# System design specification

## Part 1: Background:

This system is call "Really Useful Search Engine" or simply "RUSE". RUSE only deals with textual information, that is documents and queries are all text. It mainly has following functions: (1)fileContents query (2)modTime query (3)fileSize query (4)fileName query (5)range query (6)proximity query (7)result ranking

Let me explain the details of these functions :

(1)fileContents query:

This function returns client a list of names of files whose contents satisfies client's query expression. "Satisfies" here means the file's content contains the terms client want it to contain and do not contains the terms client don't want it to contain. The client's query expression tells the RUSE which term should be in the result files and which term should not appear in the result query. RUSE supports three boolean operators in the query expression: AND, OR, and NOT. Here is an example of the query expression: (apple OR banana) AND (NOT pear)

(2)modTime query

This function let clients search for a file by its last modified time. ModTime query supports range query, which means that client can search for files whose last modified time is within a time range.

Examples: a.modTime:20080101000000

b.modTime:[20080101000000-20011030235959]

c.modTime:{20080101000000-20011030235959}

(3)fileSize query

This function let clients search for a file by its size. FileSize query supports range query, which means that client can search for files whose size is within a time range.

Examples: a.fileSize:45

b.fileSize:[3-56]

c.fileSize:{3-56}

(4)fileName query

This function let clients search for a file by its name. FileName query supports range query, which means that client can search for files whose name is within a time range.

Examples: a.fileName:abc

b.fileName:[macrom-microsoft]

c.fileName:{macrom-microsoft}

(5)range query

Range query let client search for files whose certain attribute is within a range. The attributes that support range query are: modTime, fileSize, fileName. There are two types of range queries:inclusive and exclusive.Inclusive range queries are denoted by square brackets; exclusive range queries are denoted by curly brackets. Inclusive query means the result files' attributes must within the range inclusively, while exclusive query means the result files' attributes must within the range exclusively.

(6)proximity query

A proximity query in RUSE is a quoted string followed by an optional distance parameter. The quoted string specifies *n ordered* search terms, which we denote as *w*1, *w*2, *. . .*; and *wn*; while the distance parameter comprises a tilde symbol (˜) and a

distance number *d*. Such a query matches every document *D* containing at least one snippet *S* that satisfies the following conditions:

1. *S* ½ *D*, which we already knew.

2. *8i 2* [1*; n*]*;wi 2 S*, and they appears in exactly the same order as here.

3. In *S*, *8j 2* [1*; n ¡* 1]*; distance*(*wj ;wj*+1) · *d*

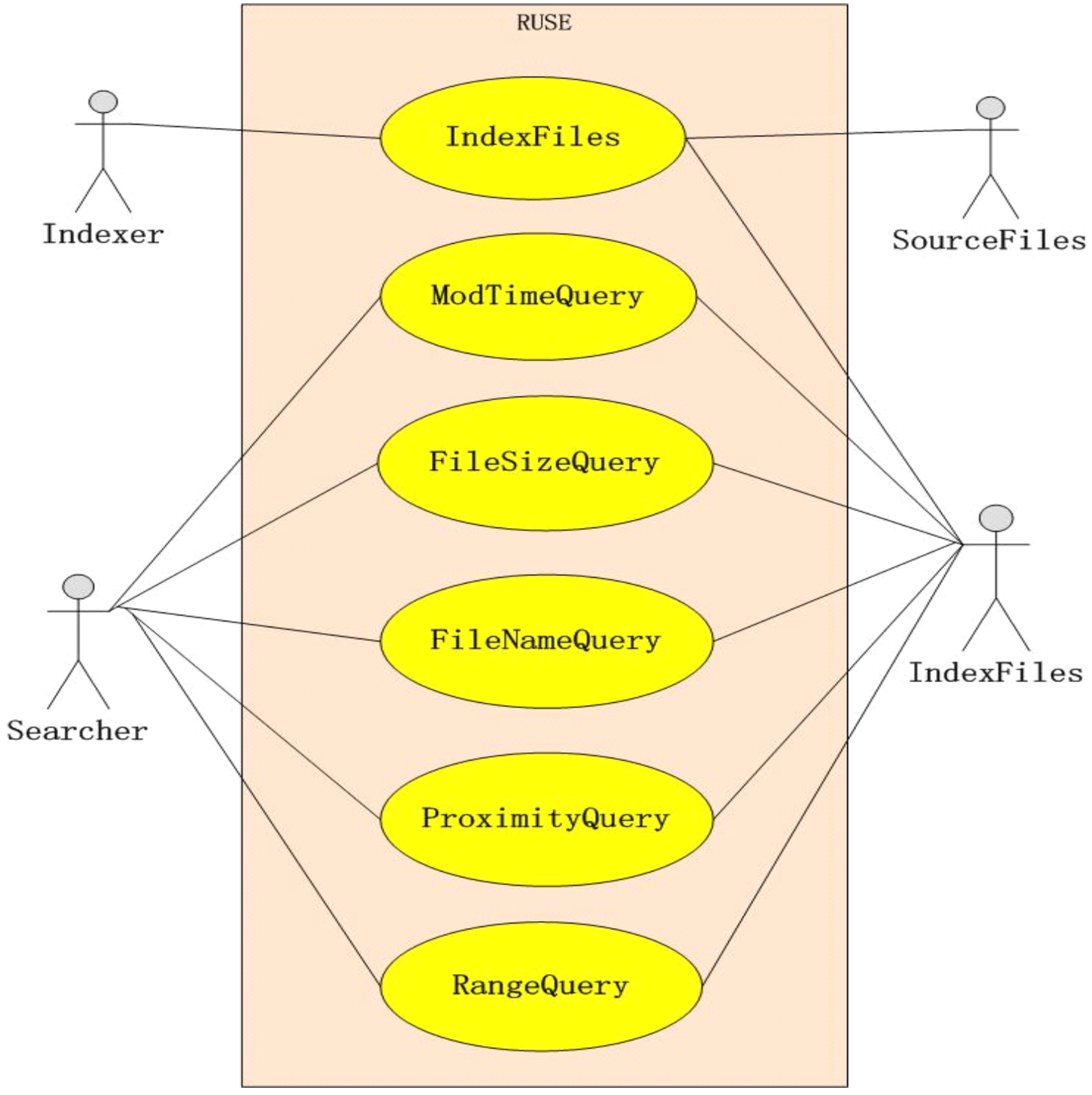
where *distance*(*u; v*) returns the number of terms between *u* and *v*. If words *u* and *v* are adjacent, then we have *distance*(*u; v*) = 0.

RUSE only supports proximity queries over the fileContents attribute.A proximity query can also be an operand of an AND or an OR operator. Hence "ipod nano" AND apple is a valid boolean query.

(7)result ranking

Result ranking means the search result can be ranked by some attributes of the files. RUSE supports two ranking strategies :One is to rank the results by file sizes in descending order; the other is to rank the results by last modified time of files also in descending order, i.e., from latest to oldest.

## Part2 : Use Case Section:




Description:

Use case1: IndexFiles

main actor: Indexer

other actors: SourceFiles, IndexFiles

In this use case, an Indexer who wants to index the contents of all the ".txt" and ".doc" files in a certain directory will give a request to the RUSE with the absolute path of the directory. RUSE will scan the SourceFiles the Indexer wants to index.Then RUSE store the index result into some files which is the IndexFiles.

Use case2: ModeTimeQuery

main actor: Searcher

other actors:IndexFiles

In this use case, the searcher wants to search the files' names by their last modified time based on IndexFiles. First, the searcher will submit a search query to RUSE. RUSE will analyse the query and scan the IndexFiles to get a result. At last, RUSE return the result to the searcher. The result is a set of file names that satisfy the query.

Use case3: FileSizeQuery

main actor: Searcher

other actors:IndexFiles

In this use case, the searcher wants to search the files' names by their sizes based on IndexFiles. First, the searcher will submit a search query to RUSE. RUSE will analyse the query and scan the IndexFiles to get a result. At last, RUSE return the result to the searcher. The result is a set of file names that satisfy the query.

Use case4: FileNameQuery

main actor: Searcher

other actors:IndexFiles

In this use case, the searcher wants to search the files' names by their names based on IndexFiles. First, the searcher will submit a search query to RUSE. RUSE will analyse the query and scan the IndexFiles to get a result. At last, RUSE return the result to the searcher. The result is a set of file names that satisfy the query.

Use case5: ProximityQuery

main actor: Searcher

other actors:IndexFiles

In this use case, the searcher wants to search a file that contains a sequence of terms in which neighbouring terms are separated by some terms whose amount is less that a certain value.To search a file that contains a term sequence in which terms are separated by no term is a special case of the proximity query. First, the searcher will submit a search query to RUSE. RUSE will analyse the query and scan the IndexFiles to get a result. At last, RUSE return the result to the searcher. The result is a set of file names that satisfy the query.

Use case6: RangeQuery

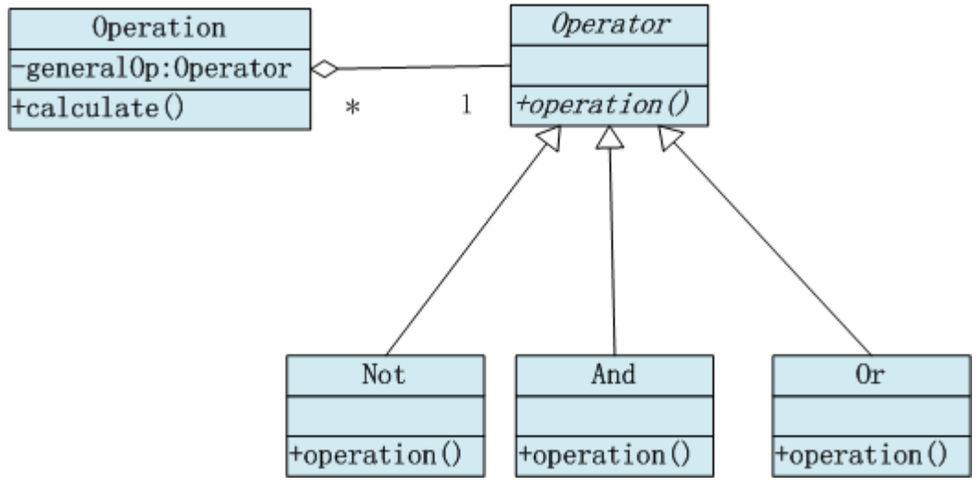
In this use case, the searcher wants to search for files whose certain attribute is within a range, for example , to search for files whose modTime is between 20080101000000 and 20011030235959. First, the searcher will submit a search query

to RUSE. RUSE will analyse the query and scan the IndexFiles to get a result. At last, RUSE return the result to the searcher. The result is a set of file names that satisfy the query.

## Part 3:Design Pattern section:

In the implementation of the RUSE, I use 6 design patterns in total. They are: Strategy Pattern, Facade Pattern, Factory method Pattern , Singleton Pattern , Command Pattern and Template method Pattern. I will explain the reason why I use them one by one:

1. Strategy Pattern




Roles:

Context—Operation

Strategy—Operator

ConcreteStrategy—And, Or, Not

I use the Strategy Pattern in the “search part” of the program. I use it to encapsulate the algorithms of different boolean operators. Because the boolean

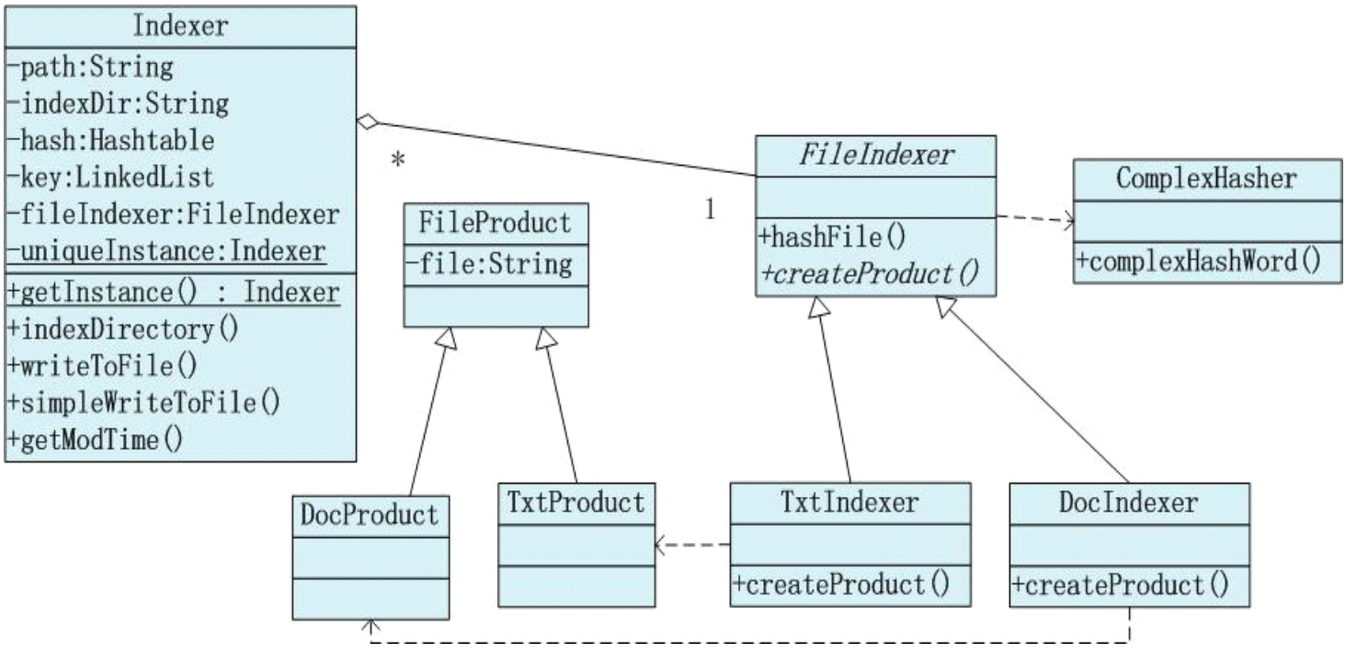
operators defined in the query syntax may be changed in the future. Perhaps some new operators will be added in the syntax or some original operators will be discarded. Besides, the way of handling a certain operator may also be changed. An operator may be defined to have a new meaning that is totally different from its primary one. Using the Strategy Pattern, if there is a new operator we just create a class of the operator and let it extend the abstract class Operator and use its defined algorithm to implement the method operation(). And if there is any change in the algorithm of an operator, we just need to change it in the operator's class. In this way, we "encapsulate what varies."

As And, Or, Not extends the abstract class Operator, it alse give us convenience to "program to interface ,not an implementation." By assigning a concrete operator object to an Operator typed object, we just call the method operation() and we get a certain concrete algorithm. When we assign a different type of concrete operator object to it, we get a different algorithm.

2. Façade Pattern

I use this pattern in two places, one is in the Indexer part and one is in the Tester part.

a. Facade Pattern in Indexer part




Roles:

Façade—Indexer

Subsystem classes--FileIndexer, ComplexHasher, TxtIndexer, DocIndexer...

I use the Façade Pattern here because I want to encapsulate the complicate implementation of the index part to enhance its reusibility and decouple the client from the subsystem of many small components.Besides, it reduces the dependency of the subsystems. The Indexer provides a simplified interface for clients to use the index function.Other subsystem only need to invoke methods in the Indexer and don't need to know the implementation details.And if there is a need to change the implementation, we only need to change it within Façade.Besides, The Façade still leaves the subsystem accessible to be used directly, which allows us to advance functionality of the subsystem classes easily.

b. Façade Pattern in the Tester part

(diagram is omitted)

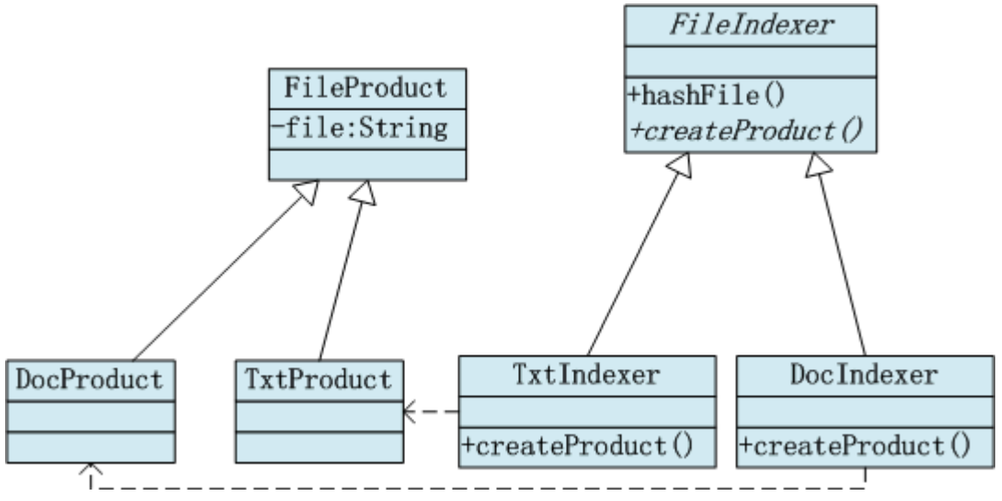
Roles:

Façade—Tester

Subsystem classes—Indexer, Searcher and all other classes in RUSE

I use the Façade Pattern here because I want to encapsulate the complicate implementation of the index part and search part to reduce the complexity of the system and decouple the client from a subsystem of many small components.The Tester provides a simplified interface for clients to use the index function and search function. A client can just submit his query by invoking methods in Tester and get what he needs.

3. Factory method Pattern




Roles:

Creator—FileIndexer

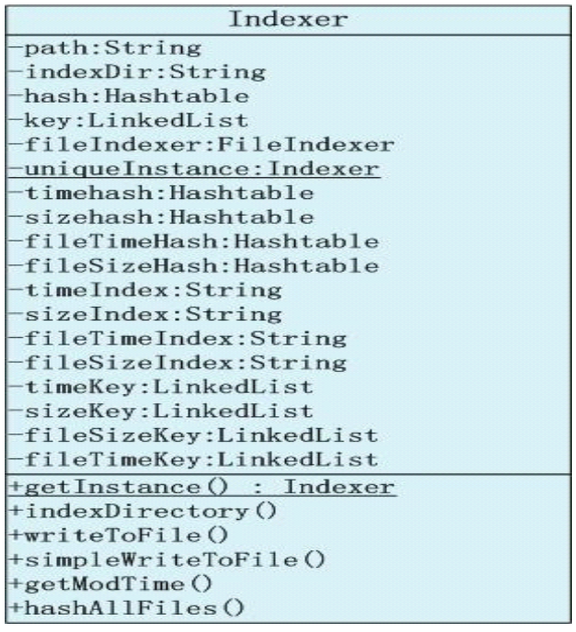
ConcreteCreator—TxtIndexer,DocIndexer

Product—FileProduct

ConcreteProduct—TxtProduct,DocProduct

I use the Factory method Pattern in the “index part” of the program.I use it here to let subclasses of FileIndexer—TxtIndexer and DocIndexer—to decide which class to instantiate. Let the FileIndexer defer instantiation to its subclasses. Maybe you think what we need the TxtIndexer and DocIndexer to product is a only a String of the context of the file and there is no complex product so there is no need to use the Factory Method Pattern .But I take a consideration of the possible change of the type of files in the future.For example, maybe we will need to handle a jar file or some complex binary files.Then we have different solutions to deal with these files and the concreteFactory will product other types of objects rather than String any more.So the Product has a potential to be complex, thus I use the Pattern to encapsulate the possible changes.Using Factory Method Pattern, I promote loose coupling by reducing the dependency of the application on concrete classes.

4. Singleton Pattern

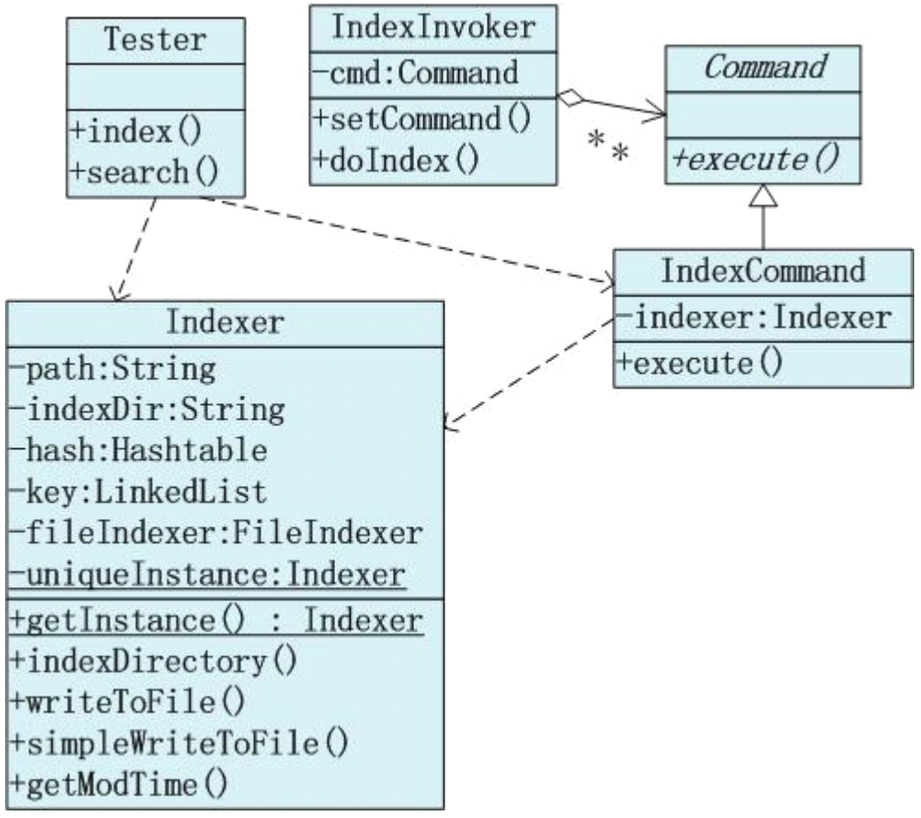


Roles:

Singleton—Indexer

I design the Indexer class to be a Singleton to enhance the performance of the system. As we know the Function of the Indexer is to handle a index request to index the contents of all the ".txt" and ".doc" files in a certain directory and record the result into a file. And searcher will use the file to search. So if an object of Indexer has finished its work of recording the index result ,it doesn't need to exist any more. We only need the index file to do search.That means we don't need to create different Indexer objects. We can have one Indexer object and change its attribute values to do different indexes. I use Singleton to ensure the Indexer class only has one instance.If there is a request with a new path and a new indexFile , the attributes of this object will be assigned with these new values and the former Index object turns to be a new one. There is no need to create a new Indexer object which will reduce the cost of memory. It is especially important if we want to change the RUSE to an application on mobile devices, such as mobile phones.

5. Command Pattern




Roles:

Client—Tester

Invoker—IndexInvoker

Receiver—Indexer

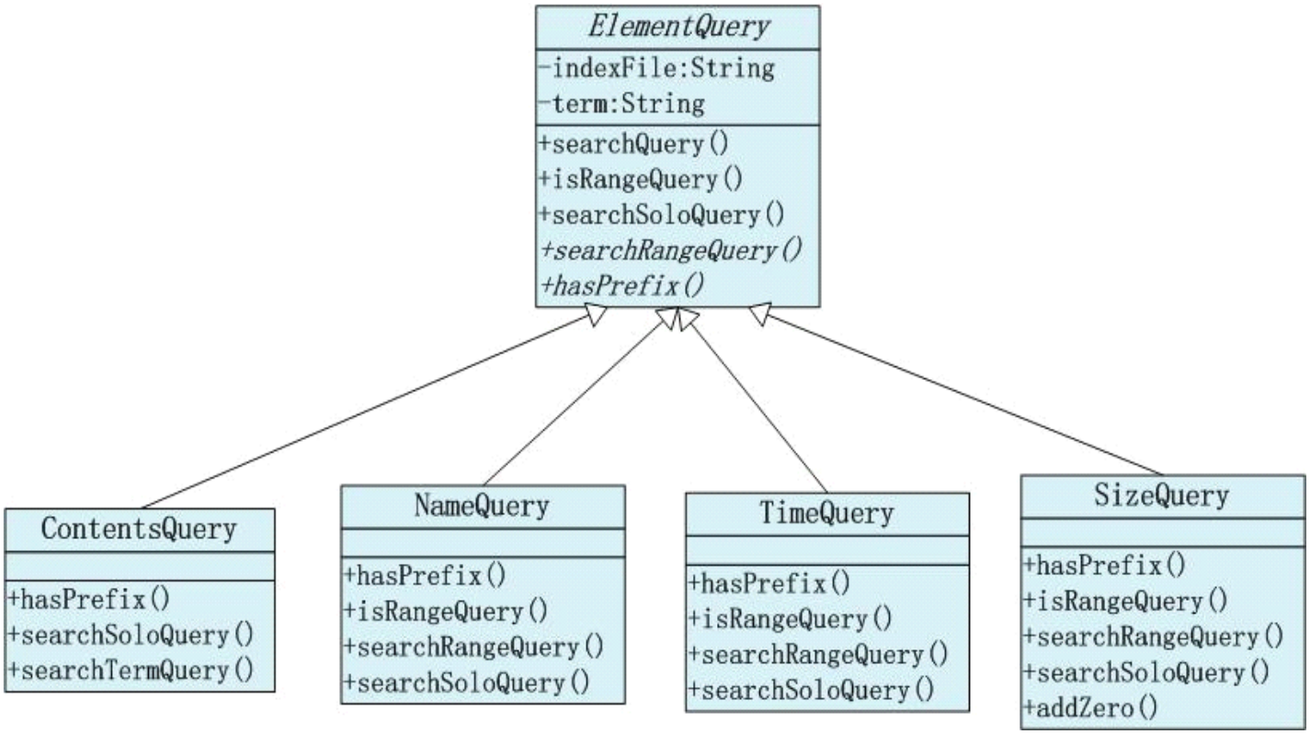
Command—IndexCommand

Command interface—Command

I encapsulte the index request to be an object to decouple the request from the "receiver"—Indexer. The Invoker doesn't know the detail of the request and how it will be handled by the Indexer.Tester does a setCommand() to store the Command object in IndexInvoker. And IndexInvoker decides whether to excute it or not. The index

request is encapsulated thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations. If the RUSE is used by many people and establishes a request queue, you can add your algortithm of handling the queue in the IndexInvoker to handle requests as you like. Besides, log requests enhance the security of the system and undoable operation provides a humanistic interface to user. All these promote the usiblity of RUSE.

6.Template Method Pattern




Roles:

AbstractClass: ElementQuery

ConcreteClass: ContentsQuery, NameQuery, SizeQuery, TimeQuery

I use Template Method Pattern here to define the skeleton of the algorithm of how

to handle a meta query in the whole query expression. The template method in the class ElementQuery is method searchQuery() which lets subclasses(ContentsQuery, NameQuery, SizeQuery, TimeQuery) redefine its internal steps without changing the algorithm's structure. I put two hooks in class ElementQuery: isRangeQuery() and searchRangeQuery(). Because not all kinds of queries support rangeQuery. Using template method ,you can have different implementations of an algorithm's individual steps,but keep control over the algorithm's structure. In this way, you won't have duplicate code.

## Part 4: Patterns' Good Performance in writing RUSE2

In writing RUSE2 , I find it quite easy to add new functions to RUSE1 because I have encapsulated many possible varies in RUSE1. I don't need to change much original code and only need to add some code to the original code. Here is my good experience in building RUSE2:

1. As RUSE2 doesn't stem words in both in index and search part , I need to delete that steps. But as I have separated stemming function in the class Stemmer, I only need to delete this class and the sentences of invoking this class.Then I delete this step.

2. As there are new types of query based on modTime, fileName, fileSize, I need to make several new index files. As I encapsulate index process in class indexer, what I need to do is just to add the creation of new index files in Indexer.

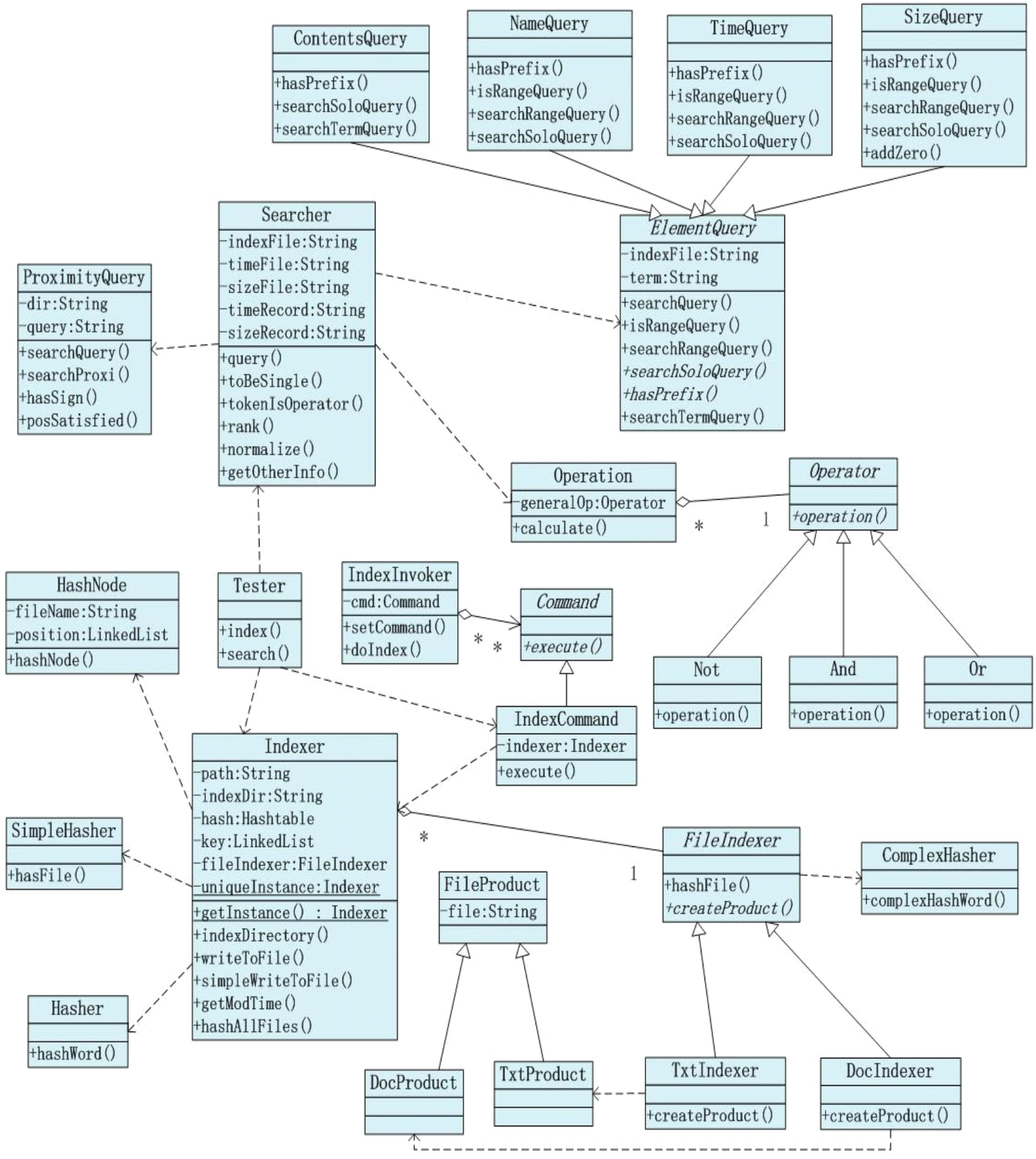
3. As I use Strategy Pattern to encapsulate the algorithm of "AND two lists " in the class "And", when I need to use it in my new class ProximityQuery, I just need to create

an "And" object and call that algorithm. Strategy Pattern enhance the reusibility of my code.

4. As I encapsulated the algorithm of hashing a file's contents to hashtable in class Hasher and decoupled the class Indexer from this detail, when I need to add terms' location information to the index file I only need to make changes in class Hasher and don't need to change code in Indexer at all.

## Part5:Class diagram section:

```
Following design uses Strategy Pattern , Facade Pattern,Factory method Pattern ,Singleton Pattern ,Command Pattern and Template method Pattern:
Their corresponding roles are as following:
In Strategy Pattern (Searcher part):
Context--Operation
Strategy--Operator
ConcreteStrategy--Not,And,Or

In Facade Pattern:
Tester--Facade--Tester
Subsystem classes--Indexer,Searcher and rest classes

Factory method Pattern(Indexer Part):
Creator--FileIndexer
ConcreteCreator--DocIndexer,TxtIndexer
Product--FileProduct
ConcreteProduct--DocProduct,TxtProduct

Singleton Pattern(Indexer Part):
Singleton--Indexer

Facade Pattern(Indexer Part):
Facade--Indexer
Subsystem classes-FileIndexer,Hasher,TxtIndexer,DocIndexer

Command Pattern(Indexer Part):
Client--Tester
invoker--IndexInvoker
Receiver--Indexer
Command--IndexCommand
Command interface--Command

Template Method Pattern:
AbstractClass:ElementQuery
ConcreteClass:ContentsQuery,NameQuery,SizeQuery,TimeQuery
```


ContentsQuery
+hasPrefix()
+searchSoloQuery()
+searchTermQuery()
NameQuery
+hasPrefix()
+isRangeQuery()
+searchRangeQuery()
+searchSoloQuery()
TimeQuery
+hasPrefix()
+isRangeQuery()
+searchRangeQuery()
+searchSoloQuery()
SizeQuery
+hasPrefix()
+isRangeQuery()
+searchRangeQuery()
+searchSoloQuery()
+addZero()
Searcher
-indexFile:String
-timeFile:String
-sizeFile:String
-timeRecord:String
-sizeRecord:String
+query()
+toBeSingle()
+tokenIsOperator()
+rank()
+normalize()
+getOtherInfo()
ElementQuery
-indexFile:String
-term:String
+searchQuery()
+isRangeQuery()
+searchRangeQuery()
+searchSoloQuery()
+hasPrefix()
+searchTermQuery()
ProximityQuery
-dir:String
-query:String
+searchQuery()
+searchProxi()
+hasSign()
+posSatisfied()
Operation
-generalOp:Operator
+calculate()
Operator
+operation()
*
1
HashNode
-fileName:String
-position:LinkedList
+hashNode()
Tester
+index()
+search()
IndexInvoker
-cmd:Command
+setCommand()
+doIndex()
Command
+execute()
* *
Not
+operation()
And
+operation()
Or
+operation()
IndexCommand
-indexer:Indexer
+execute()
Indexer
-path:String
-indexDir:String
-hash:Hashtable
-key:LinkedList
-fileIndexer:FileIndexer
-uniqueInstance:Indexer
+getInstance() : Indexer
+indexDirectory()
+writeToFile()
+simpleWriteToFile()
+getModTime()
+hashAllFiles()
SimpleHasher
+hasFile()
*
1
FileIndexer
+hashFile()
+createProduct()
ComplexHasher
+complexHashWord()
FileProduct
-file:String
Hasher
+hashWord()
DocProduct
TxtProduct
TxtIndexer
+createProduct()
DocIndexer
+createProduct()

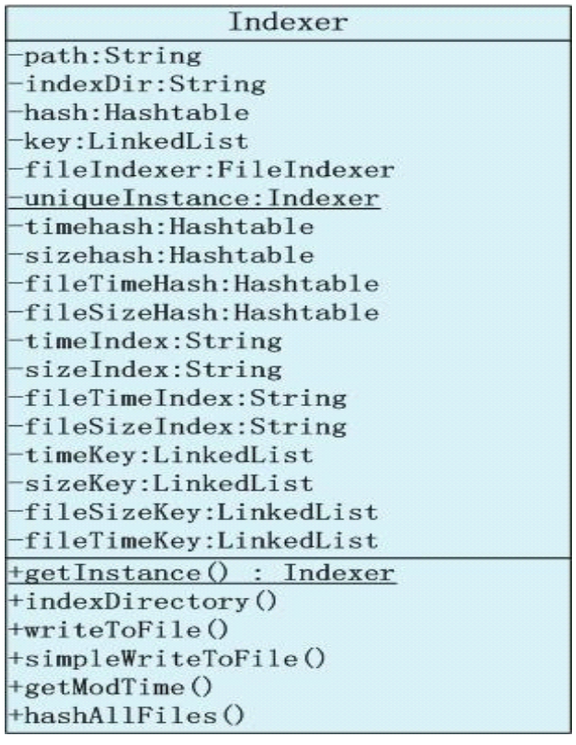
This Class Diagram shows the classes and relationship between classes of RUSE .

Note in the diagram give an illustration of the roles of corresponding classes in the design patterns used in the system.

The system has 27 classes and uses 6 patterns. The parameter and return type of most classes are omitted to make the diagram clear. The attributes of class Indexer are not all listed above as there are too many.

Now I'll explain details of some important classes for you:

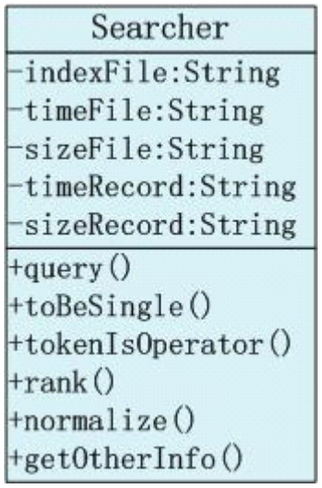
A. Class : Indexer




Indexer is the central class in index part. It deals with the main algorithm of storing useful information of each file into several index files. Method indexDirectory() is the method that controls the whole flow. It calls a recursion method hashAllFiles() to put all useful information of files in a directory into several hashtables. After that, it calls method

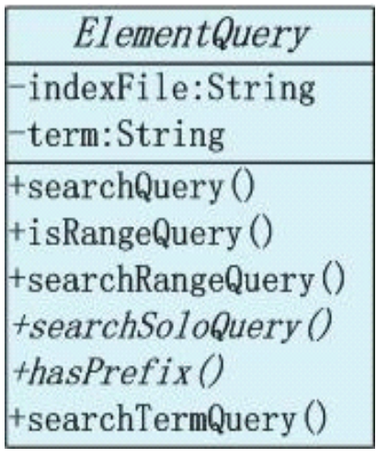
writeToFile(), simpleWriteToFile() to write the information from hashtables to index files.

## B.Class : Searcher




Searcher is the central class in search part. It contains the algorithm of analysing query expressions ,invoking proper algorithm to handle different types of query and ranking the result list. It uses two stacks to handle with expressions. One is used to store operands and the other is used to store operators. It treats every subquery separated by "and","or","not" as a meta query and invokes different algorithm to handle different meta queries. All meta query handlers return a filename list as a result. Then it invokes class And, Or,Not to deal with logic operations between two filename list. At last, it gets a result filename list. Then it invokes method rank() to rank the result by modTime or size according to client's request. After that it calls method getOtherInfo() to get the size and modTime of the files in the list and encapsulate all information to be a list of Tester.Result object and return it to client.

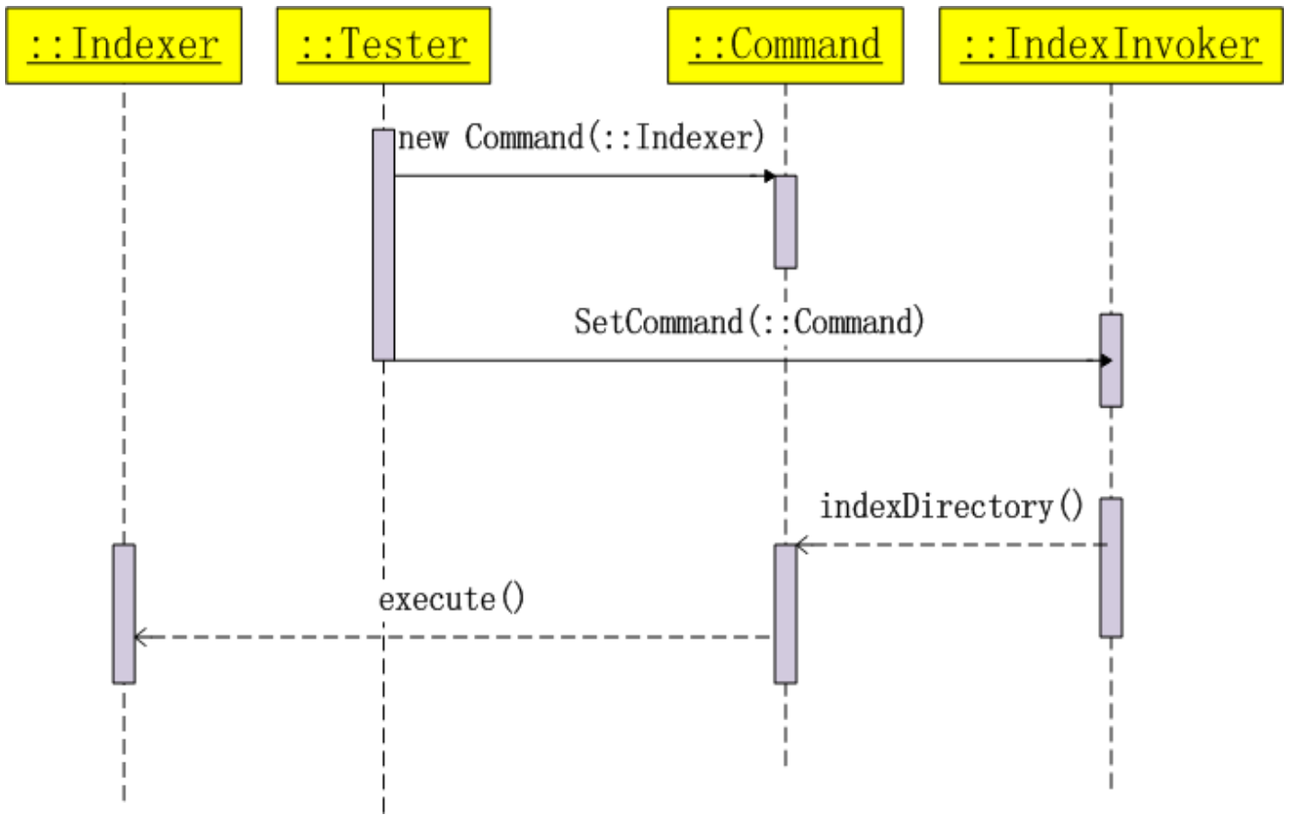
## C.Class : ElementQuery




ElementQuery is the abstract method in the template method Pattern which defines the skeleton of algorithm of handling a query including nameQuery, timeQuery, sizeQuery and ContentsQuery. It uses two hooks: isRangeQuery() and searchRangeQuery(). The skeleton method is searchQuery() and it is defined to be final. The class has two abstract methods which must be implemented by the concrete classes: hasPrefix() and searchSoloQuery().

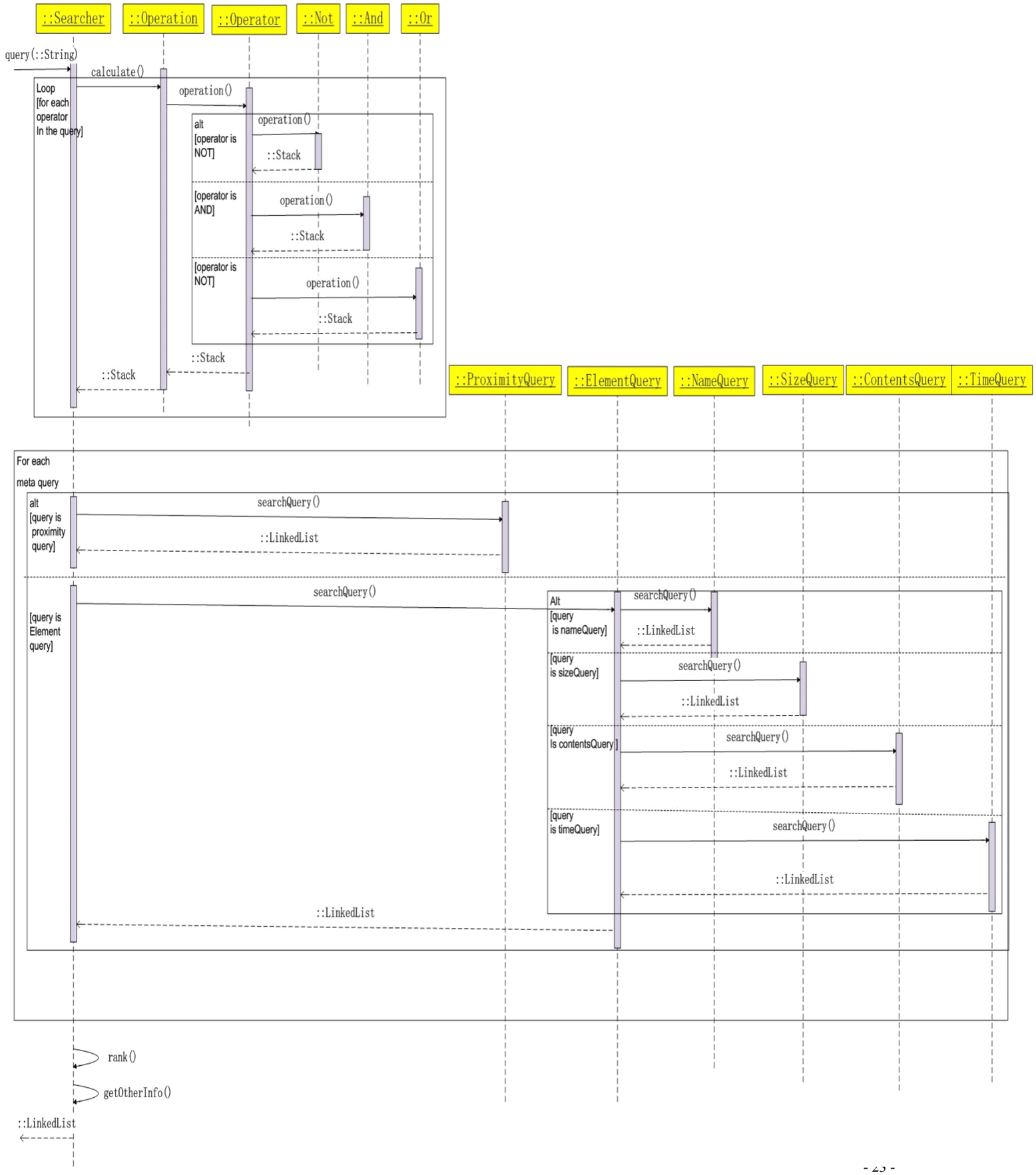
## Part6: Sequence diagram section:

### Diagram1: sequence diagram of the Command Pattern



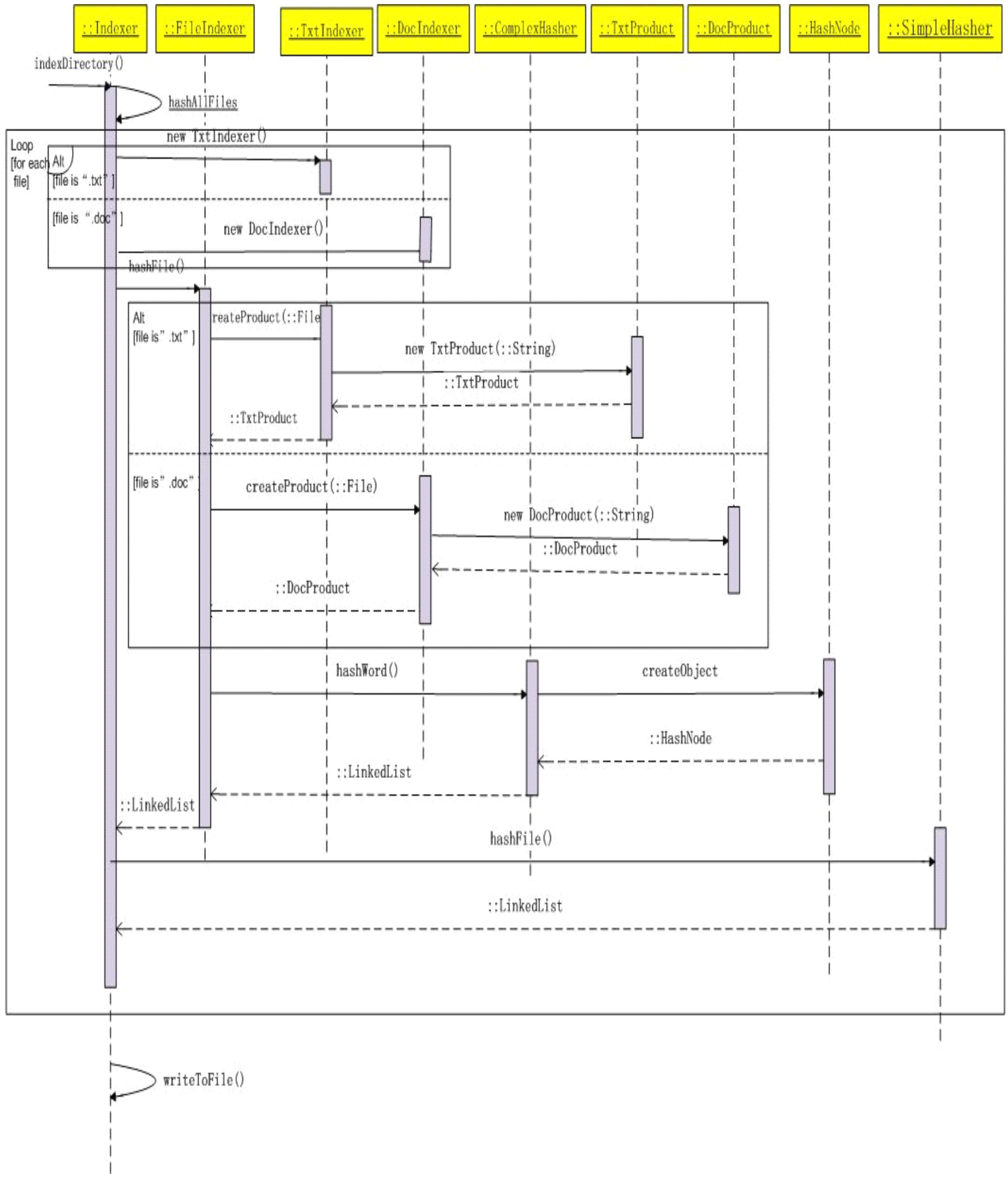

This diagram illustrates how the Command Pattern in RUSE works. It reflects the communication between classes in the Pattern.

**Diagram2: sequence diagram of the searcher part**


::Searcher
::Operation
::Operator
::Not
::And
::Or
query(::String)
calculate()
Loop
[for each
operator
In the query]
operation()
alt
[operator is
NOT]
operation()
::Stack
[operator is
AND]
operation()
::Stack
[operator is
NOT]
operation()
::Stack
::Stack
::Stack
::ProximityQuery
::ElementQuery
::NameQuery
::SizeQuery
::ContentsQuery
::TimeQuery
For each
meta query
alt
[query is
proximity
query]
searchQuery()
::LinkedList
[query is
Element
query]
searchQuery()
Alt
[query
is nameQuery]
searchQuery()
::LinkedList
[query
is sizeQuery]
searchQuery()
::LinkedList
[query
Is contentsQuery]
searchQuery()
::LinkedList
[query
is timeQuery]
searchQuery()
::LinkedList
::LinkedList
rank()
getOtherInfo()
::LinkedList

This diagram illustrates how different classes in the searcher part of the program collaborate to implement the searching function. Class searcher calls different query handlers to deal with different queries. Searcher does it by assign a proper subclass typed object to an ElementQuery object and invoke searchQuery() of the object. Then the object dynamicly calls its real-type-class's searchQuery() to handle the request.

**Diagram3: sequence diagram of the Indexer part**


::Indexer
::FileIndexer
::TxtIndexer
::DocIndexer
::ComplexHasher
::TxtProduct
::DocProduct
::HashNode
::SimpleHasher
indexDirectory()
hashAllFiles
Loop
[for each file]
Alt
[file is ".txt"]
new TxtIndexer()
[file is ".doc"]
new DocIndexer()
hashFile()
Alt
[file is ".txt"]
createProduct(::File
new TxtProduct(::String)
::TxtProduct
::TxtProduct
[file is ".doc"]
createProduct(::File)
new DocProduct(::String)
::DocProduct
::DocProduct
hashWord()
createObject
::HashNode
::LinkedList
::LinkedList
hashFile()
::LinkedList
writeToFile()

This diagram illustrates how different classes in the indexer part of the program collaborate to implement the indexing function.Indexer is responsible for storing useful information of each file into several index files. It calls method indexDirectory() to do the job. "indexDirectory()" calls the recursion method hashAllFiles() to put all useful information of files in a directory into several hashtables. Then it calls method writeToFile() to write the information from hashtables to index files.