**“ALEXANDRU IOAN CUZA” UNIVERSITY IAȘI**

**FACULTY OF COMPUTER SCIENCE**

C:\Users\Claudiu\Desktop\logo_fii3.png

DISSERTATION PAPER

**Semantic Analysis of Source Code in**

**Object Oriented Programming.**

**A Case Study for *C#***

Scientific Coordinator Candidate

***Dr. Adrian Iftene Claudiu Epure***

**Session:** July, 2015

**“ALEXANDRU IOAN CUZA” UNIVERSITY IAȘI**

**FACULTY OF COMPUTER SCIENCE**

**Semantic Analysis of Source Code in**

**Object Oriented Programming.**

**A Case Study for *C#***

***Claudiu Epure***

**Session:** July, 2015

Scientific Coordinator

***Dr. Adrian Iftene***

**Declarație privind originalitatea și respectarea drepturilor de autor**

Prin prezenta declar că Lucrarea de disertație cu titlul *Semantic Analysis of Source Code in Object Oriented Programming. A Case Study for* C# este scrisă de mine și nu a mai fost prezentată niciodată la altă facultate sau instituție de învățământ superior din țară sau străinătate. De asemenea, declar că toate sursele utilizate, inclusiv cele preluate de pe Internet, sau indicate în lucrare, cu respectarea regulilor de evitare a plagiatului:

* toate fragmentele de text reproduse exact, chiar și în traducere proprie din altă limbă,

sunt scrise între ghilimele și dețin referința precisă a sursei;

* reformularea în cuvinte proprii a textelor scrise de către alți autori deține referința

precisă;

* codul sursă, imagini etc. preluate din proiecte open-source sau alte surse sunt utilizate

cu respectarea drepturilor de autor și dețin referințe precise;

* rezumarea ideilor altor autori precizează referința precisă la textul original.

Iași, *29 iunie 2015* Absolvent: Claudiu Epure

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Declarație de consimțământ**

Prin prezenta declar că Lucrarea de disertație cu titlul *Semantic Analysis of Source Code in Object Oriented Programming. A Case Study for* C#, codul sursă al programelor și celelalte conținuturi (grafice, multimedia, date de test etc.) care însoțesc această lucrare să fie utilizate în cadrul Facultății de Informatică. De asemenea, sunt de acord ca Facultatea de Informatică de la Universitatea „Alexandru Ioan Cuza” Iași să utilizeze, modifice, reproducă și să distribuie în scopuri necomerciale programele-calculator, format executabil și sursă, realizate de mine în cadrul prezentei lucrări de disertație.

Iași, *29 iunie 2015* Absolvent: Claudiu Epure

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Contents

[1 Introduction 5](#_Toc422376828)

[1.1 Motivation 5](#_Toc422376829)

[1.2 Proposed Solution 5](#_Toc422376830)

[1.3 Keywords 6](#_Toc422376831)

[2 Concepts 7](#_Toc422376832)

[2.1 Introduction 7](#_Toc422376833)

[2.2 Definitions 7](#_Toc422376834)

[3 State of the Art 8](#_Toc422376835)

[3.1 Existing Systems for Extracting Structured Data from Source Files 8](#_Toc422376836)

[3.2 Existing Systems for Information Retrieval based on Questions 9](#_Toc422376837)

[4 C# Source Code Representation Ontology 12](#_Toc422376838)

[4.1 RDF and Graph-Oriented Databases 12](#_Toc422376839)

[4.2 Semantic Web Principles 13](#_Toc422376840)

[4.3 Building the C# Source Code Representation Ontology 15](#_Toc422376841)

[5 System Architecture 15](#_Toc422376842)

[Bibliography 18](#_Toc422376843)

# Introduction

## Motivation

Computer programs have become the most frequently used tools in our modern society. Nowadays, they are present at large scale in industry in form of applications, platforms and services, covering multiple areas such as science and education, finance, commerce, etc.

Developing a software system is not an easy action. Instead, it is a complex process comprised of several phases, which are completed during a significant period of time. However, factors like customers high demand and market competition lead to acceleration of the process with negative impact on quality.

As complex software systems are built at a fast pace, they need to remain maintainable through time. For this reason, software quality must be at its highest level, yet in most cases, it decreases as the systems are getting bigger.

Testing the code is the way for assuring the required functionality from the perspective of the users. From the programmers point of view, the code needs to be clean and easy to extend or reuse. Design patterns, coding standards, static code analysis are software engineering methodologies serving such a purpose. But still, there are old systems, hard to refactor and production source code that is not implementing any engineering technique, which is very easy to break at any small try to redesign.

Another key aspect of software development is the use of version control systems in order to keep track of changes and make possible for teams to collaborate. They also provide a general view on the projects and backup service as well. Although they help to keep track of the phisical file changes over time, they do not provide a way of tracking the logical structure inside a project.

## Proposed Solution

None of the above mentioned techniques address the problem of retrieving meta information from the code, in a semantic manner. Large software projects involving thousands of source code files would be easier to understand, control and extend if they would be complemented by a solid information retrieval system.

The idea behind the proposed solution is to incrementally build knowledge datasets from plain source code.

The first step to achieve this is to create a graph oriented knowledge base from the entities present in code. This can be built on top of specific ontologies, using a convenient format like the *Resource Description Framework* (*RDF*)[[1]](#footnote-1). Having the knowledge base in place, it would be easy to query the system (e.g. *SPARQL*)[[2]](#footnote-2) about its interacting components and services. Going further, an answering mechanism could be applied for enabling natural language questions on the knowledge base.

This idea can be applied on every programming language, no matter the paradigm. However, proposed solution is focusing on the *Object Oriented Programming* paradigm and the programming language of choise in this case is *C#*. There main reason for choosing *C#* for this project is because the new *.NET Compiler Platform (“Roslyn”)[[3]](#footnote-3)* is just fit for this purpose, having a clean and easy to use *API*.

## Keywords

C#, static analysis, ontology, RDF, triple store, linked data, natural language, SPARQL

# Concepts

## Introduction

## Definitions

# State of the Art

Some of the existing approaches that are based on similar ideas are mentioned below. They address singular or specific problems, so for the proposed system, the intent is to adapt, extend and combine some of the ideas, in order to achieve the goal.

## Existing Systems for Extracting Structured Data from Source Files

**RDF Coder[[4]](#footnote-4)**

*RDF Coder* is a tool able to generate *RDF* models of code libraries, entirely written in *Java*. *RDF Coder* can be used to perform multi-level code inspection, create code dependency graphs and generate custom documentation. Currently is supported the *Java* language only at version *1.5*. This library can be used by Java programmers as a tool to deal with huge classpaths, to find relationships across classes and objects. *RDF Coder* can be also used to develop more complex analysis tools leveraging on the flexibility of the *RDF* related technologies.

**Fuzzy Ontology Framework[[5]](#footnote-5)**

*Fuzzy Ontology Framework* is a library that helps to integrate a fuzzy ontology with object-oriented programming (OOP) classes written in .NET. It is a hybrid integration, i.e. some OWL concepts can be mapped directly to OOP classes, yet most OWL concepts are derived just from OOP instance properties, with no direct mapping to a .NET class. Hence the OOP instance-OWL concept(s) mapping can evolve dynamically in the course of time.

The implementation currently supports FuzzyOWL2 ontologies together with FuzzyDL reasoner to infer affiliation of OOP instances to particular OWL concepts. It can be however easily modified to support any fuzzy ontology notation as well as any fuzzy reasoner.

**SCRO[[6]](#footnote-6)**

*SCRO* is an ontology created to support major software understanding tasks by explicitly representing the conceptual knowledge structure found in source code.

*SCRO* captures major concepts of object-oriented programs and helps understand the relations and dependencies among source code artifacts. Supported features include, encapsulation, inheritance (subclassing and subtyping), method overloading, method overriding, and method signature information. It is designed for Java.

**Source Code Plagiarism Detection Method[[7]](#footnote-7)**

*Ion Smeureanu, Bogdan Iancu -* *Source Code Plagiarism Detection Method Using Protégé Built Ontologies.* In this paperthe authors demonstrate how source code plagiarism could be detected with the help of an ontology which models software programs.

## Existing Systems for Information Retrieval based on Questions

**Treo[[8]](#footnote-8)**

The main ideas behind this system are:

- Entity Recognition and Pivot Determination through Entity Search

- Query Syntactic Analysis: Partial Ordered Dependency Structure (*PODS*) Determination

- Spreading Activation using Semantic Relatedness

The algorithm first determines the *key entities* present in the natural language query. They can be potentially mapped to instances or classes in the *Linked Data Web*. The entity search engine receives the *key entities* and resolves *pivot entities* (URIs) in the Linked Data Web. The query is then analyzed and parsed to obtain *partial ordered dependency structure (PODS).* This is the reduced representation of the query targeted towards maximizing the matching probability between the structure of the terms present in the query and the subject, predicate, object structure of *RDF*.

The *spreading activation search* takes the URIs of the pivots and the PODS structure, and thus, starting from the pivot node, the algorithm navigates through neighboring nodes in the Linked Data Web computing the *semantic relatedness* between query terms and vocabulary terms in the node exploration process. The navigation process builds the answer of the query. The algorithm returns a set of ranked triple paths determined by the navigation from the pivot entity to the final resource representing the answer, ranked by the average of the relatedness scores over each triple path. Answers are displayed to users using a list of triple paths merged in a graph after a simple post-processing phase.

**TBSL[[9]](#footnote-9)**

The ideas of this system are:

- SPARQL Template from question

- mapping between NL expressions to the domain vocabulary

The *TBSL* approach to question answering over *RDF* data relies on a parse of the question to produce a *SPARQL* template that directly mirrors the internal structure of the question and that, in a second step, is instantiated by mapping the occurring natural language expressions to the domain vocabulary.

The user's input is a natural language question which is processed by a POS tagger. The result is the semantic representation of the natural language query, based on lexical entries that are created using a set of heuristics. In the next step, this is converted into a SPARQL query template which contains slots: missing elements of the query that have to be filled with URIs. The URIs are determined using sophisticated entity identification approaches, based on string similarity as well as on natural language patterns which are compiled from existing structured data in the Linked Data cloud and text documents.

**Squall2sparql[[10]](#footnote-10)**

*SQUALL* (Semantic Query and Update High-Level Language) is a *Controlled Natural Language (CNL)* for English that has full compliance with *Linked Open Data* (*LOD*), and covers nearly all features of *SPARQL* *1.1*, for both queries and updates. The advantage of *CNLs* is to provide a natural language syntax while retaining the precision and lack of ambiguity of formal languages like *SPARQL*.

The main drawback of *CNLs* is that users have to learn the language and its disambiguation rules. The system *squall2sparql2* is a Web application that supports the translation from *SQUALL* to *SPARQL*, as well as the direct querying of *SPARQL* endpoints, like *DBpedia*. Given a *SQUALL* sentence, the system first translates it into an intermediate logical representation using a *Montague* grammar. The intermediate representation is then translated into *SPARQL*, simply mapping logical constructs to combinations of *SPARQL* constructs. The produced query can then be sent to any *SPARQL* endpoint, and results returned.

**GFMed[[11]](#footnote-11)**

The main ideas of this system are:

- use of grammars created in *Gramatical Framework*

- use of medical datasets such as *SIDER*, *Diseasome* and *DrugBank*

*GFMed* system is focused on medical area questions and knowledge base.It consists mainly of a *Gramatical Framework* grammar for the application domain given by *SIDER*, *Diseasome* and *DrugBank* datasets. There is a minor preprocessing of questions and postprocessing of translation results, mainly in order to deal with structures involving numeric values, e.g. values for water solubility, or free text, like different names of foods. The system uses an abstract grammar and two concrete grammars. For each concrete grammar, lexicons derived from *SIDER*, *Diseasome* and *DrugBank* were generated. Both English and SPARQL grammars are based on the datatsets terminology.

# C# Source Code Representation Ontology

For many years, the traditional way of storing data was using *relational databases*. In a relational database the data is stored in tables (records) following specific fixed schemas. The entities in a table are restricted to follow this schema in order to provide consistency.

A different approach was the introduction of the *document-oriented databases* using an hierarchical model (e.g. *XML*, *JSON* files). This type of storage doesn’t use the concept of *schema*, so the records of the same type can have different properties and sizes. The drawback of a *document-oriented database* is that it uses a *hirarchy* *of elements* (nodes) in form of a *tree*, so there are elements that have a bigger importance/priority against others (e.g. parent node vs child node).

Although there are advantages and disadvantages from using each of the above database model, a new type of database is preferred when the absence of conceps like *schema* and *elements hirarchy* is required: *graph-oriented database*.

## RDF and Graph-Oriented Databases

The *Resource Description Framework* (*RDF*) is one important building block of the *graph-oriented database*. It is a framework for modelling resources and relationships between them in form of *subject-predicate-object* statements (triples). A resource can have an infinite number of properties and there is no restriction that it should follow.The underlying mathematical model is a *labeled directed multi-graph* in which the nodes are the resources and the edges are the relations. As a result, all the nodes are equal in importance/priority.

In *RDF*, a node can be an entity (such as a resurce identified by an *id*) or a data values of some type (such as a number or a string). An edge can be an object property or a data property. An object property links an entity (a resurce) to another. A data property links an entity to a plain value of some type.

For example, the following *RDF* graph express the relationships between a person identified “John\_Doe” and some information about it: type (an object property) and age (a data property) (*Figure 1*). By convention, the resource nodes are represented in ovals, the values are represented in rectangles and the properties are represented as arrows.

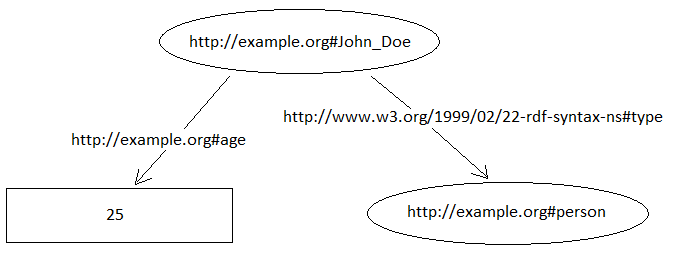


Figure : A visual representation of a RDF graph

The associated *RDF* code is:

<rdf:RDF

xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"

xmlns:ex="http://example/org#">

<rdf:Description rdf:about="http://example.org#John\_Doe">

<rdf:type rdf:resource="http://example.org#person"/>

<ex:age>25</ex:age>

</rdf:Description>

</rdf:RDF>

## Semantic Web[[12]](#footnote-12) Principles

Recently, the *Word Wide Web Consortium (W3C)[[13]](#footnote-13)* has come with the idea of *Semantic Web*. The *Semantic Web* term is used to refer to a stack of technologies that allow people to create machine-readable data stores in a common exchangeable format on the Web. The result is a global database, the new *Web of linked data* - an addition to the classic *Web of documents*. *RDF* is the data model used at the base of *Semantic Web*. As a characteristic, each resource being uniquely identified by an *URI* or and *IRI*.

The data interchange is done through protocols such as *HTTP*. For this reason, every data store created using the above characteristics needs to expose a query endpoint visible on th Web.

For querying, *RDF* has its own query language: *SPARQL* (*SPARQL Protocol and RDF Query Languag*). Because *RDF* can be seen as a collection of realtionships between resources, *SPARQL* queries are based on triple patterns, providing one or more patterns against such relationships, using variables in place of some resources. The result of the processed query is the set of resources for all triples that match these patterns.

For example, the query “Select all the people beeing 25” can be transformed in the following *SPARQL* query:

PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

PREFIX ex: <http://example.org#>

select ?s

where {

?s rdf:type ex:person .

?s ex:age "25"

}

Having in place the righ mechanism of storing data and a powerful query language for retrieving it, the next step is to provide a way to enable semantics behind the data. For this purpose, *RDF* alone is not enough. *RDFS[[14]](#footnote-14)* and *OWL[[15]](#footnote-15)* are *RDF* extensions that offer means to record semantics and meaning about the data.

The natural way to fill the semantic gap is to use *metadata[[16]](#footnote-16)* to describe the actual data. Metadata is in fact “data about data”. The data decorated with useful information about it has meaning in a specific context. For example, in the context of object-oriented source code, a method of a class can be described by properties such as *author, parameter, return type, comment,* etc.

Regarding the interchangeability principle of *linked data*, describing resources from the same domain of activity involves the existence of a common set of metadata properties. Such a collection of terms of well defined meaning is known as a *vocabulary*. If the terms are organized into a hierarchical structure, then the collection is a *taxonomy*. A certain *vocabulary* needs to be expressed in a representation language. An *ontology* is a *vocabulary* created in a representation language based on a grammar that contains formal constraints on how the terms can be combined in order to express something meaningful in a specific domain of interest.

Describing resources using the *vocabulary* and the rules from an *ontology* leads to creating a *knowledge-base*. A *knowledge-base* is a big collection of *RDF* graphs that can store data from multiple domains and it can be queried over the network via an endpoint. *Knowledge-bases* are useful for information consumers, both human or artificial, providing a common way of information retrieval.

## RDF Schema and Web Ontology Language

There are two principal syntaxes for annotating *RDF* data with semantic metadata: *RDF Schema (RDFS)* and *Web Ontology Language (OWL)*. Both of them are *W3C* specifications.

*RDF Schema* is a semantic extension of *RDF* which provides a vocabulary for data modelling of information written in *RDF*. Unlike other kind of type system, it is designed to be property centric. This means the focus is on properties, which exists on their own. They do not need to be defined as members of a class. Instead, they have a *domain* and a *range* which they apply. One property is defined once and used with multiple classes (of some domain). This way, it is easy to extend the behavior of a class without modifying it.

Some examples of frequently used *RDFS* classes and properties are shown in  *Table 1*:

|  |  |
| --- | --- |
| **Class** | **Property** |
| rdfs:Resource | rdfs:domain |
| rdfs:Class | rdfs:range |
| rdfs:Literal | rdfs:subClassOf |
| rdfs:DataType | rdfs:subPropertyOf |
| rdf:Property | rdfs:label |
| rdf:List | rdfs:comment |

Table : examples of RDFS classes and properties

*Web Ontology Language* is an ontology language for the *Semantic Web*. It provides the syntax for defining an ontology, which is a formal description of a domain of interest. The syntax is divided into three categories: *entities*, *expressions* and *axioms*.

*Entities* are the basic terms of the ontology, uniquely identified by *IRIs* and can be splitted into *classes*, *properties* and *individuals*.

*Expressions* are used to represent more complex notions. They can be used to define restrictions to classes of individuals.

*Axioms* are true statements about the domain of the ontology.

Besides the three syntactic categories of *OWL*, there is another category that is does not influence the logical structure of the ontology when used: *Annotations*. *Annotations* areused to describe the resources, for example giving them labels. These are useful for tools that help visualizing the ontology, by displaying the resources labels.

Another aspect of *OWL* is the fact that it provides a way to reason about the described data. A reasoner will detect the axioms that are not visible at first sight. For example, if *float* is an individual of class *Struct* and *Struct* is a subclass of *DataType*, then *float* is an individual of class *DataType* too.

A summarized view of *OWL* is available in *Table 2*:

|  |  |  |  |
| --- | --- | --- | --- |
| Entities | | | |
| Classes | Properties | | Individuals |
| Object Properties | Data Properties | Datatypes |
| Expressions | | | |
| Axioms | | | |
| Annotations | | | |

Table : general view of OWL

## Building the C# Source Code Representation Ontology (CSCRO)

For creating an ontology that covers C# programming language, the *SCRO* ontology, described in 3.1, was used as a model. For this reason, the new ontology is named *CSCRO* (*C# SCRO*). As a result, *CSCRO* can be considered a variation of *SCRO* that works for *C#*.

At first, the intention was to use *SCRO* concepts to annotate the resources extracted from *C#* source code. But the ontology uses entities like *package*, *java.lang.String*, *final method,* *Java* *based access modifiers*,so it was clear that it was designed to work with *Java* programming language. Therefore, a new ontology for *C#* was needed.

The new *C#* ontology, *CSCRO,* keeps the main ideas from *SCRO* ontology, but adds new concepts specific to the programming language itself, based on *MSDN[[17]](#footnote-17)* references.

There are some significant differences between the two ontologies both in structure (taxonomy) and individuals. These differeces are discussed in parallel in the following lines.

### Different Access Modifiers

In *CSCRO* ontology, *AccessModifier* is a subclass of *Modifier* (instead of *AccessControl*). Also, C*#* has a slightly different set of access modifiers. It uses *internal* and *protected internal* as *C#* specific access modifiers *(Figure 2)*.

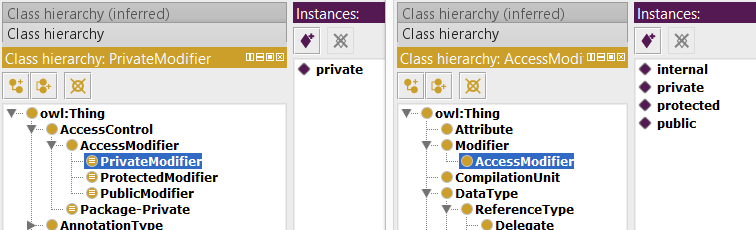


Figure : SCRO vs CSCRO (access modifiers)

### Different Modifiers Taxonomy

In *SCRO* ontology, the modifiers are not represented by class of their own. Instead, they are spreaded under the taxonomy of data types. For example, there is no *static* modifier itself, but there are *StaticMemberClass* and *StaticMemberInterface* classes. The same is valid for other modifiers too (e.g. *final – FinalLocalClass, FinalMemberClass*, *abstract- AbstractLocalClass, AbstractMemberClass*, etc.)

In *CSCRO*, there is the *Modifier* class under which there is an individual for each modifier (*abstract, static, override*, etc.). Besides, there are some *C#* specific modifiers: *async, sealed, const, new, partial,* etc*.* In the same manner, *private*, *protected*, *public* and *internal* are individuals of *AccessModifier* class.

The differences are marked with red in *Figure 3*.

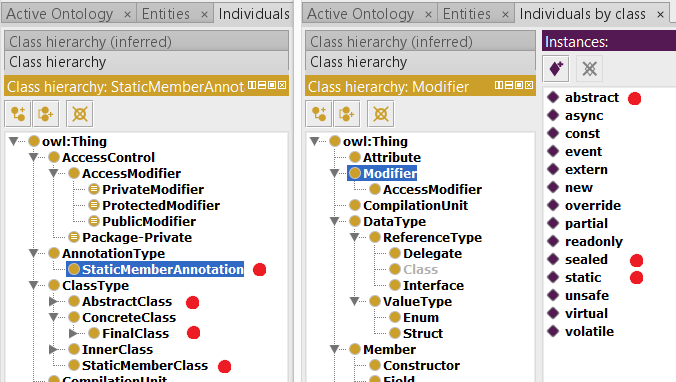


Figure : SCRO vs CSCRO (modifiers taxonomy)

### Different Data Types Taxonomy

*SCRO* has a complicate taxonomy for data types. There is no single parent node in the hieararchy from which every type is derived. Instead, there is a parent class for each type (e.g. *ClassType, EnumType, InterfaceType,* etc.). *DataType* class is used in another context than types hierarchy context (i.e. for specifying the “primitive” types: *int, float,double,* etc.).

*CSCRO* rewrites completly the taxonomy of data types. Here, there is one parent class *DataType* from which every subclass defines a new hieararchy level of types. Hence, on the second level there are *ReferenceType* and *ValueType* and on the third level there are *Delegate, Class and Interface* – as subclasses of *ReferenceType* and *Enum* and *Struct* – as subclasses of *ValueType*. Eventually, the “primitive” types are in fact individuals of class *Struct*.

The differences are explained in *Figure 4* by assigning numbers to similar concepts.

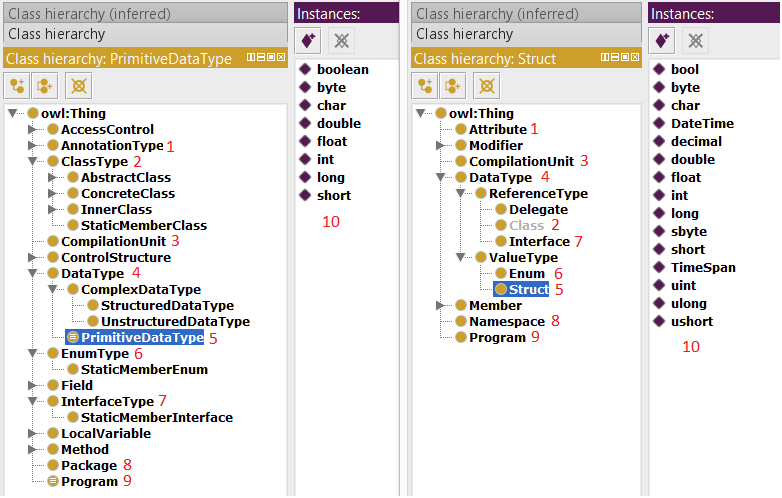


Figure : SCRO vs CSCRO (data types taxonomy)

### Different Object Properties and Data Properties

### There are also difference between the properties of the two ontologies, both object properties and data properties.

In the case of object properties (*Figure 5*), the differences appear mainly because of the changes in the class taxonomy. Moreover, there are differences because of the terminology used in *C#* versus *Java* (e.g. package vs namespace: *hasPackageMember* vs *hasNamespace*; import vs using: *imports* vs *uses*, etc.).

In the case of data properties (*Figure 6*), *CSCRO* contains fewer properties that *SCRO* (e.g. *hasName*, *hasIdentifier*, *hasComment*).

In *CSCRO*, the properties are only for basic functionality of the system. Complex properties can be added in case of complex requirements.

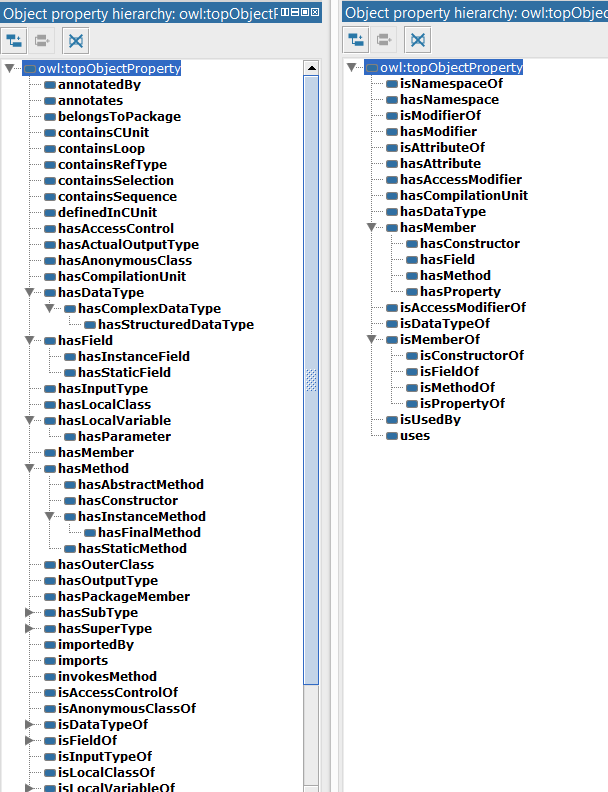


Figure 5: SCRO vs CSCRO (object properties)

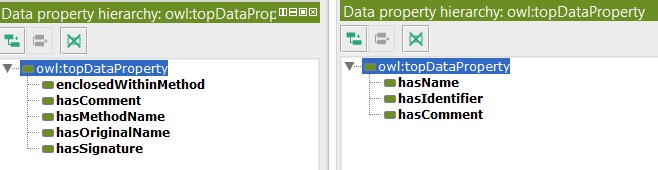


Figure 6: SCRO vs CSCRO (data properties)

# System Architecture

The proposed system *(Figure 2)* is designed to follow two main ideas.

First, there is the process of extracting structured data from C# source files, based on an existing specific ontology (e.g. C# object-oriented ontology) and store that data in a way such that it can be easily retrieved later. In this case, the data will be saved in a triple store.

Second, there is the process of retrieving data from the store, having hierarchical levels of querying. Whilst SPARQL queries are used for this purpose at the lowest level, at the highest level, the goal is the use of natural language questions. In the middle, annotated questions are used, based on NLP techniques.

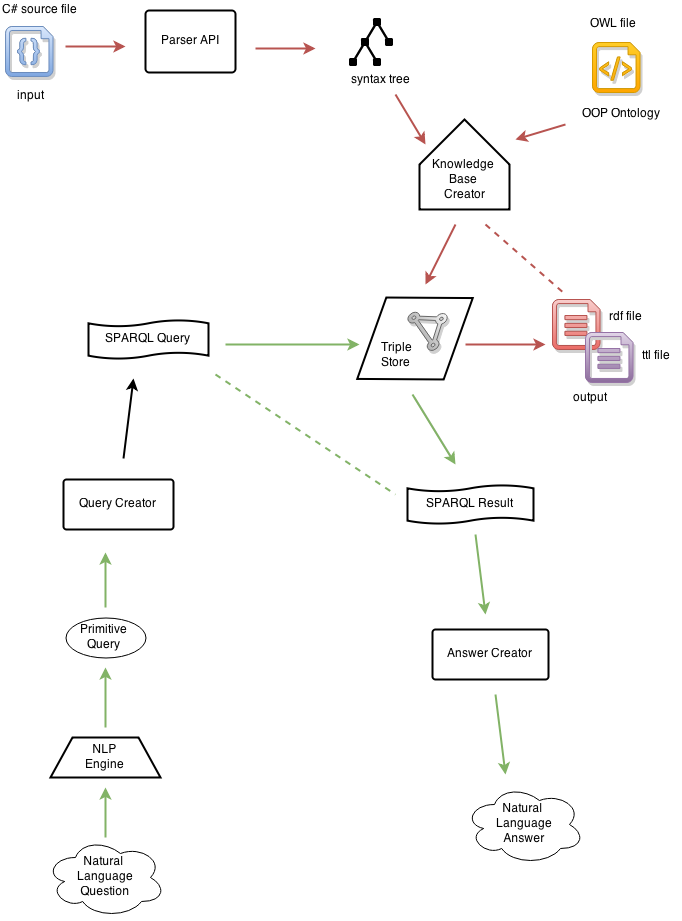


Figure : System Architecture

## The Knowledge-Base Builder Module (5 p)

This module is designed to build a *knowledge-base* of the components found in a *C#* project or in *C#* source files. The result is represented as a collection of *RDF* graphs, known as a *triple store*. The graphs are built by following the *CSCRO* ontology defined in 4.3.

The classes, interfaces and helpers.

[UML Diagram]

## The Information Retrieval Module (3 p)

Asdassadgagafsfdfd

// Monday

## Details about the Project (1 p)

Solution description

Github Repository

Nuget package

Pictures (3 p)

New chapter Conclusions (2 p)

* Personal Contributions
* Future work

# Bibliography

*WordNet*. (2012, Decembrie). Preluat pe Iunie 2013, de pe wordnet: http://wordnet.princeton.edu/

*/wiki/Levenshtein\_distance*. (2013, Mai). Preluat pe Iunie 2013, de pe rosettacode.org: http://rosettacode.org/wiki/Levenshtein\_distance

Atnafu, S., Chbeir, R., & Brunie, L. (2002, Februarie). *Efficient Content-Based and Metadata Retrieval in Image Database.* Preluat de pe jucs.org: http://www.jucs.org/jucs\_8\_6/efficient\_content\_based\_and/Atnafu\_S.pdf

Boucher, R. (2012, Noiembrie). *Windows Azure*. Preluat pe Iunie 2013, de pe msdn.microsoft.com: http://msdn.microsoft.com/en-us/library/dd163896.aspx

Datta, R., Joshi, D., Li, J., & Wang, J. Z. (2008, Aprilie). *Image retrieval: Ideas, influences, and trends of the new.* Preluat de pe infolab.standford.edu: http://infolab.stanford.edu/~wangz/project/imsearch/review/JOUR/datta.pdf

Deok-Hwan Kim, C.-W. C. (2005, Martie 9). *Relevance feedback using adaptive clustering for image similarity retrieval.* Preluat pe Iunie 2013, de pe http://islab.kaist.ac.kr/: http://islab.kaist.ac.kr/chungcw/relevance.pdf

Deselaers, T. (2003, Decembrie). *Features for Image Retrieval.* Preluat de pe thomas.deselaers.de: http://thomas.deselaers.de/publications/papers/deselaers\_diploma03.pdf

Finn, R. (1994). *Querying by image content.* Preluat de pe ibm.com: http://domino.research.ibm.com/comm/wwwr\_thinkresearch.nsf/pages/image396.html

Fischer, R. (1938). The statistical utilization of multiple measurements. *Annals of Eugenics 8*, 376–386.

Kai Song, Y. T. (2006). *Diversifying the Image Retrieval Results.* Preluat pe Iunie 2013, de pe http://idm.pku.edu.cn/: http://idm.pku.edu.cn/lunwen/2006\_2008626110631.pdf

Leetham, K. (2012, Septembrie). *Designing responsively*. Preluat pe Iulie 2013, de pe netmagazine.com: http://www.netmagazine.com/opinions/designing-responsively

Naik, A. (2010). Preluat de pe dataclusteringalgorithms: https://sites.google.com/site/dataclusteringalgorithms/

Nordbotten, J. (2008, Decembrie). *Multimedia Information Retrieval Systems.* Preluat de pe nordbotten.com: http://nordbotten.com/ADM/ADM\_book/MIRS-frame.htm

*Walkthrough: Creating a Web Site using Razor Syntax in Visual Studio*. (fără an). Preluat pe Iunie 2013, de pe msdn.microsoft.com: http://msdn.microsoft.com/en-us/library/gg606533(v=vs.100).aspx

Weyand, T. (2005, Octombrie). *Combining Content-Based Image Retrieval with Textual Information Retrieval.* Preluat de pe tobw.net: http://tobw.net/work/cbir/weyand--research-project--05.pdf

1. *Resource Description Framework (RDF)*  is the standard model for data interchange on the Web and it is the *de facto* framework for storing data in a graph-oriented manner. [↑](#footnote-ref-1)
2. *SPARQL Protocol and RDF Query Languag (SPARQL)* is a semantic query language usedfor manipulating data stored in *RDF* format. [↑](#footnote-ref-2)
3. *.NET Compiler Platform*, having the codename "*Roslyn*", is an open-source project created by *Microsoft* which includes compiler and code analysis *API*s for *C#* and *Visual Basic.NET* programming languages. [↑](#footnote-ref-3)
4. *RDF Coder* [↑](#footnote-ref-4)
5. *Fuzzy Ontology Framework* [↑](#footnote-ref-5)
6. *SCRO* [↑](#footnote-ref-6)
7. *Source Code Plagiarism Detection Method* [↑](#footnote-ref-7)
8. *Treo* [↑](#footnote-ref-8)
9. *TBSL* [↑](#footnote-ref-9)
10. *Squall2sparql* [↑](#footnote-ref-10)
11. *GFMed* [↑](#footnote-ref-11)
12. *Semantic Web* [↑](#footnote-ref-12)
13. *Word Wide Web Consortium (W3C)*  [↑](#footnote-ref-13)
14. *RDFS* [↑](#footnote-ref-14)
15. *OWL* [↑](#footnote-ref-15)
16. metadata - taken from the Greek meta- meaning “after” [↑](#footnote-ref-16)
17. *MSDN -*  [↑](#footnote-ref-17)