CSE 321 Software Performance Engineering
Project Report: Modeling a Java Application

Yi Zhang & Haibei Zhang

1. INTRODUCTION

In this project, we are aimed at evaluating the performance of a reasonably complex Object-Oriented (OO) application written in Java. We follow the three-level hierarchical performance modeling addressed by Abdel-raouf, Ammar and Sholl [1] to generate agglomerative models and obtain the overall performance function.

2. MAJOR COMPONENTS

The major effort in this practical project is invested in two phases.

First, we build performance models (i.e. Performance Image or PI) for each class and object in the system. At the bottom of this hierarchy, we use Computation Structure Model [2] method to generate the elementary performance functions for each class and object at the Object-creation Performance Layer (OPL) and Service Performance Layer (SPL).

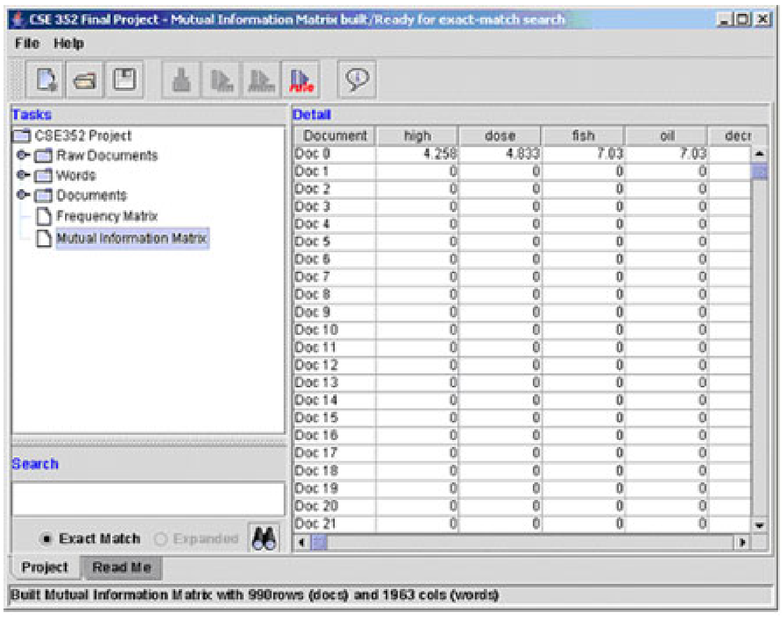
Since UML/Class Diagram is a standard way of describing the class's properties, methods and relationships with other types, we decide to make extensive use of UML diagrams when generating Related Classes/Objects Performance Layer (RPL)

Second, we evaluate the performance of each component at the master level by aggregating the PI at object level. We add up the cost of all components to estimate the overall cost of the application.

3. PROGRAM ARCHITECTURE

The application we picked is a Text Search/Mining Engine implemented as standalone J2SE application with graphic user interface (GUI). The application was developed by Haibei Zhang as a term project for CSE 352: Data Mining. The application implements a document similarity measurement algorithm based on angle between weighted feature vectors. Data mining algorithm is integrated to expand the query and retrieve semantically related but not exact-matching documents.

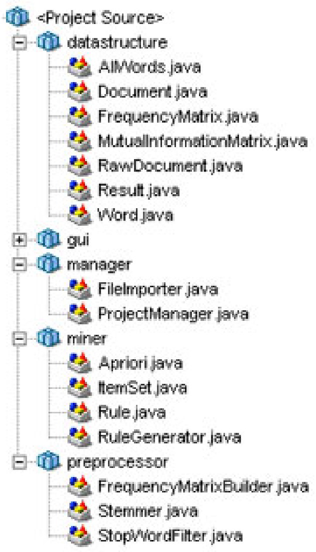
The application consists of five packages and a total of 22 classes. Classes are roughly packaged based on their standing in the class coupling or composition relationships. For instance, classes in the datastructure package are elementary data structure classes that are referenced by other classes as components. The manager.ProjectManager is the main class that organizes all components and interacts with GUI. Below are a screenshot and the package/class structure of this program:


CSE 352 Final Project - Mutual Information Matrix built/Ready for exact-match search
File
Help
Tasks
Detail
Search
Exact Match
Expanded
Project
Read Me

CSE352 Project
- Raw Documents
- Words
- Documents
- Frequency Matrix
- Mutual Information Matrix

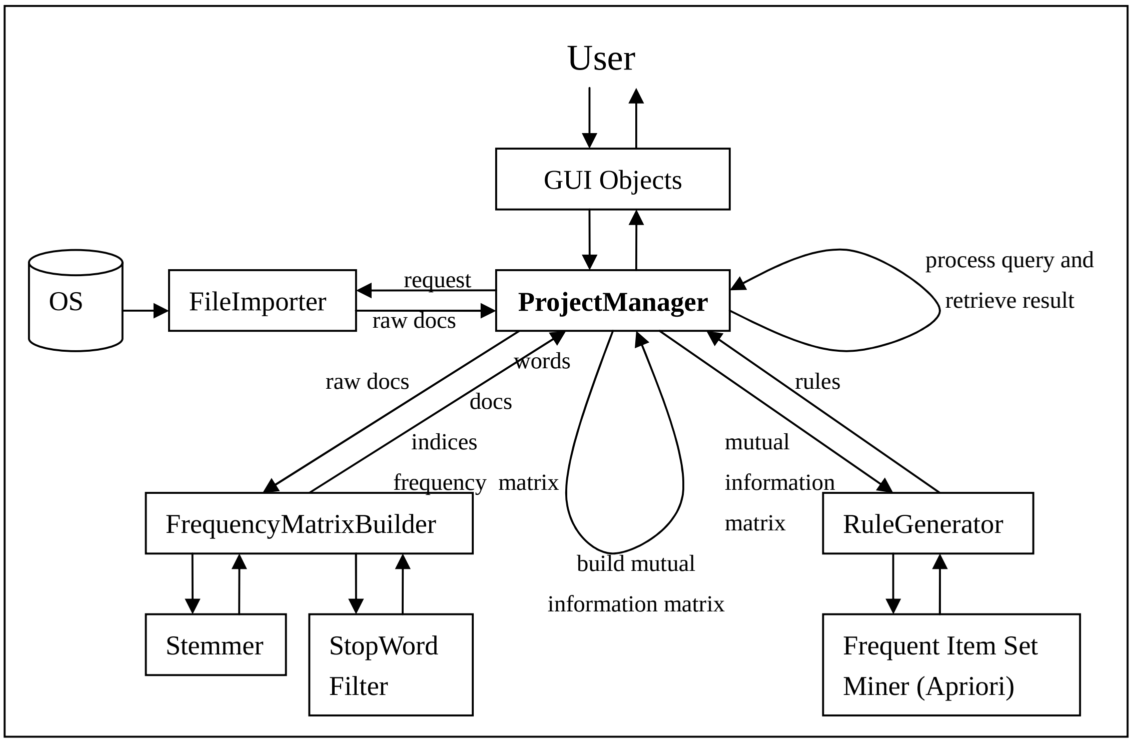
Document	high	dose	fish	oil	decr
Doc 0	4.258	4.833	7.03	7.03	
Doc 1	0	0	0	0	
Doc 2	0	0	0	0	
Doc 3	0	0	0	0	
Doc 4	0	0	0	0	
Doc 5	0	0	0	0	
Doc 6	0	0	0	0	
Doc 7	0	0	0	0	
Doc 8	0	0	0	0	
Doc 9	0	0	0	0	
Doc 10	0	0	0	0	
Doc 11	0	0	0	0	
Doc 12	0	0	0	0	
Doc 13	0	0	0	0	
Doc 14	0	0	0	0	
Doc 15	0	0	0	0	
Doc 16	0	0	0	0	
Doc 17	0	0	0	0	
Doc 18	0	0	0	0	
Doc 19	0	0	0	0	
Doc 20	0	0	0	0	
Doc 21	0	0	0	0	

Built Mutual Information Matrix with 990rows (docs) and 1963 cols (words)



```
<Project Source>
  datastructure
    AllWords.java
    Document.java
    FrequencyMatrix.java
    MutualInformationMatrix.java
    RawDocument.java
    Result.java
    Word.java
  gui
  manager
    FileImporter.java
    ProjectManager.java
  miner
    Apriori.java
    ItemSet.java
    Rule.java
    RuleGenerator.java
  preprocessor
    FrequencyMatrixBuilder.java
    Stemmer.java
    StopWordFilter.java
```

Figure bellow shows a rough architecture of this application:

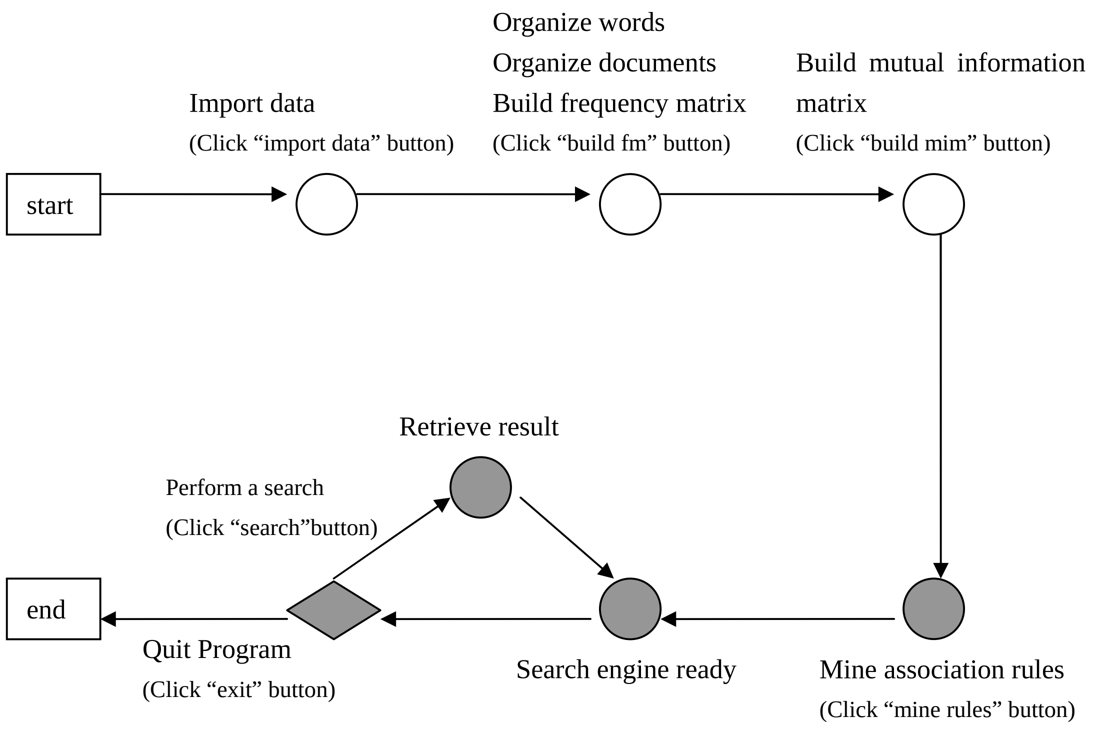


Due to complexity of code and limit on time, we do not model the performance of any class in the *gui* package. These classes consist of large number of graphic components from java.awt.* and javax.swing.* packages. These classes are provided by Sun. These robust yet complex codes pose great difficulty in performance modeling. Since GUI objects are created exactly once without any branch or loop structure, we assume they always execute in constant time and have little effect on overall performance.

We don't model the advanced part of this application, i.e. the semantic mining and query expansion functionality. These codes are very dynamic and complex. While we can assume number of words or documents, we can hardly assume the number of word associations. The number could vary from hundreds to millions.

That being assumed, we now have manager.ProjectManager as the main class residing on bottom of the modeling hierarchy, i.e. the master level (the actual main class is gui.DMProject). Methods in this class are executed in sequential order. Each method does a specific task, hence is considered a component. Once the modeling of all classes and objects are done, we may calculate the overall cost from the Computation Structure Model (CSM) [2] of manager.ProgramManager.

The manager.ProjectManager class organizes all components in a pipeline fashion such that each component uses the previous component's output as its input. The figure below shows a rough control flow graph. This graph is also known as the "driver" of the master lever, which we will discuss in section 5.5 (note that components in dark color are not modeled in this report):

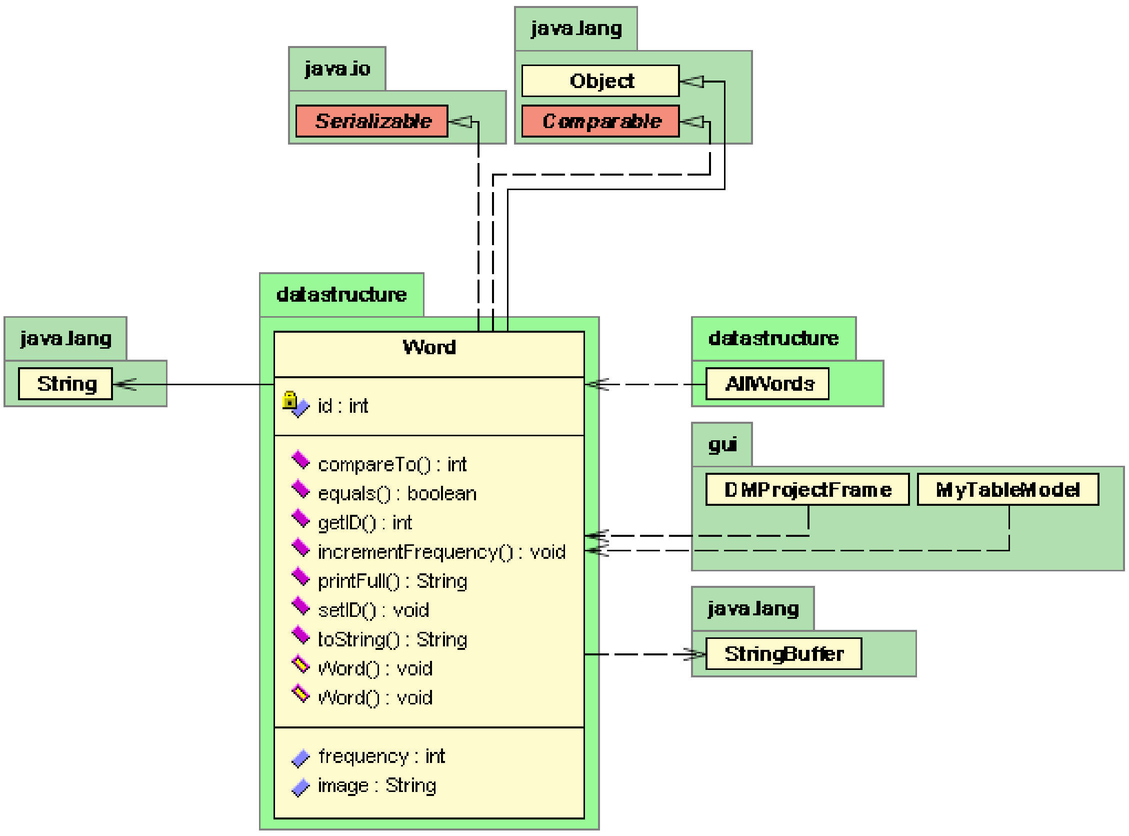


4. APPROACH

The above section shows a big picture of higher-level modeling architecture. We now discuss the approach we adopt to model each class and object.

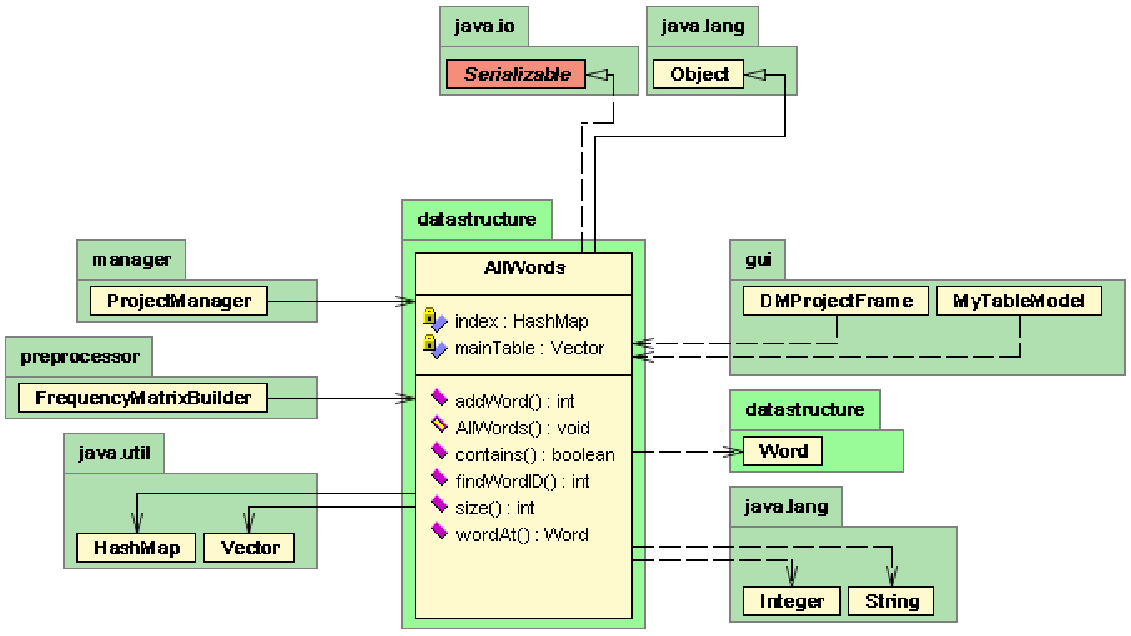
Since the input size of this application is described by the number of documents and the number of total distinct words (which mainly depends on the length of each document), our goal is to develop a performance function $f : W \times D \mapsto T$. We start from modeling the elementary classes in datastructure package, with an input size of *d* documents and *w* total distinct words.

Figure below shows the UML diagram of datastructure.Word :



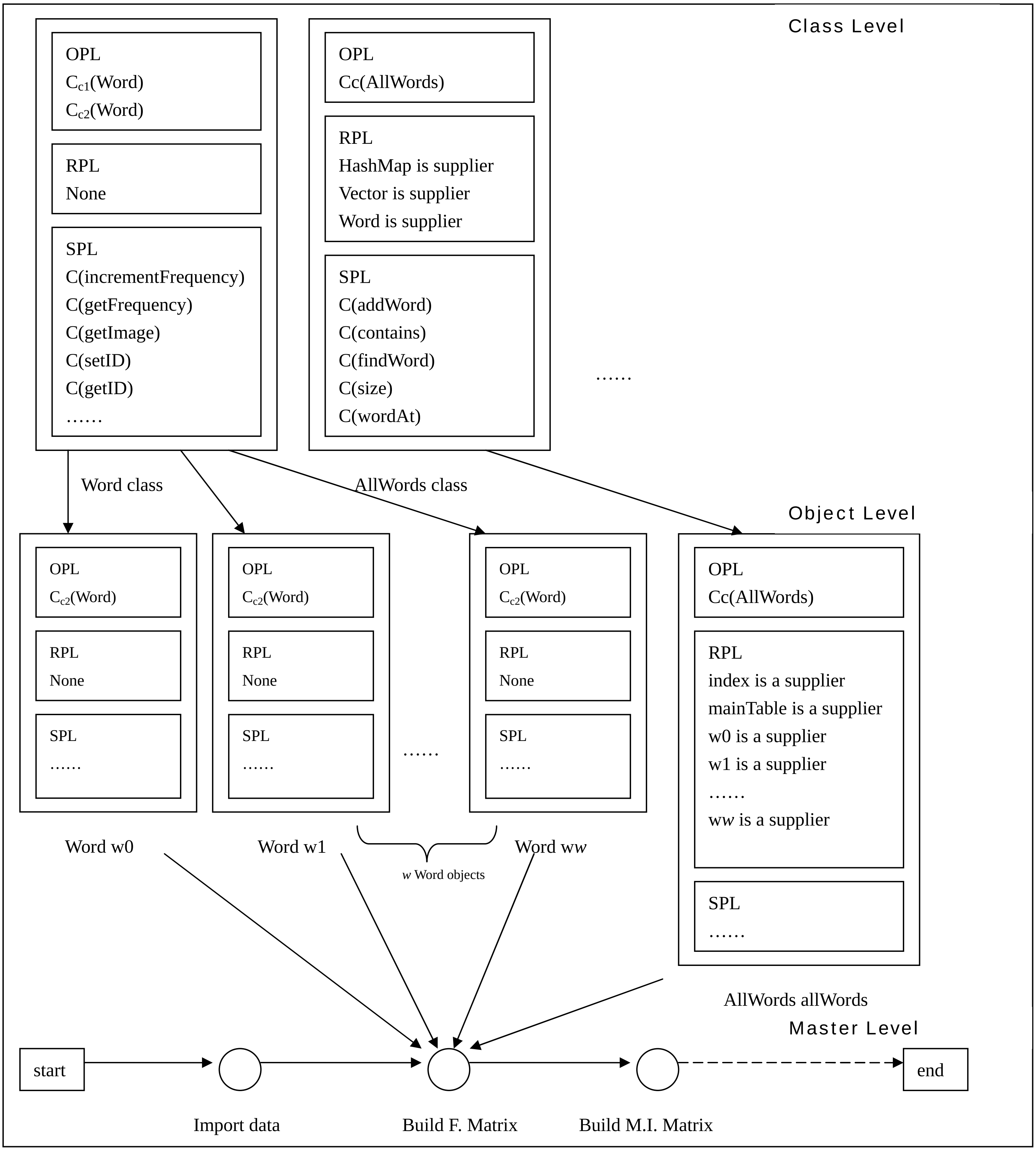
In the diagram, arrows on the left and right side with solid line denote composition relationships. Outgoing arrows denote "has" relationship meaning this class will physically create an object of the targeted class as its property; incoming arrows denote "supplies" relationship. Arrows with dotted line (weak relationship) may sometimes be ignored if one class just takes over the memory handle of another class in its method or return value instead of creating and using an object as its property. However if the class indirectly creates objects of other classes, e.g. creating objects as elements in a Vector or TreeSet, then the cost should be counted. Arrows on the top denote implement or inherit relationships. Since all Java objects implicitly inherit java.lang.Object, we ignore this relationship. Implementing any interface is also ignored because no parental constructor or method is implicitly invoked. We also treat some standard Java classes as primitive data types and assume a constant execution time, such as String and Integer. However, there are Java classes that cannot be regarded primitive, such as Vector, HashMap, TreeSet classes from java.util.* library.

The UML diagram of datastructure.AllWords reveals the composition relationship between Word and AllWords. In the application, AllWords class stores all distinct words (Word objects) in an array (Vector), and provides indexing service from the string content to the word's unique ID (the position in the array).



From the UML diagram we can easily develop a 3-layer PI for Word and AllWords by converting outgoing arrows into RPL classes. At the object level, since there are *w* numbers of distinct words, the AllWords object will create a property of type Vector with *w* elements of type Word. The partial modeling diagram concerning Word and AllWords is shown on the next page.

By illustrating the modeling of Word and AllWords classes and objects as an example, we show the agglomerative approach of modeling the whole architecture. We will replicate this process on other classes and objects and finally perform a CSM method on the main program, the ProgramManager.


Class Level
OPL
C_{c1}(Word)
C_{c2}(Word)
RPL
None
SPL
C(incrementFrequency)
C(getFrequency)
C(getImage)
C(setID)
C(getID)
……
OPL
Cc(AllWords)
RPL
HashMap is supplier
Vector is supplier
Word is supplier
SPL
C(addWord)
C(contains)
C(findWord)
C(size)
C(wordAt)
……
Word class
AllWords class
Object Level
OPL
C_{c2}(Word)
RPL
None
SPL
……
Word w0
Word w1
Word *w*w
w Word objects
OPL
Cc(AllWords)
RPL
index is a supplier
mainTable is a supplier
w0 is a supplier
w1 is a supplier
……
w*w* is a supplier
SPL
……
AllWords allWords
Master Level
start
end
Import data
Build F. Matrix
Build M.I. Matrix

5 MODELING

5.1 Assumptions and Exemptions

We assume following features of the input data:

- The input data set contains 1000 documents.
- The input data set contains 2000 distinct words.
- The input data set contains 1000 stop words which should be removed.
- 316 stop words have been defined (a, an, the, of, …… etc)
- Each word appears 5 times on average. Hence each document contains 10 words.
- The average length of each word is 5 characters.
- There is 0.3 probability that a user's search keyword does not exist in the system.

Since any class inherits *java.lang.Object*. The creation of any object implicitly creates the *Object* object. We estimate a 20ns overhead of creating any object.

We exclude following components from our modeling:

- Any class in the *gui* package. These classes use extensive java graphic interface classes and are hence complicated. These codes are executed only once during application execution. Any methods in other classes that are only called by *gui* classes are also exempted.
- The *preprocessor.Stemmer* class. This component was developed by Martin Porter [3]. It is created only once as a static object. Every incoming word is stemmed before further processing. We assume a constant 20000ns every time *Stemmer.getStemmedWord()* is called.

5.2 Estimation of Primitive Operations

Performance of primitive operations, methods of Java standard library objects are estimated using the following code:

```
long start=System.currentTimeMillis();
for(int i=0;i<1000000;i++)
{
    The statement to estimate;
}
long end=System.currentTimeMillis();
System.out.println(end-start);
```

The number printed on standard output is the execution time of 10^6 loops in millisecond, which equals the execution time of 1 loop in nanosecond. It consists of the execution time of the tested statement and the loop overhead. By executing an empty loop, we estimate that the loop overhead is 4ns.

Execution time of some primitive operations and Java library object methods is show in the following table:

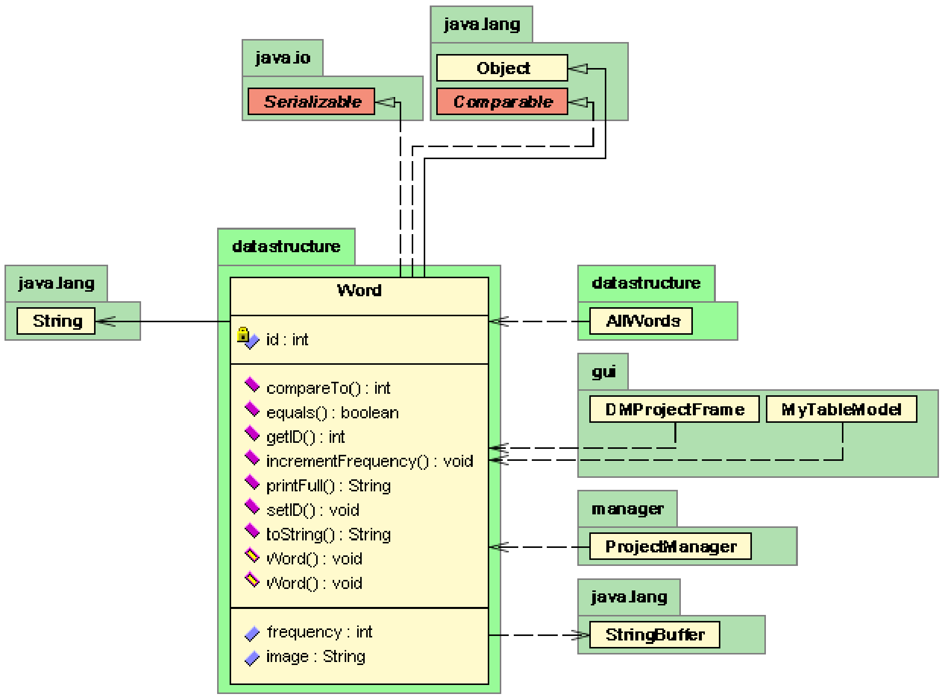
Primitive data type operations:	
Evaluation	1ns
Primitive operations and initiation of char, integer, long types	2ns
Primitive operations and initiation of float, double types	1ns
Trials (==, !=, <, >, <=, >=)	1ns
java.lang.Math	
Math.log(double n)	435ns
java.lang.String	
String concatenation	20ns
new String(String s)	58ns
string.compareTo(Object o)	30ns
string.equals(Object o)	30ns
string.toLowerCase()	142ns (on a 5-character-long string)
java.lang.Integer (int in object form)	
new Integer(int i)	27ns
integer.intValue()	12ns
java.util.StringTokenizer	
new StringTokenizer(String s)	312ns
stringTokenizer.hasMoreTokens()	57ns
stringTokenizer.nextToken()	75ns
java.util.Vector	
new Vector()	180ns
vector.add(Object o)	1000ns
vector.elementAt(int i)	60ns
vector.size()	27ns
java.util.TreeSet	
new TreeSet()	200ns
treeset.add(Object o)	3060ns
java.util.TreeMap	
new TreeMap()	74ns
treemap.containsKey(Object key)	300ns (estimated average on size 1-1000)
treemap.get(Object key)	340ns (estimated average on size 1-1000)
treemap.put(Object key, Object o)	420ns (estimated average on size 1-1000)
treemap.keySet()	11ns

java.util.HashSet	
new HashSet()	371ns
hashset.contains(Object o)	66ns
java.util.HashMap	
new HashMap()	276ns
hashMap.containskey(Object key)	90ns
hashMap.get(Object key)	110ns
hashMap.put(Object key, Object o)	126ns
java.io.FileReader	
new FileReader(String filename)	8000000ns (observed on a 100k file)
java.io.BufferedReader	
new BufferedReader(FileReader fr)	60000000ns (observed on a 100k file)
bufferedreader.readLine()	1377ns
bufferedreader.ready()	89ns

Dynamic binding overhead is estimated by calling an empty method associated to an object. The estimated value is 1ns.

5.3 Performance Image (PI) at Class Level

5.3.1 Class *datastructure.Word*



Cost of constructor: *public Word(String image)*

58+1+1+1+20=81ns

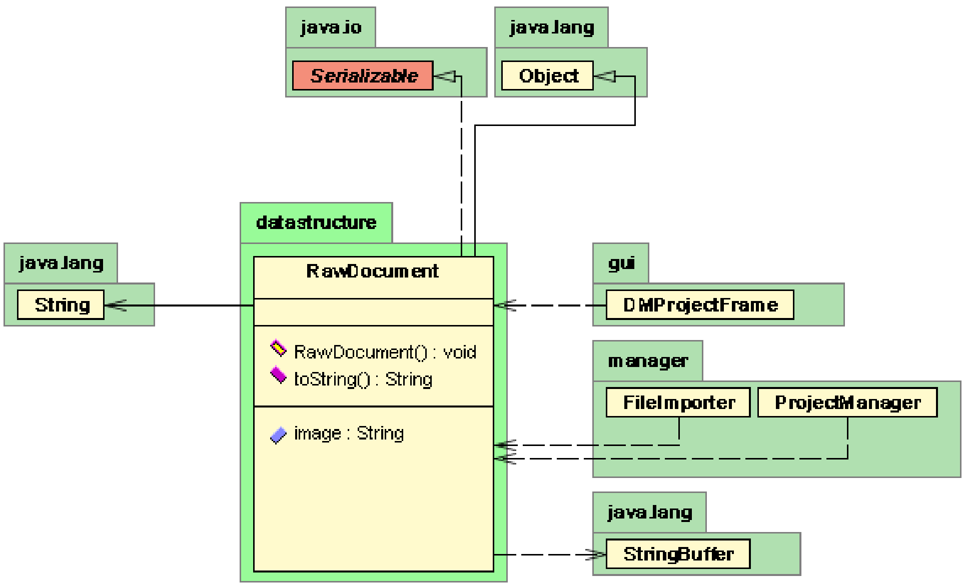
Cost of constructor: *public Word(String image, int frequency)*

58+1+1=60ns

A summary of above analysis, and cost of other simple methods is shown below:

Object-Creation Performance Layer (OPL) Cc1(Word)=81ns Cc2(Word)=80ns
Related-Objects Performance Layer (RPL) Supplies AllWords and ProjectManager
Service Performance Layer (SPL) C(getImage)=1ns C(incrementFrequency)=1ns C(getFrequency)=1ns C(setID)=1ns C(getID)=1ns C(compareTo)=43ns C(equals)=43ns

5.3.2. Class *datastructure.RawDocument*



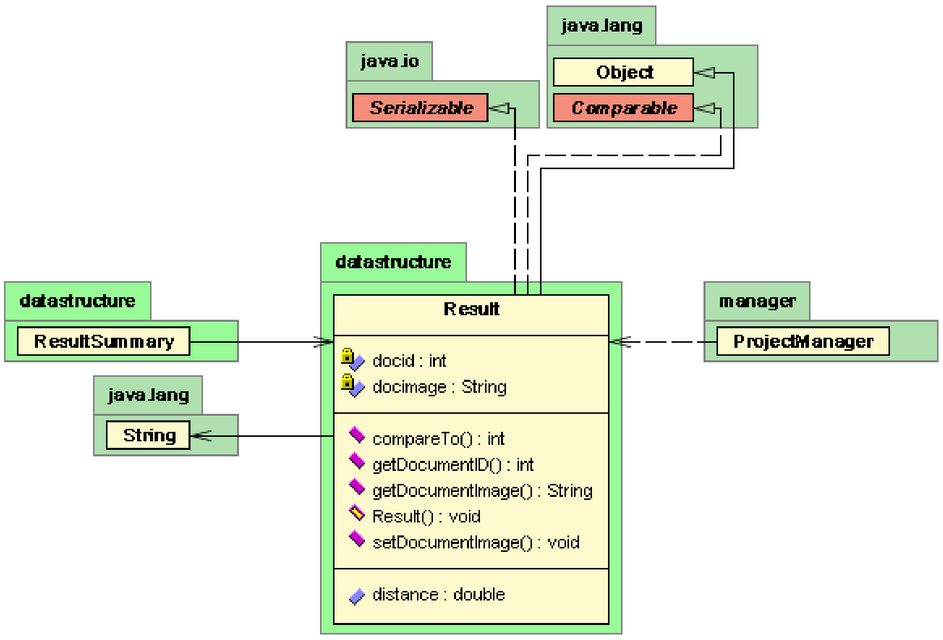
Cost of constructor: *public RawDocument(String image)*

1+1+1+20=23ns

A summary of above analysis, and cost of other simple methods is shown below:

Object-Creation Performance Layer (OPL) Cc1(RawDocument)=23ns
Related-Objects Performance Layer (RPL) Supplies FileImporter and ProjectManager
Service Performance Layer (SPL) C(getImage)=1ns

5.3.3. Class *datastructure.Result*

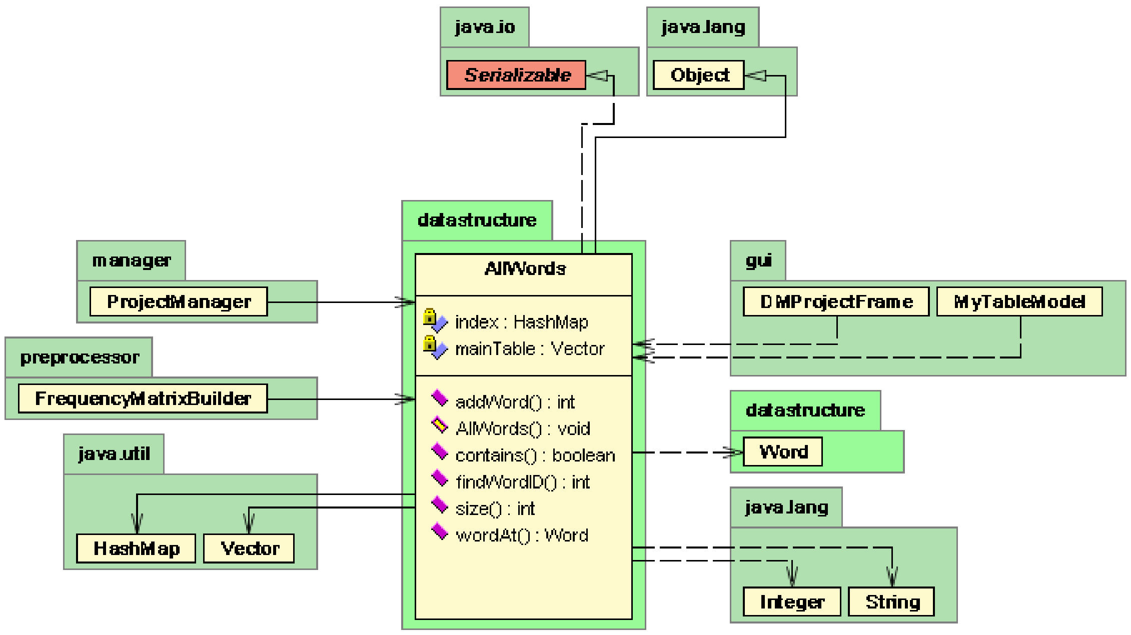


Cost of constructor: *public Result(int docid, double distance)*
1+1+1+1+1+1+20=26ns

A summary of above analysis and cost of other simple methods is shown below:

Object-Creation Performance Layer (OPL) Cc1(Result)=23ns
Related-Objects Performance Layer (RPL) Supplies ResultSummary and ProjectManager
Service Performance Layer (SPL) C(getDistance)=1ns C(getDocumentID)=1ns C(getDocumentImage)=1ns C(setDocumentImage)=1ns

5.3.4. Class *datastructure.AllWords*

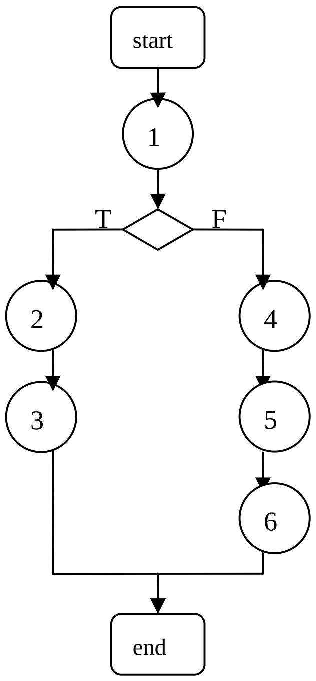


Cost of constructor: *public AllWords()*

180+276+20(object creation overhead)=476ns

Cost of method: *public int addWord(String newword)*

Control Flow Graph:



Cost of each statement:

1: 90+1(dynamic binding)=91ns

2: 110+1(dynamic binding)+12+1(dynamic binding)+1=125ns

3: 60+1+C(Word.incrementFrequency)=62ns

4: 27+1(dynamic binding)+1=29ns

5: 126+1(dynamic binding)+27+1=155ns

6: 1000+1(dynamic binding)+Cc2(Word)=1083ns

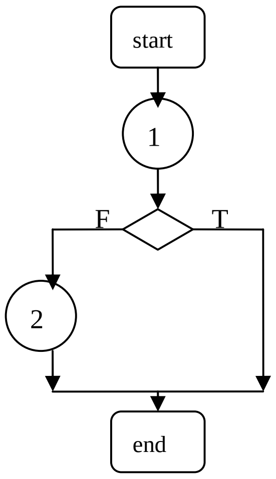
Probability of test 1 result:

True: 0.8

False: 0.2

C(addWord)=91+0.8(125+62)+0.2(29+155+1083)=494ns

Cost of method: *public int findWordID(String word)*
Control Flow Graph:



Cost of each statement:
1: 110+1(dynamic binding)+1=111ns
2: 110+1(dynamic binding)+12+1(dynamic binding)=124ns

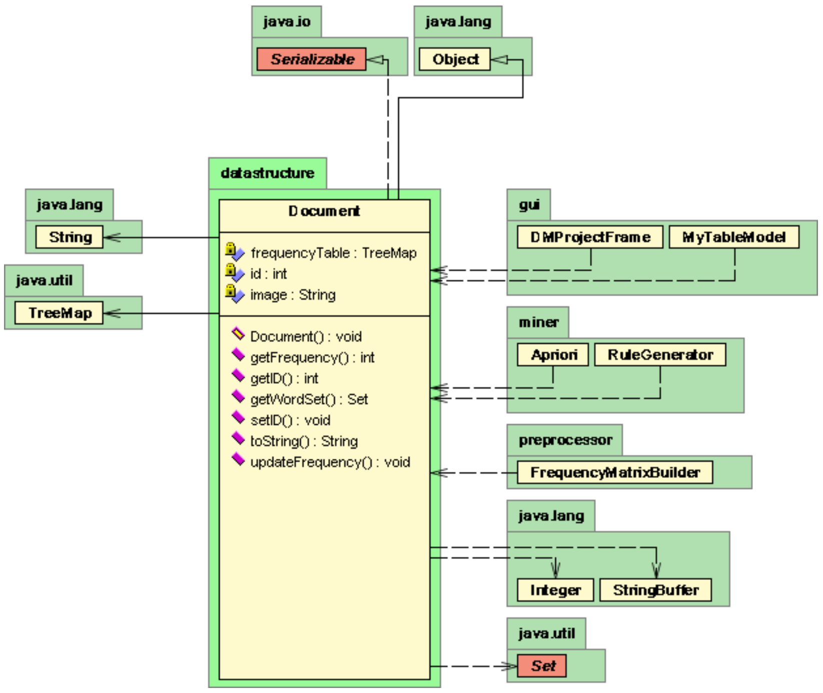
Probability of test 1 result:
True: 0.3
False: 0.7

C(findWordID)=111+0.7(124)=198ns

A summary of above analysis, and cost of other simple methods is shown below:

Object-Creation Performance Layer (OPL) Cc(AllWords)=476ns
Related-Objects Performance Layer (RPL) Supplies ProjectManager and FrequencyMatrixBuilder
Service Performance Layer (SPL) C(addWord)=494ns C(wordAt)=61ns C(contains)=91ns C(size)=28ns C(findWordID)=198ns

5.3.5. Class *datastructure.Document*

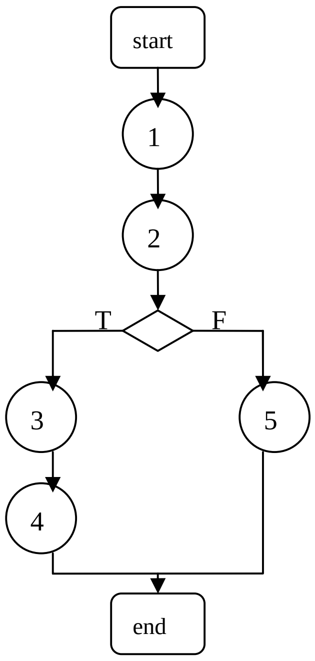


Cost of constructor: *public Document(String newimage)*

1+1+74+20(object creation overhead)=96ns

Cost of method: *public void updateFrequency(int wid)*

Control Flow Graph:



Cost of each statement:

1: 27+1=28ns

2: 300+1(dynamic binding)=301

3: 340+1(dynamic binding)+12+1(dynamic binding)+1=355ns

4: 1+27+420+1(dynamic binding)=444ns

5: 27+420+1(dynamic binding)=443ns

Probability of test 2 result:

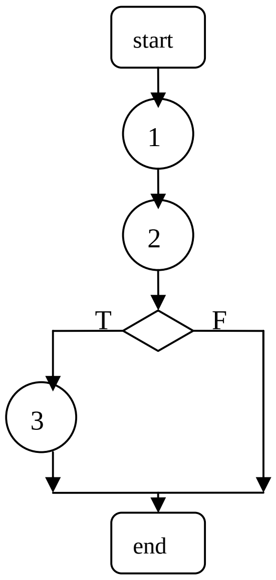
True: 0.8

False: 0.2

C(updateFrequency)=28+301+0.8(355+444)+0.2(443)=1057ns

Cost of method: *public int getFrequency(int wid)*

Control Flow Graph:



Cost of each statement:

1: 27+1=28ns

2: 300+1(dynamic binding)=301

3: 340+1(dynamic binding)+12+1(dynamic binding)+1=355ns

Probability of test 2 result:

True: 0.005

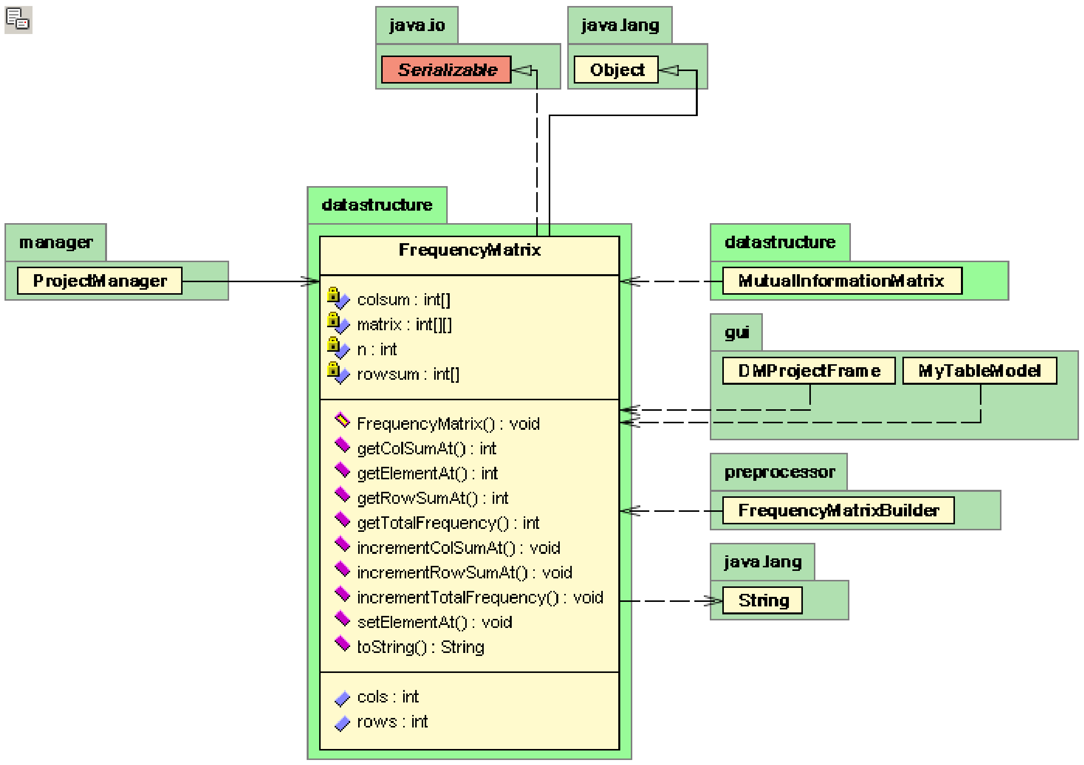
False: 0.995

C(getFrequency)=28+301+0.005(355)=331ns

A summary of above analysis, and cost of other simple methods is shown below:

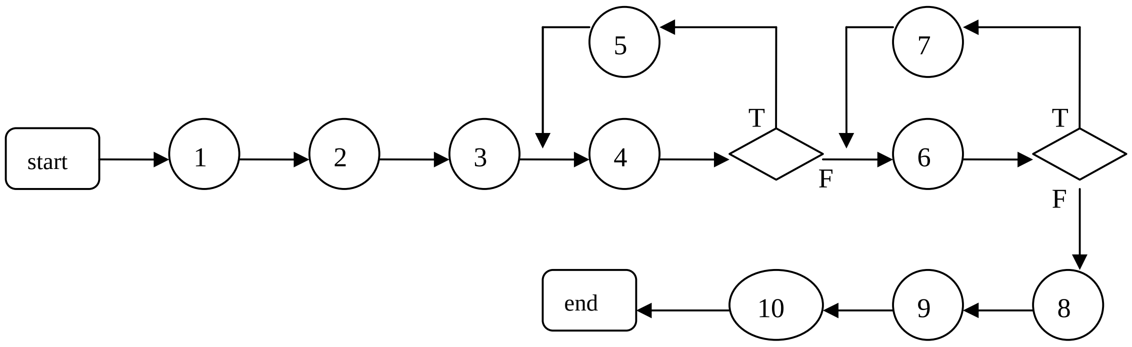
Object-Creation Performance Layer (OPL) Cc(Document)=96ns
Related-Objects Performance Layer (RPL) Supplies FrequencyMatrixBuilder
Service Performance Layer (SPL) C(updateFrequency)=1057ns C(getFrequency)=331ns C(getWordSet)=12ns C(setID)=1ns C(getID)=1ns

5.3.6. Class *datastructure.FrequencyMatrix*



Cost of constructor *public FrequencyMatrix(int rows, int cols)*

Control Flow Graph:



Cost of each statement:

1: 2000*1000+1=2000001ns

2: 1000+1=1001ns

3: 2000+1=2001ns

4: 4ns (loop overhead)*1000=4000ns

5: Constant distribution: 1000ns

6: 4ns (loop overhead)*2000=8000ns

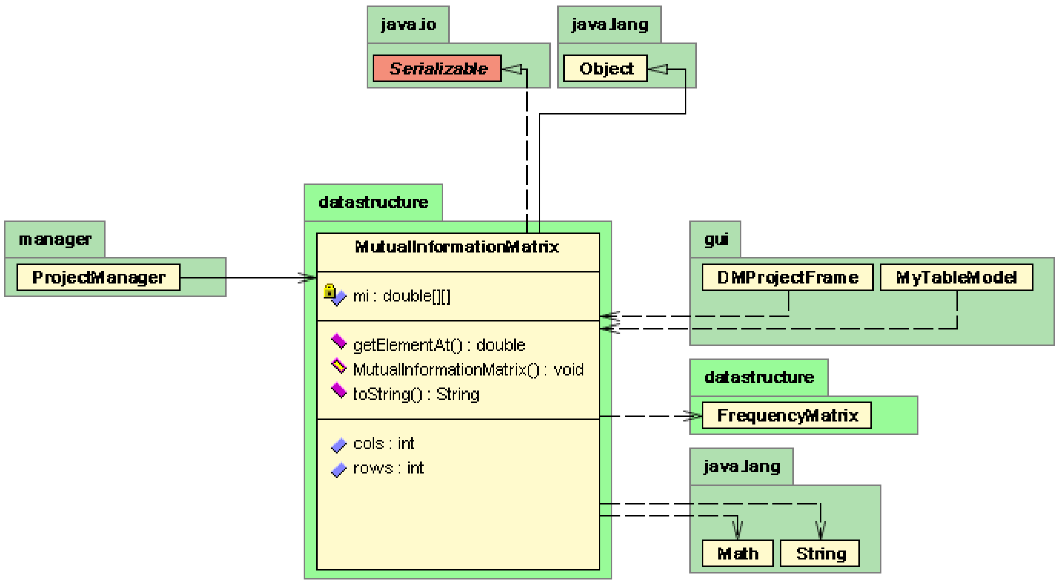
7: Constant distribution: 2000ns

8: 1ns
9: 1ns
10: 1ns
Cc(FrequencyMatrix)=2000001+1001+2001+4000+1000+8000+2000+1+1+1+20(object creation overhead)=2018026ns

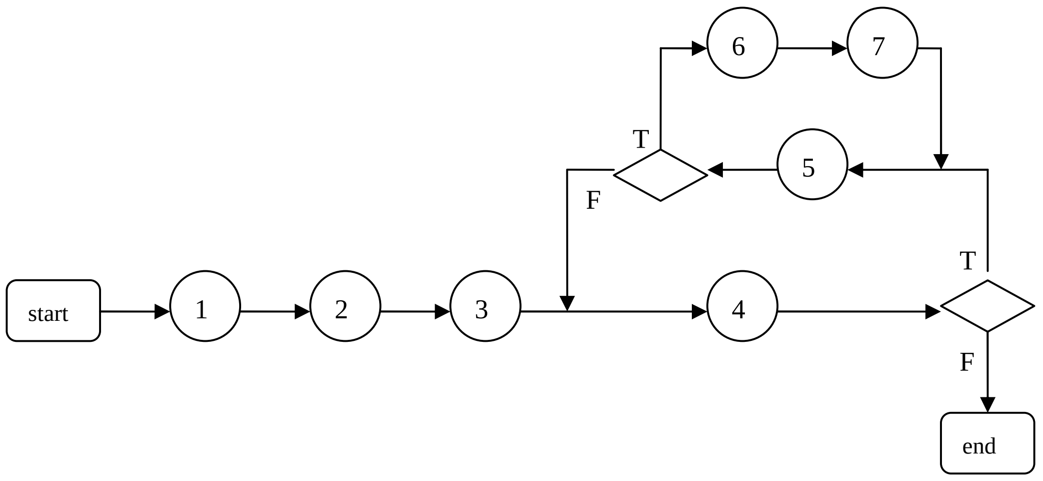
A summary of above analysis, and cost of other simple methods is shown below:

Object-Creation Performance Layer (OPL) Cc(FrequencyMatrix)=2018026ns
Related-Objects Performance Layer (RPL) Supplies ProjectManager, FrequencyMatrixBuilder and MutualInformationMatrix
Service Performance Layer (SPL) C(setElementAt)=2ns C(incrementRowSumAt)=2ns C(incrementColSumAt)=2ns C(incrementTotalFrequency)=2ns C(getElementAt)=1ns C(getRowSumAt)=1ns C(getColSumAt)=1ns C(getRows)=1ns C(getCols)=1ns C(getTotalFrequency)=1ns

5.3.7. Class *datastructure.MutualInformationMatrix*



Cost of constructor: *public MutualInformationMatrix(FrequencyMatrix fm)*
Control Flow Diagram:

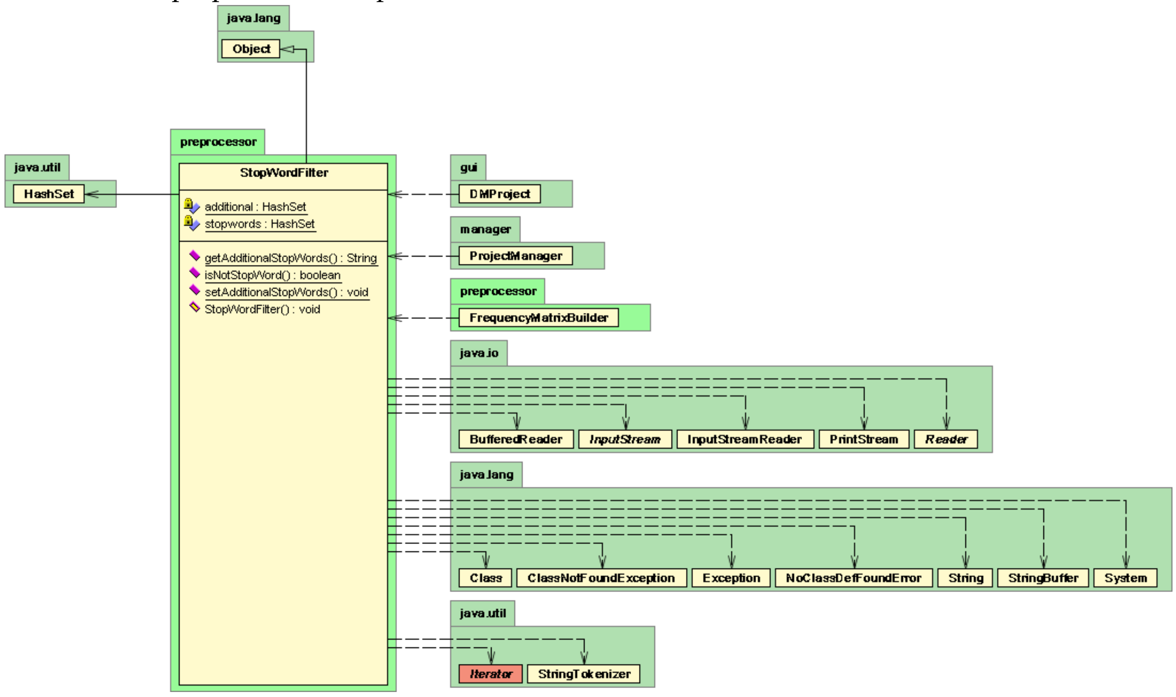


Cost of each statement:
1: 1+1(dynamic binding)+1=3ns
2: 1+1(dynamic binding)+1=3ns
3: 2000*1000+1=2000001ns
4: Constant distribution: 4*1000=4000ns
5: Constant distribution: 4*1000*2000=8000000ns
6: Constant distribution: (435+(2+2+1)+1+(2+2+1)+1+(2+2+1))*1000*2000
=904000000ns
7: Constant distribution: 3*2000*1000=6000000ns
Cc(MutualInformationMatrix)=920004007ns

A summary of above analysis, and cost of other simple methods is shown below:

Object-Creation Performance Layer (OPL) Cc(MutualInformationMatrix)=920004007ns
Related-Objects Performance Layer (RPL) Has FrequentMatrix Supplies ProjectManager
Service Performance Layer (SPL) C(getElementAt)=1ns C(getRows)=1ns C(getCols)=1ns

5.3.8. Class *preprocessor.StopWordFilter*



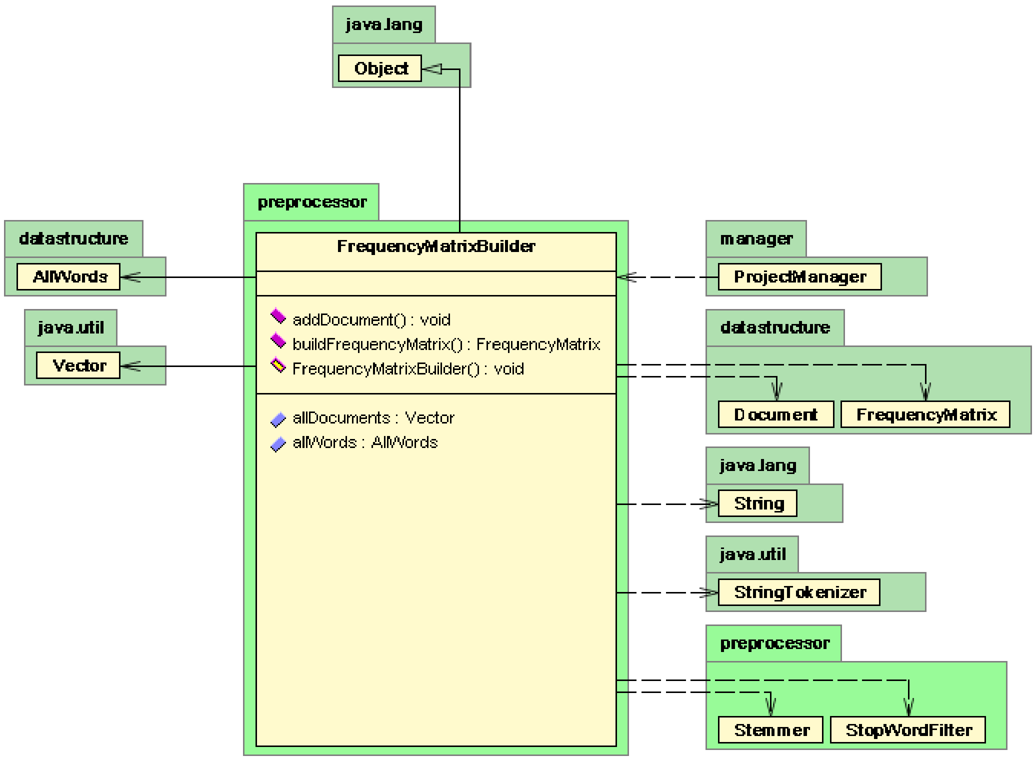
There is only one instance of this class created in the whole system. The instance is created together with GUI objects at the launching phase. Therefore we don't model the constructor of this class. The only method we need to model is a static method that checks if a word is a stop word.

Cost of method *public static boolean isNotStopWord(String word)*
The cost is estimated by observing the *hashset.contains(String s)* method with a vector size of 316 (we have 316 stop words) plus some dynamic binding overhead.

A summary of above analysis, and cost of other simple methods is shown below:

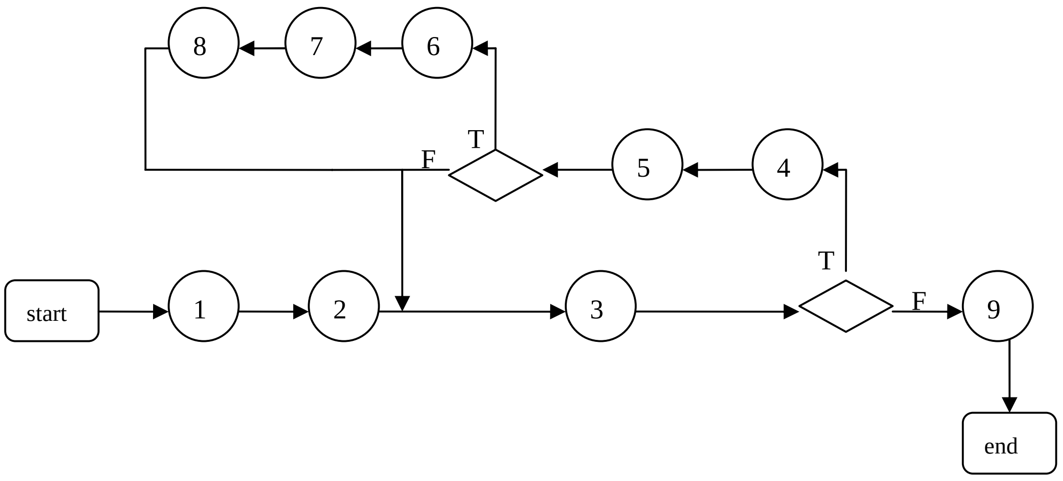
Object-Creation Performance Layer (OPL) Not required
Related-Objects Performance Layer (RPL) Supplies ProjectManager and FrequentMatrixBuilder
Service Performance Layer (SPL) C(isNotStopWord)=67ns

5.3.9. Class *preprocessor.FrequencyMatrixBuilder*



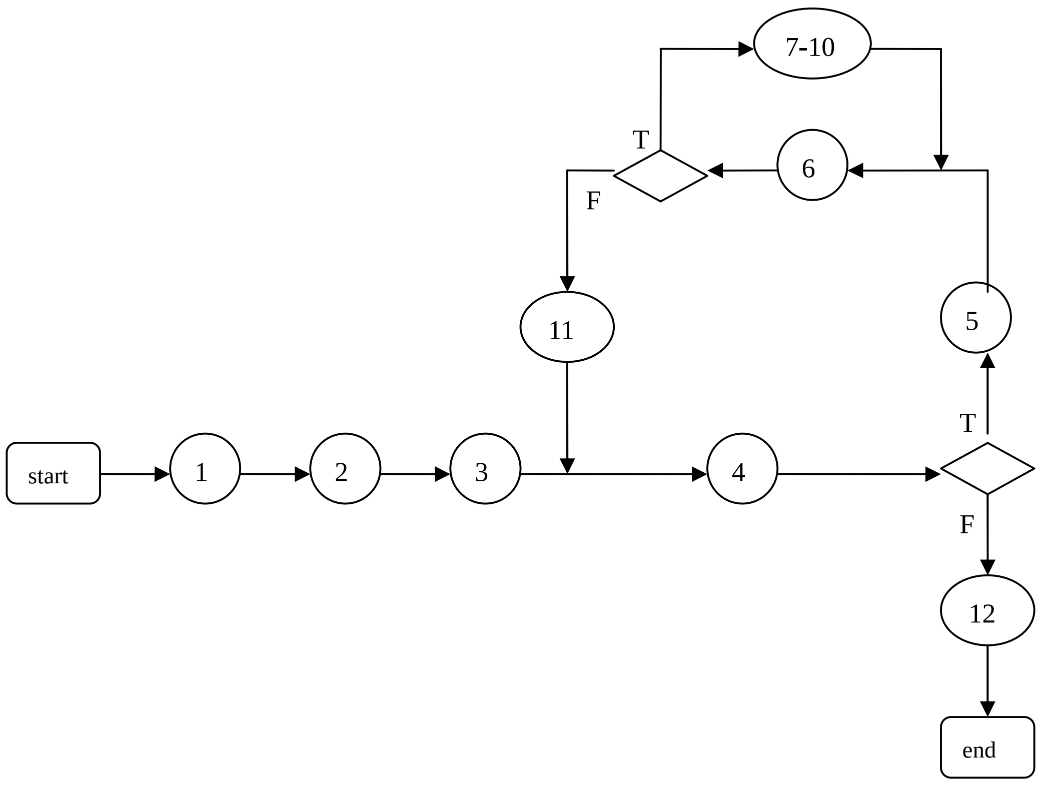
Cost of constructor: *public FrequencyMatrixBuilder()*
Cc(FrequencyMatrixBuilder)=Cc(AllWords)+1+180+1+20(object creation overhead)
=476+1+180+1+20=678ns

Cost of method: *public void addDocument(String text)*
Control Flow Graph:



Cost of each statement:
1: Cc(Document)+1=97ns
2: Cc(StringTokenizer)+1=313ns
3: Modified geometric distribution (treated as constant distribution since average document length is 11): 12*(57+1(dynamic binding))=696ns
4: 11*(75+1(dynamic binding)+142+1(dynamic binding)+1)=2409ns
5: 11*(67+1(dynamic binding))=748ns
The inner “if” trial has a probability of 10/11 being true
6,7,8: Binomial distribution: 10/11(20002+496+1058)=19596ns
9: 1000+1(dynamic binding)=1001ns
C(addDocument)=97+313+696+2409+748+19596+1001=24860ns

Cost of method: *public FrequencyMatrix buildFrequencyMatrix()*
Control Flow Graph:



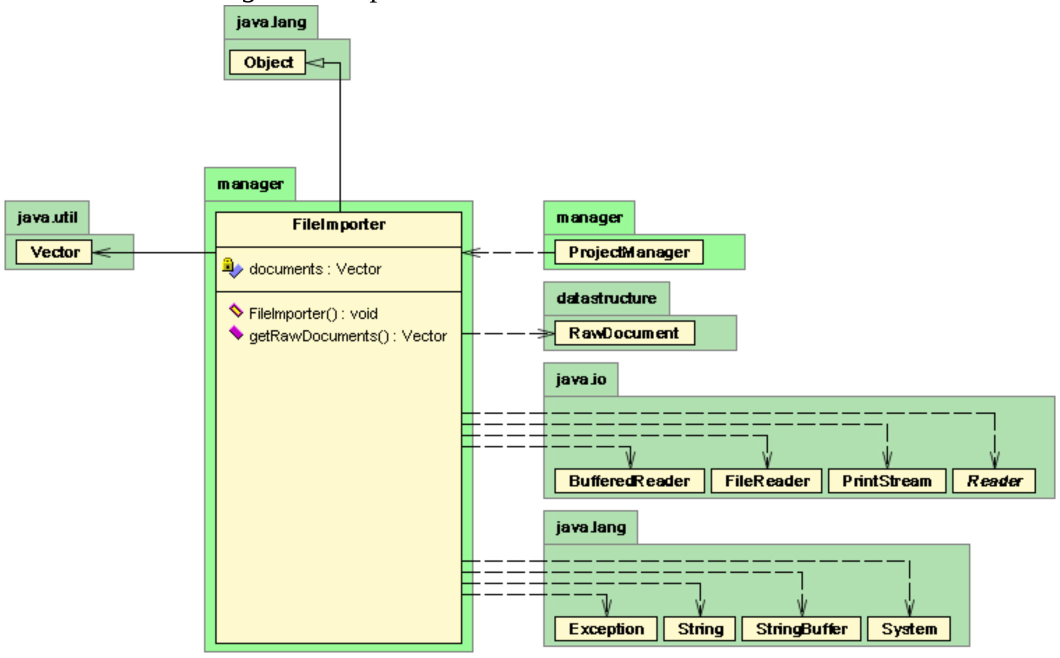
Cost of each statement:
1: 27+1(dynamic binding)+1=29ns
2: 27+1(dynamic binding)+1=29ns
3: Cc(FrequencyMatrix)+1=2018027ns
4: 4(loop overhead)*1000=4000ns
5: Constant distribution: 1000(60+1(dynamic binding)+1)=62000ns

6: 1000(4(loop overhead)*2000)=8000000ns
7-10: Constant distribution: 2000000(2+3+3+3)=22000000ns
11: Constant distribution: 1000(2+1(dynamic binding)+1+1(dynamic binding))=5000ns
12: 1ns
C(buildFrequencyMatrix)=29+29+2018027+4000+62000+8000000+22000000+5000+1
=32089086ns

A summary of above analysis, and cost of other simple methods is shown below:

Object-Creation Performance Layer (OPL) Cc(FrequencyMatrixBuilder)=678ns
Related-Objects Performance Layer (RPL) Has Stemmer, StopWordFilter, Document, FrequencyMatrix, AllWords Supplies ProjectManager
Service Performance Layer (SPL) C(addDocument)=24860ns C(buildFrequencyMatrix)=32089086ns C(getAllDocuments)=1ns C(getAllWords)=1ns

5.3.10. Class *manager.FileImporter*



Cost of constructor: *public FileImporter(int mode, String filename)*
Note that *mode* is always 0 in this version of the application, meaning the *FileImporter* reads in every line of a file as a *RawDocument*. (*mode*=1 will treat every file as a document ---not implemented.) Therefore the branching never takes effect. The code in the constructor is executed in sequential order. The *while(in.ready())* forms a constant distribution of 1000 iterations since there are 1000 lines or documents.
Cc(FileImporter)=180+1+800000+6000000+1
+1001(89+1)+1000(1377+1+Cc(RawDocument)+1000+1)+20(object creation overhead)
=9292291ns

A summary of above analysis, and cost of other simple methods is shown below:

Object-Creation Performance Layer (OPL)
Cc(FileImporter)=9292291ns
Related-Objects Performance Layer (RPL)
Has RawDocument Supplies ProjectManager
Service Performance Layer (SPL)
C(getRawDocuments)=1

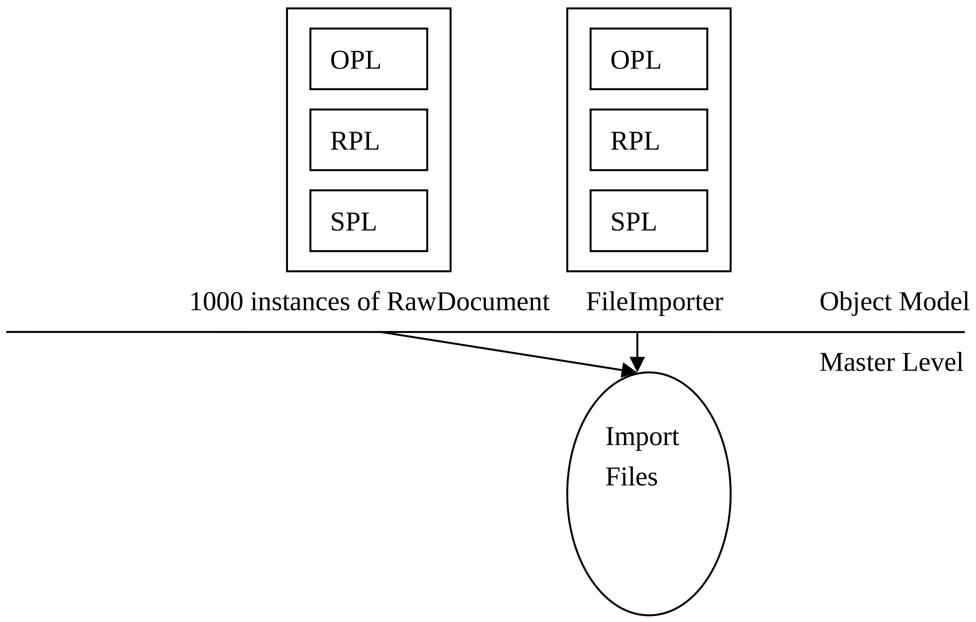
5.4 Performance Image (PI) at Object Level and Master Level

The central class of this application is *manager.ProjectManager*. This class is manipulated by a bunch of graphic interface classes. The main program, *gui.DMProject*, launches the root of the graphic interface class, *gui.DMProjectFrame*.

However, in this analysis, we consider part of the *manager.ProjectManager* class as our "main program". The modeled classes, as shown in section 5.3, form components in this main program. These components form the master level in our modeling. They are executed in sequential order without any branching or looping.

5.4.1 Import File Component

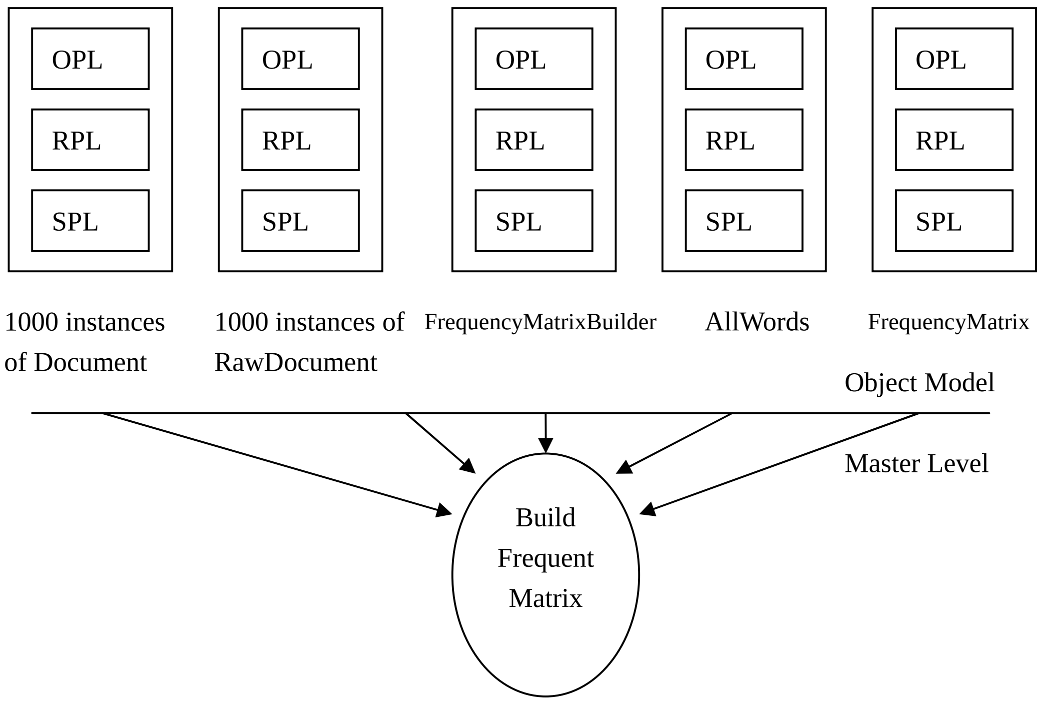
(i.e. method *public void importFromFile(int mode, String filename)*)



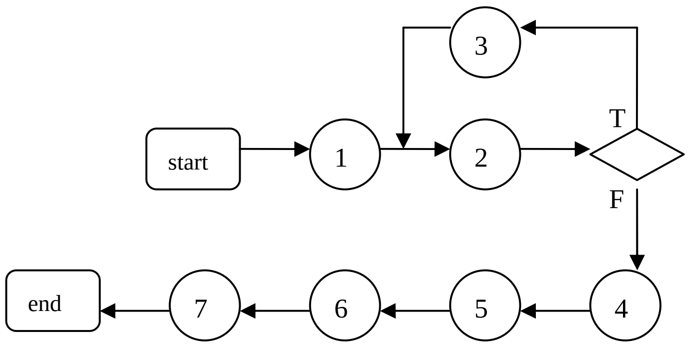
Cc	Co	C_{local}
Cc(FileImporter)=9292291ns	C(getRawDocuments)=1ns	C(assign_stage)=1ns
		C(assign_allDocumentsRaw) =1ns

The cost of this component is 9292294ns

5.4.2. Build Frequency Matrix Component
(i.e. method *public void buildFrequencyMatrix()*)



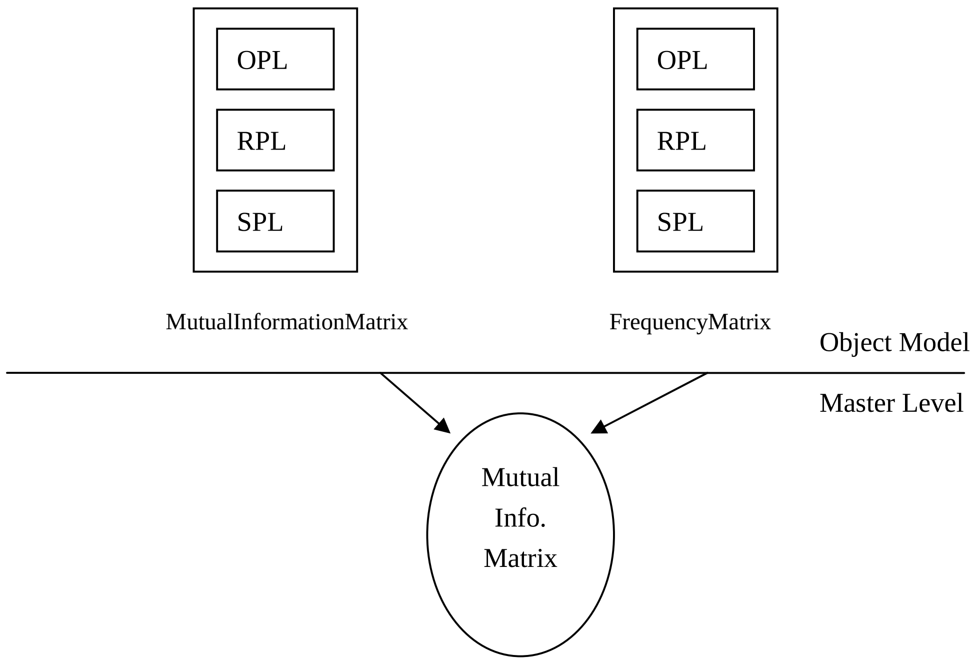
Control Flow Graph of this component:



Cc	Co	C_{local}
Cc(FrequencyMatrixBuilder)=678ns	1000C(addDocument)=24860000ns	1000C(vector.elementAt)=6000
	1000C(getImage)=1000ns	1000C(loop overhead)=4000ns
	C(buildFrequencyMatrix)=32089086ns	C(assign_fm)=1ns
	C(getAllDocuments)=1ns	C(assign_allDocuments)=1ns
	C(getAllWords)=1ns	C(assign_allWords)=1ns
		C(assign_stage)=1ns

The cost of this component is 57014770ns

5.4.3. Build Mutual Information Matrix Component
(i.e. method *public void buildMutualInformationMatrix()*)



Cc	Co	C_{local}
Cc(MutualInformationMatrix) =90004007ns		C(assign_stage)=1ns
		C(assign_mim)=1ns

The cost of this component is 920004009ns

5.5 Overall Performance

As shown in section 3, the driver (the main program) executes the import data component, the build frequency matrix component, and the build mutual information matrix component in a pipeline fashion. Each component is executed after the previous one. The total cost is the sum of the cost of the three components:

9292294+57014770+920004009=986311073ns

6. CONCLUSION AND DISCUSSIONS

In this report, we model and analyze the performance of the 3 major components of the application using the hierarchical performance image (PI) approach. UML diagrams are found helpful in identifying class composition relations and building hierarchical performance images. The total cost we estimate is 986311073ns. The application is supposed to take this long time to import data file, build frequency matrix (together with a number of word/document lists and indices) and build mutual information matrix. This is the time the application needs to make the basic exact-match searching function ready. The advanced functionality, e.g. preparation for semantic mining and query expansion, are not analyzed in this report.

We also notice following limitations of our modeling process:

- We estimate primitive operations by putting it in a loop and observe the cost of the loop. This could not be accurate since Java compiler may optimize the loop, resulting in that the statement in the loop is not executed as many times as we specified. This is the only way we know to estimate cost of primitive operations since the monitoring methods provided by Java are not fine-grained at nanosecond level.
- We make extensive estimations on Java standard library classes. These classes could be precisely modeled by the hierarchical PI approach, too.
- Methods of some Java standard classes, usually collection classes in *java.util* package, are highly dependent on data size and structure. In our report, we simply estimate the average cost of these methods using the assumed data size. However, performance functions could be derived by modeling and analyzing these methods.

7. REFERENCES

[1] A. Abdel-Raouf, R. Ammar and T. Fergany, “Performance-Based Modeling for Object-Oriented Software”, submitted to the 16th International conference on Computer Applications in Industry and Engineering, Las Vegas, Nevada , Nov. 2003.
[2] T. Booth, “Use of Computation Structure Models to Measure Computation Performance”, Proc. of Conference on simulation, Measurement, and Modeling
of Computer Systems, Boulder, CO, August 1979.
[3] M. Porter, An algorithm for suffix stripping, Program, Vol. 14, no. 3, pp 130-137, 1980.