NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

DEVELOPING A MODULAR FRAMEWORK FOR IMPLEMENTING A SEMANTIC SEARCH ENGINE

by

Brian M. Hawkins

September 2009

Thesis Advisor:	Craig Martell
Second Reader:	Andrew Schein

Approved for public release; distribution is unlimited

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE	*Form Approved OMB No. 0704-0188*

1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** September 2009	**3. REPORT TYPE AND DATES COVERED** Master's Thesis

4. TITLE AND SUBTITLE Developing a Modular Framework for Implementing a Semantic Search Engine

5. FUNDING NUMBERS

6. AUTHOR(S) Brian M. Hawkins

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Monterey, CA 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES)
N/A

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited

12b. DISTRIBUTION CODE
A

13. ABSTRACT (maximum 200 words)

Current methods of information retrieval (IR) are adequate for everyday search needs, but they are not appropriate for many military and industrial tasks. The underlying mechanism of typical search methods is based upon keyword matching, which has demonstrated very poor performance over highly technical requirements documents found within the field of acquisitions. Instead of matching keywords, IR methods that understand the meaning of the words in a query are needed to provide the necessary performance over these types of documents; this is known as semantic search.

This work utilizes sound software engineering practices to specify, design, and develop a modular framework to aid in the design, testing, and development of new semantic search methods and IR techniques, in general. The development of Modular Search Engine framework is documented in its entirety, from user needs analysis to the production of a full application programming interface.

By exploiting the powerful techniques of polymorphism and object-oriented programming in the Java programming language, users are able to design new IR techniques that will function seamlessly within the framework.

Finally, a reference implementation is provided as a proof-of-concept to demonstrate the capabilities and usefulness of the framework design.

14. SUBJECT TERMS Semantic Search, Modular Search Engine, object-oriented programming, Java, UML

15. NUMBER OF PAGES
99

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	**18. SECURITY CLASSIFICATION OF THIS PAGE**	**19. SECURITY CLASSIFICATION OF ABSTRACT**	**20. LIMITATION OF ABSTRACT**
Unclassified	Unclassified	Unclassified	UU

NSN 7540-01-280-5500
Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

DEVELOPING A MODULAR FRAMEWORK FOR IMPLEMENTING A SEMANTIC SEARCH ENGINE

Brian M. Hawkins
Captain, United States Marine Corps
B.S., United States Naval Academy, 2001

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
September 2009

Author: Brian M. Hawkins

Approved by: Craig Martell
Thesis Advisor

Andrew Schein
Second Reader

Peter Denning
Chairman, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Current methods of information retrieval (IR) are adequate for everyday search needs, but they are not appropriate for many military and industrial tasks. The underlying mechanism of typical search methods is based upon keyword matching, which has demonstrated very poor performance over highly technical requirements documents found within the field of acquisitions. Instead of matching keywords, IR methods that understand the meaning of the words in a query are needed to provide the necessary performance over these types of documents; this is known as semantic search.

This work utilizes sound software engineering practices to specify, design, and develop a modular framework to aid in the design, testing, and development of new semantic search methods and IR techniques, in general. The development of Modular Search Engine framework is documented in its entirety, from user needs analysis to the production of a full application programming interface.

By exploiting the powerful techniques of polymorphism and object-oriented programming in the Java programming language, users are able to design new IR techniques that will function seamlessly within the framework.

Finally, a reference implementation is provided as a proof-of-concept to demonstrate the capabilities and usefulness of the framework design.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

I would like to thank the staff and students of the Naval Postgraduate School for their support and assistance. I consider myself fortunate to have been stationed here over the past 27 months, and I am proud to be graduating with men and women of the graduating class of September 2009. Many of you have become friends that I will remember for the remainder of this lifetime. Specifically, I would like to thank:

My thesis advisor, Prof. Craig Martell, for all of the guidance and wisdom, recognizing when I was in over my head, and stepping in to pull me out.

Prof. Kevin Squire for all of the classes that you have taught me and putting up with my wavering thesis interests.

Dr. Andrew Schein for taking the job and becoming my second reader.

David Dreier for being King Nerd and giving me someone to compete with.

Jenny Tam for proof-reading this thesis and reminding me that it is possible to survive without Internet at home.

Johnnie Caver for all of the diverse conversations and discussions that we've had.

Jonathan Durham for being a constant source of comic relief and introducing me to Hulu.

Marco Draeger for being my sounding board on just about everything related to this thesis. Thank you for having patience and always being so polite while I barraged you with questions and took up so much of your time.

Constantine Perepelitsa for diving in and giving yourself a crash course in Java.

Finally, and most importantly, my wife, Deborah, and our children. Thank you for adapting to my constantly changing schedule over the past two years and being a constant source of pride and encouragement. All of my accomplishments are meaningless without you.

Thank you all.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

A. BACKGROUND

For many users' needs, the advent of Google has trivialized the problem of finding relevant documents on the Internet. Prior to Google, the search task was accomplished by performing a simple keyword search, which finds pages that contain the words in the query and rank orders them according to how strongly those words matched. Google's revolution came not by changing the fundamentals, as the pages returned are still those that match the keywords in the query, but instead by changing the order in which the returned pages are presented. Google evaluates the returned pages according to the PageRank algorithm and then presents those pages in order of decreasing PageRank value.

Thus, the innovation behind Google is in the PageRank algorithm. Simply put, the algorithm ranks pages according to sociological importance by observing the number of hyperlinks that point to each page. The more links that point to a particular page, the higher that page is in the "society." Additionally, some pages are given extra authority based upon the number and rank of the pages to which they point. Therefore, if several pages with high authority all refer to a particular page, it will be ranked higher than another page that has only low-ranking pages pointing to it [1]. PageRank is essentially analogous to the stereotypical high-school social popularity status: If you can become associated with a "cool kid," then your social status will be elevated respectively.

B. MOTIVATION

Despite the fact that Google works well for most search tasks, for many military and industrial tasks, *popularity* is not a sufficient metric. Consider a software engineer who is tasked with developing a sophisticated system. He separates his design into subcomponents designed to achieve particular tasks that contribute to the operation of the whole. Before he sets off to start building each subcomponent from scratch, he first searches his company's database to find out if any subcomponent (or part thereof) already exists in order to not duplicate effort.

So, he searches over the database of requirements documents with a particular search query, and if he is extremely lucky, the best component in the database that meets his needs will have been described with the same set of words in his query. Chances are, however, that those particular words were not used to describe the existing component, but rather a different set of words with the exact same meaning. In this case, the search will not return what he needs, regardless of the popularity of the documents returned: If the keywords are incorrect, he will never find the component that he is looking for. He then resorts to altering his set of keywords with synonyms, in hopes of choosing the particular words that were used to describe the relevant system in the database, a particularly time-consuming and frustrating effort.

The problem described above is the semantic search problem, and it is a particular issue in Department of Defense (DoD) acquisitions. In August 2006, Program Executive Officer of Integrated Warfare Systems (PEO-IWS) established the Software Hardware Asset Reuse Enterprise (SHARE) repository to enable the reuse of combat system software and related assets [2]. In order to make effective use of the SHARE repository, the DoD needs an effective solution to the problem of semantic search.

C. OBJECTIVES

The objectives of this thesis are to utilize sound software engineering practices to specify, design, and develop a modular framework for developing, implementing, and testing new semantic search methods and information retrieval (IR) techniques, in general. These objectives shall be accomplished through the following:

- Thorough system specification and design using UML and other software engineering practices.
- Development of a modular, object-oriented Java package whose components can be used to build a fully functional search engine consisting of one or more independent IR modules. The addition of a

single IR module should not incur a large integration effort as measured by the number of classes and methods that need to be implemented. Additionally, the framework will incorporate basic management functionality for use by administrators, such as adding and deleting documents from a corpus.

- Demonstrate the modular framework by developing a reference implementation that consists of at least two IR modules whose results are combined to produce a single list of results to the user.

D. SCOPE

The scope of this thesis focuses on the design of a modular framework that allows multiple IR methods to run simultaneously on a selected corpus of data with each method returning a list of search results. The framework also provides for the development of methods to combine the lists returned from each IR method into a single list that is returned to the user. The scope of this thesis does not include the development of a new method for IR.

E. THESIS ORGANIZATION

Chapter II establishes the system and user requirements necessary to design a comprehensive and modular framework for implementing multiple IR techniques within a single search engine. A detailed use case analysis is performed.

Chapter III formalizes the requirement specifications into an architectural design by decomposing the system into a subset of systems. The use cases from Chapter II are expanded and developed in detail.

Chapter IV describes and demonstrates the functionality of a reference implementation; in addition, this chapter describes an evaluation metric and demonstrates how to apply the measure.

Chapter V contains a summary and recommendations for future work.

The Appendix provides a UML reference key to the figures in Chapters II and III.

THIS PAGE INTENTIONALLY LEFT BLANK

II. VISION DOCUMENT

A. INTRODUCTION

1. Purpose of the Vision Document

This chapter provides the foundation, background, and reference for all future, more detailed, development. Here, the high- level user needs are gathered, analyzed, and defined to identify the required features needed for a fully functional Modular Search Engine.

2. Framework Overview

The Modular Search Engine provides the framework for future design, development, testing, implementation, and deployment of IR methods. Developers need only adhere to the design requirements, inherited via abstract super classes, in order to have a new IR technique integrate seamlessly into the Modular Search Engine.

B. USER DESCRIPTION

1. User Demographics

The primary users of the Modular Search Engine framework are any student or researcher looking to develop and test new methods of IR and/or metasearch. Specifically, Draeger used the Modular Search Engine framework to implement a new semantic search technique to help solve the problems of searching over requirements documents [3].

Additionally, the Modular Search Engine framework can be used to develop fully functional applications for end-users needing to conduct searches over text corpora. Such applications would require administrative control and functionality to update and maintain the corpora.

2. User Profiles

Students and IR researchers at NPS and other academic universities will need to be familiar with the Java programming language in order to use the Modular Search Engine framework.

End-users, for whom applications have been built using the Modular Search Engine framework, need not have any specific knowledge of the interworking of the application. Such users only need basic computer knowledge to launch the application and conduct searches over the corpus for which the application was designed.

3. User Environment

Users of the framework will need a computer system that enables development in the Java programming language. While not mandatory, a developing environment such as Eclipse or NetBeans is recommended. At minimum, users will need a text editor and a current version of the Java SE Development Kit provided by Sun Microsystems in order to write, build, and run their applications.

End-user applications developed using the Modular Search Engine framework can be run on any computer operating system utilizing a current Java Runtime Environment, also provided by Sun Microsystems.

4. Key User Needs

When conducting research in this field, comparing different IR methods against one another to determine the method with the best performance is important. The Modular Search Engine framework provides the architecture and data structures that each IR method must utilize to simplify such comparisons.

One additional and important area of study in the field of IR is known as metasearch. Metasearch is the process of fusing or merging the ranked lists of documents returned from different methods or systems in order to produce a combined list whose quality (as measured via the performance metrics mentioned above) is greater than or equal to any of the lists from which it was created [4]. Given the ability to improve the quality of results returned to the user and the modular nature of the framework, metasearch has been included in the design of the Modular Search Engine from the ground up, and users are provided with the structure in which to build their metasearch techniques.

5. Alternatives

Each student or IR researcher is certainly free to develop, test, and implement new IR techniques without the use of the Modular Search Engine framework. They would, however, be required to spend valuable time implementing the entire infrastructure themselves instead of on the development of the IR method. Additionally, it is highly unlikely that any two IR techniques developed by different authors would work cohesively in the same system without extensive modifications to one or both authors' source code.

C. FRAMEWORK OVERVIEW

1. Framework Perspective

The Modular Search Engine framework's architecture allows multiple IR techniques to run simultaneously on a user's query over a selected corpus of documents. The architecture then combines the results of each into a single ranked list that is returned to the user. The framework is designed such that each IR technique, known within the framework as a Search Module, need not be aware of any other Search Module within the Modular Search Engine.

2. Framework Position Statement

IR researchers can benefit from a common framework in which to develop and test new IR techniques. The Modular Search Engine framework provides all of the necessary overhead and design constraints necessary to streamline design efforts into the development of new IR techniques. Additionally, the framework also provides sufficient structure to develop a fully functional end-user application for searching over given data corpora.

3. Assumptions and Dependencies

The Modular Search Engine framework is written in the Java programming language and applications developed with the framework can be run on any platform with the current Java Runtime Environment installed. The data, over which a Modular Search Engine application may conduct searches, is independent of the framework itself; however, the framework provides the necessary classes in to which the data must be converted for use within the application.

D. FRAMEWORK FEATURES

1. Data Access and Management

a. Document

The basic data element within the Modular Search Engine framework is a document. At a minimum a document consists of a unique identification number, known as a document ID, and a body of text. However, a document may contain much more information e.g., an author, bibliographical information, date written, etc. For this reason, this basic document model will likely need to be extended in order to capture the additional information that may exist.

b. Corpus

A collection of documents that have similar underlying structure comprise a corpus. In the realm of IR research, a corpus is usually a fixed set of documents over which IR techniques are tested and compared against one another. To this end, read access to the data is the minimum capability required to access the data and perform these types of operations. However, all corpora need not remain static. As such, the Modular

Search Engine framework is designed with this in mind and includes the functionality to add and delete documents from a corpus. Such functions are expected to be used by an administrator needing to maintain the data in a given corpus.

2. Resource Access and Management

a. Hard Disk Access

In general, IR techniques do not read through an entire corpus of documents on the hard disk each time they perform a search. Instead, they each create an internal representation of the corpus, called an index, which each uses to conduct searches. Accordingly, each IR technique is expected to store its respective index on the hard disk for subsequent access. This use of hard disk space will save significant amounts of time and resources by preventing each technique from having to re-build its index from the original corpus every time the system is launched.

b. Threading

The Modular Search Engine framework has adopted the principle that no operation performed by any individual IR technique shall be forced to wait on the operations of another IR technique. As such, the framework has been designed to maximize the use of threading, and therefore all operations performed by individual IR techniques shall be run by independent threads.

c. Heap Space

Most IR techniques require large amounts of working memory to function and even more to be efficient at returning quality results to the user in a timely manner. By default the Java Runtime Environment allocates an initial 32 MB to the heap and allows it to grow to a maximum of 128 MB. This, unfortunately, is not likely to be enough memory for the Modular Search Engine framework to perform efficiently, especially as multiple IR techniques are added to a single system. As such, when running a Modular Search Engine application, it is recommended to use the maximum amount of memory that a given computer will allow the Java Runtime Environment to use.

E. USE CASE

Use case scenarios are a critical initial step in determining the requirements of a system by analyzing the scenarios in which actors will interact with a system and how that system should respond to the actors' actions [5]. The use cases identified in this section will become the primary functions of the Modular Search Engine framework and will be developed in detail throughout Chapter III. Figure 1 is the use case diagram for the Modular Search Engine Framework; below the figure, each of the seven use case scenarios is described in detail.

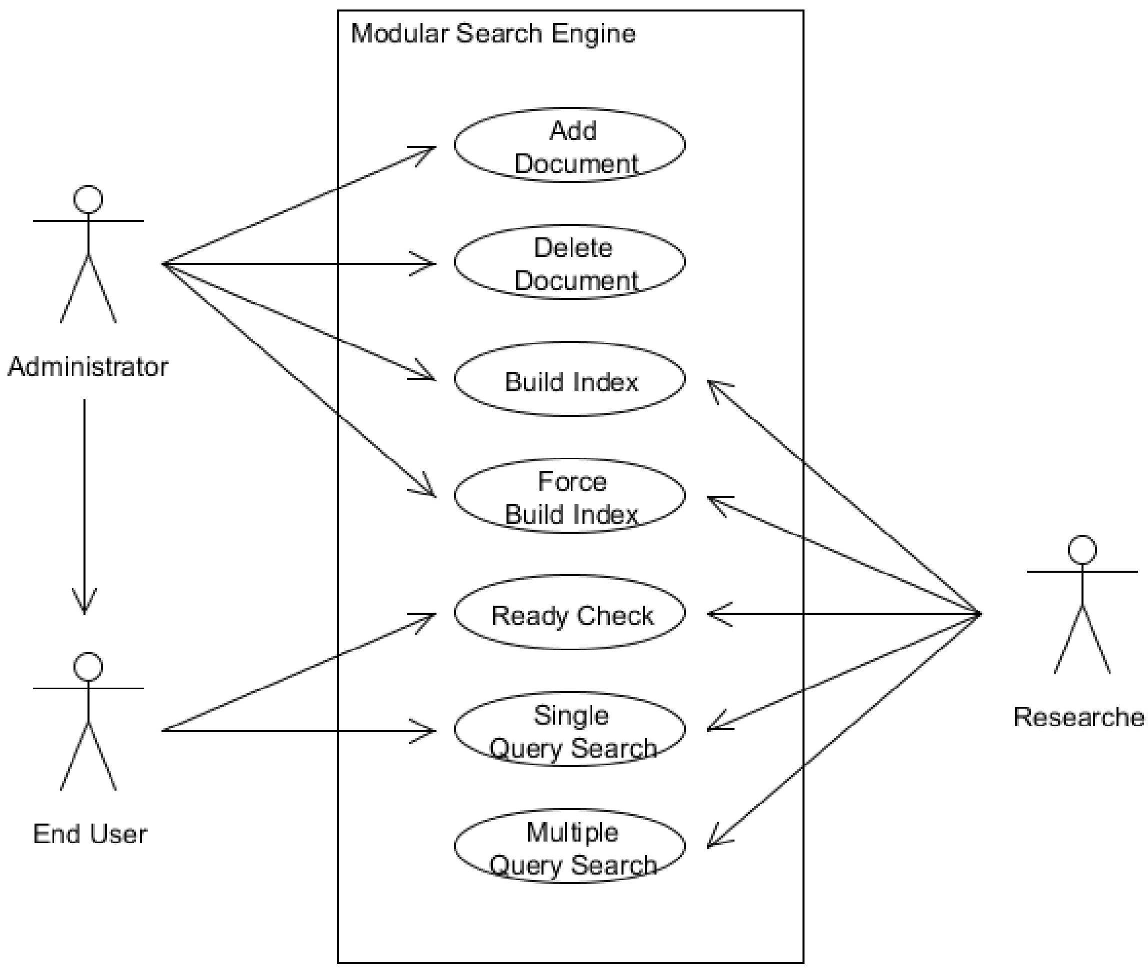


Figure 1. Use Case Diagram

1. Add Document

Use case: UC-1 Add Document

Primary Actor: Administrator

Stakeholders and Interests:

- Administrator wants to add a document into a corpus so the document can be included in search queries by the end-user.

Entry conditions:

- Administrator's application is running.
- The corpus is accessible for writing.
- Document object is created in system memory.

Exit conditions:

- Document successfully added to the corpus in memory and on disk.
- Document successfully added to each IR technique in the system.

Flow of events:

1. Administrator identifies the document to be added.
2. The document is added to the corpus on disk and in memory.
3. The document is added to each IR technique.

Special Considerations:

1. After the addition of a document into a corpus, the index models for each IR technique will need to be updated/re-built.
2. Each IR technique shall return to the system if the document was successfully added.
3. If any IR technique was not successful in adding the document, then the system as a whole is considered to have failed to add the document.
4. If the document fails to be added to the corpus in step 2 of the flow of events, above, then the failure is immediately returned to the system, and attempts to add the document to the system's IR methods are abandoned.

2. Delete Document

Use case: UC-2 Delete Document

Primary Actor: Administrator

Stakeholders and Interests:

- Administrator wants to delete a document from a corpus so that the document is no longer included in search queries by the end-user.

Entry conditions:

- Administrator's application is running.
- The corpus is accessible for writing.
- The document ID of the document to be deleted is known.

Exit conditions:

- Document successfully deleted from the corpus in memory and on disk.
- Document successfully deleted from each IR technique in the system.

Flow of events:

1. Administrator identifies the document to be deleted.
2. The document is deleted from to the corpus on disk and in memory.
3. The document is deleted from each IR technique.

Special Considerations:

1. After the deletion of a document from a corpus, the index models for each IR technique will need to be updated/re-built.
2. Each IR technique shall return to the system if the document was successfully deleted.
3. If any IR technique was not successful in deleting the document, then the system as a whole is considered to have failed to delete the document.
4. If the document fails to be deleted from the corpus in step 2 of the flow of events, above, then the failure is immediately returned to the system, and attempts to delete the document from the system's IR methods are abandoned.

3. Build Index

Use case: UC-3 Build Index

Primary Actors: Administrator & Researcher

Stakeholders and Interests:

- Administrator or researcher wants each IR technique to build its respective index of the system corpus.

Entry conditions:

- Administrator or researcher's application is running.
- The corpus is accessible for reading.

Exit conditions:

- Each IR technique in the system has built its respective index of the corpus

Flow of events:

1. Administrator or researcher provides the necessary instruction to the system.
2. Each IR technique builds its respective index of the corpus.

Special Considerations:

1. This functionality is designed to be optimized at the level of each IR technique so that unnecessary work is not performed. For example, if there has not been a change to the corpus, then there should be no need to build a new index. If an individual search technique is instructed to build a new index in this case, then it should recognize that no actual change has been made and should not spend the computer's resources to build a new index that is identical to the current index.
2. Each IR technique shall return to the system if the index was successfully built.
3. If any IR technique was not successful in building its index, then the system as a whole is considered to have failed the operation.

4. Force Build Index

Use case: UC-4 Force Build Index

Primary Actors: Administrator & Researcher

Stakeholders and Interests:

- Administrator or researcher wants to force each IR technique to build its respective index of the system corpus.

Entry conditions:

- Administrator or researcher's application is running.
- The corpus is accessible for reading.

Exit conditions:

- Each IR technique in the system has forcibly built its respective index of the corpus.

Flow of events:

1. Administrator or researcher provides the necessary instruction to the system.
2. Each IR technique forcibly builds its respective index of the corpus.

Special Considerations:

1. This use case is the complement to UC-3. It is designed to ensure that each IR technique in the system builds a new index of the corpus.
2. Each IR technique shall return to the system if the index was successfully built.
3. If any IR technique was not successful in building its index, then the system as a whole is considered to have failed the operation.

5. Ready Check

Use case: UC-5 Ready Check

Primary Actor: End-user & Researcher

Stakeholders and Interests:

- End-user or researcher wants to en sure that each IR m ethod in the system is ready to receive a search query.

Entry conditions:

- The end-user or researcher's application is running.

Exit conditions:

- Each IR method in the system has returned its ready status.

Flow of events:

1. End-user or researcher requests a ready check of the system.
2. Each individual IR method returns its ready status.

Special Considerations:

1. If any one of the individual IR methods is not ready, then the system's status, as a whole, is returned as not ready.

6. Single Query Search

Use case: UC-6 Single Query Search

Primary Actor: End-user, Researcher

Stakeholders and Interests:

- End-user or researcher wants to perform a single query search of the corpus.

Entry conditions:

- The end-user or researcher's application is running.
- The system is ready as described in UC-5.

Exit conditions:

- The system has returned the results of the single query search.

Flow of events:

1. End-user or researcher submits a single query to the system.
2. Each individual IR technique in the system performs a search using the provided query and returns its results.
3. All of the results returned from the individual IR methods are combined to return a single set of results to the user or researcher.

Special Considerations:

None.

7. Multiple Query Search

Use case: UC-7 Multiple Query Search

Primary Actor: Researcher

Stakeholders and Interests:

- Researcher wants to perform multiple query searches of the corpus.

Entry conditions:

- The researcher's application is running.
- The system is ready as described in UC-5.

Exit conditions:

- The system has returned the results of the multiple query search.

Flow of events:

1. Researcher submits a list of queries to the system.
2. Each individual IR technique in the system performs a search for each of the provided queries and returns results for each.
3. All of the results returned from the individual IR methods are combined to return a single set of results for each query to the researcher.

Special Requirements:

1. This use case is specifically designed to allow for individual IR methods to optimize the simultaneous search of multiple queries in order to preserve system resources.

THIS PAGE INTENTIONALLY LEFT BLANK

III. SYSTEM DESIGN

A. INTRODUCTION

This chapter converts the general analysis model described in Chapter II into a detailed system design. This evolution will begin with a thorough study of the use case models, and it will continue with a decomposition of the system, as a whole, into architectural and behavioral models that will eventually become objects in the design.

B. SYSTEM ARCHITECTURE

1. Goals

The primary goal of the architecture is modularity. Existing IR techniques can be encoded as SearchModule objects and built into a Modular Search Engine application. As new IR techniques are developed, they too can be encoded as SearchModule objects and seamlessly inserted into the existing Modular Search Engine application for testing and further development. As such, the SearchModule class shall be abstract, providing an existing template for extensions to inherit and follow.

In addition to new IR techniques, new methods of conducting metasearch are constantly being researched in the field, and the framework takes this into account as well. It provides researchers with the ability to encode different metasearch methods as ModuleMixer objects that can be interchanged within the system, thus keeping with the goal of modularity.

Figure 2 displays a high level, conceptual, view of the internal architecture within the Modular Search Engine framework.

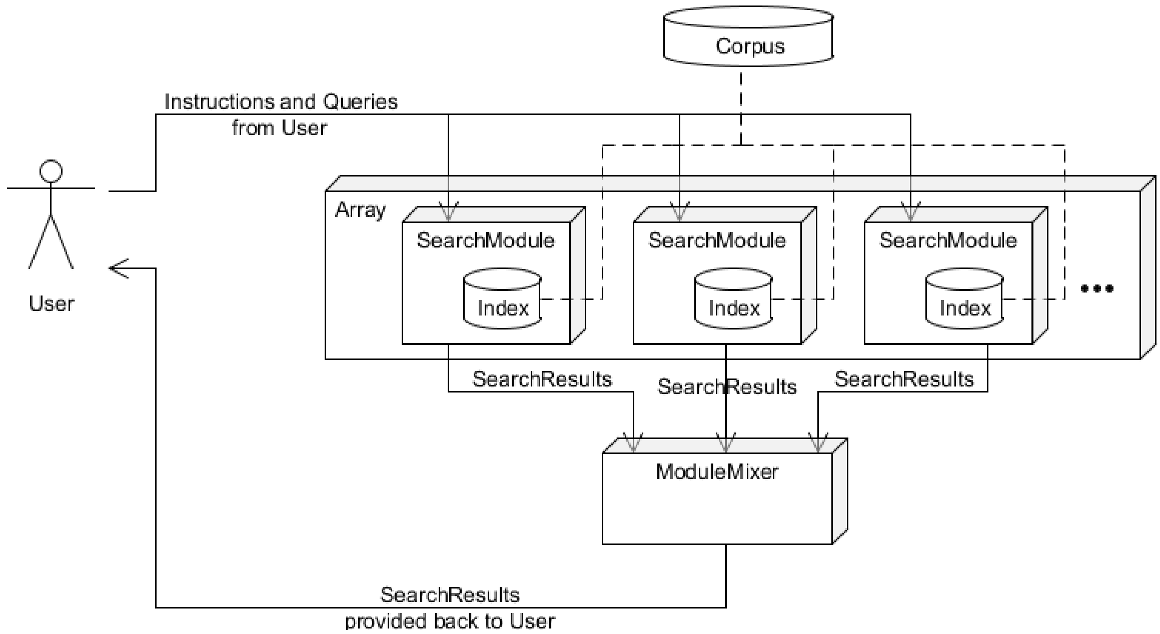


Figure 2. Modular Search Engine Architecture

As each SearchModule object completes a search request, it feeds its results, in the form of a SearchResults object, into a ModuleMixer object that combines multiple SearchResults objects into a single set of results. In general, a Modular Search Engine implementation would only use one ModuleMixer at a time; however, this is not a restriction. In fact, for the purposes of developmental testing and comparison, it may be beneficial to implement multiple ModuleMixer objects simultaneously.

2. Integration

The objects within the framework will communicate with each other by directly calling each other's procedures. However, no integration will take place between SearchModule objects because each is specifically designed to work independently of one another. As such, custom designed extensions of the java.lang.Thread class are used to handle communication both to and from all SearchModule objects for the use cases presented in Chapter II.

C. BEHAVIORAL DESIGN

1. Domain Object Model

The domain object model records the key concepts in the Modular Search Engine framework. Figure 3 depicts the various entities involved and the relationships between them. See Appendix for a key to the figure.

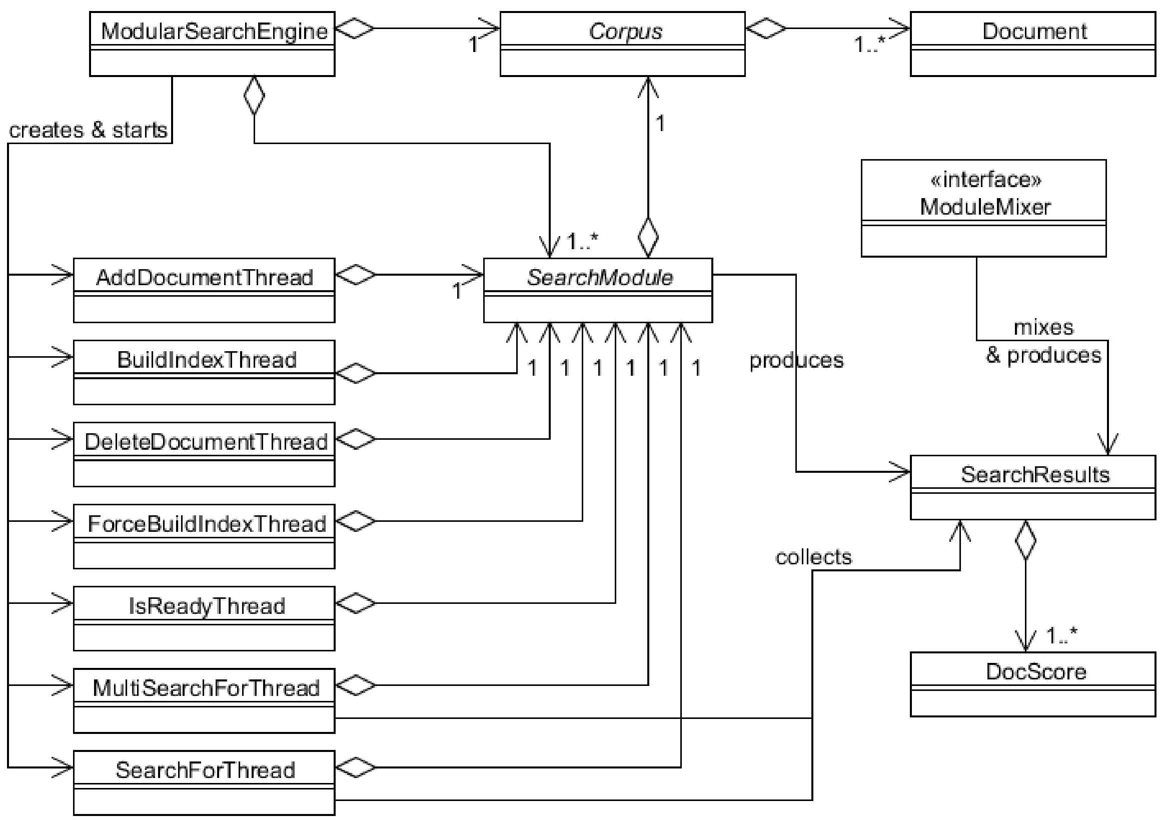


Figure 3. UML Domain Object Model

2. Sequence Diagrams

Sequence diagrams help formalize the dynamic behavior of the system by tying use cases to objects and by showing how processes operate with one another and in what order. Visualizing the communication among objects can help determine additional objects required to formalize the use cases [6]. In this regard, sequence diagrams offer another perspective on the behavioral model and are instrumental in discovering missing objects and grey areas in the requirements specification. The following sequence diagrams depict the use cases identified in Chapter II.

a. Add Document

Figure 4 displays the sequence diagram for adding a document in the Modular Search Engine framework.

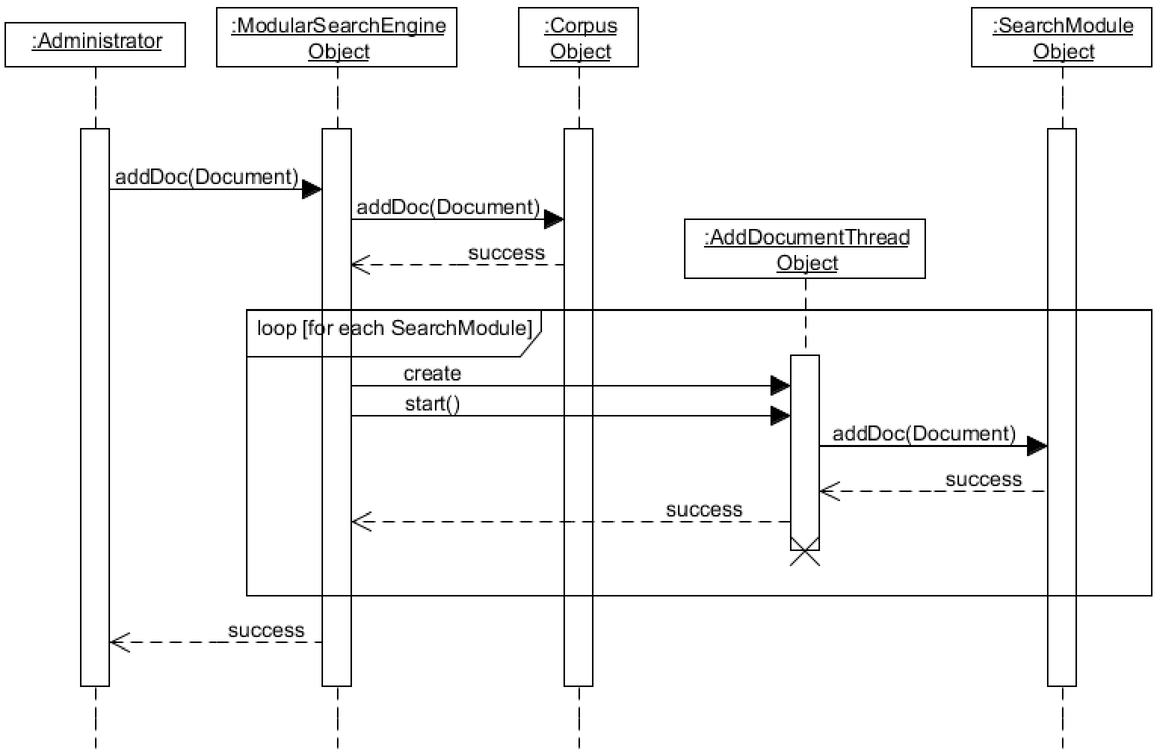


Figure 4. Add Document Sequence Diagram

b. Delete Document

Figure 5 displays the sequence diagram for deleting a document in the Modular Search Engine framework.

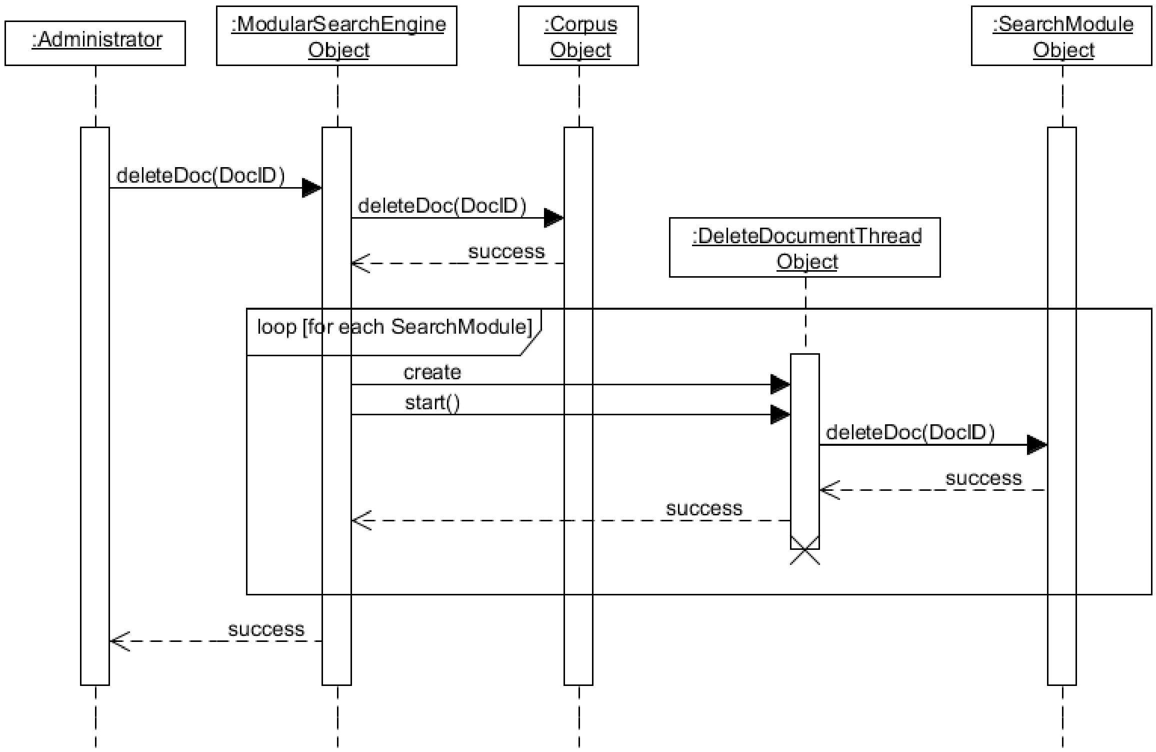


Figure 5. Delete Document Sequence Diagram

c. Build Index

Figure 6 displays the sequence diagra m for building the necessary indices in the Modular Search Engine framework.

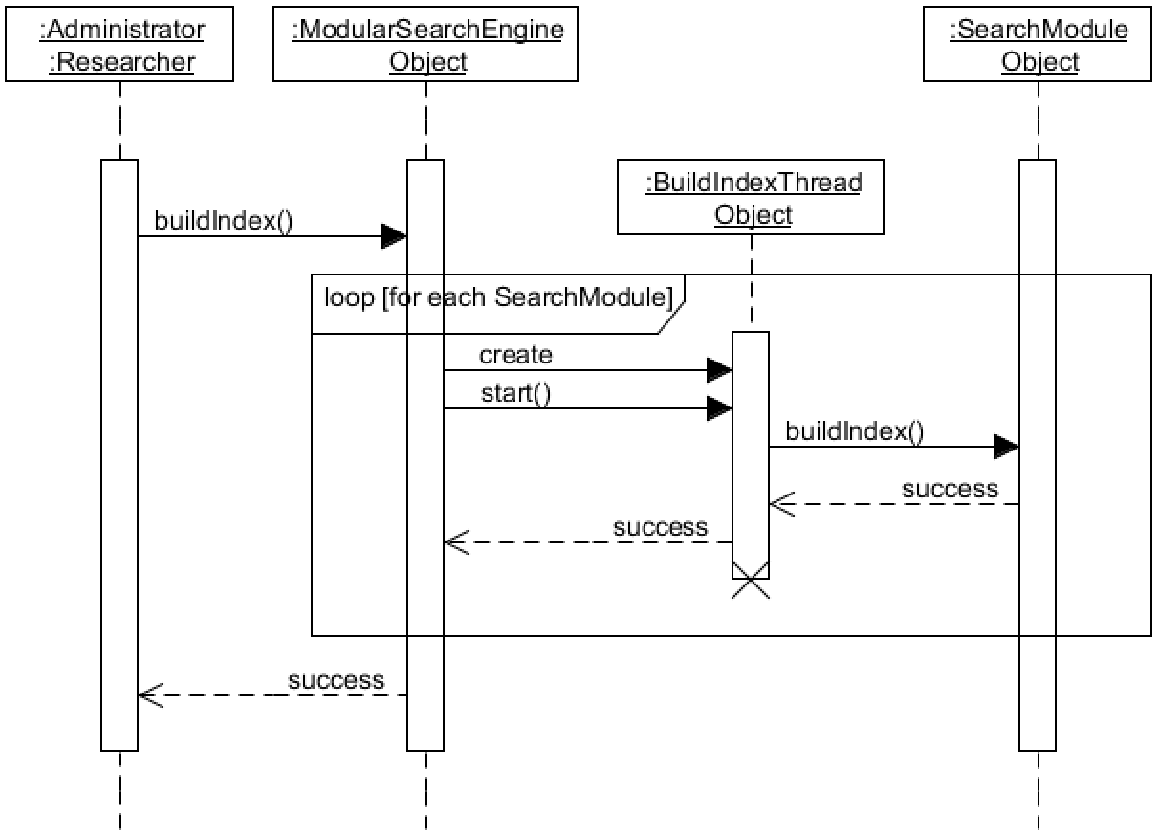


Figure 6. Build Index Sequence Diagram

d. Force Build Index

Figure 7 displays the sequence diagram for forcibly building the necessary indices in the Modular Search Engine framework.

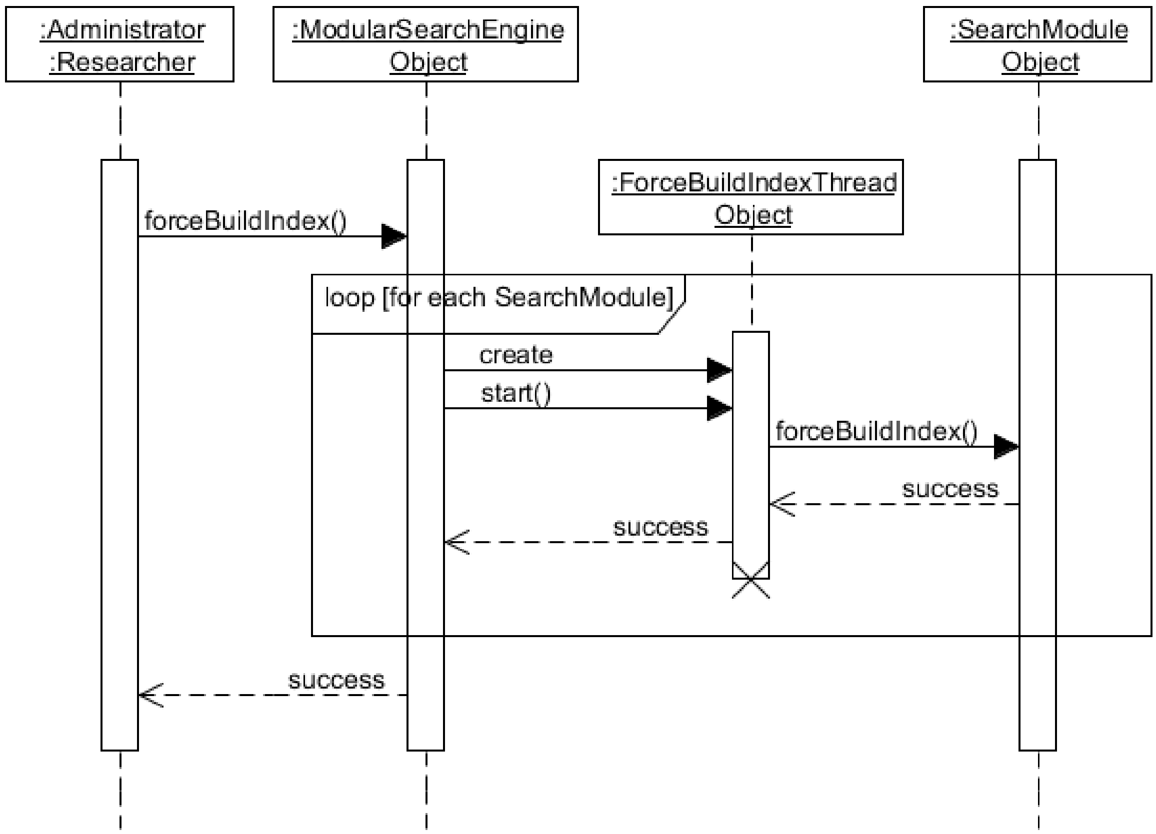


Figure 7. Force Build Index Sequence Diagram

e. Ready Check

Figure 8 displays the sequence diagram for determining that the system is ready to accept a search query in the Modular Search Engine framework.

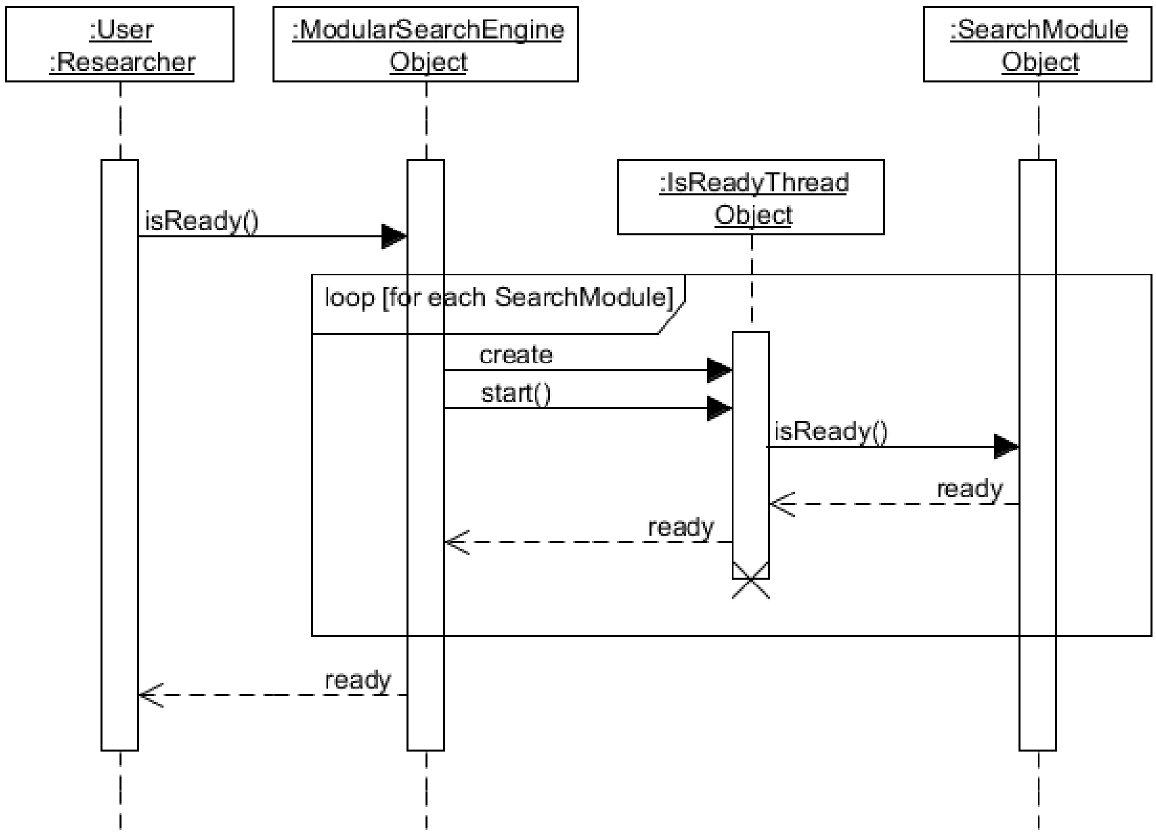


Figure 8. Is Ready Sequence Diagram

f. Single Query Search

Figure 9 displays the sequence diag ram for pe rforming a single query search in the Modular Search Engine framework.

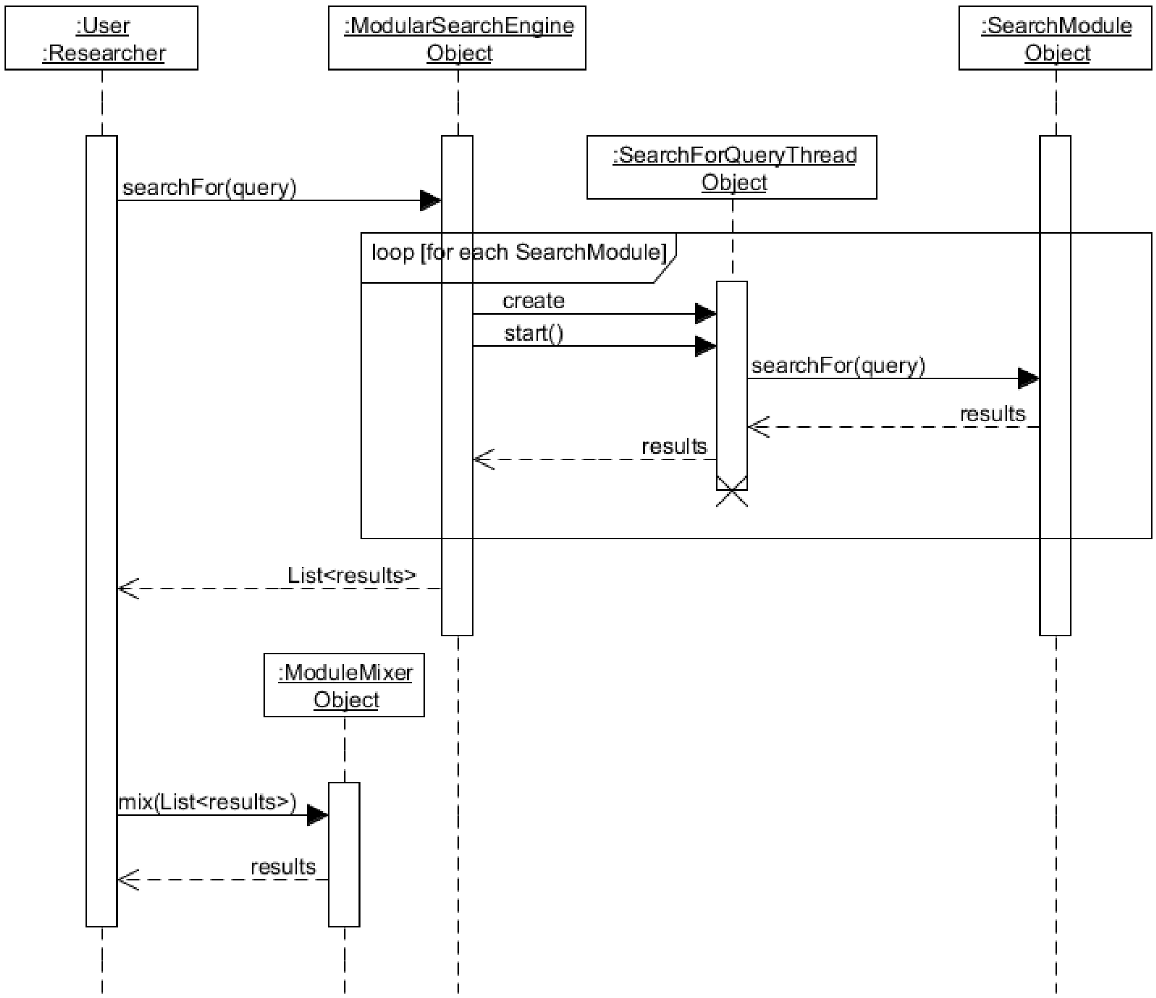


Figure 9. Single Query Search Sequence Diagram

In this case, the user is not norm ally responsible for redirecting the list of results returned from t he ModularSearchEngi ne object into the ModuleMixer object. Instead, this is performed automatically by the user's application.

g. Multiple Query Search

Figure 10 displays the sequence diagram for performing a multiple query search in the Modular Search Engine framework.

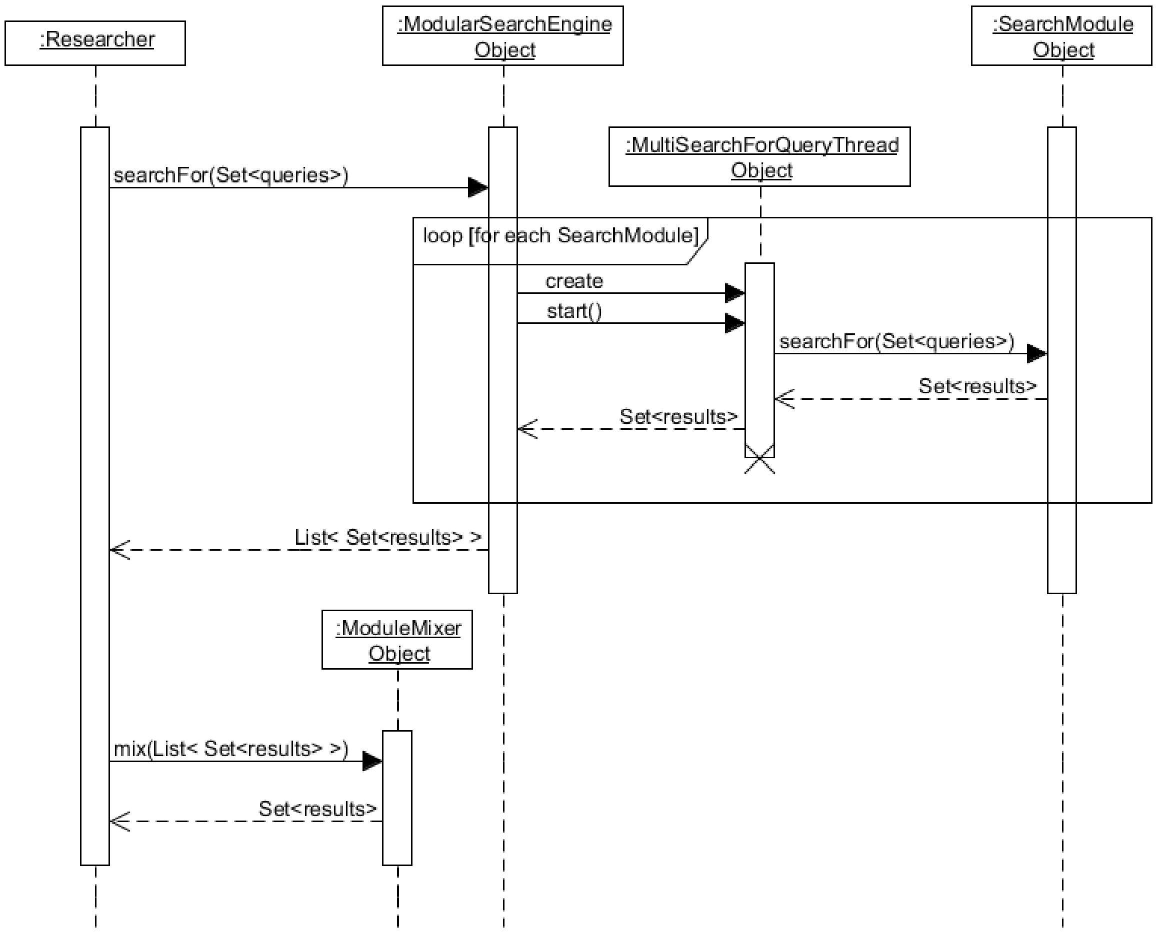


Figure 10. Multiple Query Sequence Diagram

3. Operational Contracts

Operational contracts represent the final phase of the behavioral model design; they are built on the foundations established by the use case specifications, domain object model, and sequence diagrams. These operational contracts assign concrete attributes, such as function names, parameters, and return types, to the framework components and also provide a brief definition of purpose to each. Additionally, the operational contracts precisely define the pre-conditions and post-conditions required for the proposed methods.

a. Add Document

<u>Contract</u>: C1: Add Document

<u>Method</u>: addDocument(Document d)

<u>Cross Reference</u>: UC-1: Add Document

<u>Pre-conditions</u>:

1. The Corpus object was successfully constructed.
2. All of the SearchModule objects were successfully constructed and added to an ArrayList.
3. The ModularSearchEngine object was successfully constructed with the Corpus object and the ArrayList of SearchModule objects listed in pre-conditions 1 and 2 above.
4. The system has completed a successful call to buildIndex() or forceBuildIndex().
5. The Document object to be added was successfully constructed.

<u>Post-conditions</u>:

1. The ModularSearchEngine object constructed and started an AddDocumentThread object for each SearchModule object in the system.
2. Each SearchModule object's addDocument(Document d) method has executed and terminated.
3. A status message was displayed back to the user.

b. Delete Document

<u>Contract</u>: C2: Delete Document

<u>Method</u>: deleteDocument(int docID)

<u>Cross Reference</u>: UC-2: Delete Document

<u>Pre-conditions</u>:

1. The Corpus object was successfully constructed.
2. All of the SearchModule objects were successfully constructed and added to an ArrayList.

3. The ModularSearchEngine object was successfully constructed with the Corpus object and the ArrayList of SearchModule objects listed in pre-conditions 1 and 2 above.
4. The system has completed a successful call to buildIndex() or forceBuildIndex().
5. The unique identification number of the Document object to be deleted is known.

<u>Post-conditions</u>:

1. The ModularSearchEngine object constructed and started a DeleteDocumentThread object for each SearchModule object in the system.
2. Each SearchModule object's deleteDocument(int docID) method has executed and terminated.
3. A status message was displayed back to the user.

c. Build Index

<u>Contract</u>: C3: Build Index

<u>Method</u>: buildIndex()

<u>Cross Reference</u>: UC-3: Build Index

<u>Pre-conditions</u>:

1. The Corpus object was successfully constructed.
2. All of the SearchModule objects were successfully constructed and added to an ArrayList.
3. The ModularSearchEngine object was successfully constructed with the Corpus object and the ArrayList of SearchModule objects listed in pre-conditions 1 and 2 above.

<u>Post-conditions</u>:

1. The ModularSearchEngine object constructed and started a BuildIndexThread object for each SearchModule object in the system.
2. Each SearchModule object's buildIndex() method has executed and terminated.
3. A status message was displayed to the user.

d. Force Build Index

Contract: C4: Force Build Index

Method: forceBuildIndex()

Cross Reference: UC-4: Force Build Index

Pre-conditions:

1. The Corpus object was successfully constructed.
2. All of the SearchModule objects were successfully constructed and added to an ArrayList.
3. The ModularSearchEngine object was successfully constructed with the Corpus object and the ArrayList of SearchModule objects listed in pre-conditions 1 and 2 above.

Post-conditions:

1. The ModularSearchEngine object constructed and started a ForceBuildIndexThread object for each SearchModule object in the system.
2. Each SearchModule object's forceBuildIndex() method has executed, terminated, and returned its success or failure.
3. A status message was displayed to the user.

e. Ready Check

Contract: C5: Ready Check

Method: isReady()

Cross Reference: UC-5: Ready Check

Pre-conditions:

1. The Corpus object was successfully constructed.
2. All of the SearchModule objects were successfully constructed and added to an ArrayList.
3. The ModularSearchEngine object was successfully constructed with the Corpus object and the ArrayList of SearchModule objects listed in pre-conditions 1 and 2 above.
4. The system has completed a successful call to buildIndex() or forceBuildIndex().

Post-conditions:

1. The ModularSearchEngine object constructed and started an IsReadyThread object for each SearchModule object in the system.
2. Each SearchModule object's isReady() method has executed, terminated, and returned its ready status.
3. A status message was displayed to the user.

f. Single Query Search

Contract: C6: Single Query Search

Method: searchFor(String query, int returnSize)

Cross Reference: UC-6: Single Query Search

Pre-conditions:

1. The Corpus object was successfully constructed.
2. All of the SearchModule objects were successfully constructed and added to an ArrayList.
3. The ModularSearchEngine object was successfully constructed with the Corpus object and the ArrayList of SearchModule objects listed in pre-conditions 1 and 2 above.
4. The system has completed a successful call to buildIndex() or forceBuildIndex().
5. The system has completed a successful call to isReady().
6. The user's query is contained within a String object.

Post-conditions:

1. The ModularSearchEngine object constructed and started a SearchForQueryThread object for each SearchModule object in the system.
2. Each SearchModule object's searchFor(String query, int returnSize) method has executed, terminated, and returned a SearchResults object.
3. The ModularSearchEngine object collected and passed all of the returned SearchResults objects from post-condition 1 into a ModuleMixer object via the ModuleMixer's mix(ArrayList<SearchResults>) method.
4. The ModuleMixer method from post-condition 3 returned a single SearchResults object.
5. A status message was displayed to the user.

g. Multiple Query Search

Contract: C7: Multiple Query Search

Method: searchFor(Set<String> queries, int returnSize)

Cross Reference: UC-7: Multiple Query Search

Pre-conditions:

1. The Corpus object was successfully constructed.
2. All of the SearchModule objects were successfully constructed and added to an ArrayList.
3. The ModularSearchEngine object was successfully constructed with the Corpus object and the ArrayList of SearchModule objects listed in pre-conditions 1 and 2 above.
4. The system has completed a successful call to buildIndex() or forceBuildIndex().
5. The system has completed a successful call to isReady().
6. The researcher's batch of queries is contained within a Set<String> object.

Post-conditions:

1. The ModularSearchEngine object constructed and started a MultiSearchForQueryThread object for each SearchModule object in the system.
2. Each SearchModule object's searchFor(Set<String> queries, int returnSize) method has executed, terminated, and returned a Hashtable<String,SearchResults> object.
3. The ModularSearchEngine object collected and passed all of the returned Hashtable<String,SearchResults> objects from post-condition 1 into a ModuleMixer object via the ModuleMixer's mix(Hashtable<String,ArrayList<SearchResults>> tableOfListedResults) method.
4. The ModuleMixer method from post-condition 3 returned a Hashtable<String, SearchResults> object.
5. A status message was displayed to the user.

D. OBJECT DESIGN

The system analysis conducted in the previous sections for the Modular Search Engine framework is critical for identifying the necessary objects that need to exist within the framework and how those objects should interact with one another. This section describes those objects in detail. See Appendix for class diagram reference.

1. Classes

This section describes the non-abstract classes in the framework, with the exception of the Thread classes. The customized extensions of the java.lang.Thread class are described later in this section.

a. ModularSearchEngine

The ModularSearchEngine class is the primary object on which all use cases, sequence diagrams, and operational contracts focus; it is the central object in any application developed from the framework. Figure 11 is the UML class model for the ModularSearchEngine class.

ModularSearchEngine
-corpus : Corpus -modules : ArrayList<SearchModule>
+addDocument(id : int, doc : Document) : boolean +buildIndex() : boolean +deleteDocument(id : int) : boolean +forceBuildIndex() : boolean +isReady() : boolean +nextID() : Integer +searchFor(query : String, returnSize : int) : ArrayList<SearchResults> +searchFor(queries : Set<String>, returnSize : int) : Hashtable<String, ArrayList<SearchResults> >

Figure 11. UML ModularSearchEngine Class Model

(1) Attributes

<u>Corpus corpus</u>: This private variable is the Corpus on which the ModularSearchEngine performs its operations.

<u>ArrayList<SearchModule> modules</u>: This private variable is the container for all of the SearchModules in the system.

(2) Methods

<u>boolean addDocument(Document)</u>: This public method is the interface through which a Document is added to the system. During this method's execution, the provided Document is first added the Corpus via its *addDoc* method. If adding the Document to the Corpus is not successful, this method prints an error, returns

false, and terminates. Otherwise, this method continues, creating and starting an AddDocumentThread for each SearchModule in the system. Each AddDocumentThread is responsible for calling the *addDoc* method of the SearchModule to which it is assigned. As those *addDoc* methods terminate, each AddDocumentThread returns whether or not its *addDoc* method was successful, and this method prints an appropriate message reflecting that success or failure. Once all of the AddDocumentThreads have terminated, if there were any failures, then this method displays an error message, returns *false*, and terminates. If there were no failures, then this method displays an appropriate message, returns *true*, and terminates.

boolean deleteDocument(int): This public method is the interface through which Documents are deleted from the system; the provided integer corresponds to the unique identification number of the document to be deleted. The indicated Document is first deleted from the Corpus via its *deleteDoc* method. If deleting the document from the Corpus is not successful, this method prints an error, returns *false*, and terminates. Otherwise, this method continues, creating and starting a DeleteDocumentThread for each SearchModule in the system. Each DeleteDocumentThread is responsible for calling the *deleteDoc* method of the SearchModule to which it is assigned. As those *deleteDoc* methods terminate, each DeleteDocumentThread returns whether or not its *deleteDoc* method was successful, and this method prints an appropriate message reflecting that success or failure. Once all of the DeleteDocumentThreads have terminated, if there were any failures, this method displays an error message, returns *false*, and terminates. If there were no failures, then this method displays an appropriate message, returns *true*, and terminates.

boolean buildIndex(): This public method is the interface through which a user ensures that an appropriate index is built for each Search Module. It first creates and starts a BuildIndexThread for each Search Module in the system, each of which is responsible for calling the *buildIndex* method of the SearchModule to which it is assigned. As those *buildIndex* methods terminate, each BuildIndexThread returns whether or not its *buildIndex* method was successful, and this method prints an appropriate message reflecting that success or failure. Once all of the BuildIndexThreads have terminated, if there were any failures, this method displays an error message, returns *false*, and terminates. If there were no failures, then this method displays an appropriate message, returns *true*, and terminates. This method allows each SearchModule the opportunity to optimize its *buildIndex* method so that, if possible, a new index might be built upon an existing one. This would allow the system to save resources, instead of building a new index directly from the Corpus each time.

boolean forceBuildIndex(): This public method is the interface through which a user forces each SearchModule to build a new index directly from the Corpus. It first creates and starts a ForceBuildIndexThread for each SearchModule in the system, each of which is responsible for calling the *forceBuildIndex* method of the SearchModule to which it is assigned. As those *forceBuildIndex* methods terminate, each ForceBuildIndexThread returns whether or not its *forceBuildIndex* method was successful, and this method prints an appropriate message reflecting that success or failure. Once all of the ForceBuildIndexThread have terminated, if there were any failures, this method displays an error message, returns *false*, and terminates. If there were no failures, then this method displays an appropriate message, returns *true*, and terminates. This method is the complement to the method above, and its primary purpose is to be used when the user suspects that an index has become corrupted on disk. Additionally, it may be used any time that a user has a reason to give the system a "fresh start;" however, a call to this method can be expected to take a significant amount of time to complete.

boolean isReady(): This public method is the interface through which a user determines if the system is ready to receive a search query. It first creates and starts a IsReadyThread for each SearchModule in the system, each of which is responsible for calling the *isReady* method of the SearchModule to which it is assigned. As the *isReady* methods terminate, each IsReadyThread returns the status of its *isReady* method, and this method prints an appropriate message reflecting that status. If any of the IsReadyThreads indicated that its SearchModule was not ready, then this method displays an error message, returns *false*, and terminates. If all of the SearchModules are ready, then this method displays an appropriate message, returns *true*, and terminates.

Integer nextID(): This public method is a utility to be used while creating new Documents because each Document is required to have a unique identification number, as shown later in this chapter. This method provides the user with the next available integer that can be assigned to a new Document for entry into the Corpus and each SearchModule. Specifically, it calls and returns the value from the Corpus' protected *nextID* method which is also shown later in the chapter.

ArrayList<SearchResults> searchFor(String, int): This public method is primary interface for conducting a search of the Corpus. The parameters to the method are the query String and an integer that indicates the number of results to return, e.g. if the provided integer is 100, then the each SearchModule returns the top 100 Documents that match the search query. If the provided integer is greater than the number of Documents in the Corpus, it is treated as if the user requested the results for all Documents. This method first creates and starts a SearchForThread for each SearchModule in the system, each of which is responsible for calling the appropriate *searchFor* method of the SearchModule to which it is assigned. As those *searchFor* methods terminate and return SearchResults, each SearchForThread returns those SearchResults. All of the SearchResults are collected into an ArrayList and then returned by this method.

Hashtable<String,ArrayList<SearchResults>> searchFor(Set<String>, int): This public method is the primary interface that an IR researcher uses conduct batch query searches. This method allows researchers and developers to take advantage of the way that a SearchModule computes the relevance of a document and optimize it, if possible, for performing multiple search queries simultaneously. The parameters to the method are a Set of query Strings and an integer that indicates the number of results that should be returned in the SearchResults. This method first creates and starts a MultiSearchForThread for each SearchModule in the system, each of which is responsible for calling the appropriate *searchFor* method of the SearchModule to which it is assigned. Those *searchFor* methods terminate and return a Hashtable of SearchResults which are indexed by the String used to produce them. Each MultiSearchForThread returns that Hashtable accordingly, after which all of the

Hashtables are broken down to produce a single Hashtable of ArrayLists of SearchResults such that the index of the Hashtable is the String which generated the list of results.

b. Document

The essence of conducting a search is to find documents that are relevant to the provided query, and as such, the Document class is the basic element in the Modular Search Engine framework. However, the provided class implementation represents only the minimum amount of information necessary to comprise the concept of a document. In many cases, much more information about a given document is available, and, as such, this Document class should be extended to include that additional information as required. Figure 12 is the UML class model for the Document class.

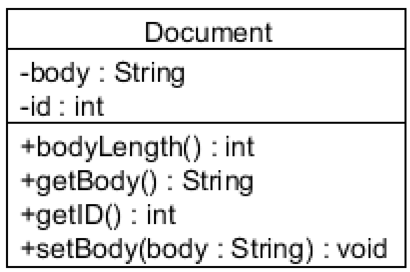


Figure 12. UML Document Class Model

(1) Attributes

String body: This private variable is the text body of a Document.

int id: This private variable is the unique identification number of a Document; it must be unique amongst all the other Documents in a given Corpus.

(2) Methods

int bodyLength(): This public method allows a user to quickly get the length of the Document's text, without having to get the entire body of the Document.

String getBody(): This public method allows a user to get the entire body of the Document.

int getID(): This public method allows a user to get the unique identification number of a Document.

void setBody(String): This public method allows a user to set the text body of a Document.

c. DocScore

Conceptually, when conducting a search, documents are considered in turn and evaluated for how relevant they are to the provided query. The DocScore class is a customized container class specifically created for the purpose of representing that evaluation. Figure 13 is the UML class model for the DocScore class.

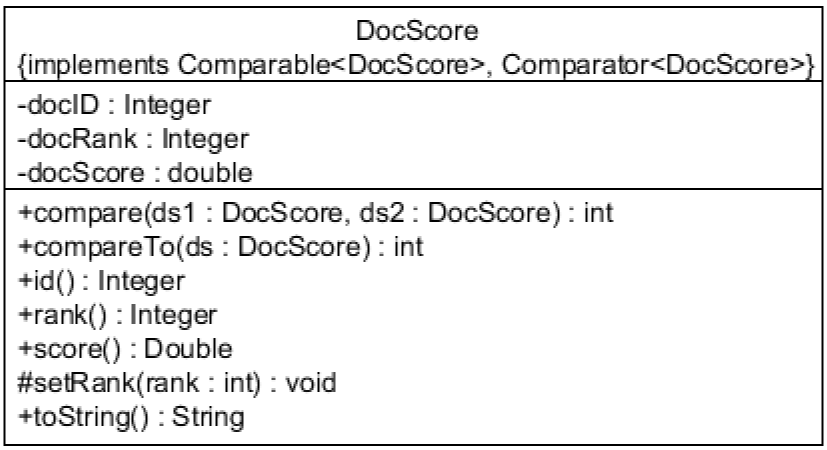


Figure 13. UML DocScore Class Model

(1) Attributes

Integer docID: This private variable is the unique identification number of the Document to which this DocScore refers.

Integer docRank: This private variable is the rank given to the Document.

Integer docScore: This private variable is the score that the Document receives from the evaluation process.

(2) Methods

int compare(DocScore, DocScore): This public method is required by the implementation of the java.lang.Comparator interface. This method assists in the sorting of DocScores. When two DocScores are compared with this method, it will return a positive integer if the first has a better score (ranked higher) than the second.

int compareTo(DocScore): This public method is required by the implementation of the java.lang.Comparable interface. This method assists in the sorting of DocScores and functions in the same manner as described above

Integer id(): This public method allows a user to get the unique identification number of the Document to which this DocScore refers.

Integer rank(): This public method allows a user to get the rank contained within the DocScore.

Double score(): This public method allows a user to get the score contained within the DocScore.

void setRank(int): This protected method allows a user to set the rank contained within the DocScore.

String toString(): This public method allows a user to get a String representation of the DocScore for display purposes.

d. SearchResults

The DocScore class above, for all practical purposes, cannot exist alone because the information contained within a single DocScore is useless without other DocScores to compare against. As such, the SearchResults class has been created as a custom container class designed to hold all of the DocScores generated from a single search query. Figure 14 is the UML class model for the SearchResults class.

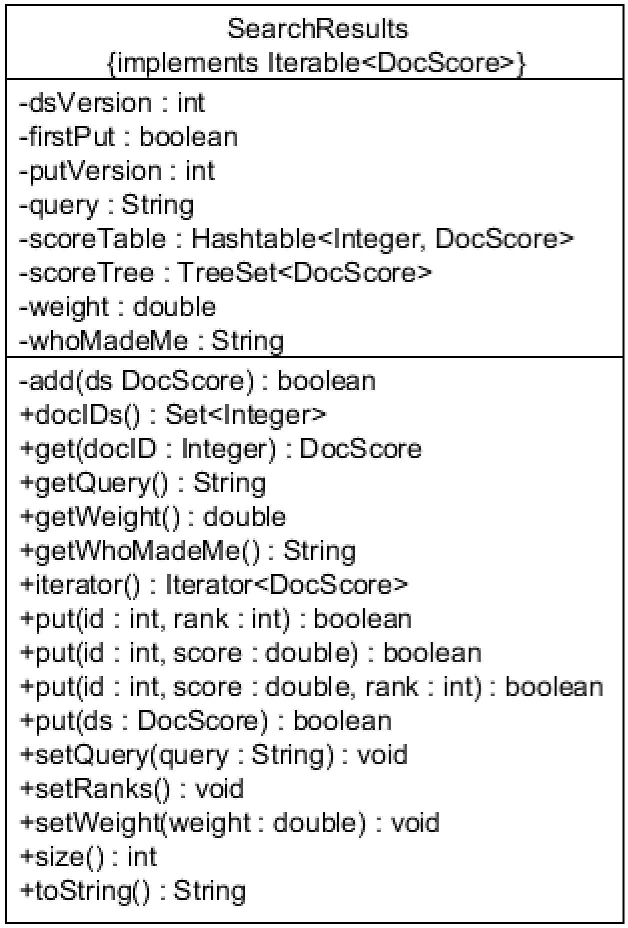


Figure 14. UML SearchResults Class Model

(1) Attributes

<u>int dsVersion</u>: This private variable ensures that all of the DocScores contained within the SearchResults are formatted the same. For example, the user is prohibited from placing a DocScore consisting of a docID and docScore into a set of SearchResults that already contains DocScores with docID and docRank.

<u>boolean firstPut</u>: This private variable is used for internal record-keeping in conjunction with the dsVersion attribute above.

<u>int putVersion</u>: This private variable is used for internal record-keeping in conjunction with the dsVersion and firstPut attributes above.

String query: This private variable is query String that produces this SearchResults.

Hashtable<Integer, DocScore> scoreTable: This private variable is one of two internal containers that hold DocScores. It allows quick access to a DocScore that is associated with a particular Document.

TreeSet<DocScore> scoreTree: This private variable is the second internal container that holds DocScores. It allows for the quick ordered retrieval of all the DocScores contained within because the DocScores are stored in sorted order according to the *compareTo* method described above.

double weight: This private variable assigns a weight to the SearchResults for the purpose of weighting different sets of results against one another.

String whoMadeMe: This private variable stores the unique String name of the object that created the SearchResults. This variable is the only way that the set of SearchResults is tied to the SearchModule or ModuleMixer that created it.

(2) Methods

boolean add(DocScore): This private method is a utility method used by the *put* methods described below.

Set<Integer> docIDs(): This public method allows a user to get all of the Document identification numbers contained within the SearchResults.

DocScore get(Integer): This public method allows a user to get the DocScore for the Document whose unique identification number corresponds to the provided integer. The null value is returned if the indicated Document does not exist in the SearchResults.

String getQuery(): This public method allows a user to get the String query that was used to generate the SearchResults.

double getWeight(): This public method allows a user to get the weight of the SearchResults.

String getWhoMadeMe(): This public method allows a user to get the name of the object that created the SearchResults.

Iterator<DocScore> iterator(): Implementing the java.lang.Iterable interface requires the definition of this public method. Calling this method returns an Iterator over all of the DocScores in the SearchResults. This function allows a user to easily create a programming loop to iterate through the results via the for-each loop construct.

boolean put(int, int): This public method is one of four that allows a user to create an entry in the SearchResults. The first parameter corresponds to the unique identification number of the Document to which the result pertains; the second corresponds to the rank of that Document when compared to the rest of the Documents. This method creates a DocScore with the provided parameters and then calls the private *add* method to store the DocScore in the SearchResults.

boolean put(int, double): This public method is the second of four that allows a user to create an entry in the SearchResults. The first parameter corresponds to the unique identification number of the Document to which the result pertains; the second corresponds to the score that the Document received from the method or object that evaluated it. This method creates a DocScore with the provided parameters and then calls the private *add* method to store the DocScore in the SearchResults.

boolean put(int, double, int): This public method is the third of four that allows a user to create an entry in the SearchResults; it is a combination of the two put methods above. The first parameter corresponds to the unique identification number of the Document to which the result pertains; the second corresponds to the score that the Document received from the method or object that evaluated it; the third corresponds to the rank of that Document when compared to the rest of the Documents. This method creates a DocScore with the provided parameters and then calls the private *add* method to store the DocScore in the SearchResults.

boolean put(DocScore): This public method is the last of four that allows a user to create an entry in the SearchResults. The user can choose to create a DocScore directly and then use this method which will call the private *add* method to store the DocScore in the SearchResults.

void setQuery(String): This public method allows a user to set the query attribute that was used to create this SearchResults.

void setRanks(): This public method allows a user to automatically set the ranks of all the DocScores contained within the SearchResults. This method is only applicable if the DocScores do not already have assigned ranks. DocScores are sorted according to their score attribute and assigned a rank, accordingly, such that the DocScore with the highest score is assigned a rank of one.

void setWeight(double): This public method allows a user to set the weight attribute of the SearchResults for later use when comparing SearchResults against one another.

2. Abstract Classes

Abstract classes are classes that cannot be instantiated; they must be extended into a non-abstract child class in order to gain this capability. Below are the two abstract classes in the Modular Search Engine framework.

a. Corpus

In the field of IR, a collection of documents that have similar structure is a corpus. As such, the abstract Corpus class has been developed for the Modular Search Engine framework. It is abstract because corpora vary greatly from one another, the details of which this author does not presume to know. Therefore, it is up to the user to extend this abstract class and conform it to the preexisting structure of a select corpus. All of the methods in the abstract Corpus class are also abstract and must be implemented to allow the functionality described below. Figure 15 is the UML class model for the abstract Corpus class.

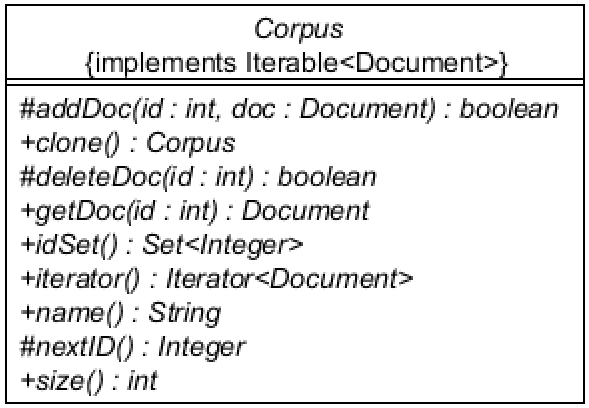


Figure 15. UML Corpus Class Model

(1) Attributes

None.

(2) Methods

boolean addDoc(Document): This protected abstract method allows a user to add a Document to the Corpus.

Corpus clone(): This public abstract method allows a user to get a deep copy of the Corpus.

boolean deleteDoc(int): This protected abstract method allows a user to delete a Document from the Corpus.

Document getDoc(int): This public abstract method allows a user to retrieve the Document who's unique identification number matches the provided integer.

Set<Integer> idSet(): This public abstract method allows a user to get all of the Document identification numbers contained within the Corpus.

Iterator<Document> iterator(): Implementing the java.lang.Iterable interface requires the definition of this public method. Calling this method returns an Iterator over all of the Documents in the Corpus. This function allows the user to easily create a programming loop to iterate through the Documents via the for-each loop construct.

String name(): This public abstract method allows the user to get the name of the Corpus. Each child extended from this abstract parent class should have a unique String returned by this function so that the Corpus can be identified at runtime.

Integer nextID(): This protected abstract method allows a user to get the next available identification number that can be used to put a new Document into the Corpus.

int size(): This public abstract method allows a user to get the number of Documents in the Corpus.

b. SearchModule

The heart of any search engine is the unique method with which it performs its primary function: to search. The goal behind the Modular Search Engine framework is to implement multiple different IR techniques simultaneously within a single search engine. As such, the abstract SearchModule class is the heart of the Modular Search Engine framework. Users are able to extend this abstract class and implement existing and new IR techniques that will integrate seamlessly with each other within the framework. Figure 16 is the UML class model for the abstract SearchModule class.

```
SearchModule
--------------------------------------------------------------
#corpus : Corpus
--------------------------------------------------------------
+addDocument(id : int, doc Document) : boolean
+buildIndex() : boolean
+deleteDocument(id : int): boolean
+forceBuildIndex() : boolean
+isReady() : boolean
+name() : String
+searchFor(query : String, returnSize : int) : SearchResults
+searchFor(queries : Set<String>, returnSize : int) : Hashtable<String, SearchResults>
```

Figure 16. UML SearchModule Class Model

(1) Attributes

Corpus corpus: This protected variable is the Corpus on which the SearchModule performs its operations.

(2) Methods

boolean addDocument(Document): This public method allows a user to add a Document to the SearchModule.

boolean deleteDocument(int): This public method allows a user to delete Documents from the SearchModule.

boolean buildIndex(): This public method allows the user to ensure that an appropriate index is built for the SearchModule. This method allows a SearchModule the opportunity to optimize its *buildIndex* method so that, if possible, a new index might be built upon an existing one. This allows the system to save resources, instead of building a new index directly from the Corpus each time.

boolean forceBuildIndex(): This public method allows a user to forcibly direct the SearchModule to build a new index directly from the Corpus. This method is the complement to the method above; it is used when the user suspects that an index has become corrupted. A call to this method can be expected to take a significant amount of time to complete.

boolean isReady(): This public method is the interface through which a user determines if the SearchModule is ready to receive a search query.

String name(): This public method allows the user to get the name of the SearchModule. Each child extended from this abstract parent class should have a unique String returned by this function so that the SearchModule can be differentiated from other SearchModules at runtime.

SearchResults searchFor(String, int): This public method is primary interface for conducting a search with the SearchModule. The parameters to the method are the query String and an integer that indicates the number of results to return, e.g., if the provided integer is 100, then the each SearchModule should return the top 100 Documents that match my search query. If the provided integer is greater than the number of Documents in the Corpus, it is treated as if the user requested the results for all Documents.

Hashtable<String, SearchResults> searchFor(Set<String>, int): This public method is the primary interface through which an IR researcher conducts batch query searches. This method allows researchers and developers to take advantage of the way in which the SearchModule computes the relevance of a document and optimize it, if possible, for performing multiple search queries simultaneously. The parameters to the method are a Set of query Strings and an integer that indicates the number of results that should be returned in each SearchResults.

3. Interface

Like an abstract class, an interface cannot be instantiated on its own. An interface must be implemented by the user, and that implementation must adhere to the structure defined in the interface. The Modular Search Engine framework contains a single interface, detailed below.

a. ModuleMixer

In the field of IR, metasearch is the process of combining multiple ranked lists of documents to produce a single list that is better than any one of the lists that generated it. Since the Modular Search Engine framework is designed to work with multiple IR methods simultaneously, integrating metasearch into the framework is essential in the design. Implementing a metasearch technique is accomplished through the ModuleMixer interface.

Figure 17 is the UML model for the ModuleMixer interface.

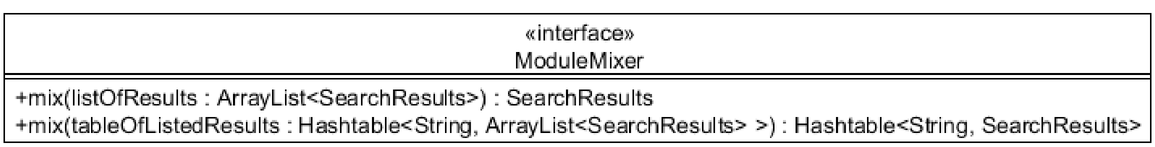


Figure 17. UML ModuleMixer Interface Model

(1) Attributes

None.

(2) Methods

SearchResults mix(ArrayList<SearchResults>): This public method is designed to accompany the single query *searchFor* method. It allows a user to create a single set of SearchResults from the provided ArrayList of SearchResults via the metasearch method implemented by the ModuleMixer.

Hashtable<String, SearchResults> mix(Hashtable<String, ArrayList<SearchResults>>): This public method is designed to accompany the multiple query *searchFor* method. It allows a user to create a single set of SearchResults for each Arraylist of SearchResults in the provided Hashtable via the metasearch method implemented by the ModuleMixer.

4. Threads

The Modular Search Engine framework contains seven class extensions of the java.lang.Thread class. Each is designed to carry out one of the use cases described in Chapter II and is responsible for handling the communication between the ModularSearchEngine and a SearchModule within the system. The details of all seven are described below.

a. AddDocumentThread

Figure 18 is the UML class model for the AddDocumentThread class.

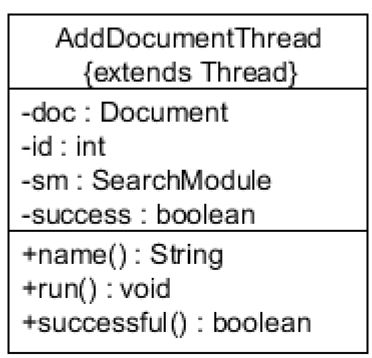


Figure 18. UML AddDocumentThread Class Model

(1) Attributes

<u>Document doc</u>: This private variable is the Document to be added.

<u>int id</u> : This private variable is the unique identifier of the Document to be added.

<u>SearchModule sm</u> : This private variable is the SearchModule whose *addDocument* method will be called by this AddDocumentThread.

<u>boolean success</u>: This private variable holds the returned result of the SearchModule's *addDocument* method.

(2) Methods

<u>String name()</u>: This public method allows a user to obtain the name of the SearchModule that this AddDocumentThread is associated with.

<u>void run()</u> : Extending the java.lang.Thread class requires the definition of this public method. It calls the *addDocument* method of the SearchModule assigned to this AddDocumentThread.

<u>boolean successful()</u>: This public method allows a user to determine if the Document was successfully added to the SearchModule.

b. DeleteDocumentThread

Figure 19 is the UML class model for the DeleteDocumentThread class.

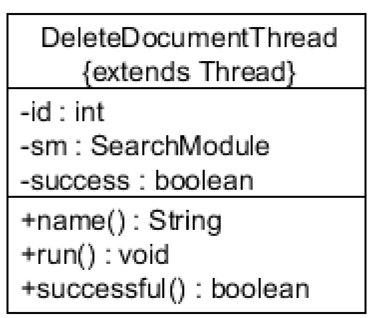


Figure 19. UML DeleteDocumentThread Class Model

(1) Attributes

<u>int id</u>: This private variable is the unique identifier of the Document to be deleted.

<u>SearchModule sm</u>: This private variable is the SearchModule whose *deleteDocument* method will be called by this DeleteDocumentThread.

<u>boolean success</u>: This private variable holds the returned result of the SearchModule's *deleteDocument* method.

(2) Methods

<u>String name()</u>: This public method allows a user to obtain the name of the SearchModule that this DeleteDocumentThread is associated with.

<u>void run()</u>: Extending the java.lang.Thread class requires the definition of this public method. It calls the *deleteDocument* method of the SearchModule assigned to this DeleteDocumentThread.

<u>boolean successful()</u>: This public method allows a user to determine if the Document was successfully deleted from the SearchModule.

c. BuildIndexThread

Figure 20 is the UML class model for the BuildIndexThread class.

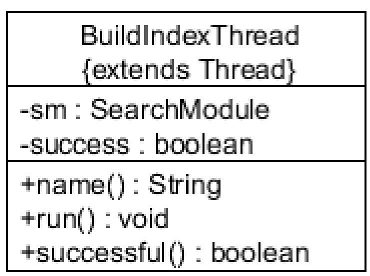


Figure 20. UML BuildIndexThread Class Model

(1) Attributes

<u>SearchModule sm</u>: This private variable is the SearchModule whose *buildIndex* method will be called by this BuildIndexThread.

<u>boolean success</u>: This private variable holds the returned result of the SearchModule's *buildIndex* method.

(2) Methods

<u>String name()</u>: This public method allows a user to obtain the name of the SearchModule that this BuildIndexThread is associated with.

<u>void run()</u>: Extending the java.lang.Thread class requires the definition of this public method. It calls the *buildIndex* method of the SearchModule assigned to this BuildIndexThread.

<u>boolean successful()</u>: This public method allows a user to determine if the SearchModule's *buildIndex* method was successful.

d. ForceBuildIndexThread

Figure 21 is the UML class model for the ForceBuildIndexThread class.

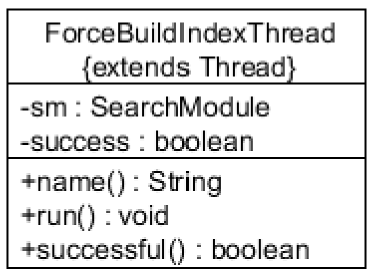


Figure 21. UML ForceBuildIndexThread Class Model

(1) Attributes

<u>SearchModule sm</u>: This private variable is the SearchModule whose *forceBuildIndex* method will be called by this ForceBuildIndexThread.

<u>boolean success</u>: This private variable holds the returned result of the SearchModule's *forceBuildIndex* method.

(2) Methods

<u>String name()</u>: This public method allows a user to obtain the name of the SearchModule that this ForceBuildIndexThread is associated with.

void run(): Extending the java.lang.Thread class requires the definition of this public method. It calls the *forceBuildIndex* method of the SearchModule assigned to this ForceBuildIndexThread.

boolean successful(): This public method allows a user to determine if the SearchModule's *forceBuildIndex* method was successful.

e. IsReadyThread

Figure 22 is the UML class model for the IsReadyThread class.

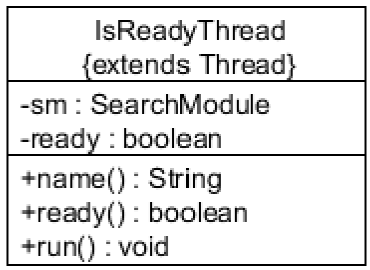


Figure 22. UML IsReadyThread Class Model

(1) Attributes

SearchModule sm: This private variable is the SearchModule whose *isReady* method will be called by this IsReadyThread.

boolean ready: This private variable holds the returned result of the SearchModule's *isReady* method.

(2) Methods

String name(): This public method allows a user to obtain the name of the SearchModule that this IsReadyThread is associated with.

boolean ready(): This public method allows a user to determine if the SearchModule is ready to receive a search query.

void run(): Extending the java.lang.Thread class requires the definition of this public method. It calls the *isReady* method of the SearchModule assigned to this IsReadyThread.

f. SearchForQueryThread

Figure 23 is the UML class model for the SearchForQueryThread class.

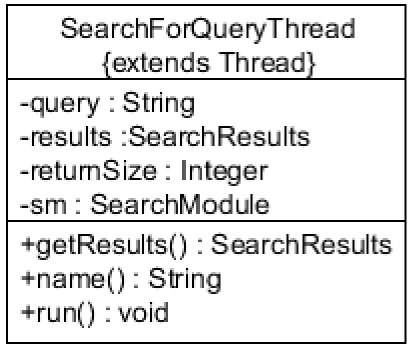


Figure 23. UML SearchForQueryThread Class Model

(1) Attributes

String query: This private variable is String to search for and is passed as a parameter to the SearchModule's *searchFor* method.

SearchResults results: This private variable holds the returned result of the SearchModule's *searchFor* method.

Integer returnSize: This private variable is passed as a parameter to the SearchModule's *searchFor* method to indicate the size of the SearchResults to return.

SearchModule sm: This private variable is the SearchModule whose *searchFor* method will be called by this SearchForQueryThread.

(2) Methods

SearchResults getResults(): This public method allows a user to get the results of the search query.

String name(): This public method allows a user to obtain the name of the SearchModule that this SearchForQueryThread is associated with.

void run() : Extending the java.lang.Thread class requires the definition of this public method. It calls the *searchFor* method of the SearchModule assigned to this SearchForQueryThread.

g. MultiSearchForThread

Figure 24 is the UML class model for the MultiSearchForQueryThread class.

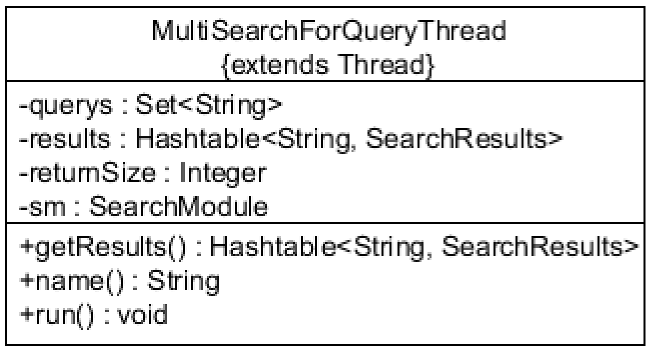


Figure 24. UML MultiSearchForQueryThread Class Model

(1) Attributes

Set<String> queries: This private variable is the Set of Strings to search for and is passed as a parameter to the SearchModule's *searchFor* method.

Hashtable<String, SearchResults> results : This private variable holds the returned result of the SearchModule's *searchFor* method.

Integer returnSize: This private variable is passed as a parameter to the SearchModule's *searchFor* method to indicate the size of the SearchResults to return.

SearchModule sm : This private variable is the SearchModule whose *searchFor* method will be called by this MultiSearchForQueryThread.

(2) Methods

Hashtable<String, Search Results> getResults() : This public method allows a user to get the results of the batch search query.

String name(): This public method allows a user to obtain the name of the SearchModule that this MultiSearchForQueryThread is associated with.

void run(): Extending the java.lang.Thread class requires the definition of this public method. It calls the *searchFor* method of the SearchModule assigned to this MultiSearchForQueryThread.

5. Packages

The Modular Search Engine framework is divided into three primary packages that serve to organize the classes, interfaces, and extensions into logical groups. The packages also serve to ensure that the protected variables are only directly accessible by objects within the same package. The three packages are described below.

a. modularSearchEngine

The modularSearchEngine package consists of the following:

- Corpus—Abstract Class
- Document—Class
- ModularSearchEngine—Class
- ModuleMixer—Interface

b. searchModule

The searchModule package consists of the following:

- DocScore—Class
- SearchModule—Abstract Class
- SearchResults—Class

c. modularSearchEngineThreads

The m odularSearchEngineThreads package consists of the following seven class extensions of java.lang.Thread:

- AddDocumentThread
- BuildIndexThread
- DeleteDocumentThread
- ForceBuildIndexThread
- IsReadyThread
- MultiSearchForQueryThread
- SearchForQueryThread

THIS PAGE INTENTIONALLY LEFT BLANK

IV. REFERENCE IMPLEMENTATION

A. OVERVIEW

As a proof of concept, we have developed a reference implementation to demonstrate the abilities of the Modular Search Engine framework. This chapter describes the internal components of the reference implementation and shows the Graphical User Interface (GUI) designed to provide the user with a simple working environment.

B. EXTENSIONS AND IMPLEMENTATIONS

As described in the previous chapter, several components of the Modular Search Engine framework must be extended or implemented. Specifically, the user must extend the abstract Corpus and SearchModule classes and implement the ModuleMixer interface. The reference implementation contains four child classes of Corpus, two child classes of SearchModule, and two implementation classes of ModuleMixer. These are described below.

1. Corpora

The reference implementation includes four standard benchmark corpora that are used frequently in IR [3]. The corpora were attained from the University of Glasgow's IR Group and are as follows: Cranfield, Medline, CISI, and Time [7]. Each of the four Corpus classes was developed by extending the base Corpus class and adapting it to the specifics of each data set. However, only one is active at a time, as chosen by the user.

2. SearchModules

There are two SearchModules included in this example application; they are individually described below.

a. TF-IDF SearchModule

Term Frequency-Inverse Document Frequency (TF-IDF) is a basic keyword matching technique and is the basis for one of the two SearchModules in the reference implementation. The essentials of TF-IDF are explained below.

One way to represent a document is as a vector of the frequencies of the words contained within it. For example, consider a document whose entirety consists of the following sentence: "The boy fed the dog." The document is five words long, but it only contains four unique words because the word "the" is used twice; we would say that that this document has five *tokens*, but only four *types*. We assign an index to each *type* and count the number of times each appears in the document. Dividing by the sum of the counts (the total number of words in the document) will yield the term frequency for each *type*. The table below shows these values for the example.

Index	Type	Count	Term Frequency
0	the	2	2/5 = 0.4
1	boy	1	1/5 = 0.2
2	fed	1	1/5 = 0.2
3	dog	1	1/5 = 0.2

Table 1. Term Frequency Example Table

We can now generalize the above process. Let $c_{i,j}$ be the count of word *i* in document *j*. We can then calculate $tf_{i,j}$, the term frequency of word *i* in document *j*:

$$tf_{i,j} = \frac{c_{i,j}}{\sum_k c_{k,j}}$$

Now that we have all of the term frequencies in a document, we can represent that document as a single column vector: $tf_j = [\ tf_{1,j}, tf_{2,j}, \ldots, tf_{V,j}\ ]^T$ where *V* is the total number of unique words in our vocabulary.

So far, the above process weights the relevance of a word according to the frequency in which that word appears in a document. This reflects the intuition that the more frequent terms in a document may reflect the meaning of that document better than the terms that appear less frequently and, thus, should have stronger weights [8, 9]. We now turn our attention to the fact that we are dealing with multiple documents that comprise a corpus.

Consider a word that appears in every document in the corpus. This word has little power when trying to identify the relevance of one document over another. Conversely, consider a word that appears in only a single document. The opposite is true because this word carries a lot of importance in identifying this particular document when compared to all the others. Thus, we should weight those words which are common across many documents lower than those which appear in only a few documents [8, 9]. As such, a new measure known as the inverse document frequency (IDF) comes into play. IDF is defined as N / n_i, where N is the total number of documents in the corpus, and n_i is the number of documents in which word i appears. In order to discount the weight of a word that appears in many documents, this measure is applied within a log function resulting in the following definition for the inverse document frequency of word i: [9]

$$idf_i = \log\left(\frac{N}{n_i}\right)$$

If word i appears in every document, then $n_i = N$, and thus $idf_i = \log(1) = 0$. When applied to every word in the vocabulary, this yields an IDF vector with dimension equal to V.

Combining term frequency (TF) with IDF results in the TF-IDF weighting scheme such that the weight of word i in document j is the product of its frequency in j with the log of its inverse document frequency in the corpus: $w_{i,j} = tf_{i,j} * idf_i$ [9]. This yields a matrix with dimension V x N, such that each column in the matrix is the TF-IDF weight vector of a single document. We then use the Euclidian norm on each of these to produce document weight vectors whose lengths are exactly one.

The TD-IDF matrix and the IDF vector together comprise the index of the corpus, and calculating these for a fixed corpus needs only take place once. They can be stored on disk and recalled for subsequent runs of the reference implementation. Up to this point, all of the above calculations have been performed on the corpus, and we now turn the attention to how to conduct a search query using TF-IDF.

First, the query string is converted into a TF vector in the same manner as each document is above. We then calculate the element-wise product of the TF vector and the corpus' IDF vector to produce a new TF-IDF vector for the query. This vector is normalized via the Euclidian norm, and now can be used to determine how relevant each document in the corpus is to the provided query. The TF-IDF SearchModule accomplishes this by computing the cosine similarity (via the dot product of normalized vectors) between the query TF-IDF vector and the TF-IDF vector for each document in the corpus (aka the columns of the matrix.) This is accomplished by a single matrix multiplication: transpose the query TF-IDF column vector into a row vector and multiply it by the TF-IDF matrix of the corpus. The resulting vector contains the scalar cosine similarity measure between each document in the corpus and the provided query. Sorting in descending order according to this measure will yield an ordered list of documents such that the most similar documents are at the top of the list [8-10].

It should be noted that the vector and matrix mathematics implemented in this implementation of TF-IDF is accomplished via the Colt Project, a set of open source java libraries published by the European Organization for Nuclear Research (CERN) [11].

b. Draeger's LDA SearchModule

As mentioned in Chapter II, Draeger used the Modular Search Engine framework to implement a new IR technique to conduct semantic search. During the course of his research, he developed a SearchModule based upon Latent Dirichlet Allocation (LDA) [3].

LDA is a parametric Bayesian model that generates a probability distribution over the *topics* covered in a document, and each *topic* is a distribution over the words in a vocabulary. These *topics* form a latent feature set that describes a document collection better than the words alone. Using this model, it is possible to perform a search by using the words in the query to infer the most likely *topics* associated with that query and then find the documents that cover these same *topics* [3, 12].

As a demonstration of the modularity of the Modular Search Engine framework, we have taken Draeger's LDA SearchModule and incorporated it directly into the reference implementation.

3. ModuleMixers

Two ModuleMixers are included in the reference implementation, however only one ModuleMixer is active for each search, as chosen by the user. The details of each ModuleMixer are described below.

a. Weighted Average Rank ModuleMixer.

This ModuleMixer simply calculates the weighted mean rank for each Document (via a DocScore). For a given document, it uses the weights assigned to each set of SearchResults and computes the weighted mean rank of that document. It then creates a new set of SearchResults whose DocScores are sorted by the new weighted average rank. This set of SearchResults is then returned to the user.

b. Condorcet Fuse ModuleMixer.

This ModuleMixer implements the metasearch technique known as Condorcet-fuse [13]. The inspiration for this technique comes from the field of Social Choice Theory which studies voting algorithms as techniques to make group decisions [14-16]. The Condorcet voting algorithm specifies that the winner of an election is the candidate that beats or ties with every other candidate in a pair-wise comparison [13, 17]. Consider a voting scenario in which ten voters are voting on five candidates in an election, and the voters must rank all five candidates in order of preference. Table 2 depicts one possible outcome of the votes for this scenario [13].

Number of Votes	Candidate Preference (in order)
3	*a*, *b*, *c*, *d*, *e*
3	*e*, *b*, *c*, *a*, *d*
2	*c*, *b*, *a*, *d*, *e*
2	*c*, *d*, *b*, *a*, *e*

Table 2. Example Voting Scenario

In the example, consider a pair-wise comparison of candidates *b* and *c*; six out of the ten voters placed candidate *b* ahead of candidate *c*. In fact, candidate *b* ranks above every other candidate in a pair-wise, head-to-head comparison; therefore, candidate *b* is the Condorcet winner [13].

This is the essence of the Condorcet-fuse metasearch method and the associated ModuleMixer in the reference implementation. Candidates are analogous to Documents, voters to SearchModules, and vote preference to Search Results. The following two pseudo-code algorithms explain exactly how the Condorcet-fuse metasearch method is applied within the Modular Search Engine framework [13].

Algorithm 1: Pair-wise Document Comparison (d_1, d_2)

1: *count* = 0

2: for each SearchModule, *sm*, do

  2a: If *sm* ranks d_1 above d_2, *count*++

  2b: If *sm* ranks d_2 above d_1, *count*--

3: If *count* > 0, rank d_1 better than d_2

4: Otherwise rank d_2 better than d_1

Algorithm 2: Condorcet-fuse

1: Create a list *L* of all the documents

2: Sort(*L*) using Algorithm 1 as the comparison function

3: Output the sorted list of documents as a SearchResults object

C. GRAPHICAL USER INTERFACE

1. Overview

The reference implementation can be divided into five different sections: Query Entry, Corpus Selection, ModuleMixer Selection, Status Display, and Results Display. Figure 25 is a screenshot of the reference implementation GUI and identifies the five basic sections, and each section is described in detail below the figure.

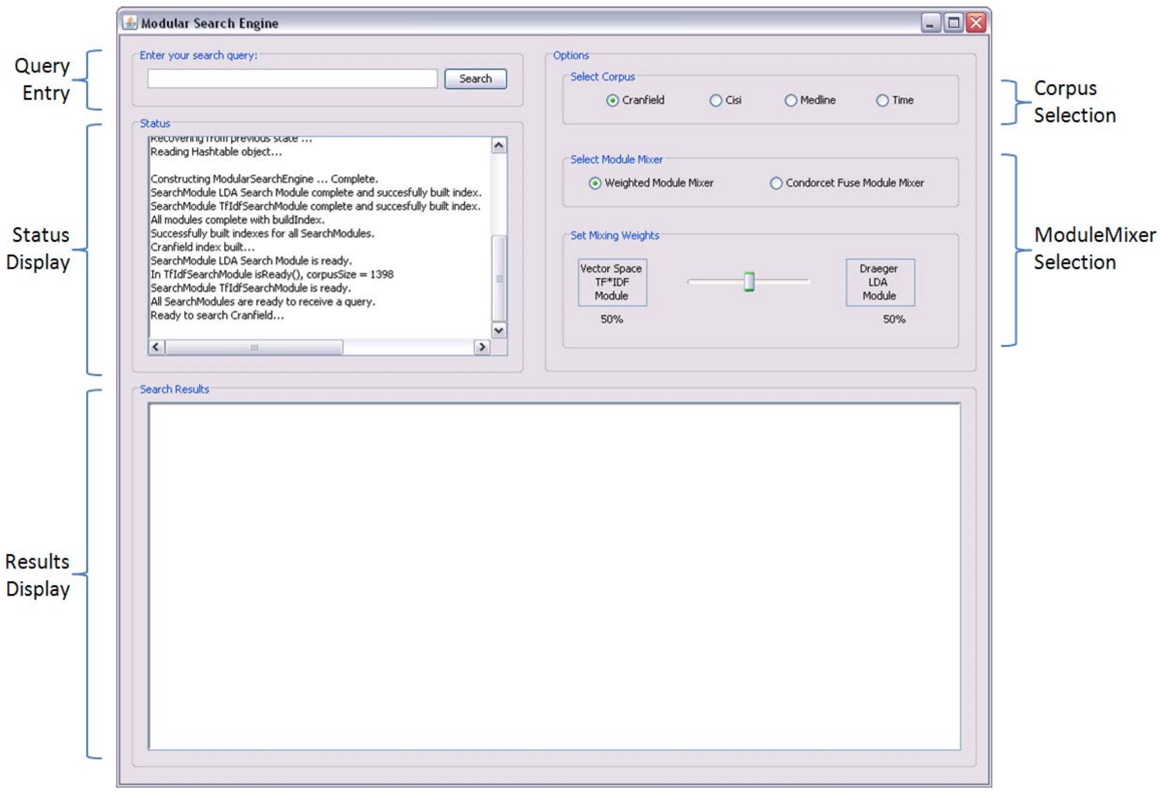


Figure 25. GUI Overview

2. Sections

a. Query Entry Section

As Figure 26 indicates, users enter their search query into the text box; typing <ENTER> or clicking the Search button will begin the search.

Figure 26. Query Entry Section

b. Corpus Selection Section

As previously mentioned, the reference implementation contains four different corpora to choose from. The Corpus Selection Section allows users to choose a corpus via Radio Button as shown in Figure 27. By default, the Cranfield corpus is selected when the application is launched.

Figure 27. Corpus Selection Section

c. ModuleMixer Selection Section

Similar to the Corpus Selection Section above, the user chooses one of two available ModuleMixers via radio button; in the reference implementation the WeightedModuleMixer is selected by default. This ModuleMixer requires additional input from the user via the slider bar. Moving the slider bar adjusts the relative mixing weight assigned to each SearchModule. In Figure 28, the TF-IDF based SearchModule will be weighted three times greater than the other.

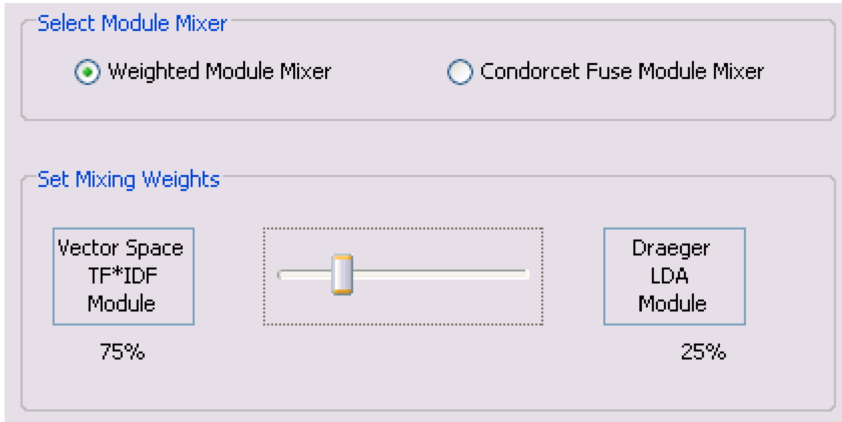


Figure 28. ModuleMixer Selection Section with Weighted Module Mixer Selected

If the CondorcetFuseModuleMixer is selected, the mixing weights are no longer applicable and that sub-section is disabled accordingly as depicted in Figure 29.

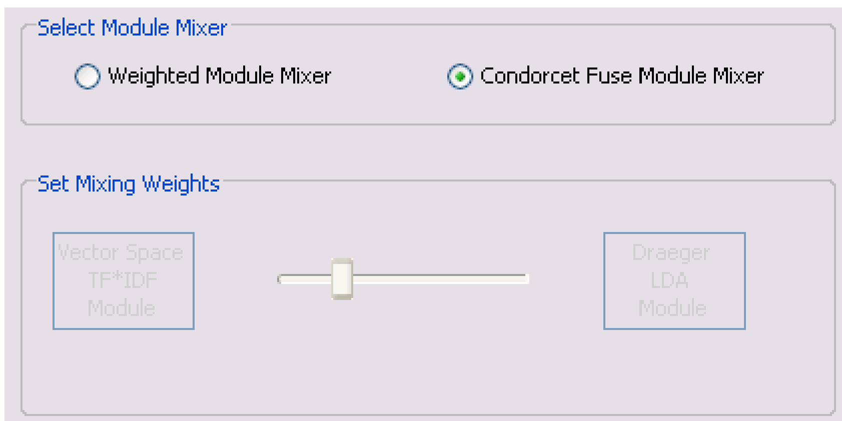


Figure 29. ModuleMixer Selection Section with Condorcet Fuse Module Mixer Selected

d. Status Display Section

When the reference implementation is running, System.out and System.err are redirected to the Status Display as shown in Figure 30 below. This area is scrollable so that a user can view older messages which may have scrolled up and out of view or longer messages that extend to the right of the view.

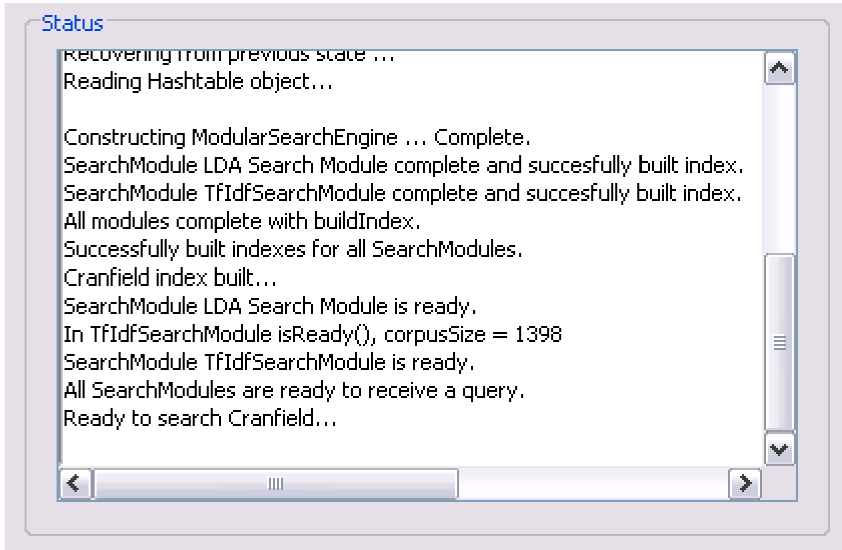


Figure 30. Status Display Section

e. Results Display Section

As the name suggests, the results of the search query are displayed in this section. In this example application, this area is simply populated with text using the *toString()* method of the final SearchResults object produced by the selected ModuleMixer. Figure 31 is an example of what this section looks like after conducting a search. Users can use the scroll bars to view the entire set of results.

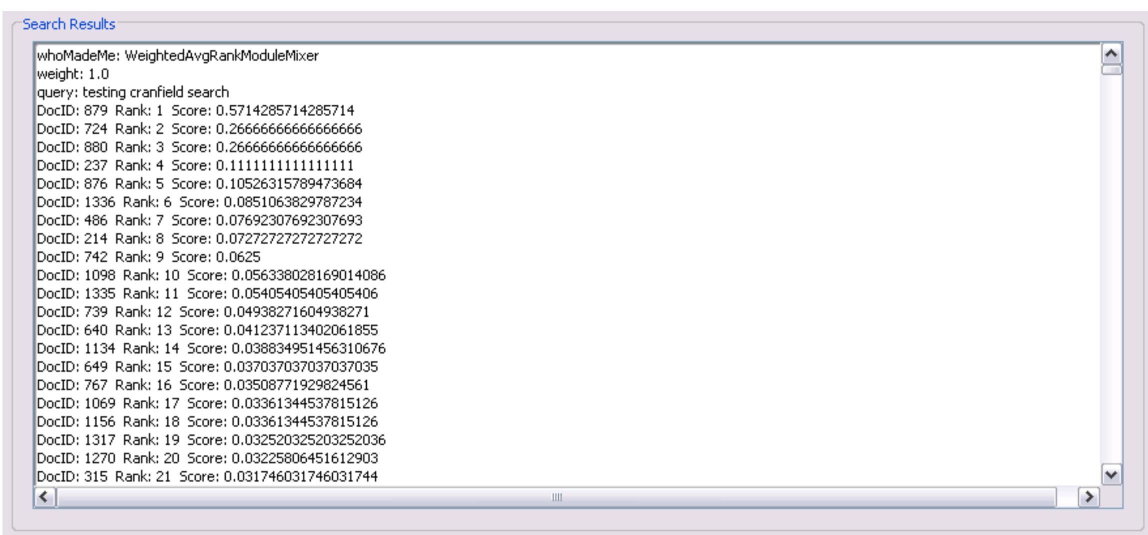


Figure 31. Results Display Section

D. PERFORMANCE EVALUATION

This section presents how the Modular Search Engine framework can help students and researchers design new IR techniques and metasearch methods by calculating and evaluating the performance of the different components within the reference implementation.

1. Average Precision

a. Definition

For a particular query, we use average precision as a metric to measure the performance of an IR technique or a metasearch method [18]. The average precision for a single query is defined as

$$AP = \frac{1}{R}\sum_{n=1}^{D} AP_n ,$$

where R is the number of total relevant documents and D denotes the total number of documents in the corpus. The contribution of document d_n to the average precision AP_n is defined as

$$AP_n = \frac{1}{n}\sum_{m=1}^{n} \delta_{m,n} ,$$

where $\delta_{m,n} = 1$, if the documents d_n and d_m are both relevant to the query, and $\delta_{m,n} = 0$ otherwise.

b. Example

Each corpus included in the reference implementation comes with a set of test queries and a relevancy list that tells which documents in the corpus that are relevant to each test query. These are provided so that different IR and/or metasearch techniques can be compared with one another. For example, the 224th test query for the Cranfield corpus is: "in practice, how close to reality are the assumptions that the flow in a hypersonic shock tube using nitrogen is non-viscous and in thermodynamic equilibrium." There are exactly nine documents identified as relevant to this query.

Using the reference implementation, one can see how each SearchModule performs compares against the other and how the ModuleMixers affect that performance when searching for this test query. Table 3 is a summary of how the two SearchModules performed independently and when mixed with the Condorcet-fuse ModuleMixer.

Relevant Document ID	LDA Ranking	TF-IDF Ranking	CondorcetFuse Ranking
656	6	15	7
1157	40	10	24
1274	113	32	43
1286	4	3	2
1313	15	23	11
1316	120	27	41
1317	26	61	15
1318	7	117	22
1319	100	33	33

Table 3. Relevant Document Rankings for the 224th Cranfield Test Query

With the information in Table 3, we can calculate the average precision for each of the three sets of results. Table 4 displays the average precision calculations for the results of Draeger's LDA SearchModule.

n^{th} Relevant Document	Relevant Document ID	LDA Ranking	AP_n
1	1286	4	1/4 = 0.25
2	656	6	2/6 = 0.33333
3	1318	7	3/7 = 0.42857
4	1313	15	4/15 = 0.26667
5	1317	26	5/26 = 0.19231
6	1157	40	6/40 = 0.15
7	1319	100	7/100 = 0.07
8	1274	113	8/113 = 0.0708
9	1316	120	9/120 = 0.075

Average Precision = 0.20408

Table 4. Average Precision of Draeger's LDA SearchModule

Table 5 dis plays th e av erage p recision calculations f or the results of th e TF-IDF SearchModule.

n^{th} Relevant Document	Relevant Document ID	TF-IDF Ranking	AP_n
1	1286	3	1/3 = 0.33333
2	1157	10	2/10 = 0.2
3	656	15	3/15 = 0.2
4	1313	23	4/23 = 0.17391
5	1316	27	5/27 = 0.18519
6	1274	32	6/32 = 0.1875
7	1319	33	7/33 = 0.21212
8	1317	61	8/61 = 0.13115
9	1318	117	9/117 = 0.07692

Average Precision = 0.1889

Table 5. Average Precision of the TF-IDF SearchModule

Table 6 dis plays th e av erage p recision calculations f or the results of th e Condorcet-fuse ModuleMixer. Note that the average precision of the m ixed results for this query is higher than both Draeg er's LDA SearchModule and the T F-IDF SearchModule.

n^{th} Relevant Document	Relevant Document ID	CondorcetFuse Ranking	AP_n
1	1286	2	1/2 = 0.5
2	656	7	2/7 = 0.28571
3	1313	11	3/11 = 0.27273
4	1317	15	4/15 = 0.26667
5	1318	22	5/22 = 0.22727
6	1157	24	6/24 = 0.25
7	1319	33	7/33 = 0.21212
8	1316	41	8/41 = 0.19512
9	1274	43	9/43 = 0.2093
		Average Precision =	**0.26877**

Table 6. Average Precision of the CondorcetFuse ModuleMixer

2. Mean Average Precision

a. Definition

In order to measure the overall performance of an IR technique or metasearch method, we use the mean average precision. Calculating the mean average precision is as simple as calculating the average precision, as shown above, for each query in the set of test queries and then taking the mean of all those.

b. Example

The Cranfield corpus contains a total of 225 test queries; using a separate application to speed the process, we calculated the mean average precision of both SearchModules independently and when mixed with the Condorcet-fuse ModuleMixer. Figure 32 shows the average precision calculations for each test query, ordered from largest to smallest for each method, and Table 7 shows the mean average precisions. Again, the Condorcet-fuse ModuleMixer outperforms both of the independent SearchModules.

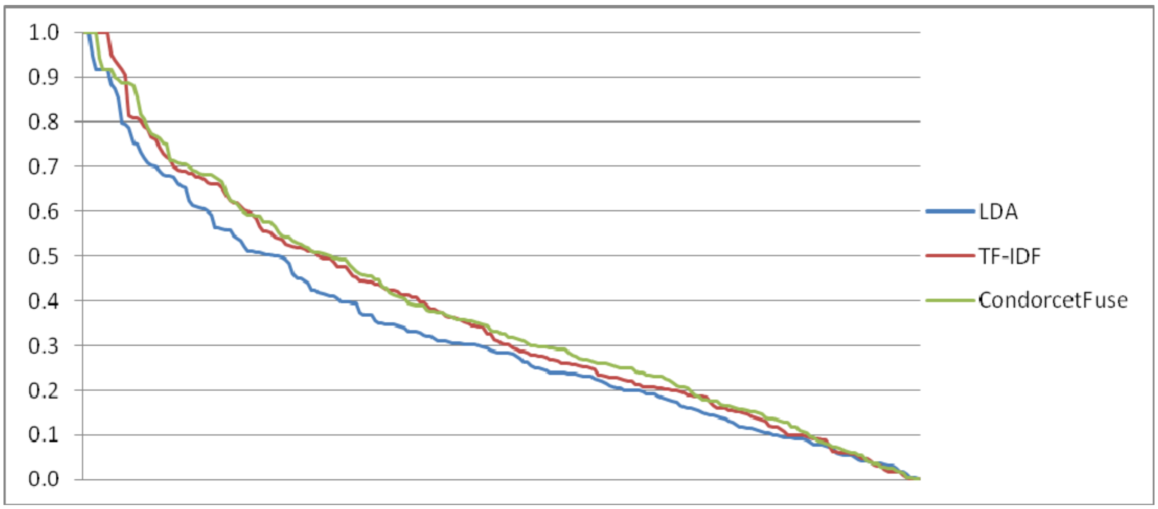


Figure 32. Average Precision of Test Queries

LDA	TF-IDF	CondorcetFuse
0.32711	0.36701	**0.37637**

Table 7. Mean Average Precisions

THIS PAGE INTENTIONALLY LEFT BLANK

V. CONCLUSIONS AND RECOMMENDATIONS

A. RESEARCH CONCLUSIONS

The overarching goal of this thesis was to develop a software API offering students and researchers a framework in which they can develop, test, and implement new IR techniques and metasearch methods, specifically targeting the development of new semantic search techniques.

Utilizing sound engineering practices, those user requirements were specified and incorporated into the overall design of the Modular Search Engine framework. Through extensive use of the Unified Modeling Language, software engineering patterns, and object-oriented features, the Modular Search Engine framework achieved the modularity goal that allows multiple IR techniques to work simultaneously within a single system and allows IR techniques to be seamlessly added and deleted from a system. Keeping with the objectives, the addition of an IR technique requires only the extension of the single abstract SearchModule class with its eight abstract methods. The framework also successfully allows for the development of different metasearch methods that can be interchanged within a system.

Furthermore, this thesis showed conclusively, using a standard metric, that the framework can be used to judge the relative performance of each individual IR technique and metasearch method.

B. RECOMMENDATIONS FOR FUTURE WORK

Overall, this research successfully accomplished its objectives as defined in Chapter I. However, several areas could benefit from further exploration, augmentation, and improvement.

As with any new software application, the framework could greatly benefit from extensive testing and debugging. If the Modular Search Engine framework were to receive greater exposure to students and IR researchers, their feedback would undoubtedly benefit the framework by providing information for patches and upgrades.

One upgrade in particular would be the development and inclusion of a set of diagnostic tools. These tools would be able to automatically calculate the metrics to analyze the performance of the different framework components using the benchmark test corpora. Such tools would make it trivial for the developer to evaluate the performance of a new IR technique or metasearch method.

Additionally, as end-user applications are developed, it is not recommended to build them as stand-alone applications designed to run on client machines. Because of the large requirement for the computer's resources, such applications will undoubtedly run extremely slow and would likely aggravate any user, especially during initialization. Instead, the framework could be used to develop a server application, possibly web-based, that clients could access to perform searches. This style architecture would provide the most responsiveness to users while preserving resources in client computers.

Finally, the framework could benefit from the incorporation of ontological information such as those suggested for the SHARE repository [2]. Such information could be used to develop a robust system that allows a user to refine search queries and navigate through documents based upon the ontological relationships of the documents.

LIST OF REFERENCES

[1] S. Brin and L. Page, "The Anatomy of a Large-Scale Hypertextual Web Search Engine," in *Computer Networks and ISDN Systems,* 1998, pp. 107–117.

[2] J. Johnson and C. Blais, "SHARE repository framework: Component specification and ontology," in *Proceedings from the Fifth Annual Acquisitions Research Symposium*, 2008, pp. 194–212.

[3] M. Draeger, "Use of probabilistic topic models for search," Master's thesis, Naval Postgraduate School, Monterey, CA, 2009.

[4] J. A. Aslam, V. Pavlu and E. Yilmaz, "Measure-based metasearch," in *Proceedings of the 28th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, 2005, pp. 571–572.

[5] C. Larman, *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development*, 3rd ed., New Jersey: Prentice Hall, 2005.

[6] B. Bruegge and A. H. Dutoit, *Object-Oriented Software Engineering: Using UML, Patterns and Java*, New Jersey: Prentice Hall, 2004.

[7] University of Glasgow, Information Retrieval Group, "Test Collections," 2004. Available at http://ir.dcs.gla.ac.uk/resources/test_collections/ (accessed September 2009).

[8] C. D. Manning and H. Schütze, *Foundations of Statistical Natural Language Processing*, Cambridge, MA: MIT Press, 1999.

[9] D. Jurafsky and J. H. Martin, *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*, 2nd ed., New Jersey: Prentice Hall, 2009.

[10] C. D. Manning, P. Raghavan, and H. Schtze, *Introduction to Information Retrieval*, New York: Cambridge University Press, 2008.

[11] European Organization for Nuclear Research, "Colt Project," 2004. Available at http://acs.lbl.gov/~hoschek/colt/index.html (accessed September 2009).

[12] D. M. Blei, A. Y. Ng and M. I. Jordan, "Latent Dirichlet Allocation," *The Journal of Machine Learning Research,* vol. 3, pp. 993–1022, 2003.

[13] M. Montague and J. A. Aslam, "Condorcet Fusion for Improved Retrieval," in *CIKM '02: Proceedings of the Eleventh International Conference on Information and Knowledge Management*, pp. 538–548, 2002.

[14] W. H. Riker, *Liberalism Against Populism*, San Francisco: W. H. Freeman, 1982.

[15] H. Moulin, *Axioms of Cooperative Decision Making*, New York: Cambridge University Press, 1988.

[16] J. S. Kelly, *Social Choice Theory*, Berlin: Springer-Verlag, 1988.

[17] M. de Condorcet, *Essai Sur l'Application De l'Analyse à La Probabilité Des Décisions Rendues à La Pluralité Des Voix*, 1785.

[18] S. Robertson, "A new interpretation of average precision," in *SIGIR '08: Proceedings of the 31st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pp. 689–690, 2008.

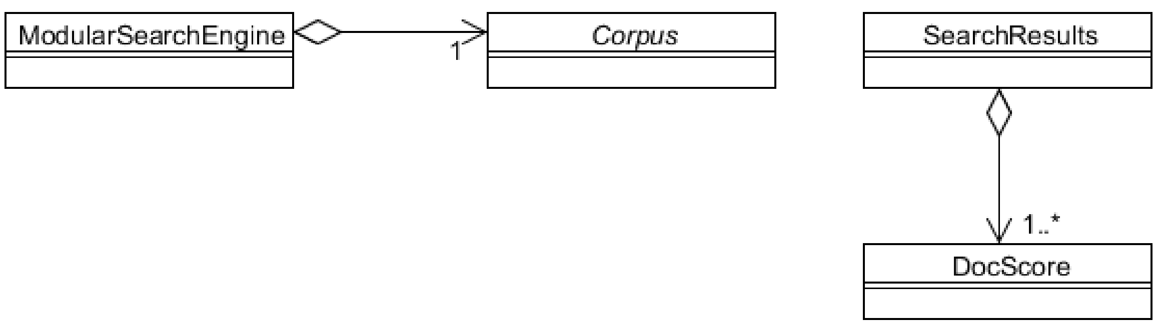
APPENDIX–UML REFERENCE KEY

This appendix contains the reference for the UML symbols used in Chapters II and III of this thesis.

A. FIGURE 3–UML DOMAIN OBJECT MODEL

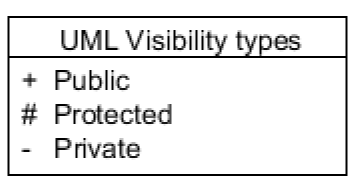
An association with an aggregation relationship indicates that one class is a part of another class. In this relationship the child class instance can outlive its parent class; the existence of the child is not dependent on the existence of the parent. The aggregation relationship is represented with a solid line drawn from the parent class to the child class with an open diamond shape on the parent class's end.

For example, a ModularSearchEngine object contains a single Corpus object, but the SearchResults object contains one or more DocScore objects:



B. FIGURES 11-24 UML CLASS MODELS

Each class member and method is preceded with one of three symbols that indicate its visibility.



Additionally, if any method name or class name is *italicized* it indicates that the method or the class is abstract.

THIS PAGE INTENTIONALLY LEFT BLANK

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Fort Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California

3. Marine Corps Representative
   Naval Postgraduate School
   Monterey, California

4. Director, Training and Education, MCCDC, Code C46
   Quantico, Virginia

5. Director, Marine Corps Research Center, MCCDC, Code C40RC
   Quantico, Virginia

6. Marine Corps Tactical Systems Support Activity (Attn: Operations Officer)
   Camp Pendleton, California