ARCHITECTURAL SOLUTION OF SYSTEM FOR SEMANTIC COMPARISON OF THE TEXTS

Vitalii Velychko

***Abstract**: The paper describes the process of designing an architectural solution of semantic text comparison. The reference text is compared with the base of texts in domain ontology. Ontology data storage is organized as a growing pyramidal network. A flexible architectural solution for semantic comparison of texts based on design pattern Visitor is proposed. Advantages of implementing this architectural solution aimed at adding comparison operations are represented.*

***Keywords**: Design patterns, semantic comparison of texts, visitor, fussy sets.*

Introduction

Software system for semantic text comparison can be developed in many areas, namely:

- possibility to recognize text from image files;
- defining the level of similarities considering grammar mistakes;
- processing hieroglyphs smiles and so on;
- to support visualization of comparison results;
- saving comparison history;
- visualize steps of comparison algorithms;
- recognize text in audio files;
- development of security aspects of the software system;
- to detect the same text in different languages.

The growing pyramidal network is one of the organizations of data information storage about problem domain. It allows storing facts about problem domain in trees of RDF triples. Thus, the operation of semantic comparison is based on

the comparison of two trees. One of them is extracted from text other trees are stored in the repository of ontologies. Otherwise, a successful semantic comparison of texts must be grounded on an extensible number of operations that should be added or removed flexibly.

Related papers and systems

Comparative analysis of software complexes of natural language processing

Measuring the similarity between texts is an important task of searching and processing texts. Software modules aimed to solve this problem are integrated into different systems in various applications. There are many different approaches to comparing texts. Let's identify their advantages and disadvantages and consider recommendations for a number of specific cases of use.

There are several groups of metrics that allow you to measure distances between texts: The following metrics allow to determine similarities based on

- editing;
- tokens;
- analysis of sequences;
- phonetic;
- hybrid.

Similarity-based editing - the more simple operations you have to perform to convert one line of text to another, the greater the distance between them. For example, the distance between the words "ert" and "wrt" is one, and the distance between "wer" and "sef" is two. This approach applies only to words and short phrases, and is not effective for longer texts. Also, this approach cannot take into account the semantics of words, because it compares only symbols. For example, the semantic distance between "banana" and "kiwi" is less than between "banana" and "mouse". This is why editing-based distances are used in applications where semantic meaning is not as important as similarity in writing. It is also worth noting that the most well-known editing-based algorithm is the Levenstein algorithm.

The similarities based on tokens are a bit more complicated. They analyze the text as a set of tokens (words). This allows you to take into account semantics and process large texts. The semantic meaning plays an important role here, as you can use word vector representations (word2vec) to describe each word in the text and then compare the vectors. Using this approach and the TF-IDF method allows you to compare the semantic similarity between whole texts (although not between independent words). Token-based similarities are widely used in various fields. However, this approach does not apply to a number of cases.

The next group of distances is sequence-based distances. This is a bit like defining syntactic distances. The longest common subsequence is ignored if there are some letters between the characters in the subsequence. For example, consider the letter sets "aebcdnlp" and "taybcrd". The longest common subsequence between these words is "abcd", while the longest common subsequence is only "bc".

Phonetic algorithms form a separate group of methods for comparing strings. In this case, it's not even a line comparison, but rather an audio comparison. These algorithms compare words based on how they are pronounced. It is difficult to compare long texts in this way. Short sentences or phrases are the maximum threshold for these algorithms. In addition, they cannot take into account the semantic meaning.

The following several algorithms (i) compare strings based on similar prefixes or postfixes; (ii) algorithms called "length distance" and "identity similarity". The first compares the strings, counting the number of characters in each of them, and the second algorithms simply check whether these strings are exactly the same or not.

The last group is hybrid algorithms. Consider the method: Monge-Elkan. This is a combined method that takes into account syntactic distance distances and tokens. The Monge-Elcan method compares every word in one text with every word in another text, but when comparing words it uses some methods based on syntactic distances. Then the distances between the words are aggregated to get a single value of the distance between the two texts. The important thing here is that this method is not symmetrical. This means that the result of the

comparison depends on which line you take as the first line and which as the second.

Comparative analysis of approaches allowing to process texts is represented in figure 1.

Text Distance Infographics

CATEGORY	APPROACH	⊕ PROS	⊖ CONS	ALGORITHM	AREAS
Edit based Similarities	compare two strings by counting the minimum number of operations required to transform one string into the other	+ Simplicity + Good for short strings	– Inefficient for longer strings – Computationally expensive – Doesn't take into account the semantic meaning	Hamming Levenshtein Damerau-Levenshtein Jaro-Winkler Strcmp95 Needleman-Wunsch Gotoh Smith-Waterman MLIPNS	Spelling correction Duplication search DNA analysis Correction systems for OCR Measuring the linguistic distance of the languages
Token based similarities	compare two strings by looking at units (tokens) of the strings (for example, words, n-grams etc)	+ Computational efficience + Applicable for long texts + Can take into account the semantic meaning	– May be not very good for single words or short phrases from several words – Magnitude of the vectors doesn't play any role in comparison for some algorithms	Jaccard index Sørensen-Dice coefficient Tversky index Overlap coefficient Tanimoto distance Cosine similarity Bag distance	Computer science Text mining Many general NLP problems Bioinformatics Information retieval
Sequence based	compare two strings by looking at different subsequences of the strings	+ Simplicity + Good for short strings	– Inefficient for longer strings – Computationally expensive – Doesn't take into account the semantic meaning	Longest common subsequence similarity Longest common substring similarity Ratcliff-Obershelp similarity	Computer science Bioinformatics Version control systems
Phonetic	compare two strings by comparing how do they sound in pronunciation	+ Allows to analyze another side of the language: pronunciation + Good for short strings	– Inefficient for longer strings – Doesn't take into account the semantic meaning – Needs special preprocessing of the strings – Applicable only for very specific tasks	MRA Soundex Metaphone NYSIIS Editex	Names retrieval Database software
Simple	compare two string by looking at some simple measurements of each string	+ The simpliest algorithms + Good for short strings + Computational efficience	– Inefficient for longer strings – Doesn't take into account the semantic meaning – Very primitive – Applicable only for very specific tasks	Prefix similarity Postfix similarity Length distance Identity similarity	Computer science
Hybrid	combines edit based approach with the token based approach	+ Can take into account the semantic meaning of the word in a text + Can be used for texts of any length + Performs better in texts with misspellings + Flexibility: you can pick any edit based similarity measure	– Can be computationally inefficient for long texts (depends on the edit similarity) – It is not symmetrical	Monge-Elkan	General NLP problems

Created by ActiveWizards

Figure 1 Comparative analysis of approaches to process texts

Figure is taken from

https // activewizards.com/blog/comparison-of-the-text-distance-metrics/

In other words, there are many cases of using text comparison methods in which symmetry is not important.

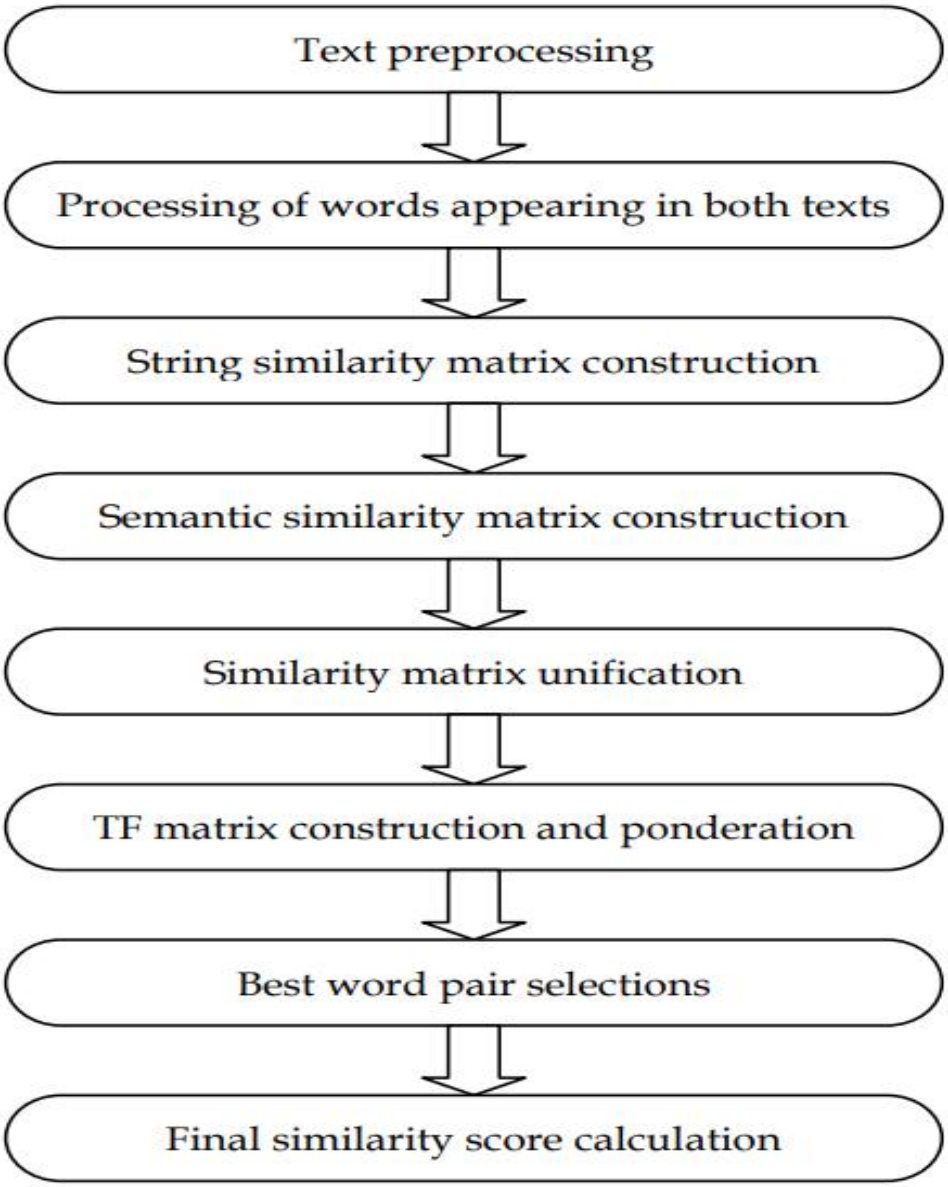
Authors of SEMILAR (Rus et al., 2013) proposed an approach to compare texts. This approach is based on a collaboration of different levels of similarities between texts. One of them is the word to word similarities. Then more complex levels of similarities are proposed, namely, sentence to sentence, text fragment to fragment. SEMILAR includes implementations of algorithms for solving the problem of defining of text-to-text semantic similarity.

Authors (Furlan et. al., 2013) proposed an approach when the comparison of texts based on their statistical analysis. Mathematical apparatus, designed by authors, aimed to calculate the differences between texts. Authors specialized in small text analysis. That's why structural differences are more related to semantic similarities or differences. The essence of the proposed approach is represented in figure 2.

Authors (Harispe et. al., 2014) proposed open-source software solutions dedicated to measures of semantic similarities of different texts. It uses Open Biomedical Ontology (OBO) Foundry or BioPortal (Smith et al., 2007; Whetzel et al., 2011) provide access to hundreds of biomedical ontologies expressed in various formats, e.g. Resource Description Framework (RDF), OBO, Web Ontology Language (OWL). These structured vocabularies are used to characterize entities through conceptual annotations. The general structure of libraries involved to the measurement of semantic similarities of the text is represented in figure 3.

Such a library can be used for computations and analyses of semantic similarities between terms/concepts defined in terminologies and ontologies. The comparison of entities (e.g. genes) annotated by concepts is also supported. A large collection of measures is available. The software project is based on JAVA library, as well as a command-line tool that can be used on personal computers or computer clusters.

The advantage of the proposed software solution is using of open source and open architecture approaches to modify the software to support a new combination of measurements.



.

Figure 2. Workflow of the process of semantic texts comparison based on statistical text analysis

Figure is taken from [Rus et al., 2013]

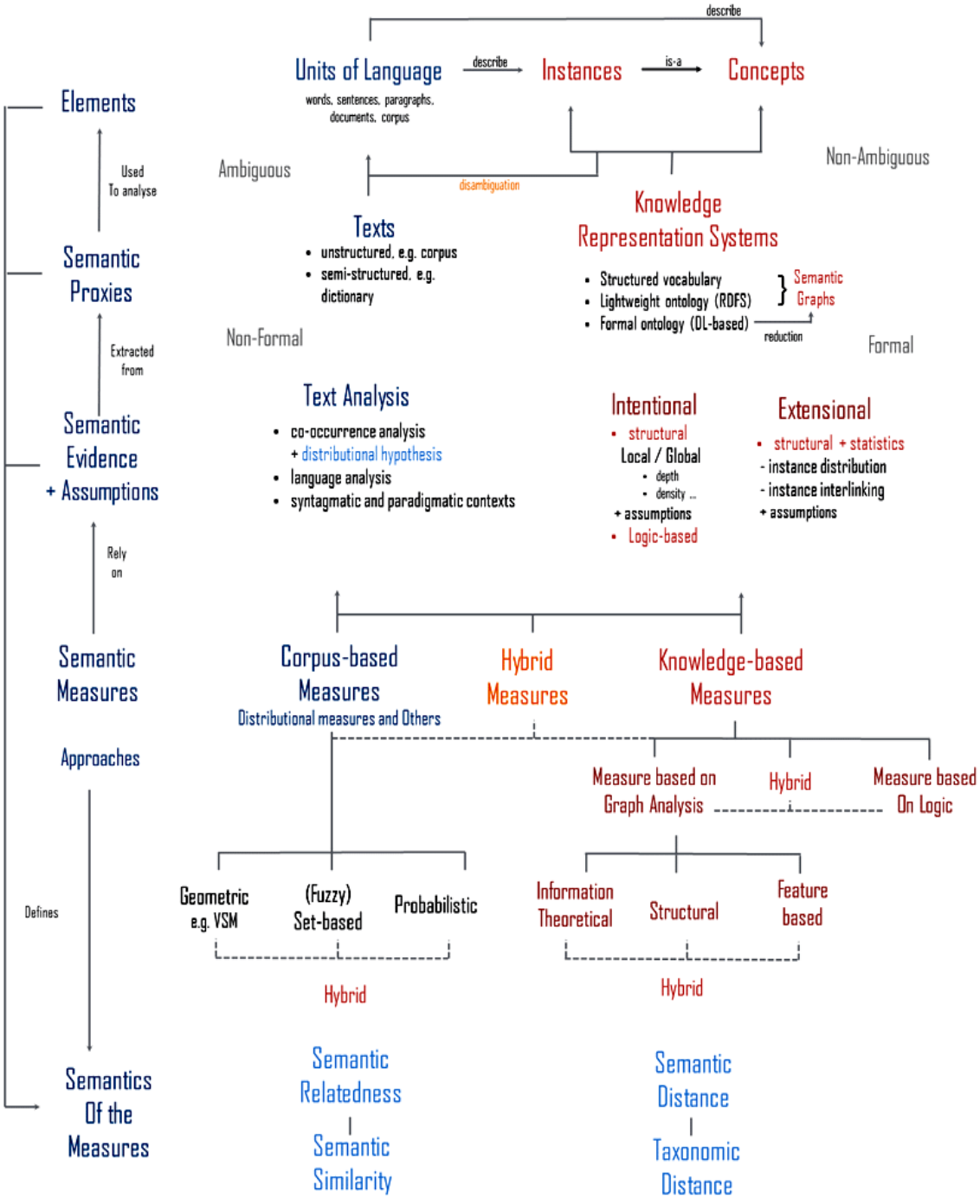


Figure 3. Overview of the types of semantic measures which can be used to compare various types of texts' elements (Harispe et al., 2013b)

Task and research questions

To design an architectural solution for flexible semantic comparison based on possibilities of adding operations and supporting principally new operations for semantic comparison of texts.

RQ1: Propose a requirement specification of a system for semantic comparison of texts.

RQ2: Choose a design pattern for designing such an architectural solution.

RQ3: Design an architectural solution, represented as UML class diagram, implementing composed requirement specification.

Requirement specifications of a system for semantic comparison of texts

Table 1 contains the requirement specification for a software system aimed to perform a semantic comparison of texts.

Table 1. Requirement specification of a software system for semantic texts comparison

RQ code	RQ textual description
F1	Design a problem domain tree Such a tree contains concepts (other words entities, represented by nouns or combination of nouns and adjectives (noun chunk example "operating system")) and RDF triples to store relationships between entities.
F1.1	Download problem domain tree from ontology storage (In order to do this the Growing pyramidal networks are transformed into an ontology of problem domain)
F1.2	Modify problem domain tree by means of adding or changing new brunch, facts, and entities.

RQ code	RQ textual description
F1.2.1	Remove keywords and corresponding processes
F1.2.2	Add keywords and corresponding processes
F 1.3	Save tree to ontology repository
F1.4	Represent problem domain tree graphically
F2	Select text and define whether its topic of matches with topics (name of the processes) of the problem domain tree.
F2.1	Design a tree from the text
F2.1.1	All entities related to the topic should be found.
F2.1.2	All facts related to the topic should be found.
F 2.2	Show text tree
F3	Compare problem domain tree and text tree
F3.1	Propose a criterion how to define whether trees are equal based on equality of the branches Think how brunch should look like to define preliminary equality Propose criterions precision equality of sub-brunches
F3.3	Represent result of comparison by the following way Text matches to "root process" on xx% Text matches to "process1" on xx% …….. Text matches to "process_n" on xx%

RQ code	RQ textual description
F6	Ability to see the most popular topic at all or specific groups of people.
NF1	web-application
NF2	Satisfy to WCAG requirements
NF2.1	Avoid malicious code attacks from client web page to server
NF2.2	Avoid fishing sites and SQL injections
NF2.3	Provide correct user input and avoid using UI components and frameworks with vulnerabilities
NF2.4	Set up the updated policy (to check for critical updates of the libraries and modules)
NF2.5	Forbid to the attacker can run malicious code through the server's API

The requirement specification for the software module "processing of ontological output by means of fuzzy sets as well as multiset" is represented in Table 2.

Table 2. Requirement specification of a software module of taking decision about different texts

RQ code	RQ textual description
F0	Implementation of common operations on multisets and fuzzy sets Functional representation of multisets and fuzzy sets Coefficient and category. For fuzzy sets the coefficient {0, 1}

RQ code	RQ textual description
	For multisets coefficient {0, + max int } value coefficient only positive The category must be a unique value
F1	Adding two types of sets
F1.1	The operation of combining sets is performed
F1.2	If the sets have the same categories then their coefficients are added
F2	Multiplying sets by a number
F 2.1	All coefficients of categories are multiplied by a certain number
F3	Subtraction of sets
F3.1	All types of categories in different sets are considered in pairs. For common categories, their coefficients are subtracted. If the coefficient is less than zero, then the category is removed from the set
F4	Determining the power of sets as the sum of all their coefficients
F5	Determination of the maximum coefficient for all categories in the set
F6	Checking the coefficients when performing arithmetic operations and set initialization operations and within the range.
F7	Division of sets by coefficient When dividing multisets, keep in mind that the division operation is possible if all the coefficients result in integers. If the result is

RQ code	RQ textual description
	not an integer number coefficient is rounding

F 8	Convert a multiset to a list of sets (fuzzy set)
	`public sets(object[] keys, T[] vals)`
	`{`
	`List<object> tempobj = new List<object>();`
	`List<T> tempval = new List<T>();`
	`if (!defineThesameLen(keys.Length, vals.Length))`
	`return;`
	`for (int i = 0; i < keys.Length; i++)`
	`{`
	`tempobj.Add(keys[i]);`
	`tempval.Add(vals[i]);`
	`}`
	`ConvertListsToSet(tempobj, tempval);`
	`}`
	`public sets(List<object> keys, int val)`
	`{`
	`List<T> tempL = new List<T>();`
	`T x = (T)Convert.ChangeType(val, typeof(T));`
	`for (int i = 0; i < keys.Count; i++) {`
	`tempL.Add(x);`
	`}`
	`ConvertListsToSet(keys, tempL);`
	`}`
	`void ConvertListsToSet(List<object> keys, List<T> vals)`
	`{`
	`// before form a set check the next conditions:`
	`// 1. number of keys and values should be the same`
	`if (!defineThesameLen(keys.Count, vals.Count))`
	`return;`
	`// 2 all values should be the specific type and inside of the allowed range`
	`cht = new checkType();`

RQ code	RQ textual description

	if (!defineTypeForVals(vals))
	return;
	a = new Dictionary<object, T>();
	
	for (int i = 0; i < keys.Count; i++)
	{
	try
	{
	a.Add(keys[i], vals[i]);
	}
	catch (Exception ex)
	{
	sayError(ex.Message);
	}
	}
	}

NF 1	Sets are stored in the Dictionary data structure

Implementation of the multiset and fuzzy set arithmetical processing

The first operation is to check the type of coefficient before performing arithmetical operations. It is realized by means of polymorphism.

```
class checkType
    {
```

Virtual method in base class id designed to verify different types of objects.

```
public virtual void CheckRange(object a, string error, string
ErrorEnd) {
     }}
    class checkInt : checkType
    {
```

```
        public override void CheckRange(object a, string error,
string ErrorEnd)
        {
            int temp = (int)a;
            if (temp <= 0 || temp > int.MaxValue)
            {
                throw new Exception(error + temp + ErrorEnd);
            }
        }
    }
    class checkdouble : checkType
    {
        public override void CheckRange(object a, string error,
string ErrorEnd)
        {
            double temp = (double)a;
            if (temp <= 0 || temp > 1)
            {
                throw new Exception(error + temp + ErrorEnd);
            }
            temp = Math.Round(temp, 5);
        }

    }
```

Example of code to perform additional operations with different types of coefficients.

Static T Add<T>(T a, T b) the realization is based on expression trees. Expression tree is compiled on predicate Func<T,T,T>.

```
   public static T Add<T>(T a, T b)
        {
            GetParam<T>();
            // add the parameters together
```

```
            BinaryExpression body = Expression.Add(paramA, paramB);
            // compile it
Func<T, T, T> add = Expression.Lambda<Func<T, T, T>>(body, paramA,
paramB).Compile();
            // call it
            return add(a, b);

        }

  public static T Sub<T>(T a, T b)
        {
            GetParam<T>();
            // add the parameters together
            BinaryExpression body = Expression.Subtract(paramA, paramB);
            // compile it
            Func<T, T, T> sub = Expression.Lambda<Func<T, T, T>>(body,
paramA, paramB).Compile();
            // call it
            return sub(a, b);
        }

        public static T Mul<T>(T a, T b)
        {
            GetParam<T>();
            // add the parameters together
            BinaryExpression body = Expression.Multiply(paramA, paramB);

            // compile it
            Func<T, T, T> mul = Expression.Lambda<Func<T, T, T>>(body,
paramA, paramB).Compile();
            // call it
            return mul(a, b);
        }

        public static T Div<T>(T a, T b)
        {
            GetParam<T>();
            // add the parameters together
            BinaryExpression body = Expression.Divide(paramA, paramB);

            // compile it
            Func<T, T, T> div = Expression.Lambda<Func<T, T, T>>(body,
paramA, paramB).Compile();
            // call it
            return div(a, b);

        }
```

In order to archive the flexible and extensible software architecture the design pattern Visitor is considered (Antoniol et. al., 2001).

Table 3 explaining the correspondence of some visitor parts and fragments of the class diagram (Gamma et al, 1993).

Table 3. Adoption of Visitor design pattern to the task of text semantic comparison.

Visitor design pattern constituent	**Part of the class diagram (figure 3)**
The **Visitor** interface declares a set of visiting methods that can take concrete elements of an object structure as arguments. These methods may have the same names if the program is written in a language that supports overloading, but the type of their parameters must be different.	Interface compare defines methods for different compare approaches One of them is based on sets, another one based on ontologies comparison
Each **Concrete Visitor** implements several versions of the same behaviors, tailored for different concrete element classes.	Concrete visitors – are classes operators and packages, implementing methods for semantic comparison based on ontology comparison. Example of concretevisitors realization `public static sets<T> operator +(sets<T> one, sets<T> two) { // to define data type int or double to check sets<T> ins = new sets<T>(); List<T> vals = new List<T>(one.a.Values); string errorend = ins.DefineT(vals[0]); foreach (KeyValuePair<object, T> kvp in two.a) { // when sets are intersects if`

Visitor design pattern constituent	Part of the class diagram (figure 3)

	(one.a.ContainsKey(kvp.Key))
	{
	// add koefficients on categories
	try
	{
	var rez = oper<T>.Add(kvp.Value, one.a[kvp.Key]);
	ins.cht.CheckRange(rez, ins.errors[1], errorend);
	ins.a.Add(kvp.Key, rez);
	}
	catch (Exception ex)
	{
	ins.sayError(ex.Message);
	}
	}
	else
	{
	ins.a.Add(kvp.Key, kvp.Value);
	}
	}
	// add those values that are unique for a set
	foreach (KeyValuePair<object, T> kvp in one.a)
	{
	if (!ins.a.ContainsKey(kvp.Key))
	{
	ins.a.Add(kvp.Key, kvp.Value);
	}
	}
	return ins;
	}

The **Element** interface declares a method for "accepting" visitors. This method should have one parameter declared with the type of the visitor interface.	Elements that accept new operations are Add, Sub, Mul, and Div

Visitor design pattern constituent	Part of the class diagram (figure 3)
Each **Concrete Element** must implement the acceptance method. The purpose of this method is to redirect the call to the proper visitor's method corresponding to the current element class. Be aware that even if a base element class implements this method, all subclasses must still override this method in their own classes and call the appropriate method on the visitor object.	Class expression plays the role of the concrete element
The **Client** usually represents a collection or some other complex object (for example, a Composite tree). Usually, clients aren't aware of all the concrete element classes because they work with objects from that collection via some abstract interface.	Client is a Program class that calls these methods.
An explanation was taken from https://refactoring.guru/design-patterns/visitor	

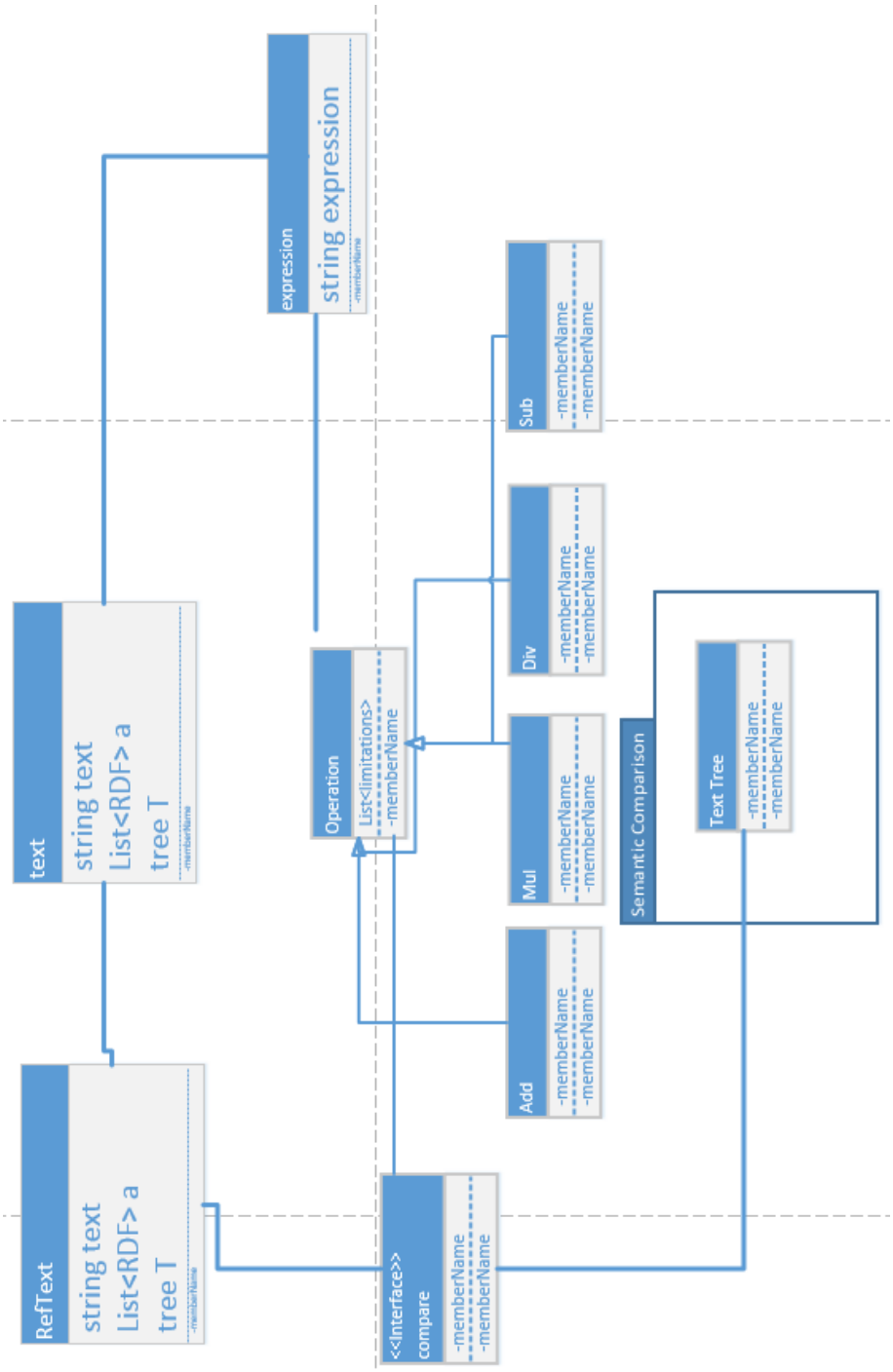


Figure 3. Architectural solution for semantic text comparison

Conclusion

Paper proposes an architectural solution of the software system for semantic comparison of texts that is based on visitor design pattern. Implementation of Visitor design patterns allows to archive the next advantages in semantic texts comparison software systems:

1. High recognition accuracy with given level of details supporting (level of words, sentences, fragments of texts);
2. Usability (flexible architectures and given speed of working) and high security level;
3. Possibility to visualize intermedia results. It will allow to visualize step of comparison algorithms for better testing and teaching of peculiarities of system functionality;
4. Estimation of different comparison algorithms effectiveness multiple solving algorithms. The system should have multiple algorithms to solve the problem because different algorithms can produce different results.

Bibliography

[Antoniol et. al., 2001] Antoniol, G., Casazza, G., Di Penta, M., & Fiutem, R. (2001). Object-oriented design patterns recovery. Journal of Systems and Software, 59(2), 181-196.

[Furlan et. al., 2013] Furlan, B., Batanović, V., & Nikolić, B. (2013). Semantic similarity of short texts in languages with a deficient natural language processing support. Decision Support Systems, 55(3), 710-719.

[Gamma et al, 1993] Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1993, July). Design patterns: Abstraction and reuse of object-oriented design. In European Conference on Object-Oriented Programming (pp. 406-431). Springer, Berlin, Heidelberg.

[Harispe et al., 2013] Harispe, S., Ranwez, S., Janaqi, S., & Montmain, J. (2013). The semantic measures library and toolkit: fast computation of semantic similarity and relatedness using biomedical ontologies. Bioinformatics, 30(5), 740-742.

[Rus et al., 2013] Rus, V., Lintean, M., Banjade, R., Niraula, N. B., & Stefanescu, D. (2013, August). Semilar: The semantic similarity toolkit. In Proceedings of the 51st annual meeting of the association for computational linguistics: system demonstrations (pp. 163-168).

[Smith et al., 2007] Smith, B., Ashburner, M., Rosse, C., Bard, J., Bug, W., Ceusters, W., ... & Leontis, N. (2007). The OBO Foundry: coordinated evolution of ontologies to support biomedical data integration. Nature biotechnology, 25(11), 1251-1255.

[Whetzel et al., 2011] Whetzel, P. L., Noy, N. F., Shah, N. H., Alexander, P. R., Nyulas, C., Tudorache, T., & Musen, M. A. (2011). BioPortal: enhanced functionality via new Web services from the National Center for Biomedical Ontology to access and use ontologies in software applications. Nucleic acids research, 39(suppl_2), W541-W545.

Authors' Information

Vitalii Velychko *– V. M. Glushkov institute of Cybernetics NAS of Ukraine; Candidate of Technical Sciences. Senior Researcher. Glushkova avenue 40, Kiev, Ukraine, 03187; e-mail: aduisukr@gmail.com*

The main directions of scientific research: computer ontologies and their use in the educational process; information-oriented information systems with processing of natural language objects, ontological approach.