Design Proposal – Online Academic Research Tool

Objective

The objective is to design an Agent-Based System (ABS) for academic research online. The criteria are to use agents to identify, retrieve, process, and store data.

The requirements have been specified as a user story in Appendix A and in the use case diagram in Figure 1. In brief, a user enters a search phrase onto a desktop application, which will search one or more online document repositories as selected by the user. The results will be stored in an offline location and the user will be notified on the User Interface (UI).

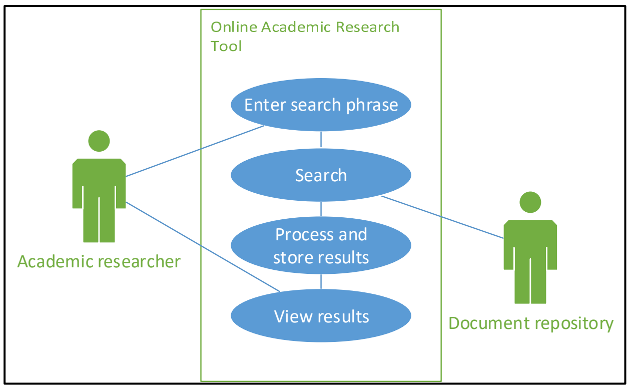

Figure 1. Use Case Diagram

Methodologies

Elements of agile development methodologies were used, including user stories and iterative design. Agent UML (AUML), an adaptation of UML sequence diagrams with the addition of state diagram notation to represent agent interactions (Bauer et al, 2001), with the further addition of queue notation suggested by van Ingen Schenau (2014) were used to describe the multi-agent system architecture.

Behaviour-driven development (BDD testing) will be applied throughout the development cycle to ensure a robust cycle of develop-test-improve.

The program will be written in Python for its extensive libraries, portability, maintainability, and simplicity (Lutz, 2011).

System architecture

A multi-agent architecture was chosen with three agents operating asynchronously as shown in Figure 2. The benefits of the multi-agent paradigm include: 1. Modularity, because each agent can be developed, updated, and maintained independently; 2. Parallelism, with improved efficiency and speed; 3. Robustness, because bugs are contained within an agent; and 4. Scalability, because it is easy to extend functionality with the enhancement of existing and deployment of new agents.

The multi-agent architecture will be implemented with multi-threading (Gustafsson, 2005) to achieve the multi-agent benefits with the simplicity of developing a single code base.

Python will be used with the following libraries:

Library	Purpose
requests	making HTTP requests
csv	enable the export agent to write a csv file
threading	allow the agents to run in separate threads
queue	create and manage the processing and export queues
os	interacting with the operating system
platform	portability between operating systems
datetime	add a timestamp to the filename

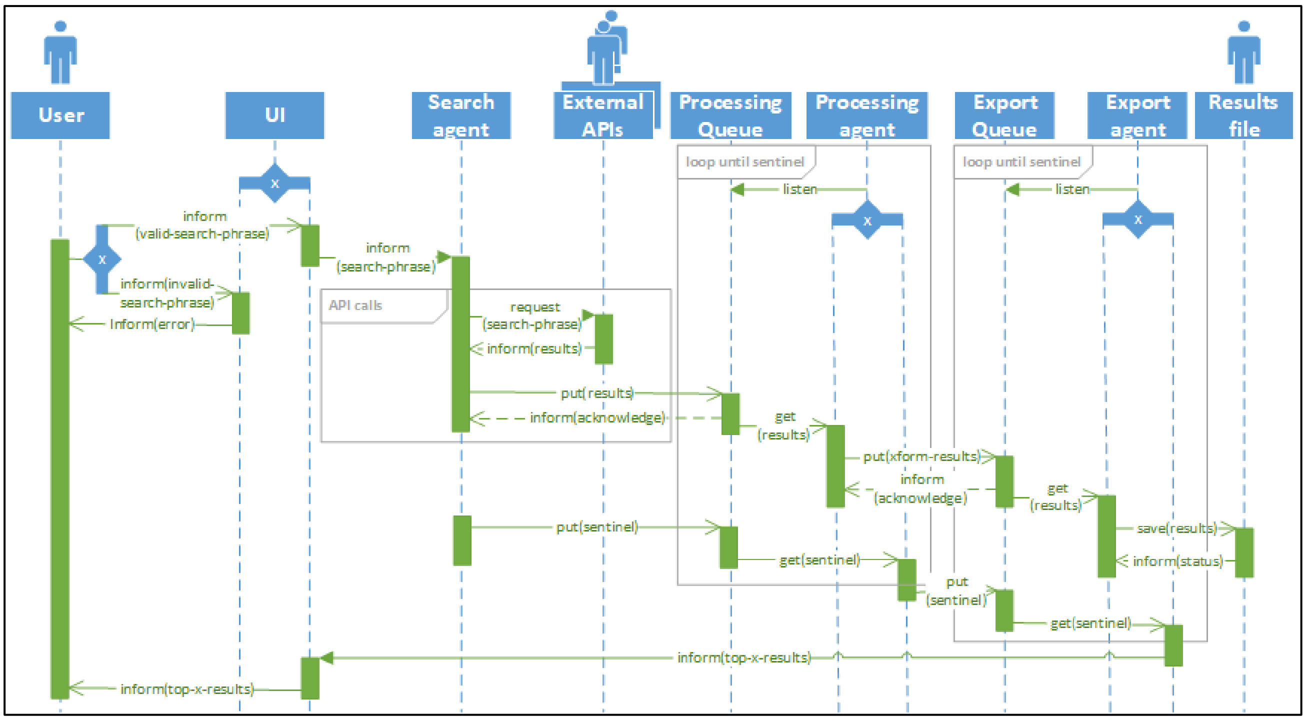

Figure 2. AUML sequence diagram

Component designs

The interactions between the components are illustrated in the activity diagram in Figure 3.

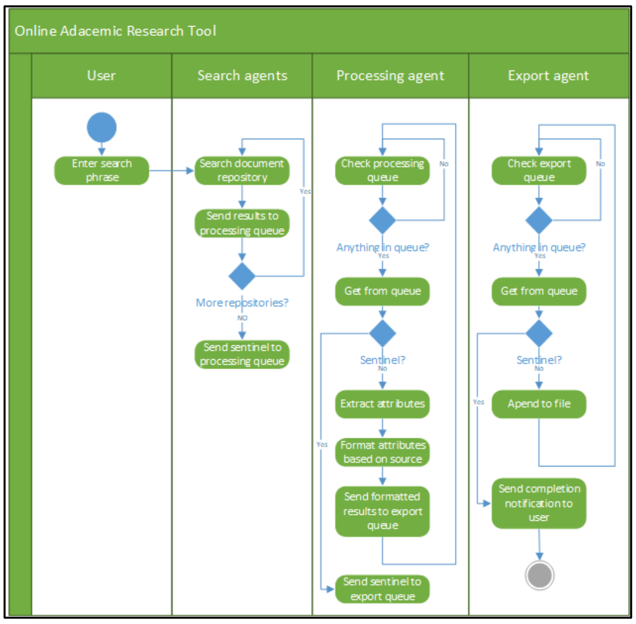

Figure 3. UML Activity Diagram

User Interface (UI)

The desktop application will be built on PyQt, a Python binding of the Qt graphical user interface toolkit because it allows powerful interfaces to be created easily.

The user will enter a search string along with which sources they wish to interrogate from a predefined list based upon the coded Application Programming Interfaces (APIs), and the maximum number of results they want from each API. Input validation will be performed after which the user-entered search string and maximum results per API will be sent to the search agent.

When the search and processing has completed the UI will display the file path and the exported data will be opened in the application associated in the operating system with .csv files.

Search Agent

The search agent will receive the parameters entered on the UI.

There will be a separate function for each document repository tailored to its API.

The APIs will be called in sequence, according to the user-entered parameters, returning results in JSON format.

The results from each API call will be added to a processing queue as soon as they are received, on a first in first out basis. This allows the processing agent to start work on the results of one document repository while the search agent is interrogating the next source.

After all APIs have been interrogated and results loaded onto the processing queue, a sentinel will be loaded onto the queue to signal the end of the search to the processing agent. The search agent will then stop.

Processing agent

The processing agent will monitor the processing queue and pick up items as they appear, while the search agent is still making API requests.

There will be a separate function for each document repository, tailored to its response format.

Each result will be processed by its respective function, extracting the required features such as article title, author, and so forth into a consistent format.

Once processed, the data will be passed to an export queue to be picked up by the export agent while the processing queue will be monitored for the next result.

When a sentinel is read from the processing queue it will signal the end of the search so the processing agent will send a sentinel to the export queue.

Export agent

The export agent will monitor the export queue and pick up the processed data while the processing agent and the search agent are still performing their functions.

It will check whether a CSV file already exists and will create one along with a header if it does not.

Each processed result will be appended as a row to the CSV file.

When a sentinel is read from the export queue this signals the end of the search so the export agent will signal back to the UI that the search is complete, passing the location of the CSV file to the UI and opening the file.

Challenges faced

We discussed developing a screen scraping capability or using APIs for data retrieval. Once we settled on using APIs we initially considered SerpAPI to plug into Google Scholar. However, we felt that this would give no advantages over simply using Google Scholar directly, plus there is a monthly limit to SerpAPI calls, so we chose to integrate with individual document repository APIs directly.

We were concerned about showing independence between multiple agents within a single program. After some research, we decided to use multi-threading to achieve this.

We found that UML didn’t sufficiently represent multi-agent architecture. After research, we found AUML was better, but when we decided to use queues to pass information between agents we found it still was not easy to represent. We therefore had to further enhance AUML using suggestions from van Ingen Schenau (2014).

Future enhancements

Interfacing directly with individual APIs means a smaller set of documents would be available initially than using SerpAPI to access everything that Google Scholar can see. However, it would create significant future development options such as the direct extraction of abstracts which is not available through Google Scholar.

When more document repositories are coded we could get more sophisticated with search and processing agents dedicated to each API running in parallel to further increase the benefits of the multi-agent architecture.

Word count: 1,091

References

Bauer, B., Müller, J.P. and Odell, J. (2001) Agent UML: A Formalism for Specifying Multiagent Software Systems. *International Journal of Software Engineering and Knowledge Engineering*, 11(03): 207-230.

Gustafsson, A., 2005. Threads without the Pain: Multithreaded programming need not be so angst-ridden. *ACM Queue*, 3(9): 34-41.

Lutz, M. (2011) *Programming Python.* 4th ed. Sebastopol: O'Reilly Media Inc.

Van Ingen Schenau, B. (2014), UML: Queue processor in a sequence diagram. Available from: https://softwareengineering.stackexchange.com/a/252773 [Accessed 28 May 2023].

Appendix A: User Story

As an academic researcher

I want to search a predefined set of online sources for a specified phrase, store the results and present them back to me

So that my research is simplified and accelerated

Acceptance criteria:

Must

- A phrase, up to a maximum of 255 characters, will be supplied by the user
- At least one source is searched for the specified word or phrase
- Where more than 50 results are returned from any source, only the first 50 results will be processed.
- Results are stored locally in a file
- The file will contain:
    - The phrase being searched for
    - The results returned
    - The passage(s) from the source(s) containing the search phrase and the source URL
- The file can be easily read by a human
- The results will all be in a similar format in the file, even when returned from different sources.
- NFR: The search will take no longer than 10 seconds to complete

Should

- Multiple sources are searched simultaneously
- The file will also contain:
    - The title of the result(s)
    - A snippet and/or abstract of the result(s)

Could

- The file will also contain:
    - The Harvard format reference for the source(s)

Won’t (potential requirements for future development)

- The solution can be run in batch mode from an input file
- The solution will run continuously, looking for new content to be available that matches the search criteria, alerting when new content is found