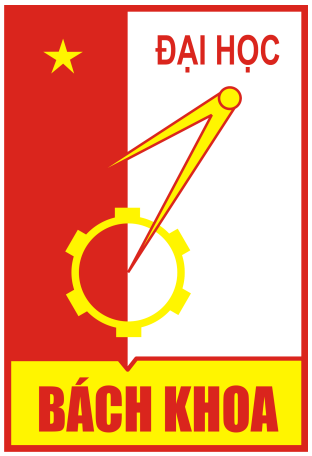
HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY
SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY



OBJECT-ORIENTED PROGRAMMING - IT3100E

CAPSTONE PROJECT: Vietnam History Wiki Application

Instructor: Ph.D. Trinh Tuan Dat
Class: 139411

Students: Nguyen Chi Long - 20210553
Ngo Xuan Bach - 20215181
Le Xuan Hieu - 20215201
Dinh Viet Quang - 20215235
Nguyen Viet Thang - 20215245
Nguyen Dinh Trung - 20215249

Ha Noi, July - 2023

Contents

1 Assignment of members

In this section, we will inform the work each group member has done by listing classes and methods where he mainly involved based on in this project.

1.1 Design and Programming

Members	Assignment	Percentage
Nguyen Chi Long	Model - Helper - Collection package	16.5%
Ngo Xuan Bach	Project structure - class diagram design, Controller - Services package	17.5%
Le Xuan Hieu	Detail class diagram design, Crawler - Model package	17.5%
Dinh Viet Quang	View package, Use case diagram design, Crawler package	17.5%
Nguyen Viet Thang	Crawler - Controller package, resources preparation	15.5%
Nguyen Dinh Trung	Use case diagram design, GUI designer, App tester	15.5%

Table 1: Member assignment

1.2 Report Writing, Slide, Readme file

- Report Writing:

  Nguyen Chi Long - 20210553 (20%)

  Ngo Xuan Bach - 20215181 (30%)

  Le Xuan Hieu - 20215201 (30%)

  Dinh Viet Quang - 20215235 (20%)

- Slide and video editor:

  Nguyen Dinh Trung - 20215249 (70%)

  Dinh Viet Quang - 20215235 (30%)

- Readme file:

  Dinh Viet Quang - 20215235 (100%)

2 Project description

2.1 Project overview

There are many websites that provide information on Vietnamese history (nguoikesu, Wikipedia, vansu, ...). Main objective is to find these sites and automatically collect data on Vietnamese history and link these data. Collected data needs to be stored as JSON or CSV format. Then build an application providing search and display the information features to the user.

Entities need to collect:

- Historical eras of Vietnam (Prehistory, Hong Bang, An Duong Vuong, Northern Domination I, ...)
- Vietnamese historical figures
- Historical sites of Vietnam
- Vietnamese cultural festivals
- Vietnam Historical events

Each entity needs identifiers, attributes, and importantly, entities that need to be associated with each other. Examples are described below:

1. Lễ hội đền Hùng
    - Địa điểm: Tổ chức ở TP Việt Trì, tỉnh Phú Thọ
    - Ngày tổ chức: 10/3 âm lịch
    - Nhân vật lịch sử liên quan: Tưởng nhớ Vua Hùng
    - Sự kiện liên quan : vua Hùng dựng nước
    - Di tích liên quan: Đền Hùng
    - ...
2. Nhân vật lịch sử Vua Hùng
    - Cha: Lạc Long Quân
    - Lên ngôi: năm 2524 trước công nguyên
    - Năm sinh: không rõ
    - Năm mất: không rõ
    - ....

2.2 Project requirement

Data collection should be automatic: The system should support automatic data collection methods to streamline the process and ensure a continuous flow of data. The automation should also be efficient and saving time.

Consistent Attribute Naming: All stakeholders must agree on a standardized naming convention for attributes associated with each entity type. This is important for data integration and analysis across different sources and systems.

Accuracy Guarantee: The data collection process should prioritize accuracy to ensure reliable insights and informed decision-making. Robust data validation and quality assurance mechanisms should be implemented to identify and rectify anomalies and inconsistencies.

Data Consolidation from Multiple Sources: The system should handle data consolidation from various sources. Efficient data mapping and transformation rules should be established to integrate the data seamlessly. By doing this, data integration techniques should be implemented to achieve a holistic view of the user's operations.

2.3 Collected data overview

2.3.1 Figures

Each figure in our database is detailed with sixteen attributes:

- Id: A unique identifier for each historical figure.
- Main Name and Alternate Names: The primary name of the figure and any other names or aliases they are known by.
- Born Year and Died Year: The year of the figure's birth and death.
- Location: The primary geographical locations where the figure lived and worked.
- Role or Job: The main roles or occupations held by the figure.
- Brief Description: A concise summary of the figure's life, accomplishments, and significance.
- Parents and Descendants: Information on the figure's mother, father, spouse(s), and children.
- Related Historical Entities: Eras, events, relics, and festivals that the figure was directly involved with or significantly impacted.

```
{
  "otherNames": [
    "Ngọc Hoa"
  ],
  "bornYear": "Không rõ",
  "diedYear": "Không rõ",
  "eras": {
    "Dựng nước (2000 - 258 trCN)": 2
  },
  "location": "Không rõ",
  "role": "Công chúa",
  "spouses": {},
  "mother": {},
  "father": {
    "Nguyễn Phúc Nguyên": 747
  },
  "children": {},
  "relatedEvents": {},
  "relatedRelics": {
    "Đền Hùng": 352,
    "Đền Tiên La": 565
  },
  "relatedFestivals": {
    "Lễ hội đình Đại Yên": 363
  },
  "id": 3,
  "name": "Bạch Hoa",
  "description": "Công chúa con gái Hùng Vương thứ XVIII, có sách chép là Ngọc Hoa, nổi tiếng sắc nước hương trời. Nhân vật gây nên cuộc \"tranh hôn kết oán\" trong thời Văn Lang."
},
```


Figure 1: An example of Figure entity in Json format

Data Acquisition:

We acquired our initial data, which included attributes such as main name, alternate names, birth year, death year, location, role, era, and brief description, from two primary sources:

- Vansu.vn: Provided data on 2391 figures.
- Thuvienlichsu.vn: Provided data on 809 figures.

After accounting for duplicated entries, we were left with a total of **2761 unique historical figures**.

For further detail, specifically pertaining to the figure's family (mother, father, spouse(s), and children), we utilized Google Search to obtain necessary data. These additional attributes were added to our original data set.

The information about related historical events, relics, and festivals was integrated during a linking process in our database. This allows for an interconnected web of information, enhancing the depth and context of each historical figure's profile.

2.3.2 Relics

Each relic in our database is defined by eight attributes:

- Id: A unique identifier for each relic.
- Relic Name: The name of the relic.
- Location: The geographical location where the relic is situated.
- Category: The classification of the relic.
- Approved Year: The year the relic was recognized as a national relic.
- Brief Description: A concise summary of the relic's significance and characteristics.
- Related Festivals: Festivals associated with the relic.
- Related Figures: Historical figures linked to the relic.


```
{
  "location": "Sam Mứn, Điện Biên",
  "category": "Lịch sử",
  "approvedYear": "22/01/2009",
  "relatedFestivals": {},
  "relatedFigures": {
    "Hoàng Công Chất": 612,
    "Lê Ý Tông": 654
  },
  "id": 8,
  "name": "Thành Sam Mứn (Tam Vạn)",
  "description": "Thành Sam Mứn hay thành Tam Vạn là một di tích lịch sử, ở bản Pom Lót, xã Sam Mứn, huyện Điện Biên, tỉnh Điện Biên. Theo tiếng dân tộc Thái thì Sam Mứn có nghĩa là Tam Vạn."
},
```


Figure 2: An example of Relic entity in Json format

Data Acquisition:

We obtained our initial data, including the relic name, location, category, approved year, brief description, and related figures, from Wikipedia, resulting in **a total of 1810 unique relics**.

Information about related festivals was integrated into our dataset during the linking process. This approach ensured that each relic was properly associated with its relevant festivals, enriching the comprehensiveness of our data.

2.3.3 Events

Each event in our database includes nine attributes:

- Id: A unique identifier for each event.
- Event Name: The name of the historical event.
- Start Year: The year the event began.
- End Year: The year the event ended.
- Location: The geographical location(s) where the event took place.
- Result: The outcome or consequence of the event.
- Brief Description: A succinct summary of the event's characteristics and significance.
- Eras: The historical era(s) to which the event belongs.
- Related Figures: Historical figures associated with the event.


```
{
  "startYear": "1406",
  "endYear": "1407",
  "location": "Thăng Long (1010 - 1831)",
  "result": "Không rõ",
  "eras": {
    "Nhà Hồ (1400-1407)": 15
  },
  "relatedFigures": {
    "Hồ Quý Ly": 2395
  },
  "id": 43,
  "name": "Cuộc xâm lược của nhà Minh 1406-1407",
  "description": "Tháng 11-1406, lấy cớ nhà Hồ cướp ngôi nhà Trần, nhà Minh đã huy động một lực lượng lớn gồm 20 vạn quân cùng với hàng chục vạn dân phu, do tướng Trương Phụ cầm đầu tiến vào nước ta."
},
```


Figure 3: An example of Event entity in Json format

Data Acquisition:

We gathered our initial data, encompassing the event name, start year, end year, location, result, brief description, and related figures, from three primary sources:

- Thuvienlichsu.vn: Provided data for 371 events.
- wikipedia: Provided data for 175 events.
- Nguoikesu.com: Provided data for 71 events.

After accounting for duplicated entries, we were left with **a total of 541 unique historical events**.

During the linking process, we compared the timeline of each event with the timeline of each era. This allowed us to assign each event to the appropriate historical era(s), thereby enriching our dataset with these additional details.

2.3.4 Eras

Each era in our database is characterized by nine attributes:

- Id: A unique identifier for each era.
- Era Name: The name of the historical era.
- Start Year: The year the era began.
- End Year: The year the era ended.
- Brief Description: A succinct summary of the era's characteristics and significance.
- Nation Names: A list of names used to refer to the nation during the era.
- Capital: The primary city or seat of power during the era.
- Kings: The rulers or monarchs that reigned during the era.
- Related Events: Major historical events that occurred during the era.


```
{
  "startYear": "40",
  "endYear": "43",
  "kings": {},
  "relatedEvents": {
    "Kháng chiến của nhân dân ta chống quân xâm lược Hán": 19,
    "Khởi nghĩa Hai Bà Trưng bùng nổ": 20,
    "Mã Viện đánh vào Giao Chỉ": 115,
    "Cuộc khởi nghĩa Hai Bà Trưng bùng nổ": 282,
    "Kinh Dương Vương lên ngôi": 296,
    "Văn hóa Sa Huỳnh": 386,
    "Văn hóa Đông Sơn": 388,
    "Văn hóa Ốc Eo": 389,
    "Chiến tranh Lĩnh Nam - Đông Hán": 397,
    "Đông Hán thôn tính Lĩnh Nam": 398
  },
  "nationNames": [
    "Lĩnh Nam"
  ],
  "capital": "Mê Linh (Vĩnh Phúc)",
  "id": 5,
  "name": "Trưng Nữ Vương (40-43)",
  "description": "Trưng Trắc đánh đuổi quân Hán, lên ngôi vua đóng đô tại Mê Linh. "
},
```


Figure 4: An example of Era entity in Json format

Data Acquisition:

We sourced our initial data, which consisted of the era name, start year, end year, brief description, and kings, from Vansu.vn, resulting in **a total of 24 eras**.

For the list of names used to refer to the nation during each era, we relied on information gleaned from Wikipedia. Capital city information was gathered from a website named Quynhluu2.edu.vn. We then compared the timeline of nation names and capitals with the start and end years of each era, enriching our dataset with these additional details.

Information about related events was incorporated during the linking process in our database, creating a more interconnected and comprehensive overview of each historical era.

2.3.5 Festivals

Each festival in our database is defined by eight attributes:

- Id: A unique identifier for each festival.
- Festival Name: The name of the festival.

- Location: The geographical location where the festival is held.
- First Time: The time when the festival was first held.
- Starting Day: The day of the year when the festival is held annually.
- Brief Description: A concise summary of the festival's characteristics and significance.
- Related Relics: Relics associated with the festival.
- Related Figures: Historical figures associated with the festival.


```
{
  "location": "Phú Yên, Ô Loan",
  "firstTime": "Không rõ",
  "startingDay": "7/1",
  "relatedRelics": {
    "đầm Ô Loan": 1369
  },
  "relatedFigures": {},
  "id": 7,
  "name": "Lễ hội đầm Ô Loan",
  "description": "Lễ hội đầm Ô Loan được tổ chức vào mùng 7 tháng Giêng (Âm lịch) tại thôn Phú Tân, xã An Cư, huyện Tuy An. Lễ hội này có tư cách là lễ hội cầu ngư của người dân quanh đầm Ô Loan."
},
```

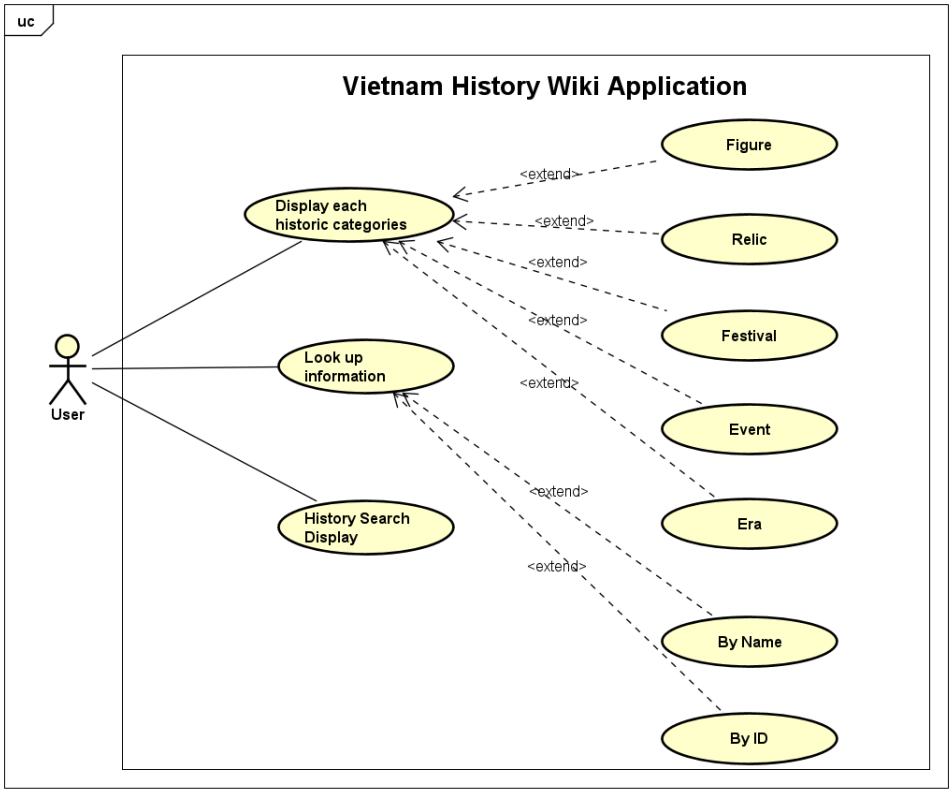

Figure 5: An example of Festival entity in Json format

Data Acquisition:
We gathered our festival data from a total of 18 different sources:

- wikipedia: Provided data for 52 festivals.
- Other sources, which offer data about festivals in specific provinces such as Vinpearl.com, Vinwonder.com, and various government websites: Provided data for 356 festivals.

After accounting for duplicated entries, we were left with **a total of 397 unique festivals**.

2.4 Use case diagram

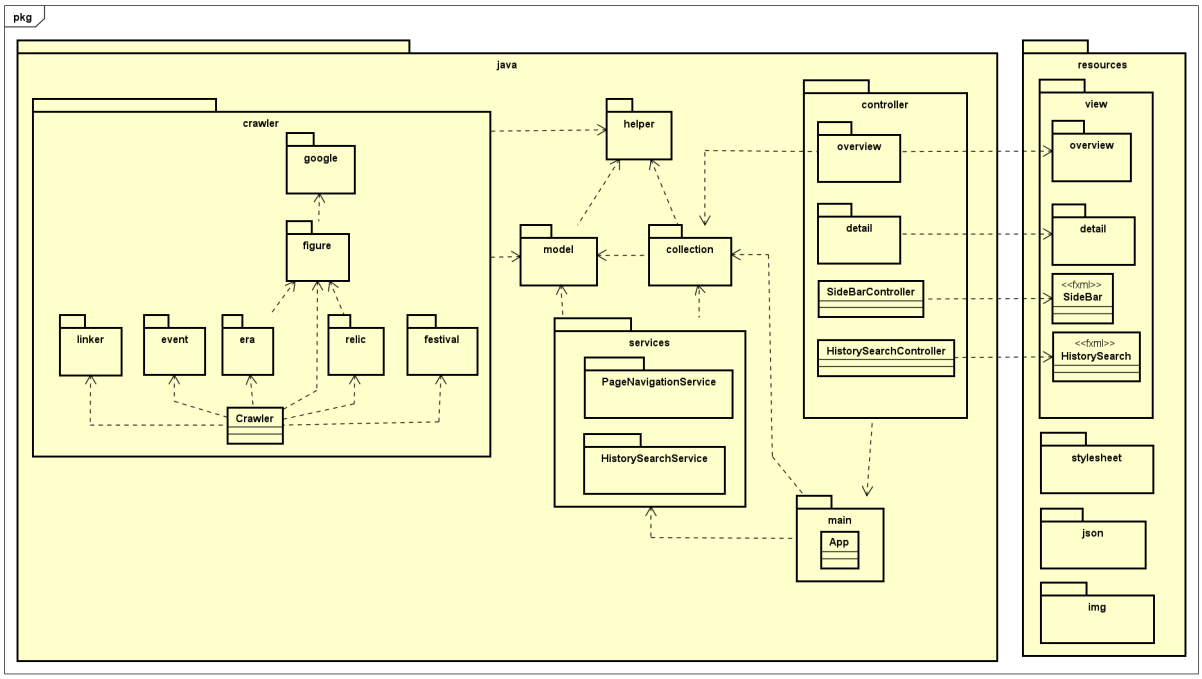


1. Display each historic categories
    - This feature in the history application allows users to view and explore different historical categories, such as figures, relics, events, eras, and cultural festivals. The display feature presents the information in a TableView, which providing a structured and organized representation of the historical entities within each category.
    - The application organizes the retrieved data into separate categories in a structured and tabular format.
    - For each category tab or section, the TableView presents the relevant information about the historical entities, such as their names, IDs, descriptions, dates, or any other pertinent attributes. It may provide sorting options to facilitate navigation and exploration. Additionally, the user can scroll through the TableView to view additional entities within the category. At the same time, user can also click on a specific entity within the TableView to access more detailed information or perform additional actions related to that entity.
2. Look up information
    - This feature in the history application allows users to search and retrieve specific information about historical entities based on their name or ID. This feature enhances the user's ability to find and access relevant information quickly and efficiently
    - The user enters the name or ID of the historical entity they want to look up and then the application validates the input and performs a search based on the provided name or ID.
    - If a matching entity is found, The application retrieves the relevant information for the entity, such as its name, description, related figures/events/eras, etc. The application then displays the retrieved information to the user. Otherwwise, The application notifies the user that no results were found for the given name or ID.
3. History Search Display
    - **"History Search Display"** is an additional feature of our application, enhances the user experience by providing easy access to previously searched historical information, allows users to efficiently navigate through their past interactions with the historical data, facilitating a seamless and personalized exploration of the application's content.
    - This feature in the history application enables users to retrieve and display the information they have recently searched for. It provides a convenient way for users to revisit and access previously viewed historical data without having to perform the search again. The feature presents the search history and displays the corresponding information in an organized manner.
    - The user interacts with the application's navigation or menu system to access the "History Search Display" feature.
    - The application retrieves the search history data in a tabular format, from the underlying storage or data source, which includes the user's past search queries, associated information and access time.
    - The user can scroll through the displayed information to explore and review the historical data, as well as access directly to the historical data.

3 Design

3.1 General class diagram

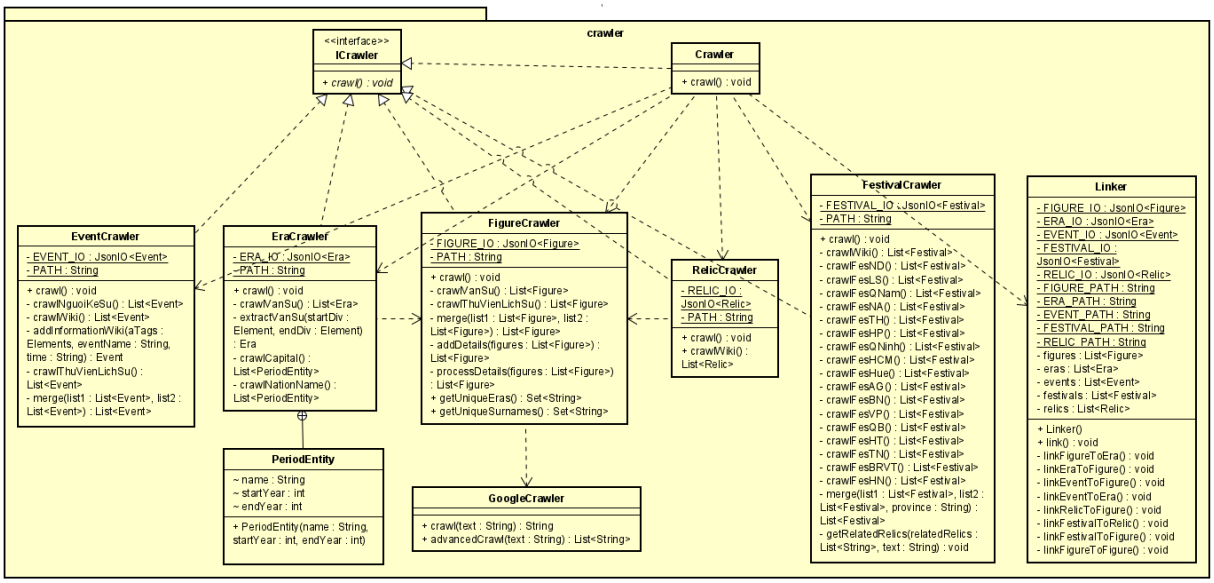
In this project, we have created the project structure so that it is easier for member assignment.



3.2 Package details

3.2.1 Crawler package

The crawler package in our history application plays an important role in gathering historical data from various sources. It ultilizes web scraping techniques and API interaction to retrieve data, which ensures a comprehensive collection of our historical information. Moreover, this package established meaningful connection between different data entities, enabling deeper navigation and exploration of historical figures, relics, eras, events and festivals.



Class and subpackage overview

1. **ICrawler** interface is fundamental to the crawler package, compelling all associated classes to implement the crawl() method.

2. **Crawler** class serves as the core within this package, offering a main method to execute the crawling tasks and subsequently store the resulting dataset. This class is dependent on the FigureCrawler, EraCrawler, EventCrawler, RelicCrawler, FestivalCrawler, each of which perform dedicated crawling tasks on their corresponding entities. Additionally, the Crawler class also leans on Linker class, which creates connections between various entities, thus facilitating the creation of an interconnected dataset.

3. **FigureCrawler:** FigureCrawler, housed within the crawler.figure package, is tasked with collecting data on historical figures from various sources, eradicating duplicates, and recording the unique figures into a JSON file.

   (a) Attributes:
   - FIGURE_IO: An instance of JsonIO<Figure> responsible for writing a list of Figure objects into a JSON file and loading them from the JSON file.
   - PATH: The path to the JSON file that stores figure data.

   (b) Methods:
   - crawlVanSu(): Collects data from Vansu.vn and returns a list of figures.
   - crawlThuVienLichSu(): Collects data from Thuvienlichsu.vn and returns a list of figures.
   - merge(List<Figure> list1, List<Figure> list2): Merges two lists of figures and removes duplicates, where list1 is the base list.
   - addDetails(List<Figure> figures): Utilizes the GoogleCrawler class to search for additional information about mothers, fathers, spouses, and children, and then adds the result to the provided list.
   - processDetails(List<Figure> figures): Removes unnecessary parts from the attributes 'mother', 'father', 'spouses', and 'children'.
   - crawl(): Calls the aforementioned methods and writes the result to a JSON file.
   - getUniqueEras(): Loads data from a JSON file and returns a set of all distinct eras to which figures belong.
   - getUniqueSurnames(): Loads data from a JSON file and returns a set of all distinct surnames that figures can possess.

4. **EraCrawler:** EraCrawler, found in the crawler.era package, specializes in harvesting data about historical eras and subsequently storing this information in a JSON file.

   (a) Attributes:
   - ERA_IO: This is an instance of JsonIO<Era>, tasked with writing a list of Era objects into a JSON file and loading the data from a JSON file back into a list of Era objects.
   - PATH: This attribute points to the location of the JSON file where era data is stored.

   (b) Methods:
   - crawlVanSu(): This method collects data from Vansu.vn and returns a list of eras.
   - extractVanSu(startDiv: Element, endDiv: Element): Utilized by crawlVanSu(), this method retrieves information about an era situated between two HTML div elements.
   - crawlCapital(): This method, invoked by extractVanSu(), returns a list of capitals.
   - crawlNationName(): Also called by extractVanSu(), this method provides a list of names referring to the nation during each era.
   - crawl(): This method executes the aforementioned methods and writes the resultant data to a JSON file.

   (c) Nested class: The PeriodEntity is a nested class that encapsulates the concept of a 'capital' and a 'nation name' during a particular historical period. Each instance of the PeriodEntity class represents either a capital or a nation name during a certain time period. It carries three main attributes - 'name', 'startYear', and 'endYear'.

5. **EventCrawler:** EventCrawler, a component of the crawler.event package, takes on the responsibility of gathering data on historical events from different sources, eliminating duplicates, and writing the list of events into a JSON file.

   (a) Attributes:

   - EVENT_IO: An instance of JsonIO<Event>, responsible for writing a list of Event objects to a JSON file and loading the data from a JSON file back into a list of events.
   - PATH: The location of the JSON file storing event data.

   (b) Methods:

   - crawlNguoiKeSu(): This method gathers data from NguoiKeSu.com, returning a list of events.
   - crawlWiki(): This method collects data from Wiki and returns a list of events.
   - addInformationWiki(aTags: Elements, eventName: String, time: String): Invoked by crawlWiki(), this method fetches additional information from the links within aTags, returning an event.
   - crawlThuVienLichSu(): This method amasses data from ThuVienLichSu.vn, returning a list of events.
   - merge(List<Event> list1, List<Event> list2): This method merges two lists of events, removing duplicates. Note that list1 is the base list.
   - crawl(): This method invokes the aforementioned methods and writes the resultant data to a JSON file.

6. **RelicCrawler:** RelicCrawler, a class in the crawler.relic package, shoulders the task of gathering data on historical relics and committing this information to a JSON file.

   (a) Attributes:

   - RELIC_IO: An instance of JsonIO<Relic>, tasked with writing a list of Relic objects to a JSON file and loading the data from a JSON file back into a list of relics.
   - PATH: The location of the JSON file storing relic data.

   (b) Methods:

   - crawlWiki(): This method crawls data from Wiki and returns a list of relics.
   - crawl(): This method calls crawlWiki() and writes the resultant data to a JSON file.

7. **FestivalCrawler:** FestivalCrawler, part of the crawler.festival package, is charged with the duty of collecting data about historical festivals from a variety of resources, weeding out duplicates, and preserving the unique festivals in a JSON file.

   (a) Attributes:

   - FESTIVAL_IO: An instance of JsonIO<Festival> that writes a list of Festival objects into a JSON file and loads the data from a JSON file back into a list of festivals.
   - PATH: The location where the JSON file storing festival data is kept.

   (b) Methods:

   - crawlWiki(): This method gathers data from Wiki and returns a list of festivals.
   - crawlFesND(), crawlFesLS(), etc.: These methods individually collect data about festivals specific to certain provinces and return lists of festivals.
   - getRelatedRelics(List<String> relatedRelics, String text): This method searches for related relics within a given text and appends them to the relatedRelics variable. This method is utilized by crawlWiki() and other crawl methods.
   - merge(List<Festival> list1, List<Festival> list2, String province): This method merges two lists of festivals based on the given province and removes duplicates. Note that list1 is the base list.
   - crawl(): This method calls the aforementioned methods and writes the resulting data to a JSON file.

8. **GoogleCrawler:** GoogleCrawler, a class within the crawler.google package, functions as a utility class, providing methods that facilitate data scraping from Google based on supplied keywords.

(a) Attributes: GoogleCrawler is a utility class that solely provides utility methods, and thus, it does not require any attributes.

(b) Methods:

- crawl(String text): returns single result based on a given query text
- advancedCrawl(String text): returns mutiple results based on a given query text

9. **Linker:** Acting as a crucial link, the Linker class serves to connect different entities based on ID. It executes this task once data about all five types of entities become available.

(a) Attributes:

- FIGURE_IO: An instance of JsonIO<Figure> responsible for writing a list of Figure objects into a JSON file and loading them from the JSON file.
- ERA_IO: This is an instance of JsonIO<Era>, tasked with writing a list of Era objects into a JSON file and loading the data from a JSON file back into a list of Era objects.
- EVENT_IO: An instance of JsonIO<Event>, responsible for writing a list of Event objects to a JSON file and loading the data from a JSON file back into a list of events.
- RELIC_IO: An instance of JsonIO<Relic>, tasked with writing a list of Relic objects to a JSON file and loading the data from a JSON file back into a list of relics.
- FESTIVAL_IO: An instance of JsonIO<Festival> that writes a list of Festival objects into a JSON file and loads the data from a JSON file back into a list of festivals.
- FIGURE_PATH: The path to the JSON file that stores figure data.
- ERA_PATH: The path to the JSON file that stores era data.
- EVENT_PATH: The path to the JSON file that stores event data.
- RELIC_PATH: The path to the JSON file that stores relic data.
- FESTIVAL_PATH: The path to the JSON file that stores festival data.
- figures: List of figures loaded from JSON file
- eras: List of eras loaded from JSON file
- events: List of events loaded from JSON file
- relics: List of relics loaded from JSON file
- festivals: List of festivals loaded from JSON file

(b) Methods:

- Linker(): This constructor method loads data into attributes such as figures, eras, events, festivals, and relics.
- linkFigureToEra(): This method links the 'eras' attribute of Figure to Era.
- linkEraToFigure(): This method links the 'kings' attribute of Era to Figure.
- linkEventToFigure(): This method links the 'relatedFigures' attribute of Event to Figure.
- linkEventToEra(): This method links the 'eras' attribute of Event to Era.
- linkRelicToFigure(): This method links the 'relatedFigures' attribute of Relic to Figure.
- linkFestivalToRelic(): This method links the 'relatedRelics' attribute of Festival to Relic.
- linkFestivalToFigure(): This method links the 'relatedFigures' attribute of Festival to Figure.
- linkFigureToFigure(): This method links the 'mother', 'father', 'spouses', and 'children' attributes of Figure to other Figure instances.
- link(): This method calls all aforementioned methods and writes the results to JSON files.

OOP techniques

1. Abstraction: The abstraction technique is leveraged in the design of this package. Each class provides a high-level functionality, hiding its internal working mechanism. For instance, methods like crawlVanSu(), crawlWiki() provide the service of fetching data without revealing how it's done.

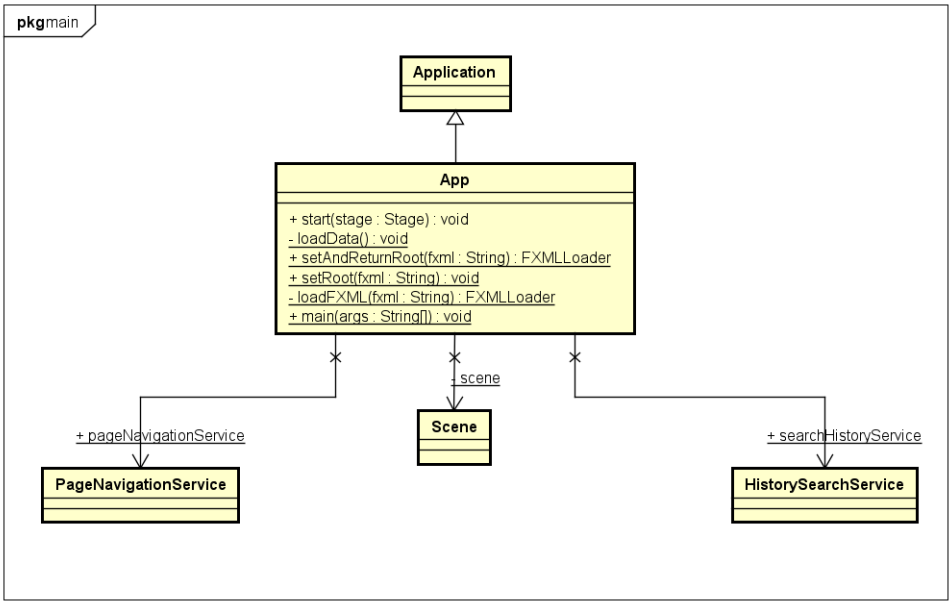
2. Interface: The ICrawler interface provides a blueprint for all crawler classes in the package, enforcing the implementation of the crawl() method and promoting a consistent structure across all crawler classes.

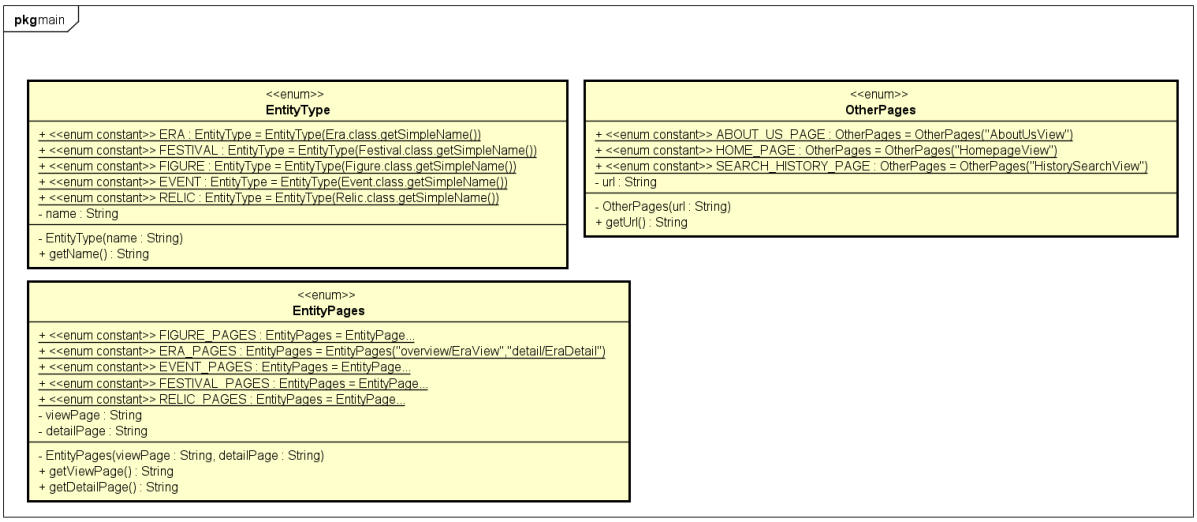
3. Nested Class: The PeriodEntity class, which is nested within the EraCrawler class, is a clear demonstration of the use of nested classes. PeriodEntity encapsulates the related attributes of 'capital' and 'nation name' along with their start and end years, forming an object-oriented structure for better data organization.

4. Exception Handling: Throughout the implementation of the crawler package, exceptions are gracefully handled to ensure the robustness of the system. Given the package's heavy reliance on external resources for data collection, it is crucial to manage potential issues such as connectivity failures, unavailable resources, or changes in the structure of these resources.

3.2.2 Main package

The main package serves as a central component in the application, containing key classes and enums that orchestrate the overall functionality and behavior of the system. It encompasses various aspects, including page navigation, entity representation, and auxiliary pages.





Class overview

1. **App** class in the main package is the main class of the application. It extends the Application class provided by JavaFX and contains the main entry point and lifecycle methods of the application.

   (a) Attributes:

   - pageNavigationService: An instance of the PageNavigationService class that handles page navigation and manages the navigation history.
   - searchHistoryService: An instance of the HistorySearchService class that handles search history functionality.
   - scene: The main scene of the application.

   (b) Methods:

   - start(Stage stage): The overridden start method from the Application class. It is called when the application is launched. This method sets up the main scene, loads initial data, configures the stage, and displays the UI.
   - loadData(): A private method that loads initial data from JSON files using the data collection classes (FigureData, EventData, FestivalData, EraData, RelicData). It also loads the search history data using the searchHistoryService
   - setAndReturnRoot(String fxml): A method that sets the root node of the scene by loading the specified FXML file using the FXMLLoader. It returns the FXMLLoader instance.
   - setRoot(String fxml): A method that sets the root node of the scene by loading the specified FXML file using the FXMLLoader.
   - loadFXML(String fxml): A private helper method that creates and returns an FXMLLoader instance for the specified FXML file.

2. **EntityPages** enum, located in the main package, provides a convenient way to represent and manage the different pages associated with entities in the application. Each enum value corresponds to an entity type and contains the URLs of the respective view page and detail page for that entity type.

3. **EntityType** enum, located in the main package, provides a representation of the different types of entities in the application. Each enum value corresponds to an entity class and contains the simple name of that class as a string.

4. **OtherPages** enum, located in the main package, represents additional pages or views in the application that are not specific to entity overviews or details. Each enum value corresponds to a specific page or view and holds the URL or identifier associated with that page.

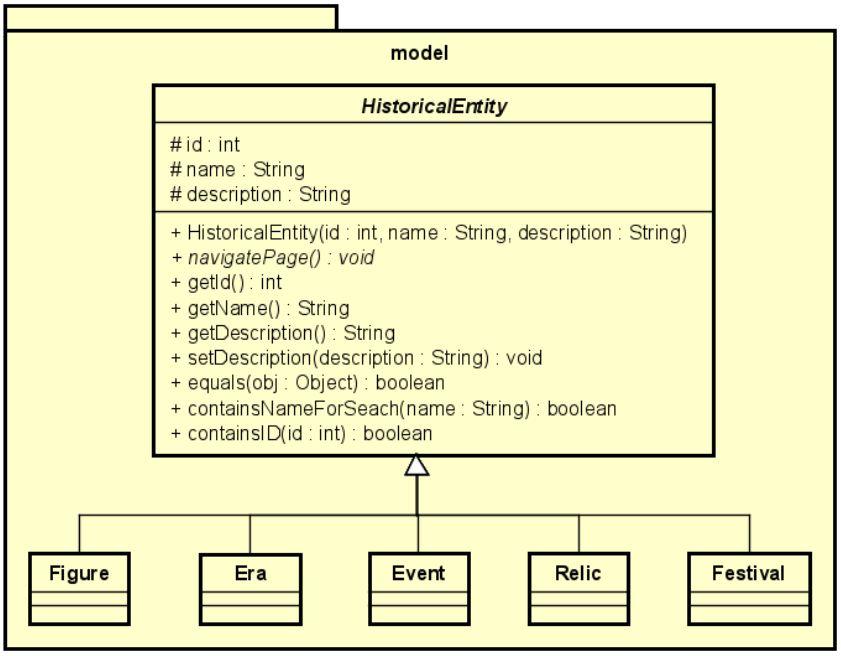
Significant OOP techniques

1. Enumerations (Enums): The package includes multiple enum classes such as EntityPages, EntityType, and OtherPages. Enums provide a way to define a fixed set of named values, allowing for better code organization and readability.

3.2.3 Model package

In our Java project, we have a package called model that serves as a foundation for modeling and representing various entities within the application. The model package is responsible for defining the essential data structures and behaviors of the core entities that our application deals with. It provides a cohesive and structured approach to representing and manipulating data objects, facilitating the implementation of business logic and application functionality.

The primary purpose of the model package is to define and encapsulate the data models or entity classes that represent the core concepts and entities within our application. These entities are Era, Figure, Event,Relic, Festival. They are all the derived classes of the abstract class HistoricalEntity.



Class overview

1. **HistoricalEntity**: The HistoricalEntity abstract class acts as a parent or base class for other classes within the model package, such as Figure, Relic, Event, Festival, and Era. Here's an overview of how the HistoricalEntity class serves as a foundation for these child classes:

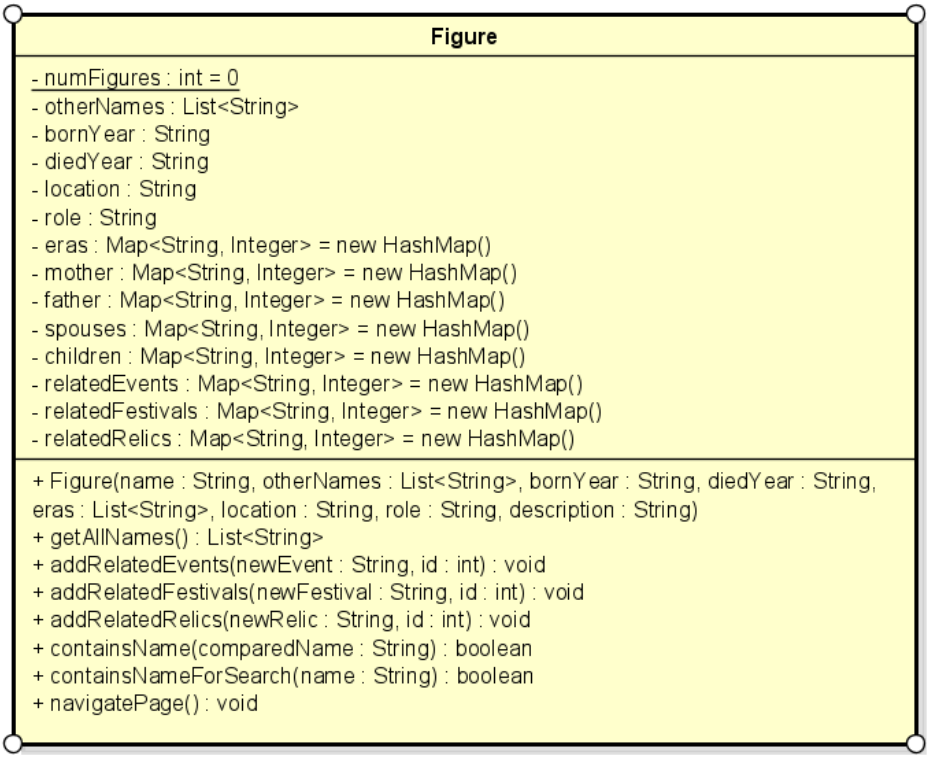
   (a) Attributes:

   - id: An integer representing the unique identifier of the historical entity.
   - name: A string representing the name of the historical entity.
   - description: : A string representing the description of the historical entity.

   (b) Methods:

   - Getter methods are provided to retrieve the values of the attributes. We have only included a setter method for the 'description' attribute to allow changes during the source merging process.
   - equals(Object obj) : Overrides the equals() method to compare the equality of historical entities based on their id and name properties.
   - navigatePage(): An abstract method that needs to be implemented by subclasses to handle navigation to a specific page related to the historical entity.
   - containsNameForSearch(String name): Checks if the provided name is contained within the instance's name (case-insensitive) for searching purposes.
   - containsID(int id): Checks if the provided id matches the instance's id for searching purposes.

2. **Figure**: This class, extending the HistoricalEntity abstract class, represents a historical figure within the model package.



(a) Attributes:

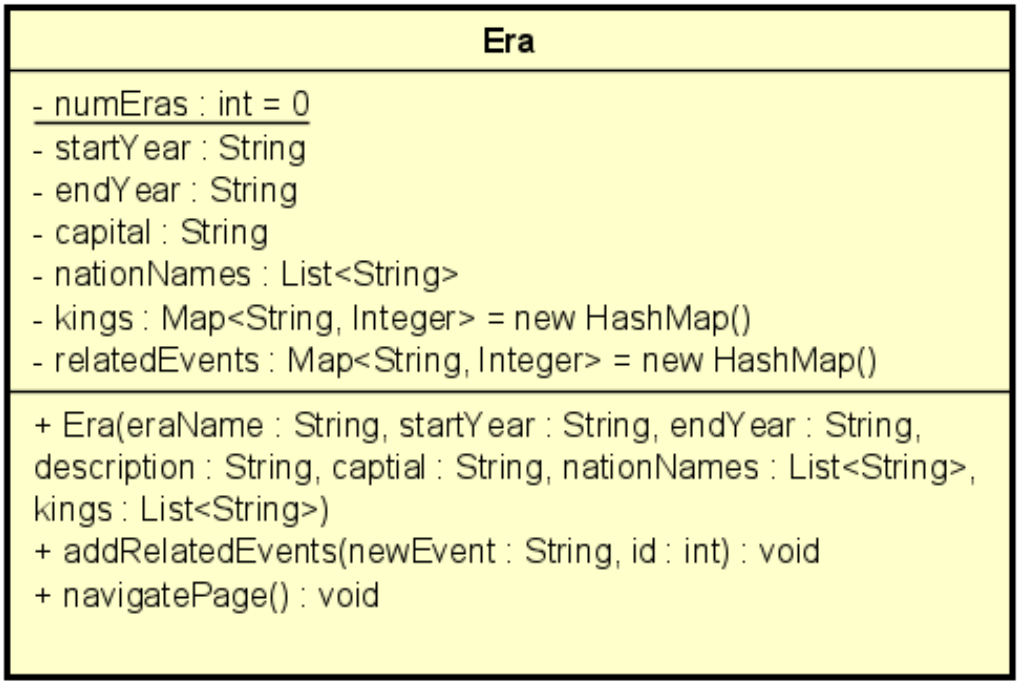
- numFigures: A static variable tracking the number of Figure instances and used to create Id automatic.
- otherNames: A list of strings representing the alternative names of the figure.
- bornYear, diedYear,location and role representing the basic information of the figure.
- eras, spouses, mother, father, children, relatedEvents, relatedRelics,relatedFestivals are maps associating related entities (strings) with their respective IDs (integers).
- *Comments: To provide greater flexibility, the related attribute in the Figure class is defined as a List<String> and Map<String,Integer> instead of a specific implementation such as ArrayList or HashMap. This allows for flexibility in choosing different types of implementations based on specific requirements and allows for easy switching between implementations without affecting the code that interacts with the original attribute.*

(b) Methods:

- All attributes provide getter methods and necessary setter methods to retrieve and modify their values.
- getAllNames(): Returns a list containing all the names associated with the figure, including the main name and alternative names.
- containsName(String comparedName): Checks if the provided name matches any of the names associated with the figure, considering alternative titles and specific naming patterns.
- containsNameForSearch(String name): Overrides the method from the “HistoricalEntity” class to check if the provided name is contained within the figure’s main name or any of its alternative names, disregarding case sensitivity.
- addRelatedEvents(String newEvent, int id): Adds a related event with its respective ID to the map of related events, if it does not exist.
- addRelatedFestivals(String newFestival, int id): Adds a related festival with its respective ID to the map of related festivals, if it does not exist.

- addRelatedRelics(String newRelic, int id): Adds a related relic with its respective ID to the map of related relics, if it does not exist.
- navigatePage(): Overrides the method from the "HistoricalEntity" class to navigate to the detail page of the figure using JavaFX.

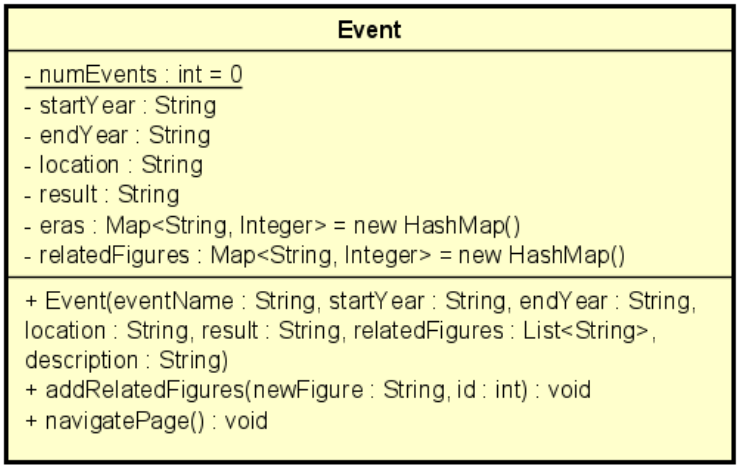
3. **Era**: This class, extending the HistoricalEntity abstract class, represents a historical era within the model package.



   (a) Attributes:
   - numEras: A static variable tracking the number of Era instances and used to create Id automatic.
   - startYear, endYear, capital, nationNames representing the basic information of the era.
   - kings, relatedEvents are maps associating related entities (strings) with their respective IDs (integers).

   (b) Methods:
   - All attributes provide getter methods and necessary setter methods to retrieve and modify their values.
   - addRelatedEvents(String newEvent, int id): Adds a related event with its respective ID to the map of related events, if it does not exist.
   - navigatePage(): Overrides the method from the "HistoricalEntity" class to navigate to the detail page of the era using JavaFX.

4. **Event**: This class, extending the HistoricalEntity abstract class, represents a historical event within the model package.



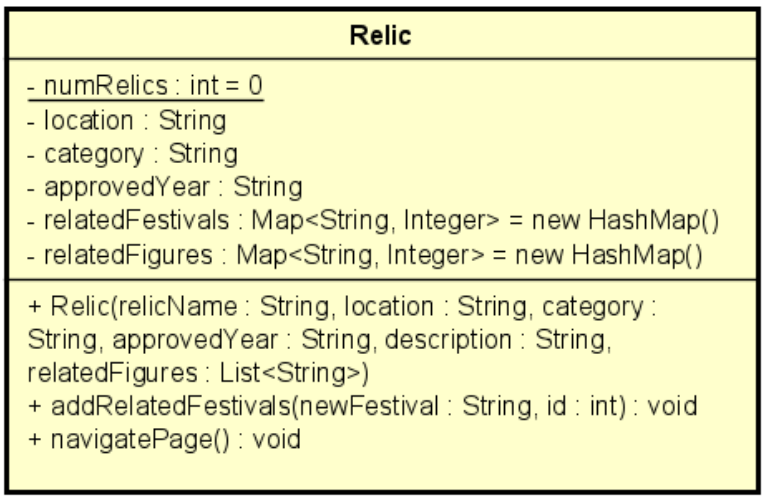
(a) Attributes:

- numEvents: A static variable tracking the number of Event instances and used to create Id automatic.
- startYear, endYear, location, result representing the basic information of the event.
- eras, relatedFigures are maps associating related entities (strings) with their respective IDs (integers).

(b) Methods:

- All attributes provide getter methods and necessary setter methods to retrieve and modify their values.
- addRelatedFigures(String newFigure, int id): Adds a related figure with its respective ID to the map of related figures, if it does not exist.
- navigatePage(): Overrides the method from the "HistoricalEntity" class to navigate to the detail page of the event using JavaFX.

5. **Relic**: This class, extending the HistoricalEntity abstract class, represents a historical relic within the model package.



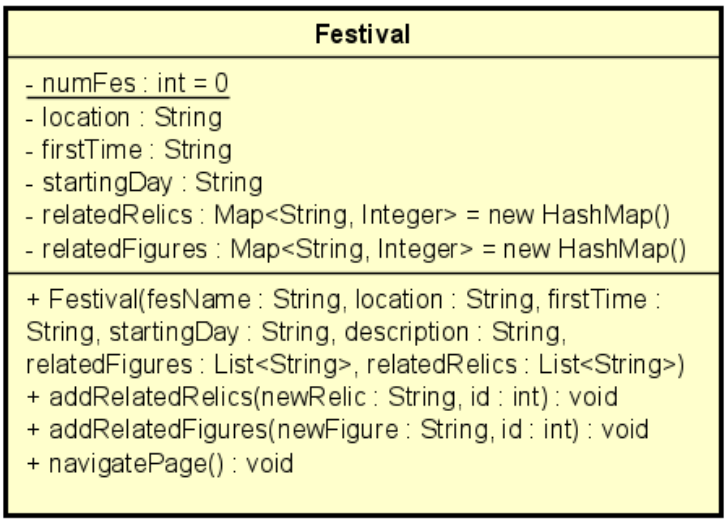
(a) Attributes:

- numRelics: A static variable tracking the number of Relic instances and used to create Id automatic.
- location, category, approvedYear representing the basic information of the relic.
- relatedFestivals, relatedFigures are maps associating related entities (strings) with their respective IDs (integers).

(b) Methods:

- All attributes provide getter methods and necessary setter methods to retrieve and modify their values.
- addRelatedFestivals(String newFestival, int id): Adds a related festival with its respective ID to the map of related festivals, if it does not exist.
- navigatePage(): Overrides the method from the "HistoricalEntity" class to navigate to the detail page of the relic using JavaFX.

6. **Festival**: This class, extending the HistoricalEntity abstract class, represents a historical festival within the model package.



(a) Attributes:

- numFes: A static variable tracking the number of Festival instances and used to create Id automatic.
- location, firstTime, startingDay representing the basic information of the figure.
- relatedRelics, relatedFigures are maps associating related entities (strings) with their respective IDs (integers).

(b) Methods:

- All attributes provide getter methods and necessary setter methods to retrieve and modify their values.
- addRelatedRelics(String newRelic, int id): Adds a related relic with its respective ID to the map of related relics, if it does not exist.
- addRelatedFigures(String newFigure, int id): Adds a related figure with its respective ID to the map of related figures, if it does not exist.
- navigatePage(): Overrides the method from the "HistoricalEntity" class to navigate to the detail page of the festival using JavaFX.

Significant OOP techniques

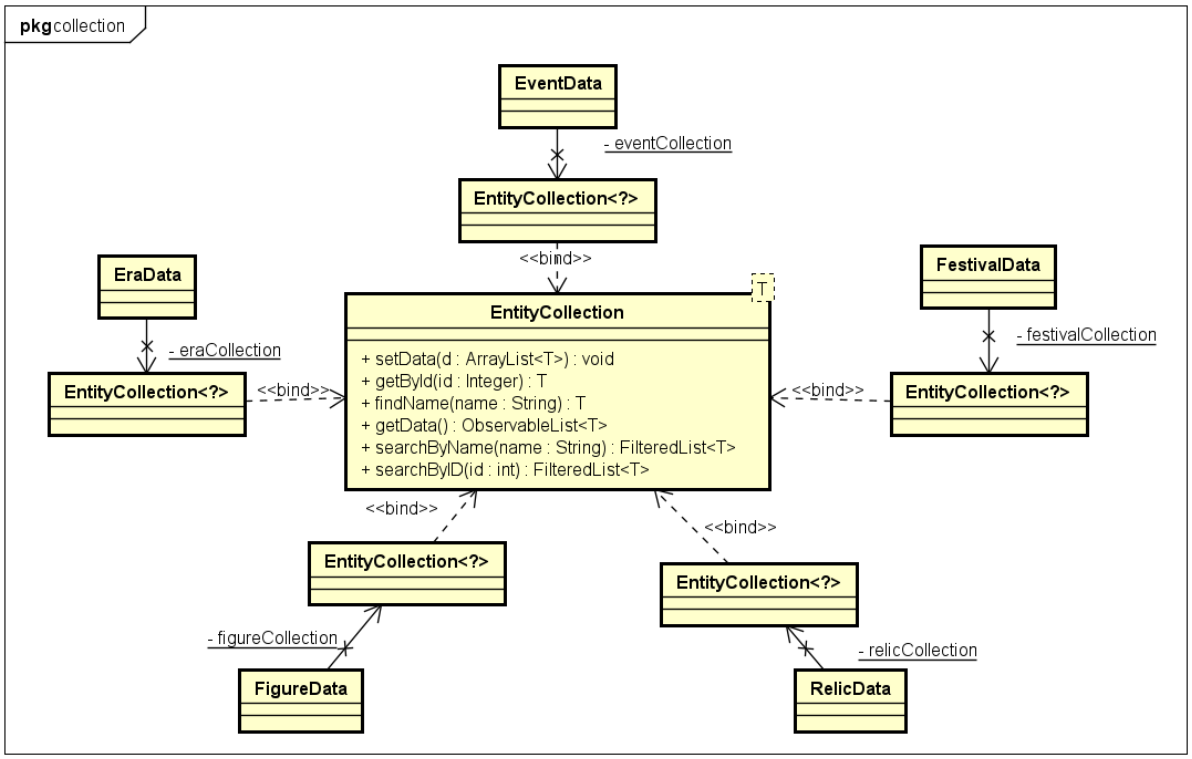
1. Inheritance:

- HistoricalEntity serves as a foundation for modeling specific historical entities, enabling code reuse and specialization.
- Figure, Era, Festival, Relic, Event extends HistoricalEntity, inheriting attributes and methods for reuse and polymorphism.

2. Encapsulation:
   - Attributes are encapsulated, promoting data integrity and controlled access through getter and setter methods.
3. Abstraction: The abstraction technique is also leveraged in the design of this package. Each class provides a high-level functionality, hiding its internal working mechanism.
4. Method Overriding:
   - The equals(Object obj) method is overridden to compare equality based on id and name.
   - containsNameForSearch(String name) overrides the method from HistoricalEntity, checking if the provided name is contained within the figure's name or alternative names.
5. Abstract class: HistoricalEntity is an abstract class that provides a common definition for all its derived classes. It also enforces the implementation of certain abstract methods, such as the navigatePage() method, in these subclasses.
6. Polymorphism: The navigatePage() method is overridden to provide a specialized implementation for navigating to the detail page of the entites using JavaFX.

3.2.4 Collection package

The collection package within our application plays a crucial role as a database, serving as a centralized storage solution for various entities. Upon application startup, the collection package loads data into memory, enabling efficient and convenient data retrieval, filtering, and more.



Class overview

1. **EntityCollection** class in the collection package serves as a database-like collection for storing entities. It utilizes JavaFX collections to provide observable and filterable lists. Here is an overview of the EntityCollection class:

   (a) Generic Type:

   - <T extends HistoricalEntity>: The generic type parameter T is bounded to the HistoricalEntity class, ensuring that the EntityCollection can store entities that inherit from HistoricalEntity.

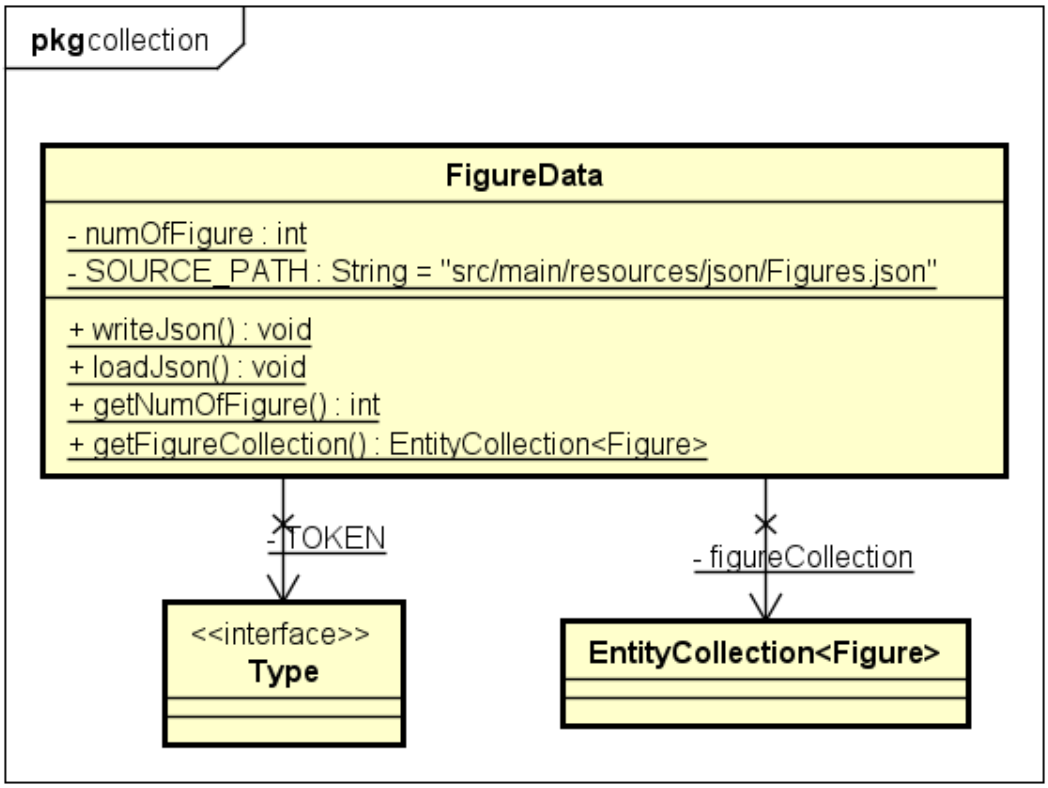
   (b) Attributes:

   - data (ObservableList<T>): An observable list that stores the entities.

   (c) Methods:

   - setData(ArrayList<T> d): Sets the data of the collection by converting the provided ArrayList to an ObservableList.
   - getById(Integer id): Retrieves an entity from the collection based on its unique identifier (id).
   - getData(): Returns the observable list of entities stored in the collection.
   - searchByName(String name): Creates a filtered list based on the entities' names that match the provided name.
   - searchByID(int id): Creates a filtered list based on the entities' unique identifiers that match the provided ID.

2. **FigureData** class in the collection package provides functionality to load and manage data for figures. Here is an overview of the FigureData class:



   (a) Attributes:

   - TOKEN: A constant Type object representing the token for serialization and deserialization of the figure data using Gson.

- figureCollection: An instance of EntityCollection<Figure> that stores the figures
- numOfFigure: An integer representing the number of figures.
- SOURCE_PATH: A constant string representing the path to the JSON file containing the figure data.

(b) Methods:

- writeJson(): This method is responsible for writing the figure data to a JSON file.
- loadJson(): This method loads the figure data from a JSON file using the JsonIO helper class. It deserializes the JSON data into an ArrayList<Figure>, sets the data in the figureCollection, and updates the numOfFigure accordingly.
- getNumOfFigure(): Returns the number of figures stored in the FigureData class.
- getFigureCollection(): Returns the EntityCollection<Figure> instance containing the figure data.

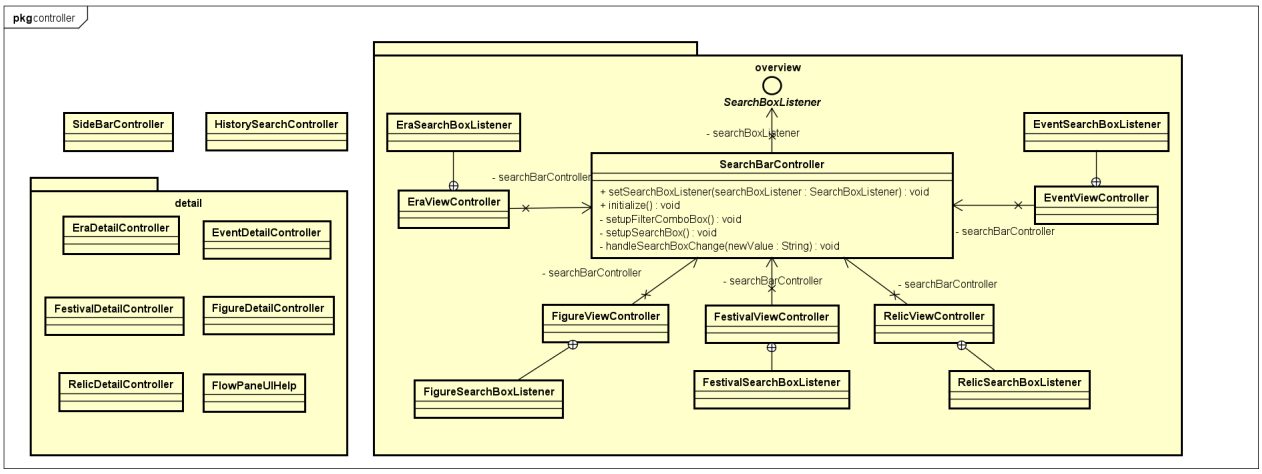
3. Other classes, including EraData, FestivalData, RelicData, and EventData, follow a similar concept as FigureData.

Significant OOP techniques

1. Generics: The class is parameterized with the generic type T to provide flexibility in storing and retrieving different types of entities.
2. Encapsulation: The class encapsulates the data list and provides public methods to manipulate and retrieve the data, ensuring data integrity and abstraction.
3. Singleton Pattern: The class implement Singleton pattern as it provides a centralized access point to the figure collection and restricts the instantiation of multiple instances.

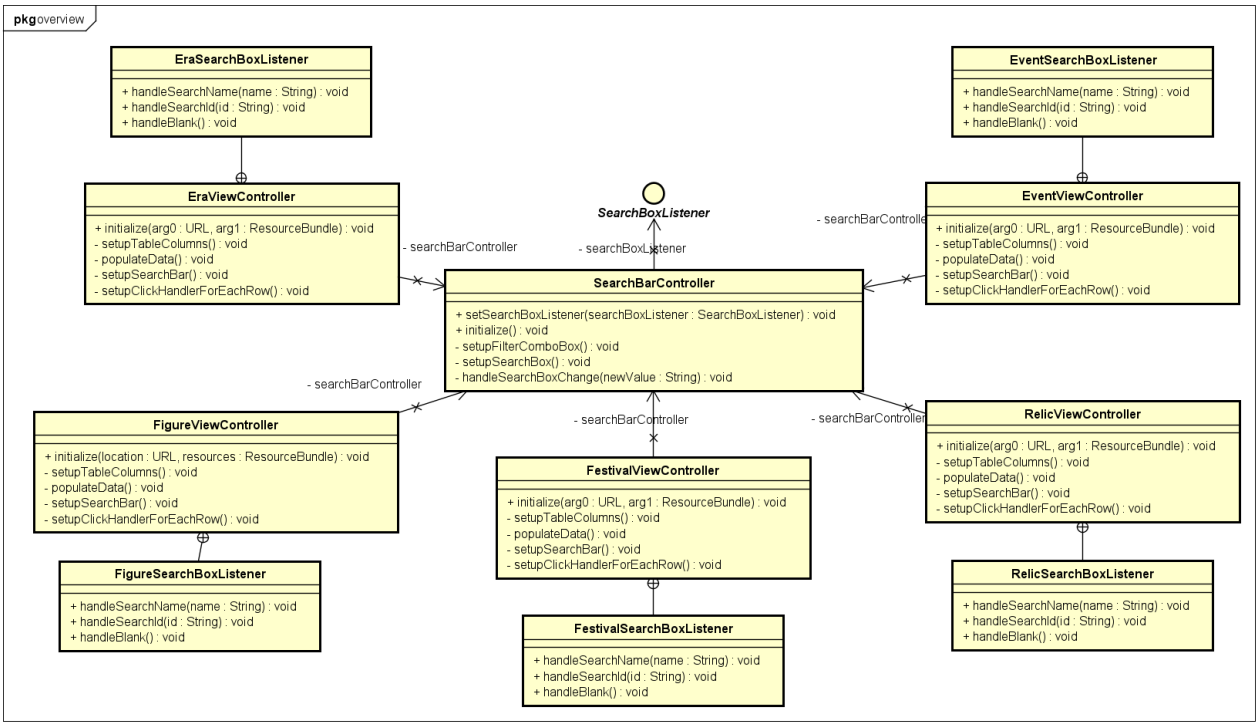
3.2.5 Controller package

The controller package in the application plays a crucial role in handling user interactions, coordinating data flow, and managing the overall behavior of the application. It contains classes that act as intermediaries between the user interface (UI) and the underlying data models.



3.2.5.a controller.overview package

The controller.overview package is an essential component of the application's controller layer. This package plays a crucial role in managing and controlling the visual presentation of data and user interactions within the application's overview views.By separating the responsibilities of handling user interactions and updating the views, the controllers in the controller.overview package help to achieve a clear separation of concerns in the application's architecture



Class overview

1. **SearchBoxListener** interface, located in the controller.overview package, serves as a contract defining methods for handling search-related actions in the application's overview functionality. This interface is designed to be implemented by classes that require search functionality and acts as a callback mechanism for notifying the implementing classes about search events.

    - handleSearchName(String name): This method is invoked when a search action is triggered with a specific name.
    - handleSearchId(String id): When a search action is initiated with a specific ID, this method is called.
    - handleBlank(): This method is invoked when the search action is performed with no specific criteria or a blank input.

2. **SearchBarController** class, located in the controller.overview package, serves as the controller for the search bar component in the application's overview functionality. This controller is responsible for handling user interactions with the search bar, managing the filter options, and invoking appropriate actions based on the user's input.

    (a) Attributes:

    - filterComboBox: A ComboBox control representing the filter options for the search. Users can choose to search by name or ID using this dropdown.
    - searchBox: A TextField control representing the input box where users enter their search query.
    - searchBoxListener: An instance of the SearchBoxListener interface that is responsible for handling search-related actions.

    (b) Methods:

    - setSearchBoxListener(SearchBoxListener searchBoxListener): A method that allows setting the SearchBoxListener instance to be used for handling search events. This method is typically called by the parent controller or component to establish the communication between the search bar and the consuming class.

- initialize(): An initialization method annotated with @FXML that is automatically called when the associated FXML file is loaded. This method sets up the filter options in the filterComboBox and configures the search box.
- setupFilterComboBox(): A helper method that initializes the filterComboBox by setting its items to a list of filter options.
- setupSearchBox(): A helper method that sets up the search box by attaching a listener to its textProperty(). This listener is triggered whenever the text in the search box changes, and it invokes the handleSearchBoxChange() method.
- handleSearchBoxChange(String newValue): A method that is called when the value in the search box changes.

3. **FigureViewController** class serves as the controller for the figure view in the application's overview functionality. This controller is responsible for managing the display of figures in a table view, setting up table columns, populating data, handling search bar interactions, and enabling row click events for navigation to detail views.

   (a) Attributes:
   - tblFigure: A TableView control representing the table view where figures are displayed.
   - colFigureId, colFigureName, colFigureEra, colFigureOverview: TableColumn controls representing the columns of the table view, specifying the data to be displayed in each column.
   - searchBarController: An instance of the SearchBarController that manages the search bar component.

   (b) Methods and inner class:
   - initialize(URL location, ResourceBundle resources): This method sets up the table columns, populates the data, and configures the search bar and row click events.
   - setupTableColumns(): A helper method that configures the cell value factories for each table column, mapping them to the corresponding properties of the Figure class.
   - setupTableColumns(): A helper method that configures the cell value factories for each table column, mapping them to the corresponding properties of the Figure class.
   - populateData(): A method that populates the table view with data from the FigureData class.
   - setupSearchBar(): A method that sets up the search bar by setting the SearchBoxListener instance in the searchBarController.
   - setupClickHandlerForEachRow(): A method that sets up a click handler for each row in the table view to navigates to the detail view using the App.pageNavigationService.

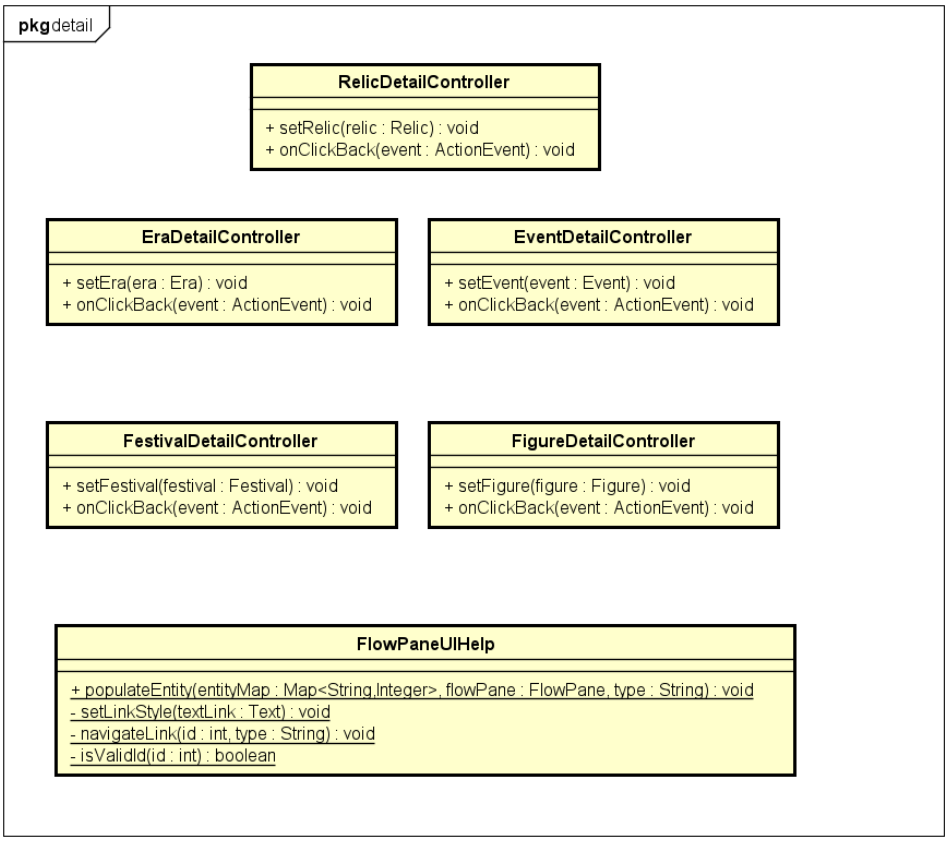
4. Other classes, such as **EraViewController, EventViewController, FestivalViewController, and RelicViewController,** follow a similar concept as **FigureViewController**.

Significant OOP techniques

1. Encapsulation: Private fields and methods are used to encapsulate the internal functionality of the class.
2. Event Handling: The class defines event handlers for mouse clicks on table rows and search box changes, utilizing event-driven programming.
3. Abstraction: The interface defines abstract methods handleSearchName, handleSearchId, and handleBlank, providing an abstraction for handling search events.

3.2.5.b controller.detail package

The controller.detail package contains controller classes that handle the behavior and functionality of specific detail views in the application. These controllers are responsible for managing the display of detailed information about specific entities and facilitating user interactions within the detail view.



Class overview

1. **FlowPaneUIHelp** class, located in the controller.detail package, provides helper methods for creating and populating links within a FlowPane based on the provided entity map. This class assists in generating clickable links and handling navigation to other detail pages within the application.

   (a) Methods:

   - populateEntity(): a key method in this class. It takes a Map<String, Integer> entityMap, representing the names of entities and their corresponding IDs, a FlowPane to contain the links, and the type of the entity. When a user clicks on a link, the navigateLink() method is invoked and navigate to the corresponding detail page based on the ID and entity type.
   - setLinkStyle(): defines the visual styling for the links, including the color, underline on hover, and cursor change on hover.
   - The navigateLink() method handles the navigation to the appropriate detail page by invoking the App.pageNavigationService and passing the ID and entity type.
   - The isValidId() method is a helper method that checks if an ID is valid.

2. **FigureDetailController** class is responsible for controlling the behavior and interaction of the figure detail view in the application. It manages the display of various text fields and flow panes that present detailed information about a specific figure.

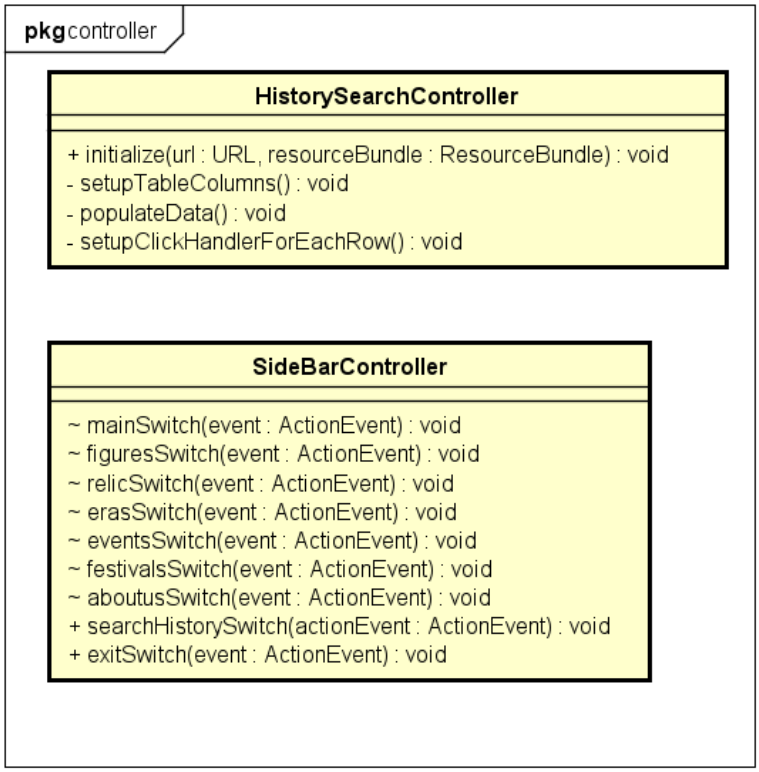
   (a) Attributes:

   - nameText, realNameText, bornText, diedText, overviewText, locText, workTimeText: Text controls representing different aspects of the figure's information.
   - eraFlowPane, fatherFlowPane, motherFlowPane, childrenFlowPane, spouseFlowPane, aliasFlowPane, eventFlowPane, relicFlowPane, festivalFlowPane: FlowPane controls used to display associated entities.

- setFigure(Figure figure): A method that sets the figure object and updates the UI elements with the corresponding data.
- onClickBack(ActionEvent event): A method that handles the back button click event.

3. Other classes such as: **RelicDetailController, EraDetailController, EventDetailController, FestivalDetailController** share the similar concept as **FigureDetailController**

3.2.5.c Other classes in controller package



1. **SideBarController** class, located in the controller package, serves as the controller for the sidebar component in the application. This controller handles user interactions with the sidebar menu and triggers corresponding actions based on the selected menu item.

   (a) Methods:
   - mainSwitch(): A method that handles the action when the "Main" menu item is selected.
   - figuresSwitch(), relicSwitch(), erasSwitch(), eventsSwitch(), festivalsSwitch(): These methods handle the actions when the corresponding menu items for figures, relics, eras, events, and festivals are selected.
   - aboutusSwitch(): A method that handles the action when the "About Us" menu item is selected.
   - aboutusSwitch(): A method that handles the action when the "About Us" menu item is selected.
   - exitSwitch(): A method that handles the action when the "Exit" menu item is selected.

2. **HistorySearchController** class, located in the controller package, serves as the controller for the search history view in the application. This controller is responsible for managing the display of search history in a table view, setting up table columns, populating data, and enabling row click events for navigation to detail views.

   (a) Attributes:
   - historyTableView: A TableView control representing the table view where search history is displayed.

- colName, colType, colTime: TableColumn controls representing the columns of the table view, specifying the data to be displayed in each column.

(b) Methods:

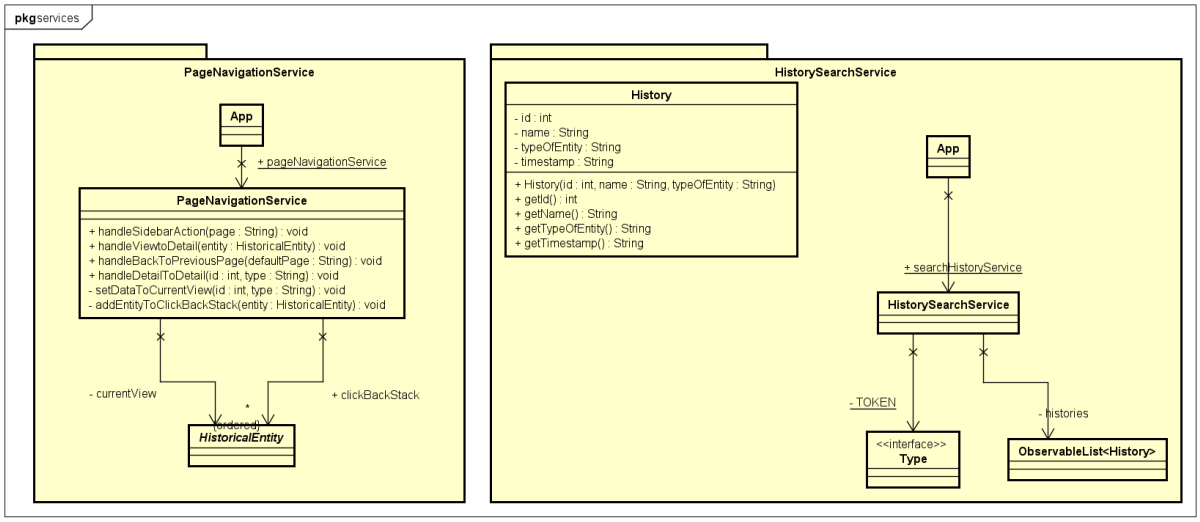
- initialize(URL url, ResourceBundle resourceBundle): This method sets up the table columns, populates the data, and configures the row click events.
- setupTableColumns(): A helper method that configures the cell value factories for each table column, mapping them to the corresponding properties of the History class.
- populateData(): A method that populates the table view with data from the search history.
- setupClickHandlerForEachRow(): A method that sets up a click handler for each row in the table view. When a row is double-clicked, it retrieves the associated History object and navigates to the related detail page.

Significant OOP techniques

1. Encapsulation: The controller classes encapsulate the behavior and data related to specific detail views. They encapsulate the UI elements, event handling methods, and data manipulation logic within each controller, providing a cohesive and self-contained unit of functionality.

2. Event-driven Programming: The controllers follow the event-driven programming paradigm by responding to user events, such as button clicks or mouse interactions. They utilize event handling methods and bind them to specific UI components to trigger actions or update the view based on user input.

3.2.6 Services package



The services package in the application plays a crucial role in providing various services and functionalities that support the core operations and features. These services encapsulate specific functionality and business logic to ensure modular, reusable, and well-organized code.

3.2.6.a HistorySearchService package

The services.HistorySearchService package provides functionality for managing the search history in the application. It consists of classes that handle the loading, writing, and retrieval of search history data.

Class overview

1. **History** class, located in the services.HistorySearchService package, represents a single entry in the search history. It encapsulates information such as the ID, name, type of entity, and timestamp of the search.

(a) Attributes

- id: An integer representing the ID of the search history entry.
- name: A string representing the name of the entity that was searched.
- typeOfEntity: A string representing the type of entity that was searched.
- timestamp: A string representing the timestamp when the search was performed.

(b) Methods:This class only contains getter methods

2. **HistorySearchService** class, located in the services.HistorySearchService package, is responsible for managing the search history functionality in the application. It provides methods to load and write the search history data from/to a JSON file, add new search entries to the history, and retrieve the list of search histories.

(a) Attributes:

- TOKEN: A constant of type Type representing the token for deserializing the JSON data into an ArrayList of History objects.
- SOURCE_PATH: A constant string representing the file path of the JSON file storing the search history data.
- histories: An ObservableList of History objects representing the search history entries.

(b) Methods:

- loadJson(): A method that loads the search history data from the JSON file.
- writeJson(): A method that writes the search history data to the JSON file.
- addToSearchHistory(HistoricalEntity entity): A method that adds a new search entry to the search history.
- getHistories(): A getter method that retrieves the histories list, allowing external classes to access the search history data.
- deleteOldHistory(): A private helper method that deletes the old history entries beyond a certain limit.

3.2.6.b PageNavigationService package

The services.PageNavigationService package provides a PageNavigationService class responsible for handling page navigation within the application and managing the navigation history.

1. Attributes:

- clickBackStack: A stack data structure that stores instances of HistoricalEntity representing the navigation history of clicked pages.
- currentView: A reference to the currently displayed HistoricalEntity page.

2. Methods:

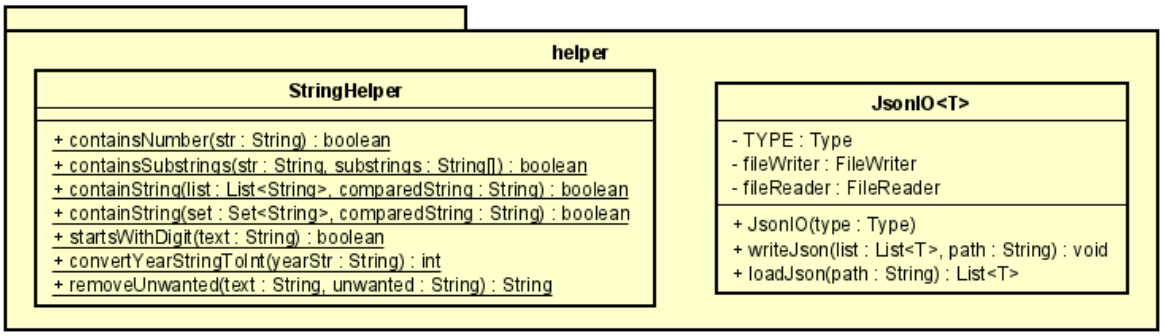
- handleSidebarAction(String page): Handles the action triggered by clicking on a sidebar item.
- handleViewtoDetail(HistoricalEntity entity): Handles the action of navigating from an overview page to a detail page.
- handleBackToPreviousPage(String defaultPage): Handles the action of going back to the previous page.
- handleDetailToDetail(int id, String type): Handles the action of navigating from a detail page to another detail page.
- setDataToCurrentView(int id, String type): Sets the currentView based on the specified id and type.
- addEntityToClickBackStack(HistoricalEntity entity): Adds the specified entity to the clickBackStack.

Significant OOP techniques

1. Encapsulation: PageNavigationService class encapsulates the logic for handling page navigation and managing the navigation history.
2. Abstraction: PageNavigationService abstracts the navigation functionality, allowing other parts of the application to navigate between pages without worrying about the internal implementation details.
3. Composition: The PageNavigationService class makes use of composition by having a Stack object (clickBackStack) as an attribute. It uses the stack to store instances of HistoricalEntity, enabling navigation history management.

3.2.7 Helper package

The Helper package serves as a utility, providing commonly used methods that are shared across various classes. These include methods for reading and writing JSON data, as well as methods for string processing.



Class overview

1. **StringHelper**: The StringHelper class provides a suite of string processing methods that are frequently employed throughout our application. It acts as a utility hub for common string manipulations and operations.
   (a) Attributes: StringHelper is a utility class that solely provides utility methods, and thus, it does not require any attributes.
   (b) Methods:
   - containsNumber(String str): Checks if a string contains any numeric digits.
   - containsSubstrings(String str, String... substrings): Checks if a string contains any of the provided substrings (case insensitive).
   - containString(List<String> list, String comparedString): Checks if a list of strings contains a specific string (case insensitive).
   - containString(Set<String> list, String comparedString): Checks if a set of strings contains a specific string (case insensitive).
   - startsWithDigit(String string): Determines if a string starts with a numeric digit.
   - convertYearStringToInt(String yearStr): Converts a year from a string format to an integer format.
   - removeUnwanted(String text, String unwanted): Splits text into different parts and removes parts that contain unwanted words.
   - *Comments: All methods are defined as static, allowing them to be called directly without the need to instantiate the StringHelper class.*
2. **JsonIO**: The JsonIO class is a generic class, designed to work with all types of Java objects. It primarily handles writing to, and loading from, JSON files, thereby serving as a crucial component in our data storage and retrieval processes.
   (a) Attributes:

- TYPE: The type of Java objects loaded from JSON files.
- fileWriter: An instance of a built-in Java class used to write text to files.
- fileReader: An instance of a built-in Java class used to read text from files.

(b) Methods:

- writeJson(List<T> list, String path): Converts a list of objects of any type into JSON format and writes the resulting JSON text to a specified file.
- loadJson(String path): Reads text from a specified JSON file and converts the text into a list of Java objects corresponding to the TYPE attribute.

Significant OOP techniques

(a) Abstraction: The JsonIO and StringHelper classes provide a simplified high-level interface to complex underlying code related to JSON file manipulation and string processing, respectively. Users of these classes do not need to understand the implementation details in order to utilize their functionalities.

(b) Encapsulation: Both StringHelper and JsonIO classes encapsulate their data (attributes) and methods into a single unit. For instance, the TYPE, fileWriter, and fileReader in JsonIO class are hidden from the outside world and can only be accessed via methods.

(c) Generics: The JsonIO class utilizes Java Generics which allows it to operate on different types, improving code reusability and type safety.

(d) Method Overloading: The containString() method in StringHelper class is overloaded to receive either a List or a Set. This enhances the class's flexibility in accommodating different data structures.

(e) Variable Arguments: The containsSubstring() method in StringHelper class demonstrates the use of variable arguments, allowing it to receive any number of substrings.

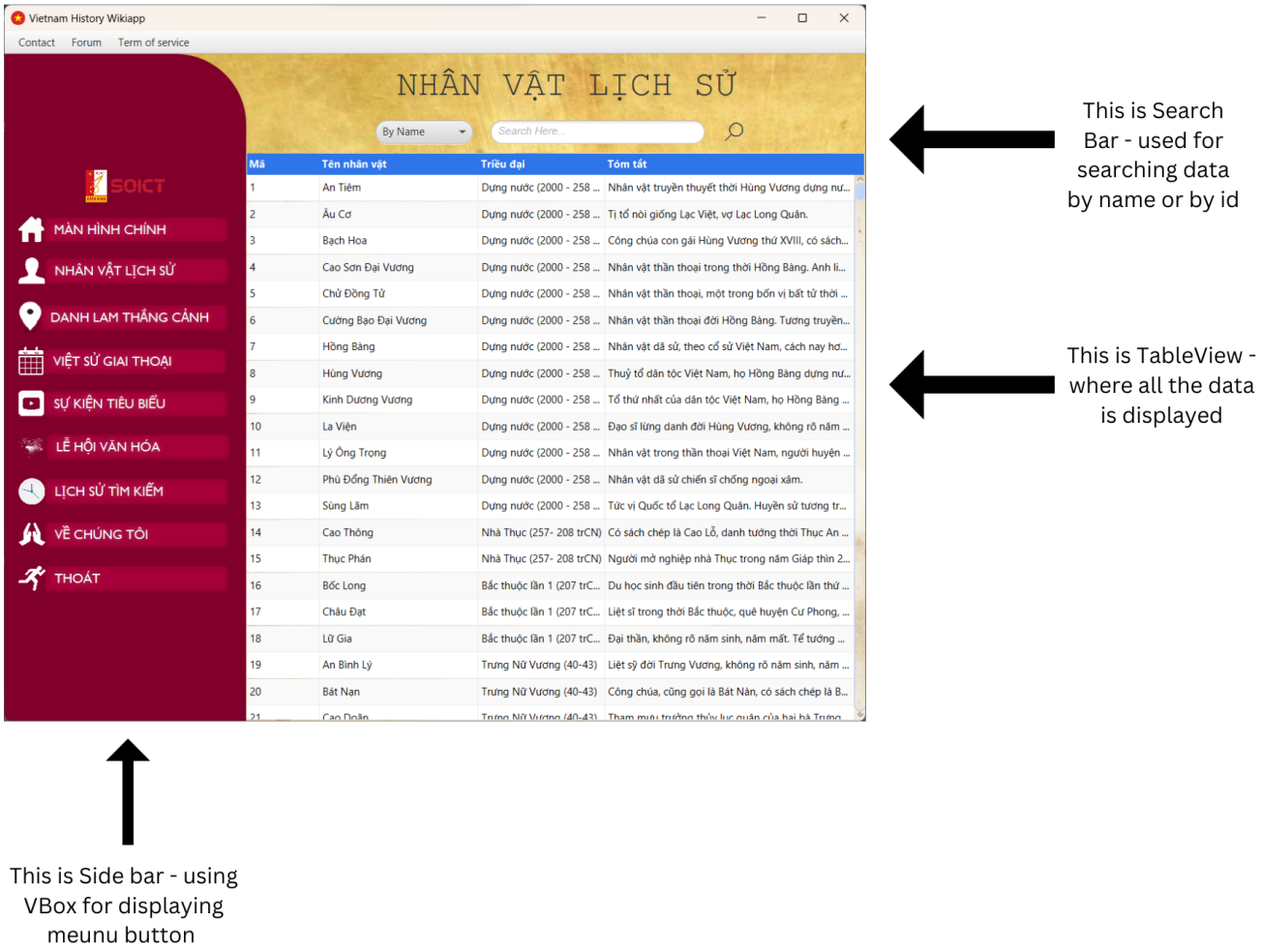
(f) Exception Handling: Both StringHelper and JsonIO classes handle exceptions, specifically in file reading and writing operations, ensuring robust and reliable operation of the program. They provide specific error messages that help in debugging and rectifying problems.

3.2.8 View

In this project, as there are many components so we want to break it to as many independent as we can for convenient and maintenance purpose, so that when we want to update some things from 1 scene, it will not affect others. The order that we list the fxml file below is the order that we add scenes to the stage.

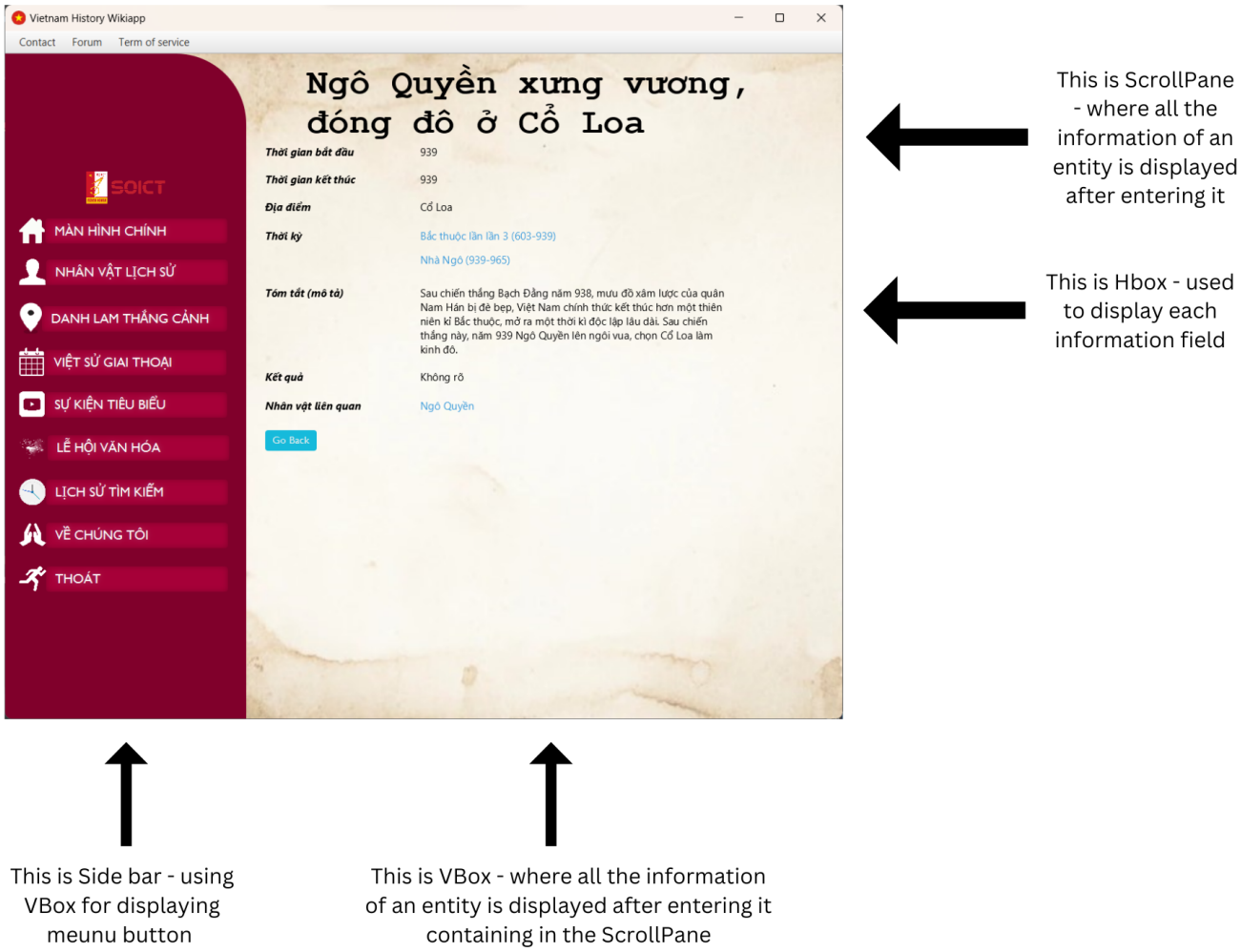
Our GUI is inspired by this: https://www.youtube.com/watch?v=cPF3qGTjYgk

Overview screen



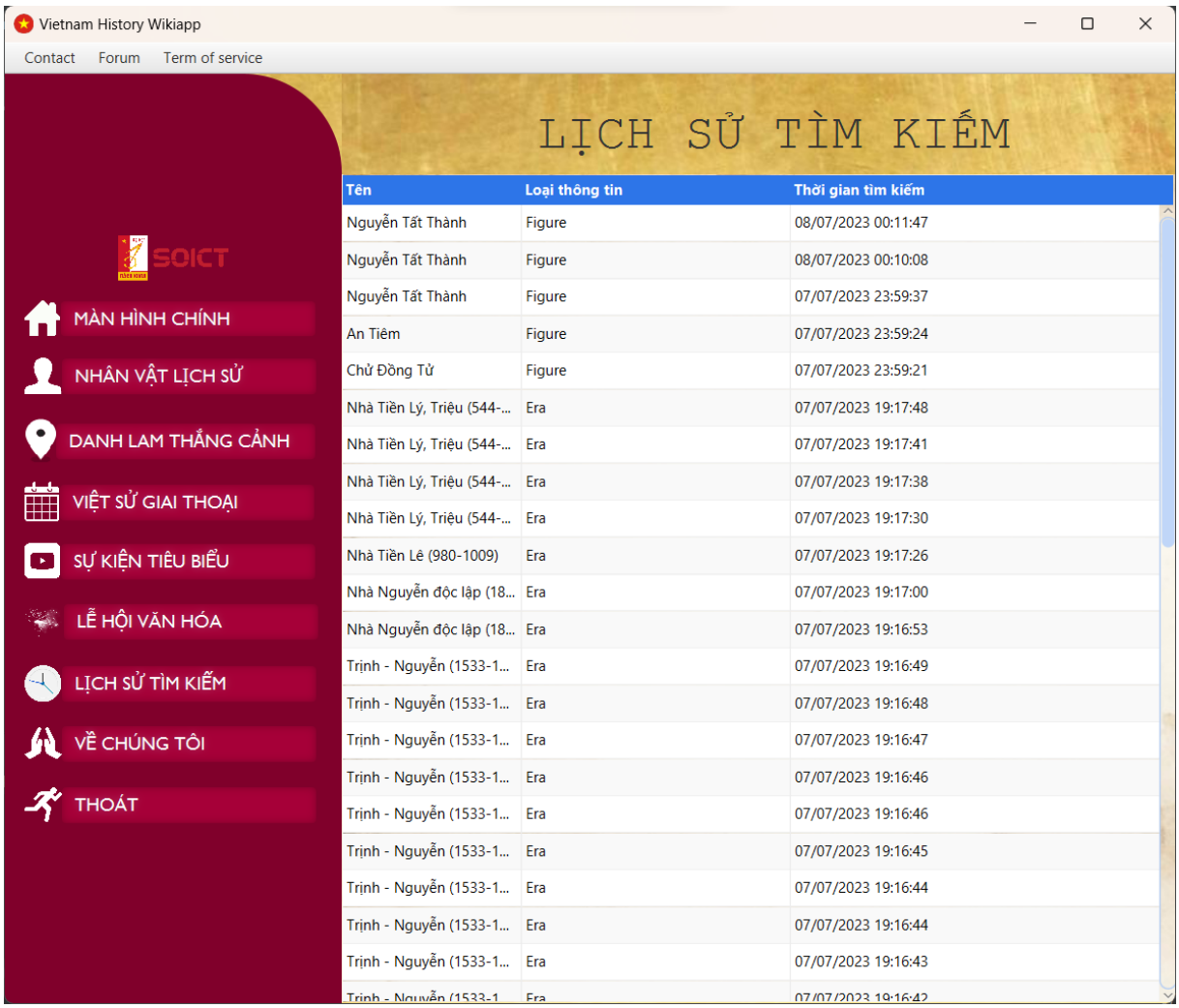
- Each historical category will have the same layout structure, contain a sidebar, searchbar and a TableView. This layout is designed to provide a user-friendly way to navigate and view data related to each history entity.
- **SearchBar.fxml**: In the "SearchBar" layout, the layout creates a simple search bar interface with a filter dropdown, text input field, and a search icon for visual representation. The layout is structured with an HBox that contains three child elements: the ComboBox, TextField, and ImageView. The ComboBox and TextField have specific styles defined using CSS classes and stylesheets. The ComboBox provides a dropdown menu for selecting different filters for searching, namely “By id” and “By name”, while the TextField allows the user to input their search query. The ImageView displays a search icon image. The Insets define the padding around the search bar, specifying the spacing between the search bar and other elements.
- **SideBar.fxml**: In the "Sidebar" layout, we created a VBox that contains multiple FlowPane elements, each representing a row in the side bar. Each FlowPane consists of an ImageView displaying an icon image and a Button representing a specific functionality or option. The buttons have specific styles defined using CSS classes and stylesheets. The side bar provides quick access to different sections or features of the application, such as the main screen, historical figures, relics, eras, events, festivals, search history, about us, and exit. Each button is associated with an action or event handler in the corresponding controller class.

Detail screeen



- Similar to Overview screen, every category will follow a consistent layout structure that includes a sidebar and a Pane for the convenient data overview.
- The BorderPane is the main container that organizes the different sections of the GUI.
- The center section is wrapped inside a ScrollPane, allowing for vertical scrolling when the content exceeds the available space. Inside the ScrollPane, a VBox is used to arrange the different components vertically. The center section includes a heading Text component inside an HBox.
- Below the heading, there is another ScrollPane that wraps a VBox containing various information of a category. In here, let us take “Event” entity as an example, the VBox includes several HBox containers, each representing a different aspect of the event, such as time interval, location, related eras, description, result and related historical figures.
- At the bottom, there is a Button for navigating back.

History view screeen



- The history view screen has the similar layout structure to Overview screen, the only difference here is that its BorderPane only contains the Side bar and the TableView with three information fields, including name, data type and searching time.

3.3 Technical Details

By leveraging the capabilities of these following tools, our Vietnam history application can effectively handle historical data retrieval, user interface interactions, data serialization, and provide an intuitive user experience.

- Java Coding Convention: Classes, packages, variable names, and functions are standardized.
- Building class and package diagrams using UML, Asta.
- JavaFX with Scene Builder: The user interface of the history application is implemented using JavaFX, a modern UI framework that allows for creating rich and interactive user interfaces. JavaFX provides various UI controls, layouts, and styling options to design and build the application's screens and components.
- Maven: Maven is used as the build tool for the history application. It manages the project's dependencies, compiles the source code, and generates the executable artifact. Maven simplifies the build process and helps maintain a consistent project structure.

- Jsoup: Jsoup is an open-source Java library that facilitates web scraping and HTML parsing. It allows the application to fetch and extract data from HTML documents, making it useful for retrieving historical information from websites or online sources.
- Gson: Gson is a Java library developed by Google that provides an easy-to-use API for converting Java objects to JSON and vice versa. It enables the history application to serialize Java objects into JSON format for storage or data exchange and deserialize JSON data into Java objects for processing and manipulation.

4 Demonstration

Setup instructions

We highly recommend the user to run this application by using Github and Git command, following these steps:

1. Download the repository files (project) from the download section or clone this project to your local machine by typing in the bash the following command in the terminal: “git clone https://github.com/ngobach26/History.git”.
2. Run the application in the main Class “/src/main/java/main/App.java”.
3. Finally, just enjoy our project and have fun exploring Vietnam’s history.