as method overloading

2. **Object polymorphism**
   We implement **Object polymorphism** in a way that it behave in multiple behaviors and have a multiple type. The **Queryable interface** exemplify this technique.

   - Up-casting an object that implement **Queryable interface** to **Queryable interface**. For example:
     Queryable object = (Queryable) ClassUtils.strToObj(className);
   - Dynamic binding, for example: both **Hotel** and **HotelHN** (which extends Hotel) implement method **Model queryModel(ExecQuery execQuery)**. So the method in **HotelHN** will override the method in **Hotel**, meaning when we create two object **Hotel o1**, **HotelHN o2** and upcasting it to **Queryable** (or any of their supperclasses), **o1.queryModel()** and **o1.queryModel()** behave differently.