**Beer Sheva Campus**

**Summary Evaluation Platform**

**Master's Degree Final Project**

**Submitted in the Department of Software Engineering**

**Sami Shamoon College of Engineering**

**Name: Sergey Mordeev**

**ID: 321175879**

**Project Leader: Dr. Marina Litvak**

Contents

[Abstract 4](#_Toc6328329)

[1. Introduction 4](#_Toc6328330)

[2. Text Summarization Task 5](#_Toc6328331)

[3. Summary Evaluation 6](#_Toc6328332)

[4. Metrics 7](#_Toc6328333)

[4.1. Implementation Forewords 8](#_Toc6328334)

[4.2. Rouge 10](#_Toc6328335)

[4.3. AutoSummENG 13](#_Toc6328336)

[4.4. Readability Metrics 13](#_Toc6328337)

[5. Platform as Library 15](#_Toc6328338)

[5.1. Chosen Approach 16](#_Toc6328339)

[5.2. Domain-Specific Language 20](#_Toc6328340)

[5.3. DI and IoC 28](#_Toc6328341)

[5.4. High Level Architecture 32](#_Toc6328342)

[5.4.1. Concurrency 33](#_Toc6328343)

[5.4.2. Immutability 35](#_Toc6328344)

[5.4.3. Convention 37](#_Toc6328345)

[5.5. Platform as Library – Conclusion 40](#_Toc6328346)

[6. Platform as GUI Client 41](#_Toc6328347)

[6.1. Prerequisites 41](#_Toc6328348)

[6.1.1. R Language Libraries 42](#_Toc6328349)

[6.2. Installation & Execution 43](#_Toc6328350)

[6.2.1. Error Analyzing – Technical Notes 44](#_Toc6328351)

[6.2.2. Source Code Compilation – Advanced 46](#_Toc6328352)

[6.3. Input Preparation 47](#_Toc6328353)

[6.4. Running Metrics Evaluation 48](#_Toc6328354)

[6.5. Analyzing Metric Results 50](#_Toc6328355)

[6.5.1. Analyzing Screen Sections – Overview 53](#_Toc6328356)

[6.5.2. System Section 53](#_Toc6328357)

[6.5.3. Notched Box Graph 53](#_Toc6328358)

[6.5.4. Topics Section 53](#_Toc6328359)

[6.5.5. Metric Heat 53](#_Toc6328360)

[6.5.6. Result Actions 53](#_Toc6328361)

[7. Conclusions 53](#_Toc6328362)

[References 54](#_Toc6328363)

# Abstract

We present here a platform for summary evaluation metrics. The purpose of such platform is to be a useful tool for the researchers in the field of text summarization. Although summary evaluation metrics come with appropriate tools (scripts) and the result could be analyzed within different data manipulation tools as Microsoft Excel or even with languages as R but taking the evaluation process under one umbrella will simplify and automate the analysis of the summary systems by our believing.

# 1. Introduction

For a person working within some field the time has a crucial meaning. No one of us wants spending a time on the same tasks again and again. The purpose of many systems is to automate such the process. For instance, no one expects that bank teller will run SQL queries to update the deposit. As well, no one expects from us working only with command line while solving our problems on computer. Today we have a plenty of tools for automating different kinds of tasks. We believe that researchers in text summarization should not be an exception. Thus, a tool for automating researcher tasks could help her for the better understanding and evaluating the results.

The word “platform” in software world has different meanings. We think that most of us agree that when one says “cross-platform software” her meaning is that the software is not dependent on the current operating system and/or chosen hardware. The expectations in such case are that the software will work similarly with a variety of operating systems or hardware. From the other hand, when one says, for instance “we have an enterprise office platform software” the meaning in such case is not related neither to operating system nor to the APIs of such office software. The meaning is the group of programs for solving a particular business’ issues. As another example, when the programmers say that they chosen a platform that might mean that the meaning of such is a software library or framework. The word “platform” in software is highly context dependent.

To make things less ambiguous, the word platform in our case should be defined. When we say platform, we mean:

1. Concrete GUI application that evaluates several metrics for the text summaries
2. Command line tool if one needs to automate his tasks
3. A software library that could be used by 3rd parties

To be honest, the project had started as the library in mind and latter it morphed into much more massive GUI application. Hence, we believe that it was a good migration since the library and the API were tested by ourselves. We cannot argue that the core library for metric evaluation is one of the best. However, we can argue that for us it was a right vision both for the code management and modeling the process of evaluation itself. The core library will be explained within this text for those who would like to use the platform as the library.

The text will be divided into a couple of part. The first part will describe the task of summary evaluation in general. The second part will give the overview of the program internally. The third part will explain the usage of the GUI application. The last part will be the experiments with the system. Although we are trying to not make the interdependencies within these parts but for better understanding of the system reading it in novel form give the better picture.

# 2. Text Summarization Task

The purpose of this section is to give an overview of what summarization process is from the flying bird point of view. Such need exists because the platform has been developed under several assumptions. No software in the world exists without taking assumptions. This piece of software should not be an exception too. From the one hand each of us wants software for all possible case in life. From the other hand, the practice says that not taking assumptions can finally bring the programming task either to the never-ending story or since it is unclear what restrictions are the software becomes bloated; program features are unclear; the code base in pursuit of generalization becomes unmaintainable and full of bugs. Thus, restrictions and/or assumptions should be taken to produce a valuable software within a given time limit.

Text summarization task depicts itself as taking a summary from the given source(s). More formally the summary could be defined as a *text that is produced from one or more texts, that contains a significant portion of the information in the original text(s), and that is no longer than half of the original text(s)* and text summarization as *the process of distilling the most important information from a source (or sources) to produce an abridged version for a particular user (or user) and task (or tasks)* (Elena Lloret, 2008). Text summarization in general could be divided into the single document or multi document. As well, the language should be considered too when we are talking about automatic text summarization (i.e. produced by machines) (Elena Lloret, 2008). It is also important to understand what formats automatic summarizer can produce and consume. These facts are important since they directly influence on the summary evaluation.

It should be mentioned that summarization task is vital in current over-information (Giannakopoulos, 2008). For instance, the development of World Wide Web brings us to state where for the person it is not possible to read every page completely (Elena Lloret, 2008). Today “simple” search in web search engine will result in short summary, see Figure 1. (It seems that it is a simplest possible summary, but it is a summary. For Google Scholar it looks like it takes an abstract of an article)



Figure 1.

As it is already said in this section those details are reflected as assumptions and crystalized as requirements in the developed platform directly. For instance, the platform expects that all documents either an original document(s) or the summary itself are a simple text encoded by UTF-8 and separated each one in its file. From the one hand it may require the preprocessing and/or after the summarization but from the other hand simplicity matters. Processing XML or other marked up format will require additional complication even for the end user to prepare documents in the needed format. As well, UTF-8 today is the de-facto standard for simple text. Our opinion that more exotic as marked up language or those that based on text script processing as PDF or office documents (or even rare Unicode encodings) bring complexity both for the end user and text processing.

# 3. Summary Evaluation

In broad term of thinking summary evaluation could be a difficult problem because of human subjectivity about what good summary is. As the result it leads to disagreement whether one summary is better than another (Giannakopoulos, 2008). From the other hand, human judgment is the best possible judgment about the summary since humans eventually should read those summaries and have an idea what they are about.

As in many science fields the evaluation is a score which is given to the object and the method which gives such score is called a metric. From mathematical point of view, it could be represented as a function from text domains (with or without additional parameters) to ℝ domain:

(1)

In many cases score range is restricted to be or normalized to have such range.

It highly dependent on the nature of the metric what expectations are. For the metrics which try show how a summary “good” or “bad” it is expected that these metrics are statistically correlated to the human judgments for those summaries. For instance, it could be a Pearson’s correlation (Lin, 2004). However, one might be interested not only how good or bad a summary is but, say, how good a summarizer is. For us as humans, it is not only important how well the information compressed into summary, but it is also important how this information is presented. Manner and matter for us have a meaning. If it is very hard to read a summary such summary might be useless eventually. Thus, we could be interested in the readability of those summaries. In (Elena Lloret, 2019) such question is asked, and it is implemented by the software platform. The readability metrics’ results are useless without any reference and as a reference for readability metrics of the original documents.

# 4. Metrics

It is already mentioned that a text metric could thought about as a function (1) from text domain to ℝ. However, for the metrics that should decide how much the summary are built on the pattern of comparisons between two texts. Although the text comparison task is interesting, but it is hard and has many applications in Natural Language Processing (Leonidas Tsekouras, 2017). The common pattern requires representing text in intermediate form, Figure 2.



Figure 2.

The second text of comparison in many cases is a human summary. Such summaries are called ‘ideal’ or ‘model’ summaries. Furthermore, because of the subjectivity of what good summary is, several summaries should be considered. As well, it should be considered what the strategy is for the multiple comparisons. For the ROUGE metric(s) (Lin, 2004) there are two strategies: either the average is taken, or the best result is taken. By our observations, the best result is not so popular. Therefore, the only strategy implemented by the evaluation platform is an average among all comparisons.

The real art of those comparisons is the intermediate form and the following algorithms based on these forms. Strictly speaking, the algorithm defines (requires the usage of) such form. Thus, for the platform it has been obligated to define a way to hold and produce various forms in generic manner.

On the other hand, obviously, the automatic summary text comparison with ideal summary is not only the way to have a metric for summary. For example, the platform also proposes variety of readability metrics for automatic summaries. Thus, one can have an overview how much considered system influences on readability. For the platform it was chosen to reflect the difference (i.e. mathematical minus) between the original text metric value and generated summary text metric value. A combination of such simplest form with some visualization techniques gives us truly beautiful results for grasping analysis without deep dive into results.

The next sections will give a brief overview of the chosen metrics for implementation by our platform. The one might wish as much as possible metrics; however, such effort would require much time with negative consequences on other planned features.

## 4.1. Implementation Forewords

While it seems a bit early to dive into the implementation details, but we should give an overview picture about implementation details to make the following text clearer. Within our platform we decided to implement several metrics for summary evaluation. Among them there are ROUGE metrics. Originally, ROUGE metrics were implemented in Perl language.

We decided to implement our platform in Java language. Specifically, it is of version 1.8. The decision was dictated firstly due to high popularity of this language. See Table 1 for top 5 languages on the time of writing this text according to TIOBE index (TIOBE, 2019). As well, according to Figure 3, this language is a leadership for almost two decades.

| **Mar 2019** | **Mar 2018** | **Change** | **Programming Language** | **Ratings** | **Change** |
| --- | --- | --- | --- | --- | --- |
| 1 | 1 |  | Java | 14.880% | -0.06% |
| 2 | 2 |  | C | 13.305% | +0.55% |
| 3 | 4 | change | Python | 8.262% | +2.39% |
| 4 | 3 | change | C++ | 8.126% | +1.67% |
| 5 | 6 | change | Visual Basic .NET | 6.429% | +2.34% |

Table 1.

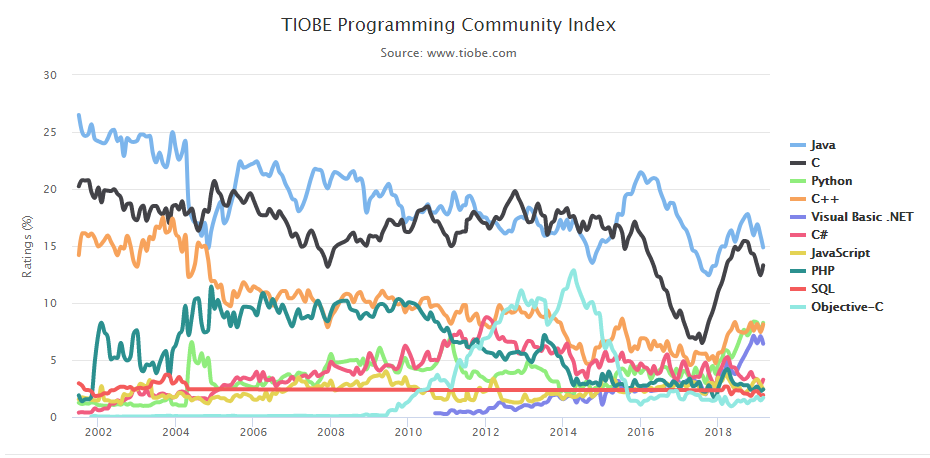


Figure 3.

The second reason for choosing the Java language was a fluent knowledge of it by authors because of professional activity of its authors for a decade. The third reason is that used by authors NLP libraries are written in Java. More specifically, for NLP processing it was chosen Stanford CoreNLP (Stanford, 2019). Fourthly, despite of common believing about the slowness of this languages, today this language (specifically, HotSpot JVM (Oracle, 2019)) is the one of fastest available choices (Debian, 2019).

Thus, the platform almost completely written in Java language. As it was mentioned, the reference implementation of ROUGE metrics is written in Perl. Therefore, we needed to decide how to integrate it within the platform. The decision might have seen non-standard, but we decided to port it to Java language. From the one hand it seems time consuming and, maybe, even useless. From the other hand, many NLP projects today start with Java or even when they do not start with, integrate it with Java is not an issue because of huge availability of integration tools and large community. Additionally, we think that Perl by today standards is outdated language. Understanding the Perl code is known to be an issue by itself. We believe that the port will help for future researchers/implementors better understanding of the source code of ROUGE metrics.

Furthermore, possibly it does not seem to be an issue, but Perl is times slower than Java (Debian, 2019). One might say that it is not an issue for today’s machines (according to (Debian, 2019) it has similar measurements to the Python language). However, we still believe that time matters, and faster solutions are better.

With all said above, we assume that the reader is familiar with one of C derived modern language as Java, C#, etc. If it will be required, the code snippets will be written in Java language. However, this knowledge is not a must.

## 4.2. Rouge

Since it has seen the light, the ROUGE metrics have a wide acceptance in the community due to it statistical correlation with human judgements. The ROUGE methods use statistical measures of similarity based on n-gram of words. The intuition behind the metric is that if two text considered having the similar meaning then they must share similar words or phrases (Giannakopoulos, 2008). One can find detailed explanation of the metrics in (Chin-Yew Lin, 2003). From our side, we should mention that all these metrics share the same “score” structure. Each metric has a precision, recall and F-measure. What precision and recall exactly meant by each metric to be the reader can find also in (Chin-Yew Lin, 2003).

For example, for the ROUGE-N metric between two text the number of common occurrences of n-grams (which tokenized words) is calculated:

**int** nGramHits(Map<String, Integer> peerGrams, Map<String, Integer> modelGrams) {  
 **int** hits = 0;  
 **for** (String modelGramToken : modelGrams.keySet()) {  
 **if** (!**"\_cn\_"**.equals(modelGramToken) && peerGrams.get(modelGramToken) != **null**) {  
 **int** peerHits = peerGrams.get(modelGramToken);  
 **int** modelHits = modelGrams.get(modelGramToken);  
 hits += Math.*min*(peerHits, modelHits);  
 }  
 }  
 **return** hits;  
}

Where peerGrams dictionary is the token n-gram against the number of its occurrences in the peer (i.e. automated summary generated by some system). Analogously, modelGrams is the n-grams against the number of its occurrences of the model. The model in this context is the human summary. Thus, the calculated value is the number of n-gram tokens shared by two texts. (The “\_cn\_” token in both mappings is an internal token indicating the total number per text on n-grams.) In order to calculate the precision, we need to divide the number of shared n-grams on the number of tokens in the peer (automated summary). Accordingly, the recall is the number of shared occurrences divided by total number of n-grams in model (human summary).

(2)

(3)

As well, by default the platform (and original the ROUGE metrics) produce the harmonic mean, i.e. the measure. It is defined as (4) with .

, (4)

Internally, the platform uses the α-based measure (Sasaki, 2007). I.e. . The equation (4) becomes:

(5)

Thus, if one wishes to calculate different F-measures she should give an appropriate α. For instance, for β=0 (, i.e. precision) α=1; for β=2 () α=0.2, etc. However, the ability to change the α parameter is only available when the platform is used as a library.

We will not go over other ROUGE metrics in detailed but will give a high-level overview of how they are built internally.

ROUGE-S has an idea of comparing skip-bigrams for two texts. Skip-bigram is a pair of words allowing an arbitrary gap. For example, for the text “the quick brown fox”, there will be following skip-bigrams: “the quick”, “the brown”, “the fox”, “quick brown”, “quick fox”, “brown fox”. Such bigrams (their statistical occurrences as with ROUGE-N) are compared to another text bigrams. I.e. the numerator is the number of common skip-bigrams. The denominator is the number of skip-bigrams either of a summary (for precision) or a reference (for recall). ROUGE-SU is an extension of ROUGE-S metrics where occurrences of unigrams are also calculated.

ROUGE-L for metric calculation uses common longest subsequence length of two texts. Precision is calculated as longest common subsequence divided by total word grams’ number in summary. In turn, recall is the longest common subsequence divided by the word grams of reference model.

ROUGE-W in some sense is very similar to ROUGE-L. The issue with ROUGE-L is that the distance of the common subsequence is not considered. Suppose we have a sequence:

: [A B C D E F G]

There are also two additional sequences:

: [A B C D J K L]

: [A J B K C L D]

While comparing and ) the common longest subsequence length will be the same in both cases (i.e. 4; thus, and ) will be the same). However, it is logically to assume that should be preferable since the internal sequence is denser. To achieve such effect, the distance of LCS tokens is accumulated though the weight function. That is, a model match is extracted from the dynamic programming matrix (Thomas H. Cormen, 2002) of LCS and the results of weight function of distance between tokens are accumulated. The weight function should have a property . As well, the weight function should have an ability for simple calculation of its inverse version (i.e. ). Thus, the weight function for the original Perl implementation is defined as and (more precise, ). Precision is defined to be

(6)

In recall the denominator has , similarly to other ROUGE metrics.

We should mention that we have ported all ROUGE metrics to the Java language: ROUGE-N, ROUGE-S, ROUGE-W, ROUGE-L. As we said before, we believe it will have good consequences either for the people that will want to integrate ROUGE metric with the code written in Java that have much wider community or to understand not only the formulas but the concrete implementation. At least for us, it was not a trivial approach to understand the Perl code. As well, one can even change the reference implementation to produce her metric with less effort because of having the sources. For instance, it was done in (Marina Litvak, 2015) to extend the ROUGE-S to get weighed version of ROUGE-S.

## 4.3. AutoSummENG

Apart the ROUGE metrics we decided to have an implementation of the AutoSummENG metrics which are based on the n-gram graph text representation. The reason for such inclusion is based on the following: there are some desired characteristics that should co-exists within a single method. More specifically:

* Language neutrality. That is, a method should not require language dependent resources and applied directly to various languages.
* Full automation. A method should not require human interception except the golden summaries.
* Context sensitivity. A method should consider contextual information, the well-formedness of the text. Random word sequences lack this quality.

The AutoSummENG method holds all these properties. As well, it has a high correlation with human judgements (George Giannakopoulos, 2009).

The idea of what n-gram graph is fully explained in (Giannakopoulos, 2008), (George Giannakopoulos, 2009) and (Leonidas Tsekouras, 2017). We will not come back to those explanations. One wishes to know the exact details could refer these sources.

Originally, AutoSummENG implemented upon the toolkit called JINSECT (<https://github.com/ggianna/JInsect>) which allows the generation of the n-gram graphs that are either word n-grams or character n-grams as well as the further comparison of these graphs.

It should be also mentioned that JINSECT toolkit is implemented in Java language. That is, it allowed us using the toolkit as a library without any additional complication as passing data though process boundaries.

## 4.4. Readability Metrics

We decided to include the readability metrics into the platform since as with AutoSummENG we believe that context sensitivity matters. That is, an automatic summarization system may produce a text with poor sematic while having good statistical measures. This issue is considered in (Elena Lloret, 2019). As the consequences, the above work proposes to use automatic readability metrics. We, in turn, implement discussed in the work metrics.

* Flesch Reading Ease (FRE)

(7)

The idea of this metric is that the long sentences and/or long words are hard to read.

* Word Variation Index (OVIX)

(8)

It is assumed that the metric could measure the idea of the text.

* Proper Noun Ration (PNR)

(9)

The reason for this measurement is that the number of proper nouns should increase the readability.

* Unique Proper Noun Ratio (distPNR)

(10)

The variation of PNR.

* Average Word Length (AWL)

(11)

The intuition behind this is that words with fewer characters are more readable.

* Average Sentence Length (ASL)

(12)

Long sentences depict harder text for understanding.

* Noun Ration (NR)

(13)

The idea is that noun massive text is harder for understanding.

* Pronoun Ratio (PR)

(14)

As more a text contains pronouns as less it is clear.

We should mention that in our platform we have decided to normalize FRE, OVIX, AWL, ASL into interval. The normalization is done by linear interpolation into the interval by taking the maximal value within all metric results and minimal value within all metric results. I.e. suppose we evaluate original texts with N automatic summarization systems. We find the maximum and minimum value with all these resources (original texts and all summaries).

(15)

The reason for doing is firstly while doing experiments we found that FRE could give negative results in systems which produce very long sentences. In addition, we believe that values that in [0,1] range are much preferable for the researcher. Knowing that an average sentence length is, for example, 5.42 does not say a much without comparing it to other value. Working with [0,1] range values seems for us a more standard way.

It is also should be said that not all metrics are considered better with higher values. Table 2 explains whether the higher metric value is better, or it is the opposite.

|  |  |
| --- | --- |
| Metric | Is higher value better? |
| FRE | Yes |
| OVIX | Yes |
| PNR | Yes |
| distPNR | Yes |
| AWL | Opposite |
| ASL | Opposite |
| NR | Opposite |
| PR | Opposite |

Table 2.

We need to add POS (part of speech) tagging has done with Stanford CoreNLP package (Stanford, 2019) as well as sentence recognition. Recalling that POS tagger is identifying what the concrete word stands for (noun, pronoun, proper noun, etc.). Syllables counting is done with MorphAdorner package (NIUT, 2013) as in the source article (Elena Lloret, 2019). Both are written in Java language therefore the integration with them was smooth. However, the MorphAdorner package seems to be limited to English language. Thus, FRE for non-English texts is inaccurate.

# 5. Platform as Library

Software engineering is known to be a hard and challenging task. There are many aspects of software engineering. For instance, it would be a time management, a development methodology, system analysis, requirement analysis, system design, code design and so on. While developing a software many properties should be considered to produce well behaving product. However, according to (Standish Group, 2014) canceled project statistics for small companies is 21.6%, medium companies 37.1% and 29.5% in large companies. That is, 29.4% in average a company produce no (planed) software at all. We present those value to highlight the fact that the chances that some software will see the light in general is about . We do not claim but we think it is logically to assume that in those project that are delivered on time and budget, the situation of post-delivery stage is not so rainbow if it would be measured. What we are trying to say is that a software after delivery is still alive. It requires a maintenance, adding features, reusability. We assume that if project success measurement would include these parameters the success factor could be less.

We think that it is pretty obvious that maintenance, adding of features and reusability are highly dependent on software design. We also think that software design is a broad term. Different sources define it with a different meaning (Paul Ralph, 2009). When we say ‘software design’ we mean internal software structure. Although one could not agree with us because she can also consider the chosen technologies (for example, a programming language), data storages (variety of data bases, file systems, etc.) and so on; but comparing to these adjacent software properties, internal software structure may become a bottleneck – for new features, maintenance and reusability – much earlier. For instance, (Grady Booch, 2007) takes the approach of internal program structuring interchangeably with design. Such conclusion comes up because Object Oriented Design by our opinion is no more than an internal program structuring based on modeling by objects and reflecting it as a code.

## 5.1. Chosen Approach

There are many approaches for modeling a program. For instance, it should not be new for one that today we have a raise of functional programming approach. The more or less similar processes were with object-oriented programming (OOP) two decades ago. There are benefits of using one modeling or another as well as its weakness.

By our opinion, for example, the strongest OOP design is for ‘static’ world knowledge. The word static used in sense of ‘no or almost no changes during the software lifecycle’. That is, if we take such field as modeling of mathematics, GUI libraries or game programming. Each of these examples could be modeled with OOP almost natively. For instance, if we want to model mathematical set, we know the finite sets of its operators. It should be union, intersect, difference, cartesian product. Sure, the operators themselves could be programmed in variety of ways. It could be either hash or tree-based structures with an appropriate find, add and remove operators. However, the strongest part of such modeling is that the operators’ set is static. It is close to impossible that one will invent a new operator which is impossible to express within the provided operators. The same could be valid for linear algebra modeling (which is heavily used in graphics) or whatever else mathematical field. GUI libraries also share the same properties. We could predict what exactly operator set is needed for GUI widgets library. It could have a common methods’ set as getting/changing size, color, font, background, etc. It is also very understood how to build a hierarchy of such objects. For instance, a toggle button will just extend a button (i.e. it is ‘is a’ relationship/inheritance). Games fall in the ‘easy to model with OOP’ approach since the game behavior and object interaction has a limited and predicted set of methods. For example, if an arcade game is being developed such set could be: ‘find’, ‘detect collision’, ‘draw self’ and so on. Each character will have almost the same methods with different implementation (which one will put in ‘game character’ class). All these objects will be orchestrated in ‘game’ object.

We assume that the situation is different with modeling a world when it has fuzzy method set. Such assumption is based on some experience in the field that we will talk about. Fuzzy method set we define as unpredictable set (i.e. a developer cannot predict what exactly operator set could be for the object class). The first example we wish to consider is a ‘string’ class. Although any person having a minimalistic experience with programming sees this entity within modern OOP languages modeled as a class, but it is only a first approximation. For instance, Java language ‘String’ class has several methods exposed. There are two questions. First, one could ask herself whether this set is enough. The second question whether this set is orthogonal (i.e. minimalistic number of operations such that other methods could be derived from minimalistic set). The answer is no for both. If the set were enough it was not required to use utility functions for string either written by 3rd parties or in-house. The set is also non-orthogonal. We think that it is obvious that minimalistic set of methods for class ‘String’ could be only getting character at some position or getting character in a stream manner. Class ‘String’ in Java language in order to be ‘extended’ provides the method ‘charAt(int i)’ where the ‘i’ is a position of requested character. The conclusions of all said above are such that, first, it is understood that ‘String’ cannot provide all possible variety of required operators. Simply because it is impossible to include all of them under one hood; for example, what if we want to compute longest common subsequence of two strings, should or should not such method be exposed by ‘String’ class? If all possible methods would be included into class ‘String’ it would be hard to say what exactly the responsibility of it. Second, if we would model the ‘String’ class with minimalistic number of methods, does it stay a class? Such modeled class is a reduction to a character list with or without random access. This, in turn, violates the main principle of information hiding proposed by OOP. However, the fact stays the fact that in all modern object-oriented languages ‘string’ is a class (or whatever similar entity as with prototype-based languages, but this changes nothing eventually). We leave the ‘String’ entity modeling question for further researches whether the taken approach is suitable if at all.

The second example of fuzzy method set could be business application programming. By business application we mean applications that mimic business processes within an enterprise. For instance, one can imagine CRM, Bank or bookkeeping application. This kind of applications requires a storage many entities with many parameters for an entity with many rules connecting these entities which could even produce some artificial entities (for example, modeling ‘salary’ as some number property on ‘employee’ could be wrong since salary should have a date, bonuses and so on and, thus, has to be reflected as a separate entity). The ‘sad’ thing about modeling this kind of software by object-oriented techniques is that no one can predict all fluctuations. For instance, one would like to model a virtual enterprise. She takes the current snapshot of some existing enterprise. First, she creates an ‘employee’ class. The next thing that looks logically for her is to model the enterprise hierarchy. She creates following classes: ‘factory worker’, ‘manager’, ‘bookkeeper’, ‘cleaner’, ‘security guard’. For her it seems consistent since ‘factory worker’ *is* ‘employee’. A ‘manager’ *is* an ‘employee’ too, and so on. What exactly methods an ‘employee’ should have? Say, we want to model a payroll. Sure, it seems indisputably to put the payroll method on ‘employee’ since all employees will receive a salary (i.e. it could be an abstract method ‘payroll’). She even thinks that for new positions new classes will be created. What she does not consider is that she took only the *current snapshot*. Maybe such mental model has chances to exist for a period. However, there is probability that the enterprise will evolve. What if the enterprise does not want ‘bookkeeper’ being the direct worker but a contractor company? This enterprise decision breaks the mental model based on the OOP approach. For example, could the ‘payroll’ method be the part of employee? No, since contractor company pays salary for its workers. She can even try to fix it by splitting the ‘employee’ to ‘direct employee’ and ‘contractor’ having the ‘payroll’ method on ‘direct employee’ only. Though, it will fit only for the given time snapshot. Each time an enterprise will move workers to pay directly or under contractor the code will require changes. We assume that most will agree that the discussed kind of modeling is not suitable for a library code because of unpredictable code changes. As well, one should grasp that it is only one example of change. Other changes could break the following OOP based approach in a similar manner.

Some people can argue that, first, our synthetic example is not so correct since ‘factory worker’, ‘manager’, ‘bookkeeper’, ‘cleaner’, ‘security guard’ look more like the concrete objects than classes and they should be distinguishable by an inner ‘employee’ indicator. The counter question, where does the demarcation lie? With the same argument we can say that ‘employee’ is no more than ‘person’. ‘Customer’ is a ‘person’ too. Thus, we can reduce all participating humans to ‘person’. However, we think it is unnatural since while modeling an enterprise we agree that they have very different roles*. Thus, we argue that finding a demarcation between class and object is not trivial for modeling. Within some types of software, it may lead to fragile class hierarchy that will require fixing the code on even small context change. The preferable solution could be modeling the processes based on generic data structures than nouns with operations on them*. Yes, in some cases it requires a data redundancy. For instance, all employees will have ‘reporters’ while only ‘manager’ has a meaning of this property. *However, when processes involving entities are unclear it will lead to less changes over the time and these changes will be more concentrated*. That is, for our artificial example, ‘payroll’ becomes a ‘payroll service’. ‘Direct employee’ entity does not need to exist. Instead, before ‘payroll service’ is about to execute, ‘employees’ should go over ‘direct employee filter’.

The shown above approach is different for modeling. Surely, one might say that it is functional approach and will be right. The verbosity of the above text is to give a feeling that library code writing is not trivial. The library code should be generic. On the other hand, the generalization should adopt the modeled domain and the domain itself dictates the more suitable form. Object-oriented modeling should be taken carefully.

Thus, the chosen by us approach to model a platform is rather functional than object-oriented. We think even that when one imagines the platform, she imagines it as a functional pipeline. In general, what we need to do is to present a way to map from one domain (for example, a text) to another and have comparing mechanism for result domain. While modeling a platform for evaluation it is not only difficult to predict methods for the entities but also it is not always possible to predict the data structures. That is, there are infinitely many ways and results that we can produce from a text. For example, we want to present a text as word n-grams with number of occurrences for an n-gram (which eventually word n-gram histogram). We can directly write a function that will just split words and calculate such histogram. We can also firstly preprocess the text to have some intermediate form as list of tokens and then calculate a histogram. Such histogram we might wish to reuse in calculations later or immediately compute some score of evaluation. Having put the text into some object-oriented class (for example, it would be ‘document’) will require continuously changes in this and derived classes.

For example, suppose we have modeled it in object-oriented way. We have a ‘document’ that expect to get a text as a string. We can have on this ‘compute histogram’ method. We can also have a ‘convert to token list’ method. Since we want work with token list as with ‘document’, how do we represent it as a document too? We can either put a list to be a member of a document or to have a ‘document’ that will hold a list of tokens. If we put it to be a member, how do we distinct whether we want ‘compute histogram’ to be from the string or from tokens? If we create another entity as ‘document for tokens’ list’, how do we create a class hierarchy? I.e. how do we care not to have a ‘convert to token list’ method? Do we treat a histogram as a ‘document’ too? Thus, to answer these questions we present in a next section a generic and robust way to handle variety of cases.

## 5.2. Domain-Specific Language

From the ancient ages, humanity tries to pass information and ideas through specific languages. Despite of speaking, reading and writing abilities of the native language of a person, today it is impossible to imagine that the person is not exposed to specific languages. For instance, when he learns mathematics, in addition to learn abstract ideas what number is, what set is, etc. he also learns how to express those ideas within mathematical notation. The mathematical notation is no more than a specific language to pass those ideas. We think that everyone agrees that without math notation language, passing those ideas within only the reading, writing and speaking his native language would be hard, even impossible.

With varying degrees of success, the idea of domain-specific languages periodically appears in software engineering (the word ‘domain’ is used in context of humanity knowledge domain; e.g. typesetting domain has its own set of meanings and rules therefore). For instance, Unix has a long tradition having small languages for specific purpose. It is gratefully covered in ‘Minilanguages’ (Eric S. Raymond, 2003). For instance, utilities *troff, eqn, tbl* and *pic* are used for typesetting; *awk, sed* and *dc* are for shell and text processing. As well, there are minilanguages for software development and so on. Strictly speaking, it is a part of Unix philosophy having small strict utilities passing among them a text data.

Although following Unix philosophy of creating small languages could give truly wonderful results but it has its drawbacks as masterly possession of the compiler/interpreter writing. As well, while writing a code within a chosen programing language one does not want to mix numerous languages and ‘jumping’ across languages boundaries. (Honestly speaking, in Unix shell scripting is ‘invented’ to allow gluing together those ‘minilanguages’. However, most people exposed to shell programing could agree that it is a cumbersome matter.) Thus, the second way of domain-specific language creation is so called ‘internal domain-specific languages’. The topic is covered in (Martin Fowler, 2010). The idea is to use a programming language for somehow expressing the modeled language (there is no strict way how to do it). One of the well-known patterns for language modeling is so called *fluent interface*. The idea of this pattern is very simple: an object method should return the reference to itself or one of language modeled tokens. It will allow a chaining of the modeled operators.

In order not to be unfounded, we will present a small language example modeled within a fluent interface. It will give a feeling and one can grasp the idea behind it. Suppose we need to write a calculator program. (Say we want to have a linear algebra modeled. Surely, one exposed to programming in C++ language can model it with operator overloading, but it is synthetic example which only should show the ability of different modeling approach. As well, operator overloading is not a case for many languages, particularly for Java. Moreover, because of the explanation reason the values will be float point numbers.) If we would model directly with class and methods approach, then complex expressions would be hard to understand because they will be split to many unrelated pieces (lines). To not overcomplicate the example with operator precedence, it is also chosen that the calculator will be built on top reverse polish notation (RPN). Thus, the code could look like:

**public class** SmallRPNCalculator {  
  
 **private final** Deque<Double> **stack** = **new** ArrayDeque<>();  
  
 **public** SmallRPNCalculator push(**double** val) {  
 **stack**.push(val);  
 **return this**;  
 }  
  
 **public** SmallRPNCalculator dup() {  
 **double** val = **stack**.peek();  
 **stack**.push(val);  
 **return this**;  
 }  
  
 **public double** pop() {  
 **return stack**.pop();  
 }  
  
 **public** SmallRPNCalculator add() {  
 **double** left = **stack**.pop();  
 **double** right = **stack**.pop();  
 **stack**.push(left + right);  
 **return this**;  
 }  
  
 **public** SmallRPNCalculator sub() {  
 **double** left = **stack**.pop();  
 **double** right = **stack**.pop();  
 **stack**.push(left - right);  
 **return this**;  
 }  
  
 **public** SmallRPNCalculator mul() {  
 **double** left = **stack**.pop();  
 **double** right = **stack**.pop();  
 **stack**.push(left \* right);  
 **return this**;  
 }  
  
 **public** SmallRPNCalculator div() {  
 **double** left = **stack**.pop();  
 **double** right = **stack**.pop();  
 **stack**.push(left / right);  
 **return this**;  
 }  
  
 **public** SmallRPNCalculator sqrt() {  
 **double** val = **stack**.pop();  
 **stack**.push(Math.*sqrt*(val));  
 **return this**;  
 }  
  
 **public static void** main(String[] args) {  
 SmallRPNCalculator smallRPNCalculator = **new** SmallRPNCalculator();  
 **double** a = 1, b = 1, c = -6;  
 **double** sqrtDet = smallRPNCalculator  
 .push(b)  
 .dup()  
 .mul()  
 .push(-4)  
 .push(a)  
 .push(c)  
 .mul()  
 .mul()  
 .add()  
 .sqrt()  
 .pop();  
 System.***out***.println(sqrtDet);  
 }  
}

As we can see almost all methods of RPN calculator except the ‘pop’ return the reference to itself. Thereby, it allows us to chain the methods. The example in the main static method is calculation of the square root of square equation determinant. More specifically, it is and the calculation is .

One could say that the same effect she could achieve if she will make all those methods to have a ‘void’ type. We might reply that yes; the effect of calculations will be the same. However, it will have much more *syntactic noise* because it will be required (at least) to repeat the variable of the calculator object. The most important thing is that *it looks like an expression for human eye* but method invocation on the object does not look so. It is not occasionally that, for example, Java models the *higher-order functions* of *functional programming* primitives (from Java version 8) for collections exactly in the same way (‘Stream API’). Strictly speaking, Java 8 Steam API only mimics the functional programming primitives by using of fluent interfaces. Additionally, we would like to add that by our experience once a person is exposed to domain-specific language programming, *the code written with domain-specific languages becomes almost self-documented since once the domain is correctly modeled within a language additional comments add no more than a noise.* Again, introducing functional programming higher-order primitives for collections in almost all modern languages in last decade is due to fact that *the code is much more readable and maintainable* because *we say what happens* to collection and not write the for-if loops which should be explained (for-if loops say how it should happen).

As one could already guess we use the domain-specific language. Especially we use it for the pipeline definition of what should happen to text to get a final form for comparison. Expressing the text transformations as a pipeline gives an ability to reuse the pipeline fully or partially in different places in the code. We think that it also gives an ability to extend the pipeline much simpler way than it would be modeled and coded with a ‘pure’ object-oriented approach. We are going to explain a conceptual part of such pipeline. Although it could be possible to give UML class diagrams, but we think that it will not give any benefits and even confuse. Eventually, usage of diagrams here will not give an idea since we do not model objects and classes, but it is a language. However, a given code will be reduced only for understanding and not fully copied from the code-base.

Thus, the first entity is Text. It is defined as:

**public class** Text<T> {  
  
 **private final** String **textId**;  
 **private final** T **textData**;  
  
 **public** String getTextId() {}  
  
 **public** T getTextData() {}  
  
}

The overall idea of this entity is to represent a form or derived form (generic T type) of the text with an id. The identifier could be any string that somehow distinct a text from another. Within the platform the text id is the absolute file name.

The next entity is a ‘text processor’. It is defined as:

**public interface** TextProcessor<X, Y> {  
  
 Text<Y> process(Text<X> data);  
  
}

The purpose of this entity is to be a mapping function to change a form of text. I.e. X is the domain type and Y is codomain type of the text form.

One possible example of ‘text processor’ could be a processor that maps from string to list of tokens. In code it looks so:

**public class** TextToTokensProcessor **implements** TextProcessor<String, List<String>> {  
  
 @Override  
 **public** Text<List<String>> process(Text<String> data) {  
 String text = data.getTextData();  
 String[] split = text.split(**"\\s+"**);  
 List<String> tokens = Arrays.*asList*(split);  
  
 **return new** Text<>(data.getTextId(), tokens);  
 }  
}

More interesting and special example is a gluing of two text processing. That is, we want to have a processor that maps from domain X to domain Z while passing the domain Y (i.e. ). The code of such entity looks like:

**public class** PipelineProcessor<X, Y, Z> **implements** TextProcessor<X, Z> {  
  
 **private final** TextProcessor<X, Y> **first**;  
 **private final** TextProcessor<Y, Z> **second**;  
  
 **public** PipelineProcessor(TextProcessor<X, Y> first, TextProcessor<Y, Z> second) {  
 **this**.**first** = first;  
 **this**.**second** = second;  
 }  
  
 @Override  
 **public** Text<Z> process(Text<X> data) {  
 Text<Y> firstProcessed = **first**.process(data);  
 **return second**.process(firstProcessed);  
 }  
}

While looking into the code there are two interesting things. First, ‘pipeline processor’ has three generic types. Java type safety mechanism will not allow to use an object constructor if the first type of second processor is different from second type of the first processor. The second interesting thing is that after all transformations it is still a processor from (look at the ‘implements’ part and the ‘process’ method itself).

Additional processor that has a special meaning as pipeline processor is a caching processor. Caching is a vital component of the system. Many transformations while working with text should be only really processed once and then just retrieved from the cache. For instance, running Core NLP is a heavy operation. It is much better to cache the result of execution somehow and then just retrieve the processed result. Another essential application of caching is in the comparison itself. For example, in ROUGE we compare a summary to a same set of human summaries. There is no any reason to get human summaries reprocessed for every machine summary. *However, if it would be programmed ‘directly’ it is an issue how do we decide what exactly cache type we want, and how we do control its execution. By our experience in most cases caching mechanics is threaded into the code itself which makes the usage and understanding cumbersome*. (The people familiar with programming know how much the problem of caching is not trivial. The when, why and how in many cases lead to salad in code) Decoupling of caching mechanics from the code itself could be considered as an achievement or benefit. In code the place responsible for caching is defined as:

**public abstract class** AbstractCacheTextProcessor<X, Y> **implements** TextProcessor<X, Y> {  
  
 **private final** TextProcessor<X, Y> **textProcessor**;  
  
 **public** AbstractCacheTextProcessor(TextProcessor<X, Y> textProcessor) {  
 **this**.**textProcessor** = textProcessor;  
 }  
  
 @Override  
 **public** Text<Y> process(Text<X> data) {  
 Text<Y> result = getCached(data);  
 **if** (result != **null**) {  
 **return** result;  
 }  
 result = **textProcessor**.process(data);  
  
 setToCache(result);  
 **return** result;  
 }  
  
 **protected abstract** Text<Y> getCached(Text<X> data);  
  
 **protected abstract void** setToCache(Text<Y> computed);  
  
}

The code only defines sufficient self-documented micrologic for the caching mechanics. All derived entities should define its logic of how and where store and retrieve requested text forms. This, in turn, abstracts the underlying details of what cache is. For instance, it could be a memory, filesystem or even a database. For now, there are two concrete available implementations for the caching in the platform. One is responsible to store in memory and the second uses filesystem.

Filesystem cache expects that the stored/retrieved object will be of Java type java.io.Serializable and uses the standard mechanism of Java marshaling (serialization). As the key it uses the text key. As it was already mentioned the key itself is the absolute file location of originally loaded text. Thus, there should be some mechanism that will use an absolute file location as a key. The solution for this is to compute cryptographic hash (specifically, SHA-256) of the key (absolute file name) and store it as a directory on file system (more specifically it is Base-64 encoding of the key where ‘/’ is replaced to ‘-’).

In general, it is enough to have those entities to express any text pipeline. However, it is much better to move further to a language that will glue together processors and the result will look like an expression. That is, eventually we have come to a domain-specific language defining a text pipelining:

**public class** TextPipeline<X, Y> {  
  
 **private final** TextProcessor<X, Y> **current**;  
  
 **public** TextPipeline(TextProcessor<X, Y> textProcessor) {  
 **this**.**current** = textProcessor;  
 }  
  
 **public** <Z> TextPipeline<X, Z> pipe(TextProcessor<Y, Z> next) {  
 PipelineProcessor<X, Y, Z> pipedProcessors = **new** PipelineProcessor<>(**current**, next);  
 **return new** TextPipeline<>(pipedProcessors);  
 }  
  
 **public** TextPipeline<X,Y> pipeIf(**boolean** condition, TextProcessor<Y, Y> next) {  
 **if** (condition) {  
 PipelineProcessor<X, Y, Y> pipedProcessors = **new** PipelineProcessor<>(**current**, next);  
 **return new** TextPipeline<>(pipedProcessors);  
 }  
 **return this**;  
 }  
  
 **public** TextPipeline<X, Y> extract(TextPipelineExtractor<X, Y> extractor) {  
 extractor.setTextProcessor(**current**);  
 **return this**;  
 }  
  
 **public** TextPipeline<X, Y> cacheIn(Function<TextProcessor<X, Y>, AbstractCacheTextProcessor<X, Y>> cacheProducer) {  
 AbstractCacheTextProcessor<X, Y> cache = cacheProducer.apply(**current**);  
 **return new** TextPipeline<>(cache);  
 }  
  
}

The language has only 4 operators: pipe, pipe if, cache in and extract. However, this set of operators is enough to have the pipeline as expression. We will comment on ‘extract’ and ‘cache in’ operators. To understand what ‘extract’ method is, one should come to example of RPN calculator previously described in this chapter. There is a ‘pop’ method there. The issue with pop method is that it breaks the fluent interface. The example is written so to not overcomplicate it. However, within real life we might want to continue the fluent interface while having extracted the intermediate result. This is the purpose of this method – we can define long pipeline however part of this should be used in different places. The second operator required to be commented is the ‘cache in’. The issue is that when we describe what should happen, we do not want the code being executed. It does not exist in the system but say, we want to have a database as a storage system. It will require a preparation to use a database. Where this code should be stored? To avoid confusions and decouple the cache initialization from the caching processor it was decided that this block should come in functional closure. For the simple scenarios it does not add overhead too as it will be shown.

Following examples will show code snippets how it is used. The first example is the initial pipeline both for ROUGE and AutoSummENG processing:

TextPipeline<String, List<String>> initialPipeline = **new** TextPipeline<>(**new** FileToStringProcessor())  
 .pipe(**new** TextToTokensProcessor())  
 .pipeIf(filters.contains(Constants.***LOWER\_CASE\_FILTER***), **new** LowerCaseFilterProcessor())  
 .pipeIf(filters.contains(Constants.***PUNCTUATION\_FILTER***), **new** PunctuationFilterProcessor())  
 .pipeIf(filters.contains(Constants.***STOP\_WORDS\_REMOVAL\_FILTER***), **new** StopwordsRemovalFilterProcessor())  
 .pipeIf(filters.contains(Constants.***PORTER\_STEMMER\_FILTER***), **new** PorterStemmerFilterProcessor())  
 .cacheIn(CacheMemoryTextProcessor::**new**)  
 .extract(tokensExtractor);

This is as simple as it seems. However, it allowed to us setup caching in ‘cherry picking’ manner. Once the initial pipeline is calculated the needed us result is cached in memory. (The result is a list of token – please take a look on the second generic parameter of the text pipeline) We might be able to set even file loading to cache after the first processor (which is defined in the constructor of the text pipeline) but it is redundant in the given circumstance since what we really want is a list of tokens.

This initial pipeline is used later in the code in the following manner in ROUGE-N:

initialPipeline.pipe(**new** NGramTextProcessor(rougeNParam))  
 .cacheIn(CacheMemoryTextProcessor::**new**)  
 .extract(rougeNGramXExtractor);

For ROUGE-S the pipeline is continued too:

initialPipeline.pipe(**new** SGramTextProcessor(useUnigrams, skipDistance))  
 .cacheIn(CacheMemoryTextProcessor::**new**)  
 .extract(rougeSGramExtractor);

Summarizing this and partially the previous chapter we want to have a philosophical retreat on the shown above code. While working in industry for a while it is not a secret for software engineers that some ideas are overengineered while others are poorly or completely not engineered. The KISS principle is discussed and suggested in many articles and books. However, what exactly meant is staying unclear for many software engineers (for example, it is mentioned in chapter one (Eric S. Raymond, 2003), but today one can find tons of information in Internet). For instance, if we take the text pipeline modeling it is clearly understood for one that it is a kind of functional pipeline. Let us try to model it with regular Java methods (functions). We think that when following KISS principle people may come to more or less same design which could be expressed as Java-like pseudocode. In the first part we need tokens:

String loadFile(File file) {}

List<String> tokenize(String text) {}

List<String> convertToLowerCase(List<String> tokens) {}

List<String> applyPunctuation(List<String> tokens) {}

List<String> removeStopWord(List<String> tokens) {}

List<String> applyPorterStemming(List<String> tokens) {}

List<String> getInitialTokens(boolean lowerCaseEnabled, boolean punctuationEnabled,

boolean stopWordsEnabled, boolean porterStemmingEnabled) {

String fileText = loadFile(file);

List<String> tokens = tokenize(fileText);

if (lowerCaseEnabled) {

tokens = convertToLowerCase(tokens);

}

if (punctuationEnabled) {

tokens = applyPunctuation(tokens);

}

if (stopWordsEnabled) {

tokens = removeStopWord(tokens);

}

if (porterStemmingEnabled) {

tokens = applyPorterStemming(tokens);

}

return tokens;

}

Thus, now we need to somehow write design calculations of n-gram and skip-grams. One might suggest that those functions should look like:

Map<String, Integer> calculateNGrams(List<String> tokens, int nGramLength) {}

Map<String, Integer> calculateSGrams(List<String> tokens, boolean useUnigrams, int skipDistance) {}

The dilemma is whether these functions are written correctly. From the one hand they are standalone and just expect tokens as an input. From the other hand how do we express the calculation of the initial tokens before and then run functions that calculate n-grams and skip-grams? We think that design with passing only the tokens is preferable because it decouples n-gram/skip-gram calculations from how tokens are generated. However, even so, we are obligated to have some functions before that will calculate the initial tokens and pass token further. The most interesting part is, first, we should pass the context (all Boolean variables) to these wrapping functions. Second, if we will need to add some addition processing for initial tokens, it, in turn, will require changing of the wrapping functions. Third, where do we put the caching exactly? In n-gram calculation itself, in the wrapping functions? How do we change the caching type smoothly? Even if we are going to externalize the caching, we will have to either pass it as a function parameter or a wrapping context (say, we can put all this code into classes). Eventually, such design will lead to inconsistent spaghetti code in which each change will require additional changes over the code-base (one will put caching in s-gram function, someone else will put it in wrapping functions, etc.). However, one should understand that *this code having a vision of function pipeline* maybe fine as a code for closed system but as a library code ‘it has no chances to live’. The *design within domain-specific language has the same vision of functional pipeline* but it allows dynamicity being straightforward. *The KISS principle says about having non-overengineered code, but it does not propose non-engineered solutions which we tried to avoid*.

## 5.3. DI and IoC

These acronyms stay for Dependency Injection and Inversion of Control respectively. We have given acronyms not by occasion since today one may hear more often about those things in this form than fully pronounced. These principles are de-facto standard of the (software) industry in general. A good description of them could be found in (Martin Fowler, 2005). We will have their brief overview and then will explain how them related to code.

While we (and not only) saying that they are principle the most important thing it means is that principles do not have strictly defined rules and forms (that is, they are not engineering patterns which have a concrete form, for example). By a set of external signs, we might say whether those principles are followed or not.

We think that various forms of DI principle have been existing from the programming appearing as a human activity. However, intentional and comprehended usage could give truly wonderful results as self-contained, decoupled, testable and independent code pieces. We will use several examples for giving a feeling of what DI is to have better understanding of the chosen approach.

The first example we will take is a C language approach of programming. The C compiler has an ability to split source files into two types: headers and source code. This distinction is not by occasion. In header file we can only declare the set of used functions and/or ‘extern’ variables. However, the concrete implementation can be various. Thus, when linking the final executable file, we can choose what implementation we will use. The dependency (functions or variables) injection (we do not really care how functions implemented) in this case is expressed by linker. The second example we will take from the world of functional programming. While working with higher order functions for collections, higher order function expected to get a function for one element processing. For the higher order function, the function for one element processing is an injected dependency (for instance, map, filter, reduce work exactly in the same way expecting a function as input). Third, and last, example is factory method pattern. While asking factory method pattern by concrete identifier we do not care about how exactly concrete object instance programmed. For code using factory method pattern produced entity is an injected dependency. There are myriads of such examples. *The main principle could be expressed as a pluggable piece of code that could be changed simply*.

Inversion of Control is a form of dependency injection. As well, we will give a couple of examples to give a filling of this principle. Suppose we have an interface for string utility functions (the example assumes the Java language).

public interface StringUtils {

int len(String s);

int isAlphaNumeric(String s);

int[][] calculateLongestCommonSequenceDPmatrix(String s1, String s2);

...

}

public class A {

void someMethod() {

StringUtils stringUtils = new StringUtilsVendorX();

...

}

}

public class B {

void someMethod() {

StringUtils stringUtils = (StringUtils) ServiceLocator.get("veryFastStringUtils");

...

}

}

public class C {

private final StringUtils stringUtils;

public C (final StringUtils stringUtils) {

this.stringUtils = stringUtils;

}

void someMethod() {

...

}

}

The class A in this example cannot be considered a code that follows dependency injection principle (in general). This is because once we would like to change the concrete usage of ‘string utilities’ we will be obligated to change the source code using it. The class B in the example can be considered as a dependency injection entity since actual string utility implementation is externally controlled (service locator pattern is a ‘brother’ of factory with the only difference that service locator does not require returning a concrete type). The class C also follows the dependency injection, but it is also an example of inversion of control. The huge difference between C and B is that B ‘pulls’ actual instance while C is ‘pushed’ by an actual instance. *This is exactly the meaning of inversion of control, i.e. the client code does not control the actual instance – the code just expects that it somehow gets the working instance*.

The design with concrete instances requires code changes on if the dependent code changes. The design with factories and/or service locators has the benefit of being decupled from the concrete implementation. However, it has drawbacks of being dependent on the factory/service locator and asking for the concrete instance. That is, suppose two entities B1 and B2 are dependent on an artifact via a service locator. *For B1 and B2 being able to work service locator must exists*. For testing or composition, it will require either mocking or separating service locators (which in turn brings to ‘monstrous’ constructions as factories of factories or locators of locators that sure do not make a code more maintainable/scalable). As well, if B1 and B2 are dependent on the same artifact separating artifacts for B1 and B2 will require code changes of B1 and B2. Suppose, C1 and C2 are dependent on an artifact but this artifact is injected externally (i.e. control is inversed for C1 and C2). Within inversion of control, C1 and C2, firstly, do not dependent on intermediate entities. Thus, no mocking or factories of factories are required. The separation of the common dependency does not require direct code intervention for C1 and C2. However, the drawbacks of the approach are that it supposes an upper entity that will care about composition. The second issue is that if the code should be dynamically requested the C class solution does not meet the requirements (though, such cases are rare in general, but vital in some type of applications, for example, compiler writing require different behavior for different tokens type).

The example of class C is the basis for the modern frameworks for inversion of control dependency injection. For instance, both Spring Framework (<https://spring.io>) and more lightweight as Google Guice (<https://github.com/google/guice>).

The chosen by us approach is similar to those frameworks since DI IoC philosophy has many benefits as it was discussed (testing, granularity and overall application high-level viewing). However, by our opinion the dependency on the framework might be redundant in the library code. The solution for it is simply to define a place a code place as a ‘building kit’.

Such place is a Container abstract class:

**public abstract class** Container {  
  
 **private final** Map<String, Object> **container** = **new** ConcurrentHashMap<>();  
  
 **public** Object getBean(String beanKey) {  
 **return container**.get(beanKey);  
 }  
  
 **protected void** setBean(String beanKey, Object bean) {  
 **container**.put(beanKey, bean);  
 }  
  
 **public abstract void** build();  
 **public abstract void** setConfiguration(Configuration configuration);  
 **public abstract** Configuration getConfiguration();  
}

The class defines three main entities: some storage for the inner entities (‘container’ class field), configuration and the building of container. The purpose of the ‘container’ field is to behave like as service locator. The purpose of configuration is to have some external configuration while building the container. As by now, there is only once concrete instance of container. Though, one can define its own container (actually this is a purpose of this chapter to explain a flexibility of such replaicement).

The second very important class place of the platform is the ‘main algorithm’ entity:

**public interface** MainAlgo **extends** Runnable {  
  
 **void** setContainer(Container container);  
 Container getContainer();  
}

Since the interface is a descendant of runnable, it expects to get a container and run the main logic based on this container within the ‘run’ method of Runnable. The run method expected to get some executable instance which could be retrieved through the ‘get bean’ method and executed. (As for today, each metric is exposed as some executable ‘reporter’ entity (i.e. multiple entry points). The why behind is a historical reason. Broadly speaking, it is possible to combine the execution of the container by adding it an executable method. From the other hand the solution will be tightly coupled with the execution)

## 5.4. High Level Architecture

Eventually we have come to the high-level architecture and one that wish to extend the system might be able to do with less effort.

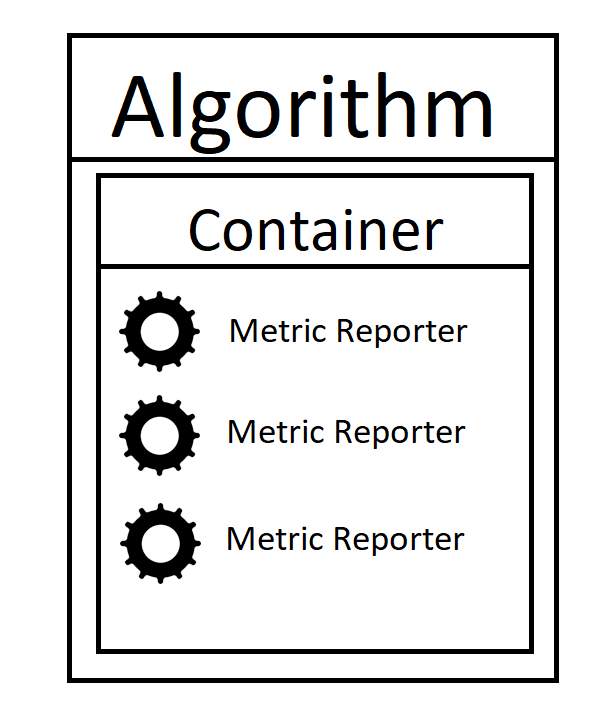


Figure 4.

Figure 4 represents the design of the container and main algorithm described in previous sections. Each reporter is a class that programmed to process one metric. A reporter expects to get:

* Text processors which explained in domain-specific language.
* Score calculators – entities to perform actual comparisons.
* Thread pool to post the score calculation.

### 5.4.1. Concurrency

Although it is possible to have a reporter that will work in single-threaded mode (which within current architecture is pretty straightforward) but one of our vision has been performance of the platform. To achieve this goal, we need to have concurrent calculations – we assume that today the platform will run on multicore CPUs and concurrency is an obvious way for performance. However, we should choose a granularity of each concurrent task. We think the most natural granularity for the platform is each comparison that may return a score to be expressed as a chunk of work. For example, using a reporter as a working chunk could be coarse grained solution. What happens if we want to run just one metric? Although having a granularity less than just one score calculation is possible, but we think it will lead to inconsistency and will be hard for understanding and maintaining.

One of the first things defined in Container are thread pool and ‘arbiter’. Both entities are passed to underlying reporters. The idea of using thread pool is simple:

* Each score calculation is wrapped in reporter as work chunk (AsyncScoreCalculator class in code) and submitted to a thread pool.
* Such submit produce a ‘future’ (an entity that will block until the result of work is not available – the regular Java thread pool implementation)
* All reporter’s futures are collected to list.
* The list is submitted to a thread pool via additional chunk of work – AsyncPeerAllResultsProcessor/AsyncAllResultsProcessor in code (this is done so in order not to block next reporter).
* After all scores of a reporter is processed, it signalizes about its finish via ‘arbiter’

The ‘arbiter’ entity is a kind of mutex which waits till all registered on it entities are done. The idea behind this entity is to know when all reporters are done. The need of this is dictated by the fact that we need to process some results to calculate something based on the all metrics (or part of them). For example, as it mentioned earlier some readability metrics might not have values in interval (FRE, for example). Normalization can be done in many ways, but it is better to have all values of a metric to be normalized. For instance, if we normalize separately readability metrics of topics (i.e. original texts) and metrics produced by machine summarizers we will not be able to compare those normalized values. That is, we need to normalize by all results we get for a specific metric. The step after all metrics are done is called *reducer* (the name is inspired by functional programming – there is a ‘reduce’/ ‘fold’ function there which reduces the list to a one value). As one can guess, reducers are defined in container itself and used in the algorithm after all reporters are done.

We would like to say a note about chosen approach in general. Originally, we would like to use so called ‘stealing pool’. The idea of stealing pool is that it knows about work chunks and some threads, instead of doing nothing, steal the work chunks. We know that it sounds too fuzzy, let us introduce an example to clarify the said. One who has carefully read the last paragraph maybe noticed that reporters’ list of futures is submitted to the same thread pool. Why to do so? Why can not we just wait on till one reporter is done and then starting the next reporter? This is done to avoid thread pool starvation on the finishing of this list. In the end, we will wait till the last job of the pool is done but other threads will do nothing. Moreover, we need to process the result in some manner – we need to combine all results and save to disk, the thread pool is idle in this time too. Thus, if we post combining and further processing of the results, we avoid the pool starvation. However, the issue with such approach is that the thread pool should be bigger than the number of reporters because it is possible to have a deadlock (if all thread pool threads are busy by ‘finishers’ there is no place to get the actual job done – the thread pool is blocked forever). Naturally, we are not first who have been thinking about this. More general approach is to have a Fork/Join pool. This is a stealing pool which recursively forking the jobs and joining the results together after all results processed. The idea of stealing here is on the join phase – that is, the joining thread instead of waiting steals jobs from others. The beauty about this pool is in word recursively. If in our case the design is still flat (only reporters fork the task into jobs), Fork/Join pool allows a nested level of fork/join operation. One who is curious enough can look here (Doug Lea, 2000). Java from the version 7 has built-in fork/join pool. Why we would not taken this approach lies in the implementation details. The huge problem of practice that breaks the rainbow theory lies in the fact that Java Virtual Machine is still not an Operating System. How exactly join phase steals the jobs? It should create/use an addition thread to *simulate* stealing. All JVM implementations have a threading model built on top of the real OS threads. Context switching is out of scope of a JVM. However, *the correct implementation requires context switching being a part of the fork/join mechanics*. More on this one can read here (Edward Harned, 2016). It has a big list of reference for one who want to extend her knowledge on the subject. For us, in practice, while trying to implement concurrency within fork/join mechanics the journey ended up by uncontrolled number of threads. This is, in turn, hard to maintain, debug and understand. As well, this may even slow the execution since threads are OS Kernel primitives which are not so lightweight by our opinion and their creation maybe much heavy than a processing of metric comparison. We do not claim that it is totally useless, but we have preferred a manually controlled solution from the above reasons.

### 5.4.2. Immutability

One of issues while going into concurrency is data consistency. From the one hand thread pools almost eliminate explicit control of execution through the synchronization primitives (in case of Java it is either built-in synchronization on monitors – binary semaphores (each of which lives in every object in Java and controlled by ‘synchronized’ keyword) or ‘java.util.concurrent’ package that defines more primitives and constructions). From the other hand, nothing/nobody but programmer controls the data consistency in the program.

More specifically, within wide spreading of multi-core CPU how do we guaranty that one thread on one CPU core will use the correct data of another CPU core? Suppose we had prepared a chunk of work and submitted it into our thread pool. How do we guaranty that a processing thread from the thread pool will get the right prepared data? This is not a fantastic scenario. There could be situation when other thread will get inconsistent data. The issue is fully described in (Brian Goetz, 2006). *One of possible solution is to use immutable data structures. Immutable data structures (following their name) must not change after creation. The new state of object is a new object. As well, designed correctly they should be safe for inter-thread communication*. The overall design is described in (Joshua Bloch, 2018).

For example, let us look on the Score class of ROUGE metrics. Partially it looks so:

**public class** Score **implements** Serializable, ReportedProperties {  
  
 **private final double precision**;  
 **private final double recall**;  
 **private final double alpha**;  
 **private final** Double **f1**;  
  
 **public** Score(**double** alpha, **double** precision, **double** recall) {  
 **this**.**alpha** = alpha;  
 **this**.**precision** = precision;  
 **this**.**recall** = recall;  
 **double** factor = (1 - alpha) \* precision + alpha \* recall;  
 **if** (factor > 0) {  
 **f1** = (precision \* recall) / factor;  
 } **else** {  
 **f1** = **null**;  
 }  
 }  
  
 **public** Score(**double** precision, **double** recall) {  
 **this**(***DEFAULT\_ALPHA***, precision, recall);  
 }  
  
 **public double** getAlpha() {}  
  
 **public double** getPrecision() {}  
  
 **public double** getRecall() {}  
  
 **public** Double getF1Measure() {}  
}

This class fulfills requirement of being immutable. No member of the class instance can mutate after its initialization. As well, ‘final’ keyword on class members forces immutability within the object itself (‘final’ keyword has a special meaning in JMM – Java Memory Model in general that really makes the object immutable for cross-core CPU passing. One interested on subject could look for more information for so called ‘memory barriers’).

Although the pattern for immutable objects in Java is simple, from the consumer perspective their usage is not so comfortable. Immutable object should be initialized completely by the object constructor. One can see the first object constructor. It expects three double parameters. Within real development remembering of what all of them are could be cumbersome, especially when the number of parameters is not a trivial list. From the other hand having setter methods on the object is impossible (as with Java beans). The solution for that in Java is defining additional layer of abstraction – the Builder pattern. More information is available in (Joshua Bloch, 2018). In two words it is a fluent interface which already discussed which is not immutable but build the immutable object (regularly, this class is an inner class of immutable class). For instance, for the discussed Score it is defined as:

**public static class** Builder {  
  
 **private** Double **precision**;  
  
 **private** Double **recall**;  
  
 **private** Double **alpha**;  
  
 **public** Builder precision(**double** precision) {  
 **this**.**precision** = precision;  
 **return this**;  
 }  
  
 **public** Builder recall(**double** recall) {  
 **this**.**recall** = recall;  
 **return this**;  
 }  
  
 **public** Builder alpha(**double** alpha) {  
 **this**.**alpha** = alpha;  
 **return this**;  
 }  
  
 **public** Score build() {  
 **if** (**precision** == **null** || **recall** == **null**) {  
 **throw new** IllegalArgumentException(**"Either precision or recall is not provided."** +  
 **" Both should be provided"**);  
 }  
 **if** (**alpha** == **null**) {  
 **return new** Score(**precision**, **recall**);  
 }  
 **return new** Score(**precision**, **recall**, **alpha**);  
 }  
}

Its usage may look so:

Score score = **new** Score.Builder()  
 .precision(precision)  
 .recall(recall)  
 .build();

For a big list of parameters, it could be a benefit of using Builders.

Summing up this section, we should mention that immutable objects have many benefits and, in many cases, have cross-system influence. For instance, even designing mutable abstract data types with assumption that inner objects will be immutable is much painless and memory consuming than assuming that inner data could change. For instance, all Java collection framework has a weak requirement that it works with immutable types (weak since this check is not forced but assumed). If one pass into, say, Hash Set an object which is mutable and do not care about not changing the object further will have bugs since the object sits in certain bucket of the hash set. Once the object is changed it is (could be) already not belonging to the bucket it settled in, and the hash set will be corrupted. The same is about tree-based data structures. Changing the object which is settled in the tree will not force for tree reconciliation. A tree will be corrupted. Assuming mutability within abstract data types will require either deep cloning or another exotic strategy. By the way, cloning for immutable data types absolutely has no meaning since two objects with identical state are identical. Within some circumstances, equality of two objects can be checked only by checking the object reference identificatory. More information could be found in already mentioned (Joshua Bloch, 2018). The more advanced topics can be found here (Chris Okasaki, 1999) which explains how to construct abstract immutable data structures. I.e. how, for instance, do we construct immutable list? Simple copy strategy for new list state is a naïve strategy. However, one designing a program solution should be careful to not overcomplicate the logic with immutability. Designing a picture pixel matrix with immutable object is an overkill.

### 5.4.3. Convention

By our opinion the topic of convention is underestimated in software engineering. By convention we mean a set of rules/agreements that one should follow. We think that the subject is not so popular in academic researches. However, there is some (weak by us) attention in open source and commercial product worlds. Although one could say for instance object-oriented patterns are convention and have well acceptance even in academic word, but patterns are not convention in broad terms of speaking. If you follow OOP patterns as they are suggested (maybe) you will get an elegant solution for your problem, but it is only your problem and your part. Others are free not to solve their solutions in the same way. Here, by convention we mean almost an ‘obligation’ to follow the same agreements in overall project.

It is not by occasion that we have not call this section ‘code convention’. Although code conventions are important part of the code but by code convention, it is regularly assumed the agreements of how to give names for types, variables, constants, starting code blocks, etc. For the Java language it is strange if the variable contains underscore as word delimiters, but no official code guideline exists as far as we know (The Python language goes further and has an official Python Enhancement Proposal – PEP, which defines the language style – PEP 8). There is a work (Michael Smith, 2011) that proposes following (Java) code conventions as a metric for measuring maintainability.

We think that following common conventions not only in code but in overall structure helps, first, in reducing ‘entropy’. The, second, more important by us thing is to create similarity within all part of the system. *We think that human cognitive properties as repeatability if used properly should reduce the learning curve as well help with understanding*. For example, Java today has almost completely switched to declarative build systems as Maven/Gradle. The heart of those systems is a slogan ‘convention over configuration’. In the simplest form it means the common directory structure for the projects: ‘src/’ for sources and ‘target/’ for binaries output. As well, source directory is structured in the following way: ‘src/main/java/’, ‘src/main/resources/’, ‘src/test/java/’, ‘src/test/resources/’. *Such a small mental shift to convention instead of procedural builds saves now tons of time because whoever familiar with one project in this structure exactly knows what to expect in another project without any need to learn the actual build pipeline*.

We have taken it is a good practice. The source code packages are defined in the same structure. The project source code is split into two parts: ‘core/’ and ‘concrete\_metric/’. Core contains (abstract) entities which already discussed as processors, reporters, etc. It also contains other things for GUI, statistics and so on. Concrete metrics contain the metric related implementations of discussed entities. Figure 5. shows the high-level package structure.

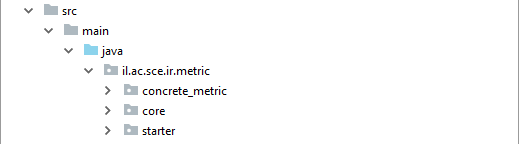


Figure 5.

Figure 6. shows the structure of core sub package.

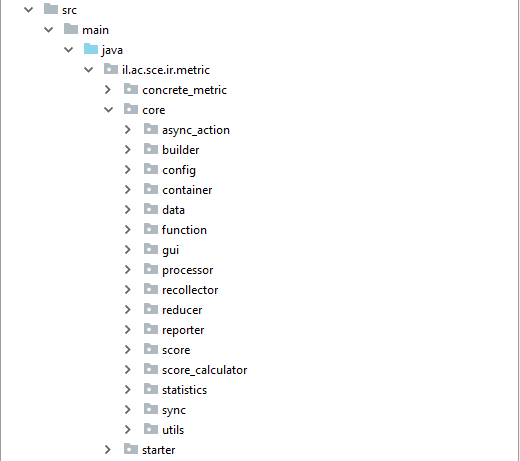


Figure 6.

The most things are already familiar to one who has read previous section. Utilities directory is an unavoidable directory which either contains some common functionality that should be used by concrete topic or an entity that hard to decide about classification.

On Figure 7., one can observe the ‘proof’ of said – concrete metrics are mimicking the core. If one in a future would like to extend the library. We suggest him to follow the convention. (‘elena’ sub package stays with that name because of historical reasons. This package is responsible for readability metrics implementation)

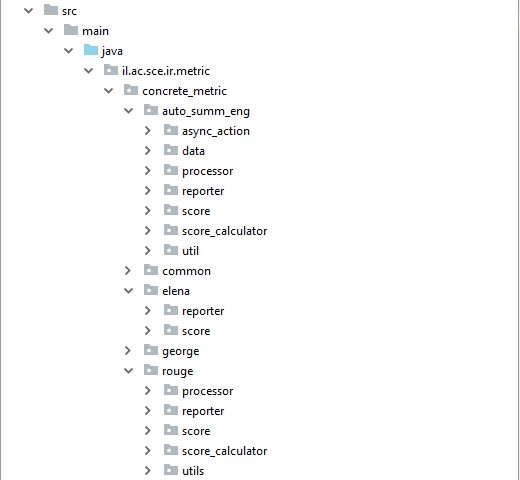


Figure 7.

In the end of this section we want to suggest a book (Donald A. Norman, 2013) that is not related to programming. It is related to design of things in general. The design of source code in many aspects more an art and experience than science (a computer does not care about directory structures – this information helps people). The design of surrounding us things in many cases can be an inspiration for the code design (at least reducing the number of strange sometimes even scary or ridiculous places in code).

## 5.5. Platform as Library – Conclusion

The project has the open source code. The amount of the source code has a nontrivial volume. Instead of trying to have every feature and class to be described we have given a conceptual overview for chosen design. We think that a one wishing to use the project as a library or even to extend it in some manner will be able to do so with almost no effort. We think that the project has a straightforward design. The pipeline definition of domain-specific language either allows the definition of custom pipelines in comfortable and easy manner or, if somebody would like to define additional operators she could do it easily too. We also think that IoC approach allows to see overall code structure in one place and it should not be an issue to understand how inner parts work. As well, concurrency and immutability should not scare away since they are explained too. After all, the convention principle that we tried to follow should allow faster understanding of different parts of the system. Eventually, we tried to describe the design foundation that will allow easy navigation and understanding of source code which by our opinion is not hard to understand and extend because of chosen design principles. There are many other parts of system that are out of scope. Their usage will be explained in next chapters (as well the reference to source code will be given too).

# 6. Platform as GUI Client

In this chapter we will present the usage of the platform from the user perspective. We will explain how to install, run and use the GUI client. As well, we will explain how to prepare input data for processing.

## 6.1. Prerequisites

These are the basic requirements for running the software.

1. The software requires JRE 1.8 installed on the system.
2. Some features are only available with R language installed (tested with version 3.5.2).
3. Both Java and R binaries should be set into system Path environment variable.
4. The system tested on a machine with 8GB RAM. There should be available free disk space (~2GB).
5. *It is assumed that the software runs on Windows OS*. Some features (R language integration) assume Windows OS and will not work on other operating systems.

Both Java and R language could be freely downloaded and installed. The Java language downloads are available on <https://www.oracle.com/technetwork/java/javase/downloads/index.html> [accessed at April 2019]. The R language is available here: <https://www.r-project.org/> [accessed at April 2019].

After installation, directories with binaries of Java and R language should be available on Path environment variable. This page <https://www.java.com/en/download/help/path.xml> [accessed at April 2019] explains how to set Path variable on system level.

After the Path variable is updated a small check could be performed to ensure their availability.

1. Press WinKey + R on keyboard. Run dialog window should appear (Figure 8.).
2. Type “cmd” and press Enter. System command line should appear (Figure 9.).
3. In command line window type “rscript --version” and then “java -version”. Both commands should end up successfully by replying its product version (Figure 9.).

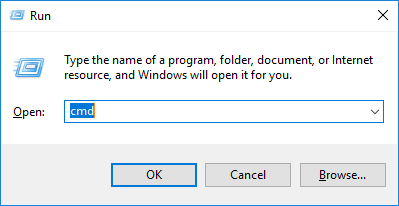


Figure 8.

exit

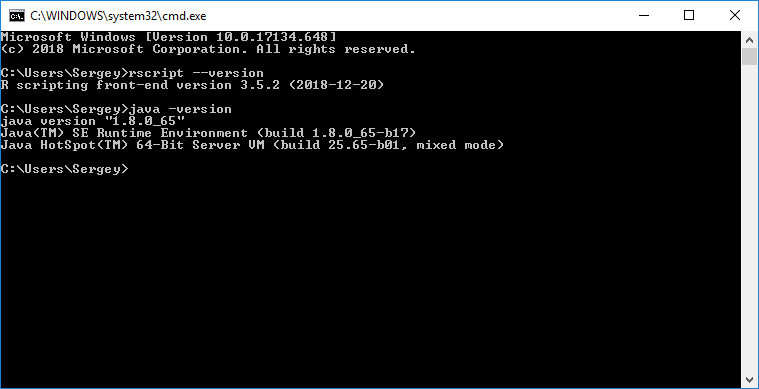


Figure 9.

### 6.1.1. R Language Libraries

R language distribution should include additional libraries for all features of the GUI client functioning properly. Two additional libraries should be included:

* agricolae
* jsonlite

The simplest way installing them is through the R GUI client:

1. Press WinKey + R on keyboard.
2. On the run dialog window type “rgui” and press enter. R GUI client should appear (Figure 10).
3. Choose in menu “Packages” -> “Install package(s)”. The list with available packages should appear. Find and install the needed packages. (You may be required to set up the repository. Under “Packages” -> “Select repositories” choose CRAN. Under “Package” -> “Set CRAN Mirror” choose an appropriate mirror if default does not work)

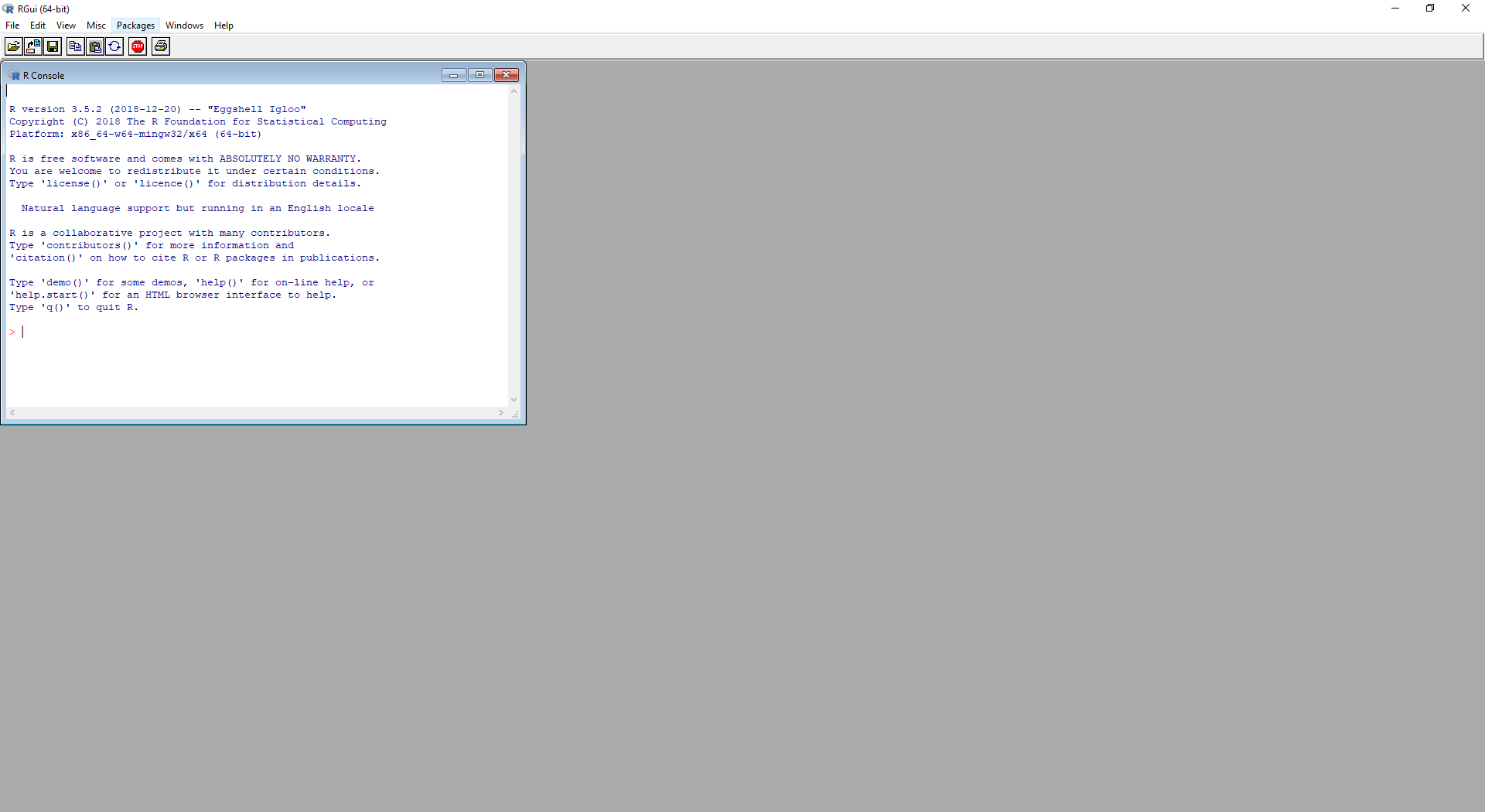


Figure 10.

## 6.2. Installation & Execution

The executable artifact of the platform is so called ‘uber jar’. Uber jar contains all libraries a software is required for. From the one hand, it significantly increases the size of artifact. From the other hand, it simplifies to a total minimum deployment/installation procedure. It does not require installation wrappers. We have been trying to follow the deploy everywhere and zero configuration approaches. However, we suppose that one willing to run the platform has the artifact. Thus, the procedure looks like:

1. Copy the artifact into the desirable directory (to not mix things together we assume that the directory is initially empty – Figure 11.)
2. Double-click on the artifact (correct Java installation should register handler for running jar files in Windows OS Explorer). The main platform window should open (Figure 12).

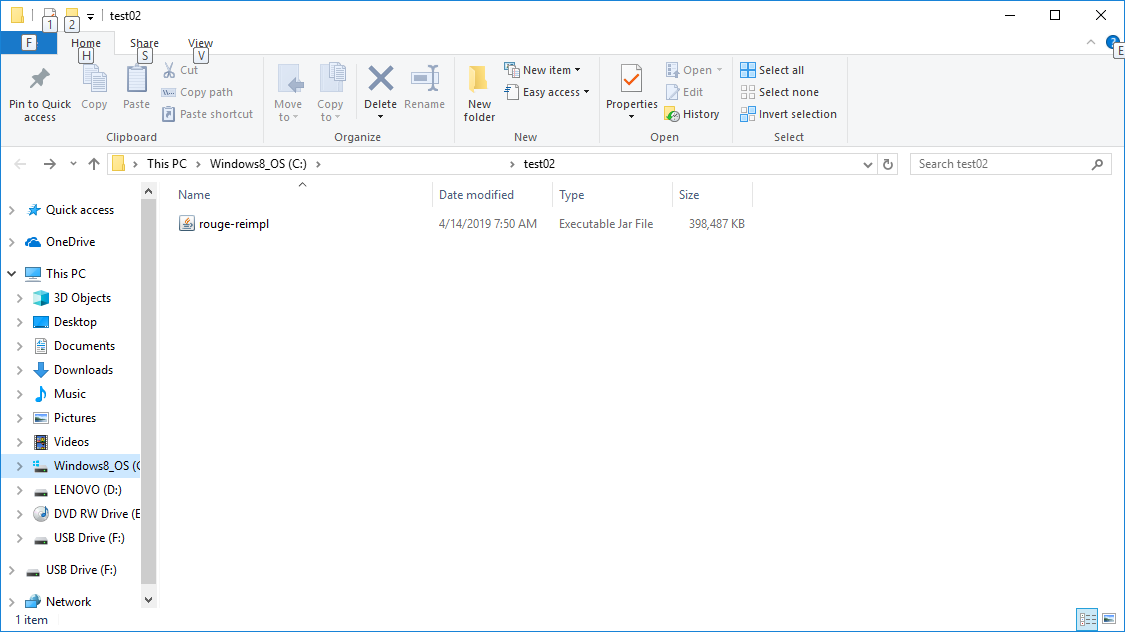


Figure 11.

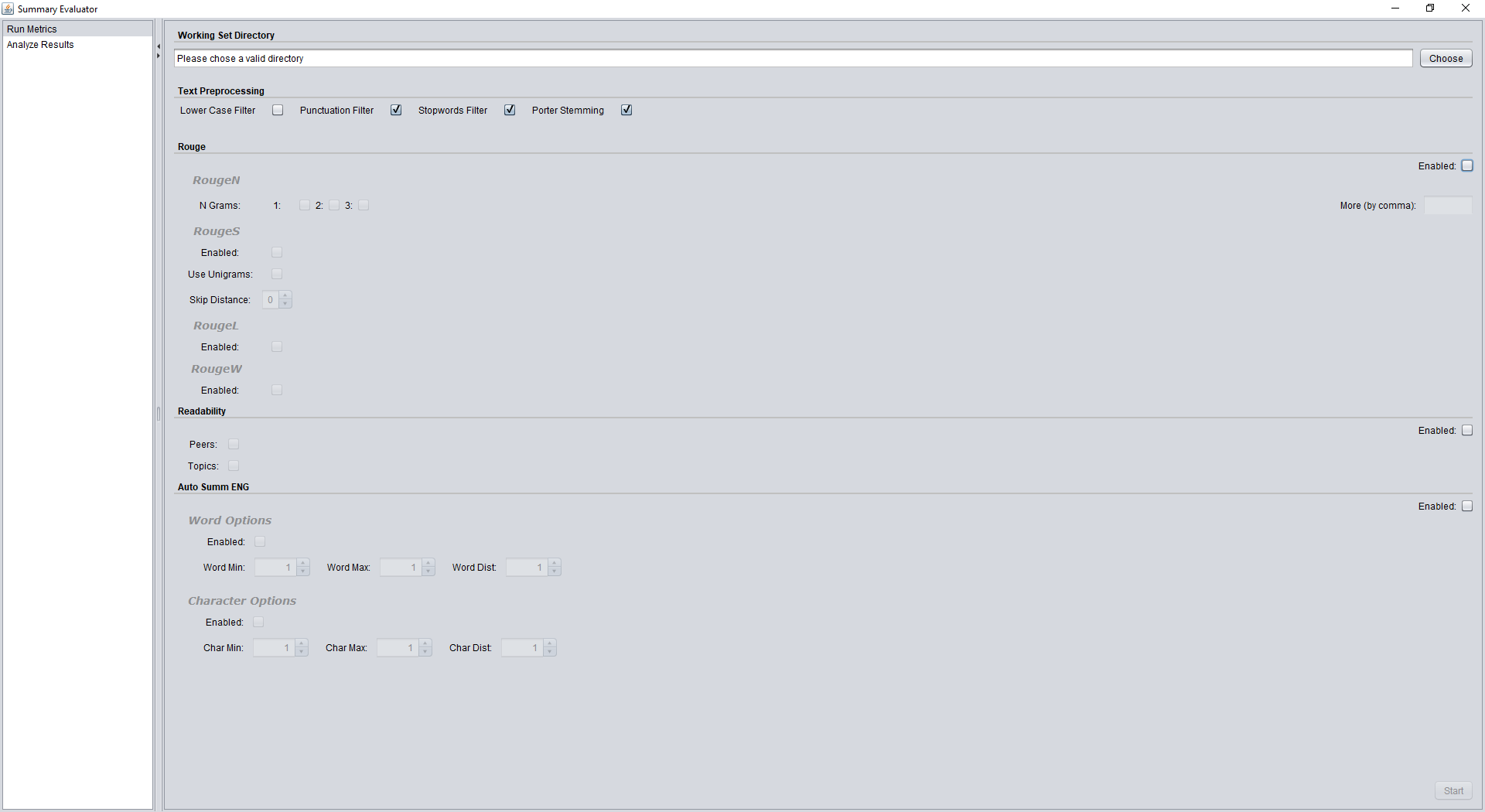


Figure 12.

### 6.2.1. Error Analyzing – Technical Notes

Error logging and analyzing is one of the cornerstones of application development. A ‘silent’ application would be hard both for understanding and tracking underlying processes. We would not allow to ourselves to create such an application. Therefore, we tried to follow a correct error handling within the code itself:

* Use logger mechanics (technically, the bundle of SLF4J + Java Util Logger is used)
* Log the exception on upper levels only (some programs suffer from too expensive logging by printing one exception almost on every method which makes very hard to understand what original exception is)
* Use ‘one source of truth’ logger output (log messages printed out to one dedicated place)

The SLF4J within current configuration prints into the standard output only (output error stream of process). This is by design – the program now is not a long running server process. If one will change the code to run under server logger configuration for her will be correct bundling SLF4J with an appropriate configured logger. Secondly, one can run the client either with redirected output or just printing into console. (Console output could be achieved by executing uber jar from the console. Figure 13 shows how to run the command line with Windows explorer, Figure 14 depicts how the command to run from console (“java -jar”). Figure 15 shows the state of console after running.)

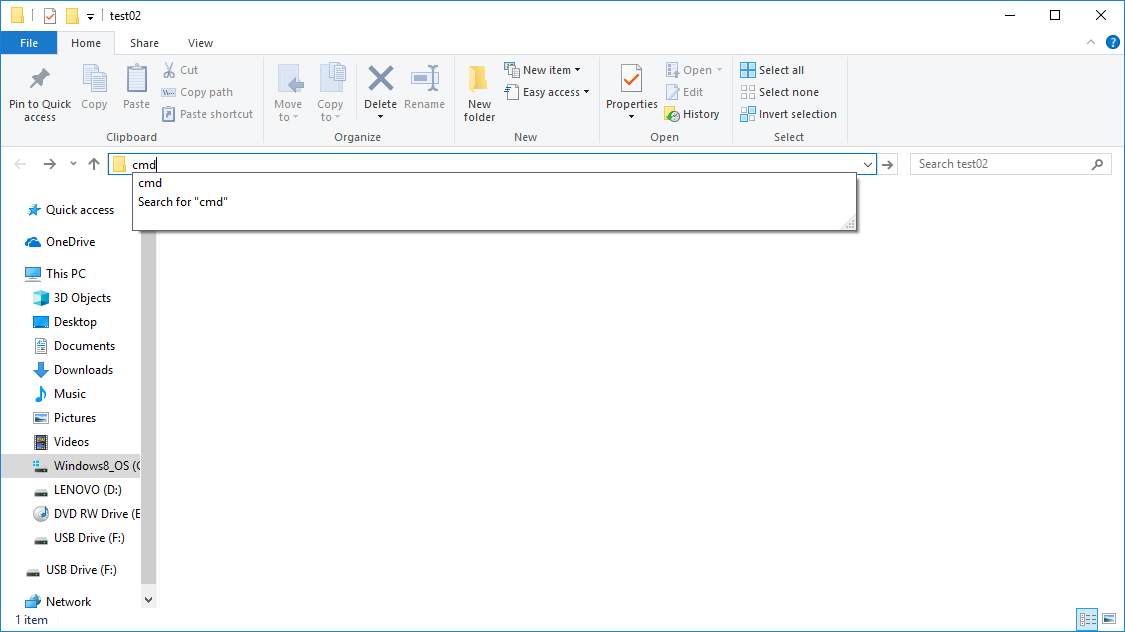


Figure 13.

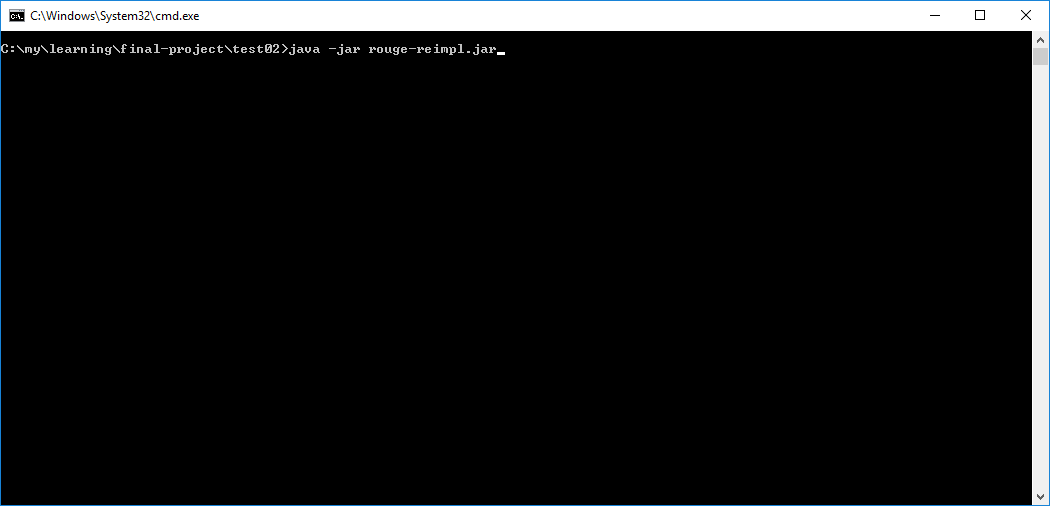


Figure 14.

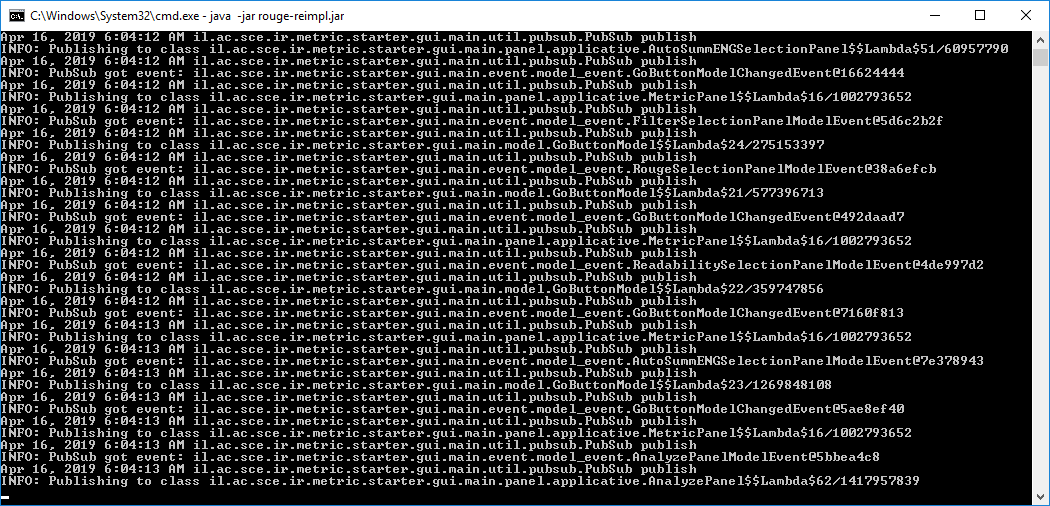


Figure 15.

### 6.2.2. Source Code Compilation – Advanced

In the upper section of installation, we omitted and assumed that user has the final artifact somehow available for him. However, honestly speaking, we do not have such distribution mechanism. This is only possible if user will copy from somewhere the artifact. From the other hand compiling of source code should not be hard too.

These are prerequisites for compiling:

* JDK of version 8 is installed. If not, it may be downloaded from <https://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html>
* Maven build system is installed. If not, it may be downloaded from <https://maven.apache.org/download.cgi> (version 3.2.5 is used for development)
* Git SCM is installed. If not, it may be downloaded from <https://git-scm.com/downloads>
* The source code is located at <https://github.com/karno-bh/rouge-reimpl>

Execute the list of following commands from the command line:

1. git clone https://github.com/karno-bh/rouge-reimpl.git
2. cd rouge-reimpl
3. cd lib-deploy
4. <run all .bat files in directory>
5. cd ..
6. mvn clean install -DskipTests=true
7. cd target

In ‘target’ directory there should be ‘rouge-reimpl.jar’ which is the executable artifact.

## 6.3. Input Preparation

Once the executable artifact is available and has been successfully executed, the input should be prepared to run within in the client. The input is a directory that should have the following structure:

* category01
  + models
  + peers
    - system01
    - system02
  + topics

‘category01’, ‘system01’ and ‘system02’ are given as an example. ***They might have any name but the usage of underscores and spaces within names is not recommended since underscores are used internally as separators for further processing and the results are unexpected. If the word separation is required, dashes or camel-case names should be used***. ‘models’, ‘peers’ and ‘topics’ are hard-coded expected names of the input structure. ‘peers’ directory should include the set of summarizing system summaries. ‘models’ directory is the place for the human (or golden) summaries. ‘topics’ directory should have original documents.

The system is designed so that it works with multi models and multi topics. The starting point for comparison is always a machine summary of the system. That is, suppose ‘system01’ has a file named ‘M000’. For metrics expecting a golden summary to be used it will try to find files that start with ‘M000’ ***prefix with a dot after prefix*** (for example, “M000.A.250”). The same for metrics that expect to have original documents as a reference (currently, these are readability metrics. For example, “M000.0.english”). It is possible to run and have results only for topics, but further, the name plays a key role with matching results of summarizing system to original document. The bottom line, all original documents and golden summaries are correlated by the name of automatic summary which is found in concrete system.

## 6.4. Running Metrics Evaluation

Once the GUI client of the platform is running (Figure 12) and input is prepared the platform is ready to run metrics. The screen is divided into two main functionalities (which is represented by list on the left side):

* Run Metrics
* Analyze Results

The “Run Metrics” screen is divided horizontally into 5 sections and “Start” button. These 5 sections are:

* Working Set Directory
* Text Preprocessing
* Rouge
* Readability
* Auto Summ ENG

The enablement of “Start” button is dependent on:

1. Selecting a working set directory (i.e. the directory where the prepared input is located)
2. Selecting at least one metric. (There is a checkbox “Enabled” on the right side of the metric section. A metric processing should be enabled. The concrete metric should be enabled too. For example, if Rouge is selected then at least one of Rouge-N, Rouge-S, Rouge-L or Rouge-W need to be enabled.)

In addition, the text preprocessing could be controlled from the “Text Preprocessing” section. Possible configuration of all available metrics is depicted in Figure 16.

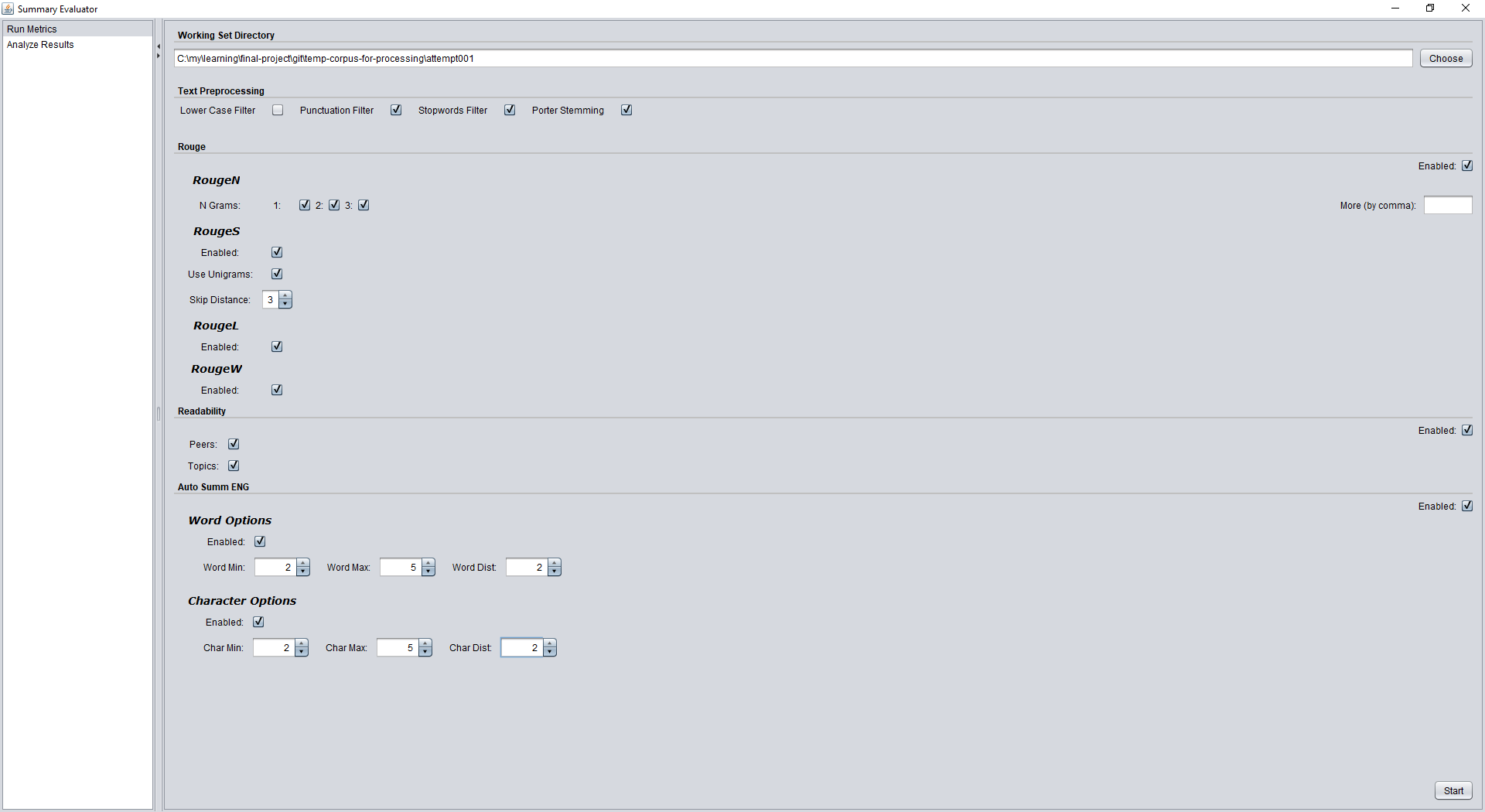


Figure 16.

Once the directory is set and “Start” button pressed, the actual processing starts. Until the processing is not done, “Start” button will be disabled. The progress of the metric calculations could be observed in logs.

The platform within a run will create following directories:

* cache (Inner purpose directory for caching heavy calculations based on text identifier. Technically, cache text processor described in Domain-Specific Language section)
* result

Result directory is an actual output of metrics’ execution. The output of metrics’ execution is a Microsoft Excel CSV friendly file (comma separated values). The CSV format chosen by design to allow smooth integration with Excel.

All metrics except AutoSummENG are split by files for automatic system summary. File name convention for such metrics is “<category>\_<system>\_<metric>.csv”, for example, “category01\_sysid01\_rougesu3.csv”. For the topics’ readability metrics (original documents) the pattern is “<category>\_<topic>\_<metric>.csv”, for example, “category01\_M000\_elena\_topics\_readability.csv”. In addition, there will be “reduced” directory under results. The purpose of this directory is to include for some metrics “reduced” values (i.e. calculated from the whole or part of metrics’ set results – explained in ‘Concurrency’ section). Figure 17 and 18 show the “result” and “reduced” directories accordingly.

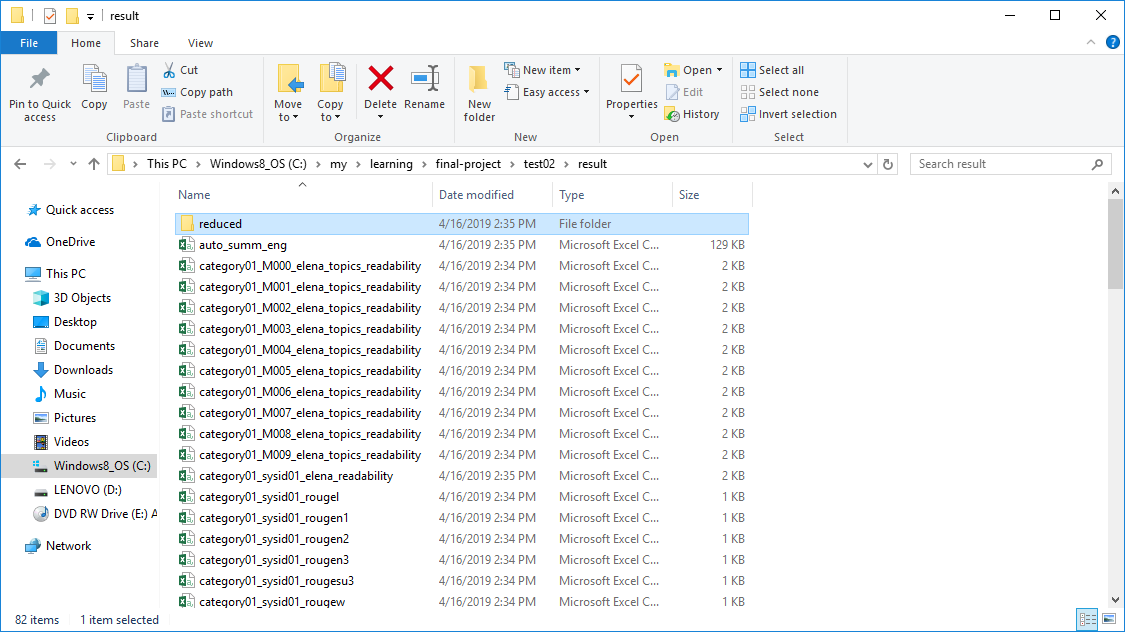


Figure 17.

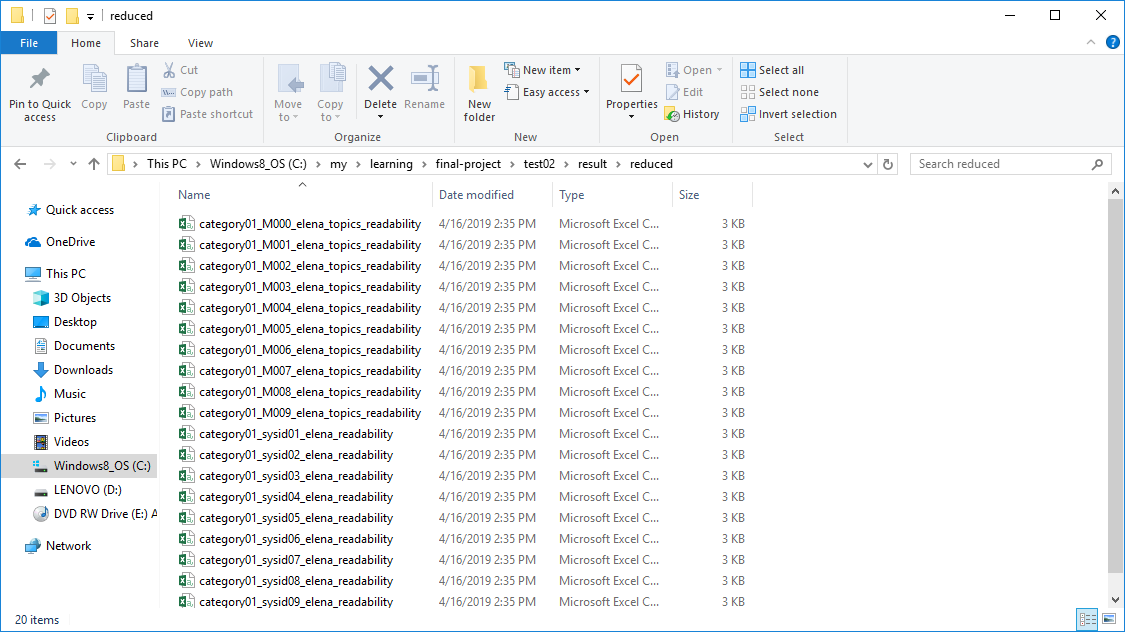


Figure 18.

## 6.5. Analyzing Metric Results

Other big feature of the software besides the metric evaluation is results analyzing. This feature is vital in the platform and allows graphics visualizations and table representation for further data understanding and analysis. The switch to result analyzing is available through the left list of main features. Once the user switched there the screen looks as on Figure 19.

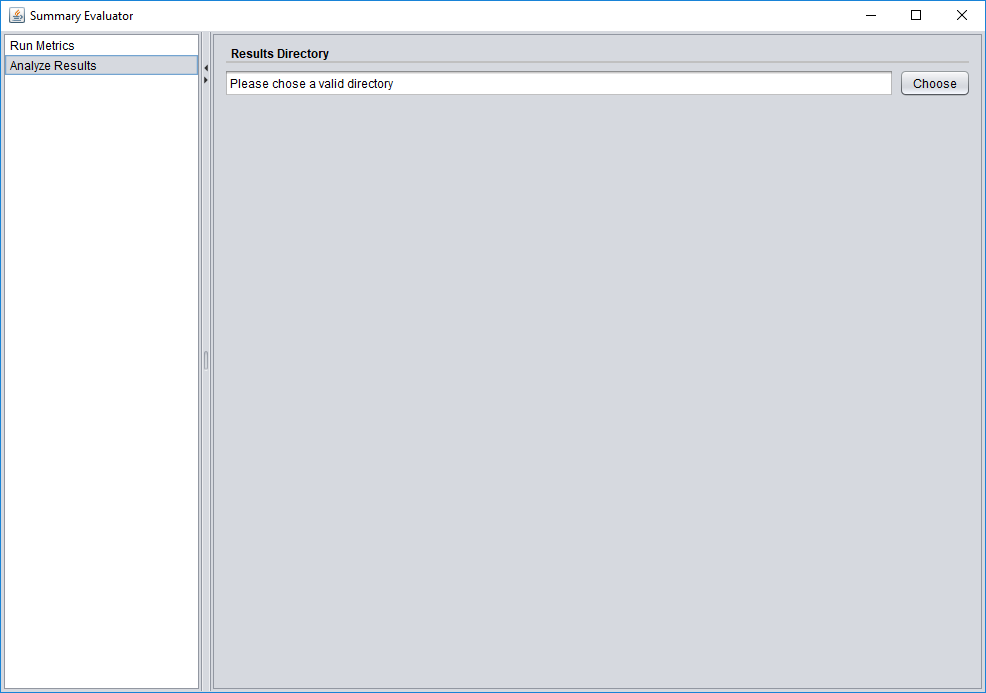


Figure 19.

Sure, to analyze results the platform, first, should be pointed to the results and expects some format of them. The pointing to results is done by specifying the “result” directory. The format is the set of CSV files mentioned in “result” directory. If there are results in “reduced” directory with the same file names those results overrides the results of “result” directory. Pointing to result directory and not directly passing the results from evaluation stage allows persistency of the results and decoupling from analysis.

Once the “result” directory is chosen the system load and parses the available metrics. The system builds dynamically the screen according to available metrics. For example, if all metrics are chosen (Figure 16) the result will be as on Figures 20 and 21 (it is the same screen in two different scrollbar positions).

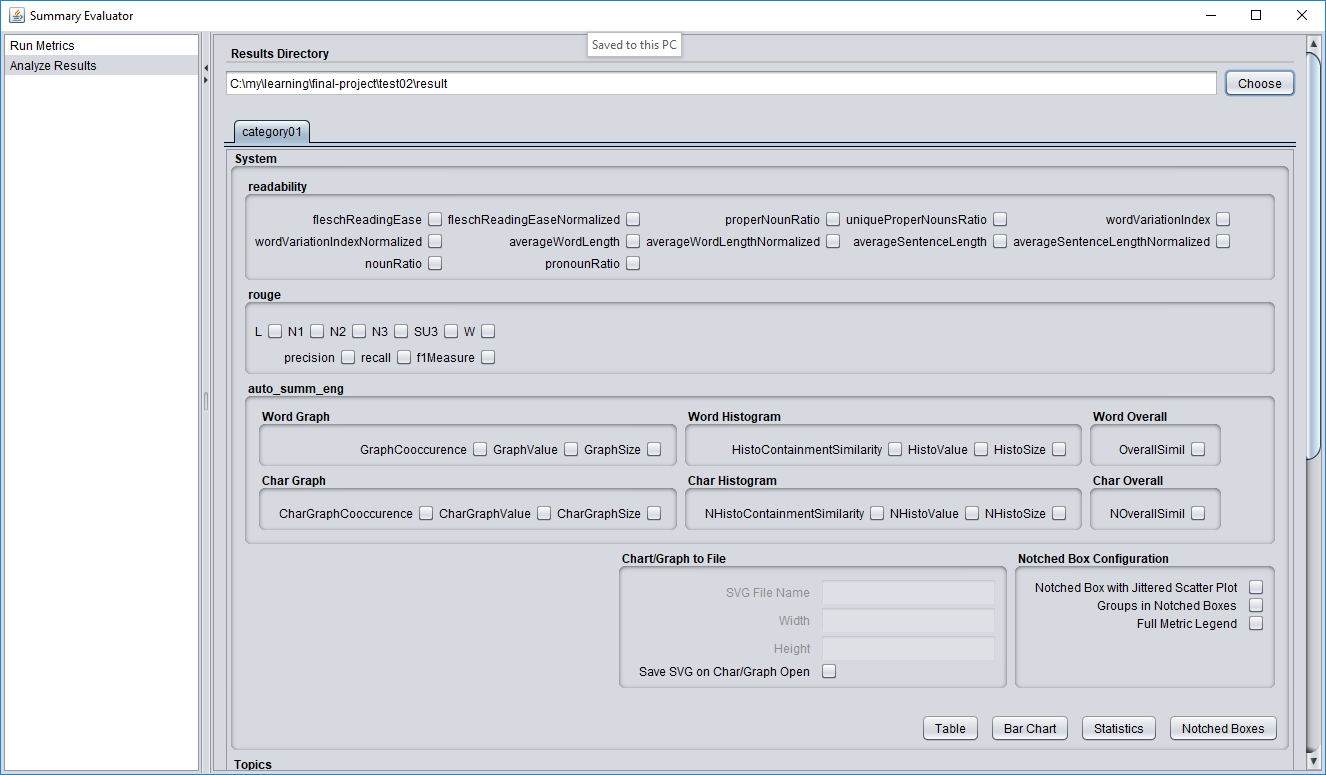


Figure 20.

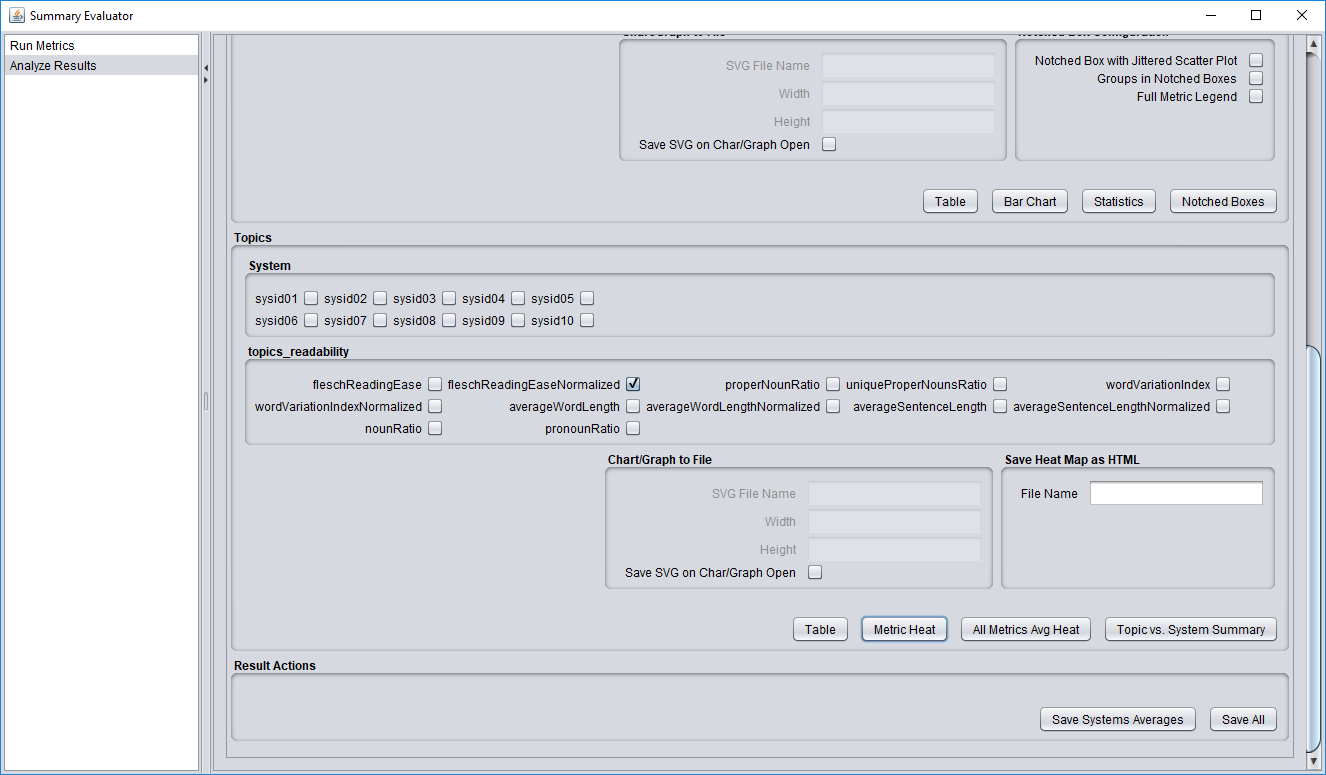


Figure 21.

As one can see the main screen is tabbed by category and the internal area is divided into three categories.

### 6.5.1. Analyzing Screen Sections – Overview

There are three sections as it was mentioned:

* System
* Topics
* Result Actions

Each section is a logical set of features. “System” section is responsible for analyzing results for summaries generated by summarizing systems. “Topics” section is responsible for analyzing metrics related to topics (i.e. original documents) themselves (currently there are only readability metrics). “Result Actions” section is a utility section that allows saving metrics in more programming language friendly format.

### 6.5.2. System Section

<place-holder>

### 6.5.3. Notched Box Graph

<place-holder>

### 6.5.4. Topics Section

<place-holder>

### 6.5.5. Metric Heat

<place-holder>

### 6.5.6. Result Actions

<place-holder>

# 7. Conclusions

<place-holder>

# References

Brian Goetz, T. P. (2006). *Java Concurrency in Practice.* Addison-Wesley Professional.

Chin-Yew Lin, E. H. (2003). Automatic Evaluation of Summaries Using N-gram.

Chris Okasaki. (1999). *Purely functional data structures.* Cambridge University Press.

Debian. (2019). *The Computer Language Benchmar Game*. (Debian) Retrieved March 2019, from https://benchmarksgame-team.pages.debian.net/benchmarksgame/

Donald A. Norman. (2013). *The Design of Everyday Things: Revised and Expanded Edition.* Basic Books.

Doug Lea. (2000). A Java fork/join framework. *Java Grandle*, 36-43.

Edward Harned. (2016, July). *A Java Fork-Join Calamity*. Retrieved from http://coopsoft.com/ar/CalamityArticle.html

Elena Lloret. (2008). *Text summarization: an overview.*

Elena Lloret, T. V. (2019). Are Better Summaries also Easier to Understand? Analyzing Text Complexity in Automatic Summarization. In *Multilingual Text Analysis: Challenges, Models And Approaches* (pp. 337-371). World Scientific.

Eric S. Raymond. (2003). *The Art of UNIX Programming.* Addison-Wesley.

George Giannakopoulos, V. K. (2009). N-gram Graphs: Representing Documents and Document Sets in Summary System Evaluation.

Giannakopoulos, G. K. (2008). Summarization System Evaluation Revisited: N-Gram Graphs.

Grady Booch, R. A. (2007). *Object-Oriented Analysis and Design with Applications, Third Edition.* Addison-Wesley.

Joshua Bloch. (2018). *Effective Java, 3rd Edition.* Addison-Wesley Professional.

Leonidas Tsekouras, I. V. (2017). A Graph-based Text Similarity Measure That Employs Named Entity Information.

Lin, C.-Y. (2004). ROUGE: A Package for Automatic Evaluation of Summaries.

Marina Litvak, C. A. (2015). HEADS: Headline Generation as Sequence Prediction Using an Abstract Feature-Rich Space.

Martin Fowler. (2005, June 26). *InversionOfControl*. Retrieved from https://martinfowler.com/bliki/InversionOfControl.html

Martin Fowler, R. P. (2010). *Domain-Specific Languages.* Addison-Wesley Professional.

Michael Smith, B. G. (2011). Code Convention Adherence in Evolving Software. *27th IEEE International Conference on Software Maintenance (ICSM)*, (pp. 504-507).

NIUT. (2013, September). *MorphAdorner*. Retrieved from http://morphadorner.northwestern.edu/morphadorner/

Oracle. (2019). *Java SE HotSpot at a Glance*. (Oracle) Retrieved March 2019, from https://www.oracle.com/technetwork/java/javase/tech/index-jsp-136373.html

Paul Ralph, Y. W. (2009). A Proposal for a Formal Definition of the Design Concept. In L. P. Lyytinen K., *Design Requirements Engineering: A Ten-Year Perspective, vol 14.* Springer, Berlin, Heidelberg.

Sasaki, Y. (2007). The truth of the F-measure.

Standish Group. (2014). *CHAOS Report*. Retrieved from Project Smart: https://www.projectsmart.co.uk/white-papers/chaos-report.pdf

Stanford. (2019). *Stanford CoreNLP*. (Stanford) Retrieved March 2019, from https://stanfordnlp.github.io/CoreNLP/

Thomas H. Cormen, C. E. (2002). Introduction to Algorithms, Second Edition. In *Introduction to Algorithms* (pp. 350-356). The MIT Press, McGraw-Hill Book Company.

TIOBE. (2019). *TIOBE Index*. (TIOBE) Retrieved March 2019, from https://www.tiobe.com/tiobe-index/