IJCSC 0973-7391 Volume 7 • Number 2 March 2016 - Sept 2016 pp. 157-168 available online at www.csjournalss.com

Class-Oriented Model Graph Design Based on Abstract Syntax Tree

Appala Srinuvasu Muttipati[1], Durga Prasad Dakineni[2]
[1,2]Assistant Professor, Department of Computer Science and Engineering, Baba Institute of Technology and Sciences, Visakhapatnam, India
srinuvasu.mutti@gmail.com, info2ddp@gmail.com

Abstract: Software model graph might play an important role in the representation of software system. To achieve this, essential initial works are found in the literature. For instance, construction of the model graph for software system includes different types of classes and their relationships between the classes. It is now essential to find a system such that it will be possible to implement model graph to visualize a software system or software design. In this paper, proposes an approach for constructing of a class-oriented model graph and analyzing source code using abstract syntax tree. This approach recovers the high-level structure of software architectures solidly from source code to produce a Class-Oriented (software) Model Graph. The obtained model graph will endow with the profoundly understanding of relations between the classes while compared with Unified Modeling Language class diagram.
Keywords: abstract syntax tree, source code analysis, class-oriented model graph, class diagram.

Introduction

In several software engineering disciplines, source code analysis represents a basic and preliminary step needed to perform activities such as software maintenance and program transformation. Programming building design, in addition, assumes a key part in refactoring procedures, that represent the receptive a part of maintenance assignments. Refactoring enhances the inward design structure of software system by keeping the generation of low-quality product. Recognizing these kinds of identical non- customary design patterns and customary design defects may provide a significant advantage in terms of reducing the cost of maintenance; the rationale is that the most commonly-used structures in software design are the most effective places to look for refactoring opportunities that they influence numerous elements of the design. for instance, non- customary structures that prefer to design patterns is also changed to fits normal forms, and common design defects are directly distinguished, which allows them to be mounted in multiple areas at once. Likewise, frequent recurrent indistinguishable design structures are typically the most reusable elements of the design; these elements will provide sensible candidates for added use in future design.

This paper proposes an Abstract Syntax Tree based Class-Oriented model Graph (AST-COG) tool for identifying and representing a graph with classes and their relationships or association between the classes for object-oriented language programs written in C#. It also makes use of the Nfactory libraries (open source) to extract the information at various levels (class/method/assembly) in order to generate the abstract syntax tree from which the class-oriented model graph is constructed. The graph visualization software is utilized to show the model graph. This paper is organized as follows: section 2 presents the background studies, section 3 discusses the proposed approach used to construct the abstract syntax tree from the C# source code and then constructed the model graph. Section 4 talks about the implementation of the tool, section 5 discusses the outcome of the project. Finally, section 6 represents the conclusion.

Related Work

In this section, we will discuss the existing graph notations and their relationships. We will also describe the existing libraries that are used for abstract syntax tree and graph.

A. Unified Modeling Language

The Unified Modeling Language (UML) is proverbial by all software system architects, developers, and testers. UML is used for documenting use cases, class diagrams, sequence diagrams, and activity diagram, sequence diagram will only offer the sequence flow of the application however class diagram could be a bit completely different. Thus, it is the foremost widespread UML diagram within the software engineer community [1]. The class diagram could be a static diagram. It represents the static view of an application. the class diagram is not only used for visualizing, describing and documenting completely different options of a system however additionally for constructing an executable code of the software system application.

The class diagram describes the attributes and operations of a class and also the constraints obligatory on the system. The class diagram is widely utilized in the modeling of object-oriented systems as a result of they are the sole UML diagrams which may be mapped directly with object oriented languages.

The class diagram shows a set of classes, interfaces, associations, collaborations and constraints. The class diagram is also called a structural diagram. The rationale of the class diagram is to model the static view of an application. The class diagrams are the exclusive diagrams which may be directly mapped with object-oriented languages and therefore widely used at the time of construction.

There are several software system tools that assist software system engineers to achieve this either by forward engineering or reverse engineering.

- Forward engineering could be an ancient method of moving high-level abstracts and logical, implementation-independent designs to the physical implementation of a system.
- Reverse engineering could be a method of analyzing an existing system to identify its elements and their interrelationships and to form representations of the system at a high level of abstraction. In most cases, reverse engineering is used to retrieve missing design documents from the prevailing source code in an abstract model UML format for finding out each the static structure and dynamic behavior of a system.

B. Abstract Syntax Tree in Java

The Abstract Syntax Tree is the base framework for many powerful tools of the Eclipse IDE, including refactoring, Quick Fix and Quick Assist. The Abstract Syntax Tree maps plain Java source code in a tree form [2]. This tree is more convenient and reliable to analyze and modify programmatically than text-based source. This article shows how you can use the Abstract Syntax Tree for your own applications.

To start off, you provide some source code to parse. This source code can be supplied as a Java file in your project or directly as a char[] that contains Java source. The source code described is parsed. All you need for this step is provided by the class org.eclipse.jdt.core.dom.ASTParser. This phase is called “Parsing source code”.

The Abstract Syntax Tree is the result. It is a tree model that entirely represents the source you provided by parser. If requested, the parser also computes and includes additional symbol resolved information called "bindings".

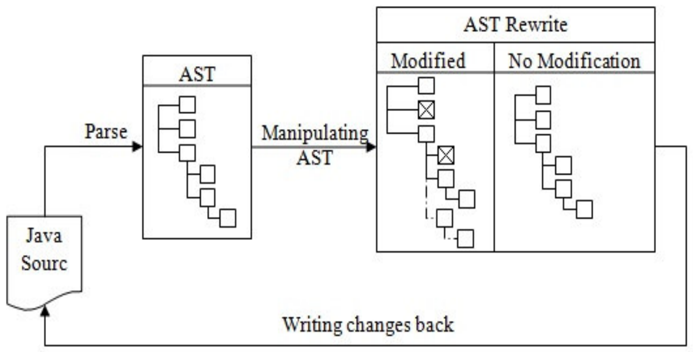


Figure1. Abstract Syntax tree in Java

Manipulating the AST is needs to be changed; this can be done in two ways:

1. By directly modifying the AST.
2. By noting the modifications in a separate protocol. This protocol is handled by an instance of ASTRewrite.

If changes have been made, then they need to be applied to the source code that was provided by java source.

Clone detection using abstract syntax trees was suggested by Andrew Yahin et al [3]. A functional system for recognizing close-miss and sequence clones on scale has been introduced. The methodology is taking into account varieties of strategies for compiler common sub expression elimination using hashing. The method is implemented directly by standard parsing technology which identifies clones in arbitrary language constructs, and computes macros that permit evacuation of the clones without influencing the operation of the program. The method is applied to a genuine use of moderate scale, and affirmed past appraisals of clone density of 7-15%, suggesting there is a “manual” software engineering process “redundancy” consistent. Automated methods can recognize and remove such clones, lowering the value of this constant, at concomitant savings in software engineering or maintenance costs. Clone discovery tools additionally have good potential for supporting domain analysis.

C. NRefactor

NRefactory is the C# analysis library used in the SharpDevelop and MonoDevelopIDEs. It allows applications to easily analyze both syntax and semantics of C# programs. It is quite similar to Microsoft's Roslyn project; except that it is not a full compiler-NRefactory [4] only analyzes C# code, it does not generate IL code. The .NET Compiler Platform "Roslyn" uncovered an arrangement of Compiler APIs and Workspaces APIs that gives affluent information about source code and that has full dedication with the C# and Visual Basic dialects [5]. The move to compilers as a stage drastically brings down the obstruction to passage for making code centered devices and applications. It makes numerous open doors for advancement in areas, for example, meta-programming, code generation and change, intuitive utilization of the C# and VB language, and installing of C# and VB in area particular language. The goals of the rewrite were to have a more accurate syntax tree (including exact token positions) and to integrate the IDE-specific semantic analysis features and refactorings into the NRefactory library. This allows us to share them between SharpDevelop and MonoDevelop. The architecture of the NRefactory with two system types is shown in Figure 2.

NRefactory 5.0 has recently shipped with MonoDevelop 3.0. NRefactory is also used in the ILSpy decompiler, in Unity's IL-to-Flash converter, and as a front-end in the C#-to-JavaScript compiler Saltarelle.

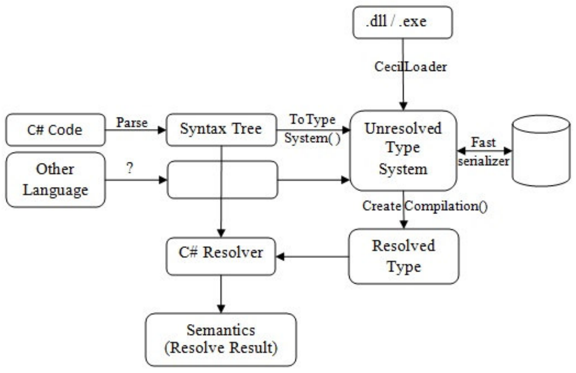


Figure 2. Architecture of NRefactory with its two system types

Software quality tool for measuring the different code metrics for C# source code using Abstract syntax tree is proposed by Pavitdeep singh et al., [6]. Author generated abstract syntax tree of the source code by using Nfactory libraries.

D. Graph Libraries

Graphs are mathematical abstractions that are helpful for finding many varieties of issues in engineering science. Consequently, these abstractions should even be delineating in computer programs. A uniform generic interface for traversing graphs is of utmost importance to encourage utilized of graph algorithms and data structures. A part of the Boost Graph Library [7] could be a generic interface that enables access to a graph's structure, however, hides the main points of the implementation. this can be an “open” interface within the sense that any graph library that implements this interface are going to be practical with the BGL generic algorithms and with different algorithms that conjointly use this interface. The BGL provides some general purpose graph categories that adapt to the present interface, however, they are not meant to be the only graph categories, there actually are going to be different graph classes that are better for certain things.

E. Graph Visualizes

Graphviz is open source graph representation software [8]. Graph visualization is a way of representing structural information as diagrams of abstract graphs and networks. It has vital applications in systems administration, bioinformatics, software engineering, database and web design, machine learning, and in visual interfaces for other technical domains.

Proposed Approach

The proposed design of the tool consists of various phases from C# source code to syntax tree creation. Once the syntax tree is generated it is resolved to using the type system to generate the semantic tree, which is further utilized by the tool to recognize the class names, their relation and graph representation. The overall architecture of the proposed system is described as shown in Figure 3.

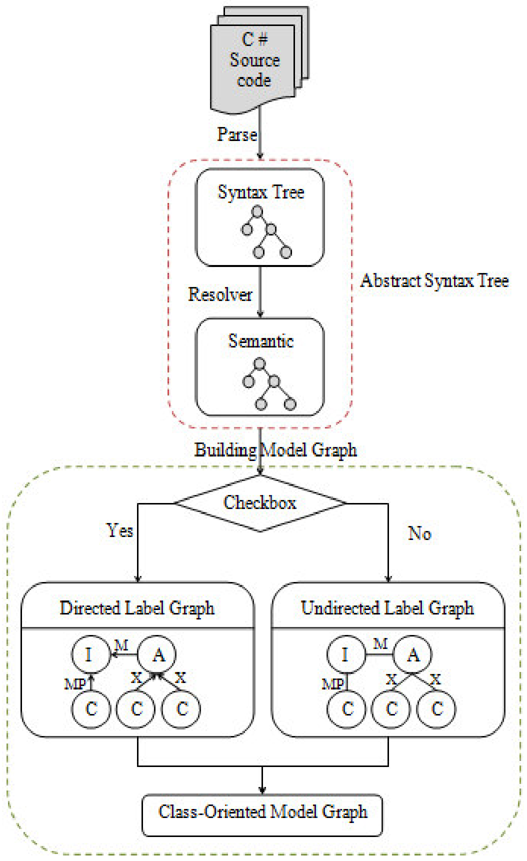


Figure 3. Block Diagram for Construction of AST-COG

A. C# Source code

The proposed tool initially takes a single file as input and then reads all the projects within that single file (Solution file) and then parses the project files to find all the source files within the projects. During traversal of various files present it will also filter out the files which are marked for exclusion during parsing. Once all the required files are read by the system they are passed to the language parser for parse or syntax tree creation.

B. Syntax Tree

C# source code is simply a string. Parsing the string into a syntax tree tells us that it is an invocation expression, which has a member reference as target. The syntax tree would be similar to the tree shown in Figure 5.

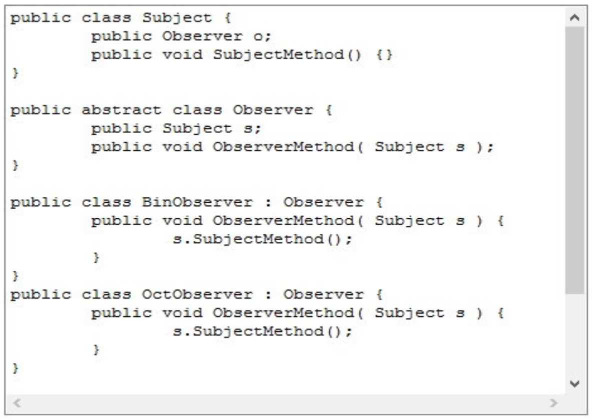
```
new InvocationExpression
{
    Tar = new MemberReferenceExpression
          {
          Tar = new IdentifierExpression("obj"),
           MemberName = "DoThing"
          },
           Arguments =
          {
           new PrimitiveExpression(0)
          }
}
```

C. Semantic Tree

A syntax tree doesn't provide us the complete information regarding what Object or DoThing. DoThing most likely is an instance method, and Object seems to be a local variable, parameter or a field of the current class. It may be that Object could be a class name and DoThing a static field containing a delegate. The semantic tree provides the information regarding these attributes. The semantic tree constructed from the above syntax tree would to be similar to the tree shown as follows

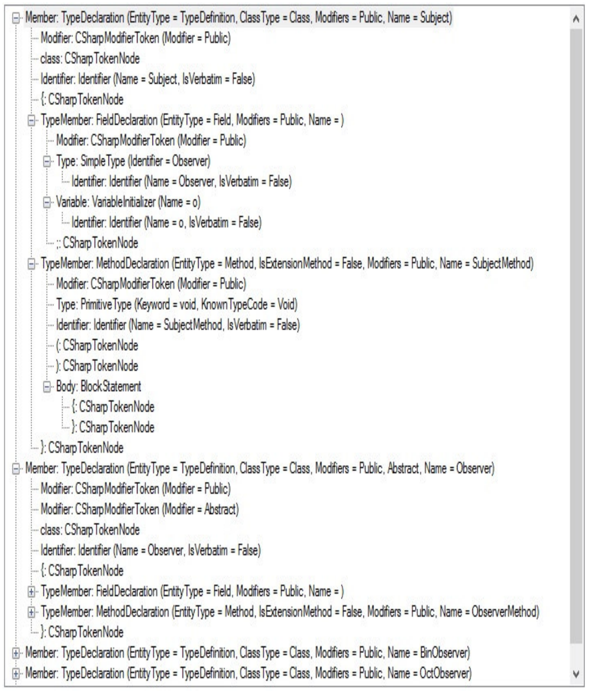
```
new InvocationResolveResult  {
TatResult = new LocalResolveResult('myclass Object;'),
Member = 'public void TestClass.DoThing (int)',
Arguments =
{        new ConstantResolveResult('System.Int32', 0)      }
 }
```

Figure 5 exemplify an instance of the parser or syntax tree for the sample C# source code from Figure 4.



```
public class Subject {
        public Observer o;
        public void SubjectMethod() {}
}

public abstract class Observer {
        public Subject s;
        public void ObserverMethod( Subject s );
}

public class BinObserver : Observer {
        public void ObserverMethod( Subject s ) {
                s.SubjectMethod();
        }
}
public class OctObserver : Observer {
        public void ObserverMethod( Subject s ) {
                s.SubjectMethod();
        }
}
```

Figure 4. Sample Source code



```
⊟ Member: TypeDeclaration (EntityType = TypeDefinition, ClassType = Class, Modifiers = Public, Name = Subject)
    Modifier: CSharpModifierToken (Modifier = Public)
    class: CSharpTokenNode
    Identifier: Identifier (Name = Subject, IsVerbatim = False)
    {: CSharpTokenNode
  ⊟ TypeMember: FieldDeclaration (EntityType = Field, Modifiers = Public, Name = )
      Modifier: CSharpModifierToken (Modifier = Public)
    ⊟ Type: SimpleType (Identifier = Observer)
        Identifier: Identifier (Name = Observer, IsVerbatim = False)
    ⊟ Variable: VariableInitializer (Name = o)
        Identifier: Identifier (Name = o, IsVerbatim = False)
      ;: CSharpTokenNode
  ⊟ TypeMember: MethodDeclaration (EntityType = Method, IsExtensionMethod = False, Modifiers = Public, Name = SubjectMethod)
      Modifier: CSharpModifierToken (Modifier = Public)
      Type: PrimitiveType (Keyword = void, KnownTypeCode = Void)
      Identifier: Identifier (Name = SubjectMethod, IsVerbatim = False)
      (: CSharpTokenNode
      ): CSharpTokenNode
    ⊟ Body: BlockStatement
        {: CSharpTokenNode
        }: CSharpTokenNode
    }: CSharpTokenNode
⊟ Member: TypeDeclaration (EntityType = TypeDefinition, ClassType = Class, Modifiers = Public, Abstract, Name = Observer)
    Modifier: CSharpModifierToken (Modifier = Public)
    Modifier: CSharpModifierToken (Modifier = Abstract)
    class: CSharpTokenNode
    Identifier: Identifier (Name = Observer, IsVerbatim = False)
    {: CSharpTokenNode
  ⊞ TypeMember: FieldDeclaration (EntityType = Field, Modifiers = Public, Name = )
  ⊞ TypeMember: MethodDeclaration (EntityType = Method, IsExtensionMethod = False, Modifiers = Public, Name = ObserverMethod)
    }: CSharpTokenNode
⊞ Member: TypeDeclaration (EntityType = TypeDefinition, ClassType = Class, Modifiers = Public, Name = BinObserver)
⊞ Member: TypeDeclaration (EntityType = TypeDefinition, ClassType = Class, Modifiers = Public, Name = OctObserver)
```

Figure 5. Syntax tree for sample source code

D. Abstract Syntax Tree based Class-Oriented Model Graph

The proposed tool takes a single file as input and then reads all the projects within the solution file and then parses the project files to find the entire source file within the projects. During the traversal of various files present, it will filter out the files which are marked for exclusion during parsing. Once all the required files are read by the system, they pass the language parser of syntax tree creation. The vertices of this graph are classes, abstract classes, and interface classes. The edge of the graph represents directed relations between these entities (node/ vertices).

Relation types in the software model graph are based on UML-like relations. At this point, especially consider class and sequence diagrams of the UML. Moreover, to handle some important relations that is visually hidden in the UML diagrams. For example, if a method of a class has the same signature with a method of the parent class, then there is an "override" relation between these classes that is Number of visually observable in UML class diagrams. We also include some important high-level relations from UML sequence diagrams, such as the "create" and "method call" relations between entities. Possible entity types, relation types and their labels are given in Table 1.

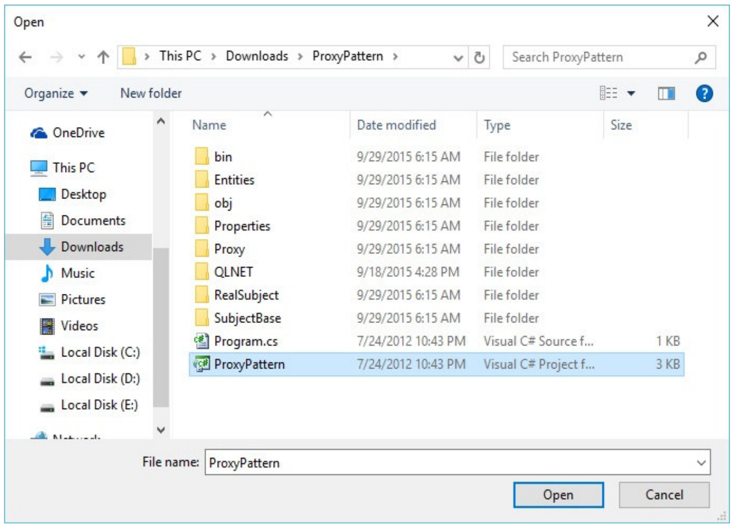
Table 1: Labels of nodes and edges of AST-MOC

Node Label	**Node Type**
C	Class
I	Interface
A	Abstract class
Edge Label	**Relation Type (Edges from P to Q)**
X	Extends
I	Implements
A	Has field type
T	Uses generic type declaration
L	Method has a local parameter
P	Uses methods parameter
R	Methods has been return type
M	Has method call
F	Access field
C	Creates
O	Overrides methods

The nature of the object oriented design is that, there can be more than one relation between the vertices (classes and interfaces). To build a simple and understandable graph, we collect all of the labels of parallel edges between two vertices into a solitary set of labels, such that "L_{ij}" is a set of labels of directed edge "e_{ij}" that contains all relation labels from vertex "v_i" to vertex "v_j". For example, if two entities have both method call {M} and method parameter {P} relations in the same direction, then the combined label set for this edge becomes $\{M\}\cup\{P\} = \{M,P\}$.

Implementation

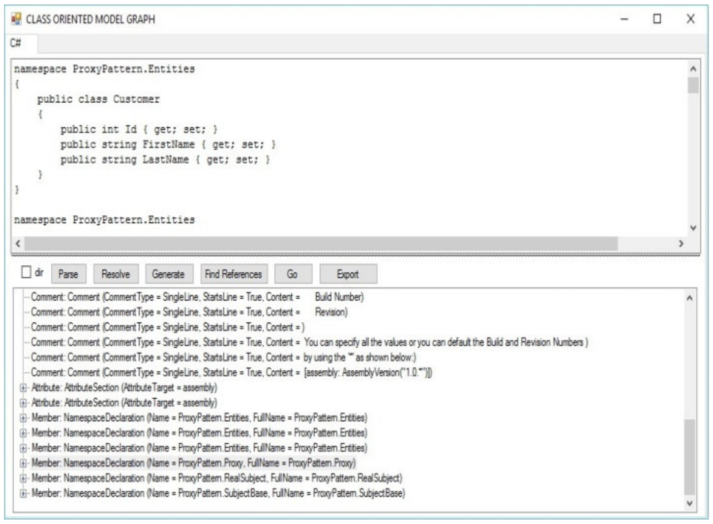
In this section we describe about our tool's (AST-COG) simple interface with a focus on its main features: analyzing and display are activated by clicking the respective buttons. The user interface contains parse, resolve, generate, find reference, go and export buttons. Figure6 shows the browsing of project file. Figure 7 shows the source code and syntax tree. Figure 8 shows the semantic tree for loaded file.



Name	Date modified	Type	Size
bin	9/29/2015 6:15 AM	File folder	
Entities	9/29/2015 6:15 AM	File folder	
obj	9/29/2015 6:15 AM	File folder	
Properties	9/29/2015 6:15 AM	File folder	
Proxy	9/29/2015 6:15 AM	File folder	
QLNET	9/18/2015 4:28 PM	File folder	
RealSubject	9/29/2015 6:15 AM	File folder	
SubjectBase	9/29/2015 6:15 AM	File folder	
Program.cs	7/24/2012 10:43 PM	Visual C# Source f...	1 KB
ProxyPattern	7/24/2012 10:43 PM	Visual C# Project f...	3 KB

File name: ProxyPattern

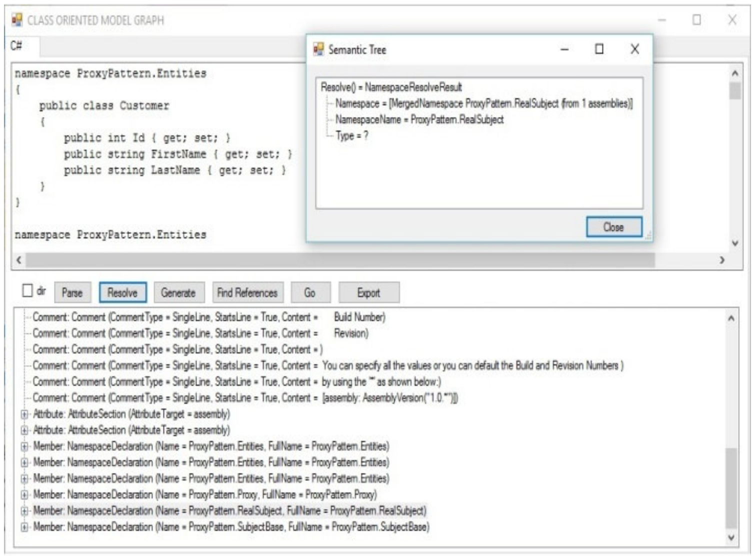
Figure 6. Browsing project file



```
namespace ProxyPattern.Entities
{
    public class Customer
    {
        public int Id { get; set; }
        public string FirstName { get; set; }
        public string LastName { get; set; }
    }
}

namespace ProxyPattern.Entities
```

```
 -Comment: Comment (CommentType = SingleLine, StartsLine = True, Content =      Build Number)
 -Comment: Comment (CommentType = SingleLine, StartsLine = True, Content =      Revision)
 -Comment: Comment (CommentType = SingleLine, StartsLine = True, Content = )
 -Comment: Comment (CommentType = SingleLine, StartsLine = True, Content =  You can specify all the values or you can default the Build and Revision Numbers )
 -Comment: Comment (CommentType = SingleLine, StartsLine = True, Content =  by using the '*' as shown below:)
 -Comment: Comment (CommentType = SingleLine, StartsLine = True, Content =  [assembly: AssemblyVersion("1.0.*")])
⊞-Attribute: AttributeSection (AttributeTarget = assembly)
⊞-Attribute: AttributeSection (AttributeTarget = assembly)
⊞-Member: NamespaceDeclaration (Name = ProxyPattern.Entities, FullName = ProxyPattern.Entities)
⊞-Member: NamespaceDeclaration (Name = ProxyPattern.Entities, FullName = ProxyPattern.Entities)
⊞-Member: NamespaceDeclaration (Name = ProxyPattern.Entities, FullName = ProxyPattern.Entities)
⊞-Member: NamespaceDeclaration (Name = ProxyPattern.Proxy, FullName = ProxyPattern.Proxy)
⊞-Member: NamespaceDeclaration (Name = ProxyPattern.RealSubject, FullName = ProxyPattern.RealSubject)
⊞-Member: NamespaceDeclaration (Name = ProxyPattern.SubjectBase, FullName = ProxyPattern.SubjectBase)
```

Figure 7. Source code and syntax tree for loaded file



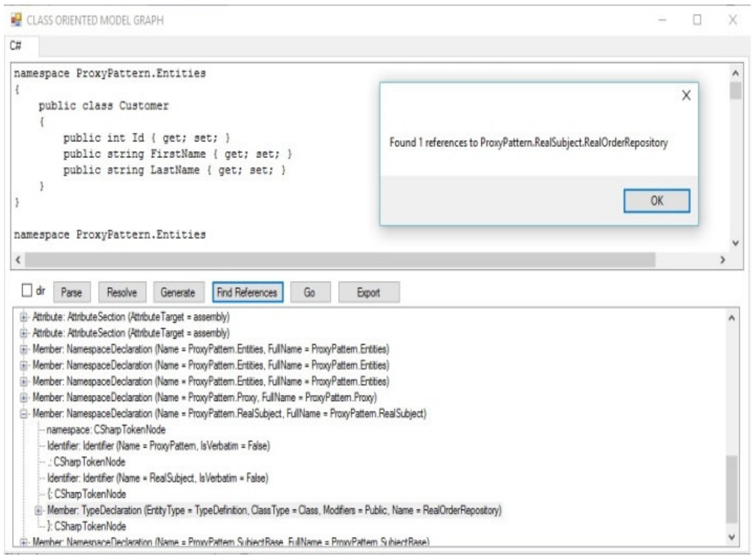
```
namespace ProxyPattern.Entities
{
    public class Customer
    {
        public int Id { get; set; }
        public string FirstName { get; set; }
        public string LastName { get; set; }
    }
}

namespace ProxyPattern.Entities
```

```
Resolve() = NamespaceResolveResult
  Namespace = [MergedNamespace ProxyPattern.RealSubject (from 1 assemblies)]
  NamespaceName = ProxyPattern.RealSubject
  Type = ?
```

```
Comment: Comment (CommentType = SingleLine, StartsLine = True, Content =      Build Number)
Comment: Comment (CommentType = SingleLine, StartsLine = True, Content =      Revision)
Comment: Comment (CommentType = SingleLine, StartsLine = True, Content = )
Comment: Comment (CommentType = SingleLine, StartsLine = True, Content =  You can specify all the values or you can default the Build and Revision Numbers )
Comment: Comment (CommentType = SingleLine, StartsLine = True, Content =  by using the '*' as shown below:)
Comment: Comment (CommentType = SingleLine, StartsLine = True, Content =  [assembly: AssemblyVersion("1.0.*")])
Attribute: AttributeSection (AttributeTarget = assembly)
Attribute: AttributeSection (AttributeTarget = assembly)
Member: NamespaceDeclaration (Name = ProxyPattern.Entities, FullName = ProxyPattern.Entities)
Member: NamespaceDeclaration (Name = ProxyPattern.Entities, FullName = ProxyPattern.Entities)
Member: NamespaceDeclaration (Name = ProxyPattern.Entities, FullName = ProxyPattern.Entities)
Member: NamespaceDeclaration (Name = ProxyPattern.Proxy, FullName = ProxyPattern.Proxy)
Member: NamespaceDeclaration (Name = ProxyPattern.RealSubject, FullName = ProxyPattern.RealSubject)
Member: NamespaceDeclaration (Name = ProxyPattern.SubjectBase, FullName = ProxyPattern.SubjectBase)
```

Figure 8. Semantic tree with resolve ()

Figure 9, find references of a member type declaration in the file shown below

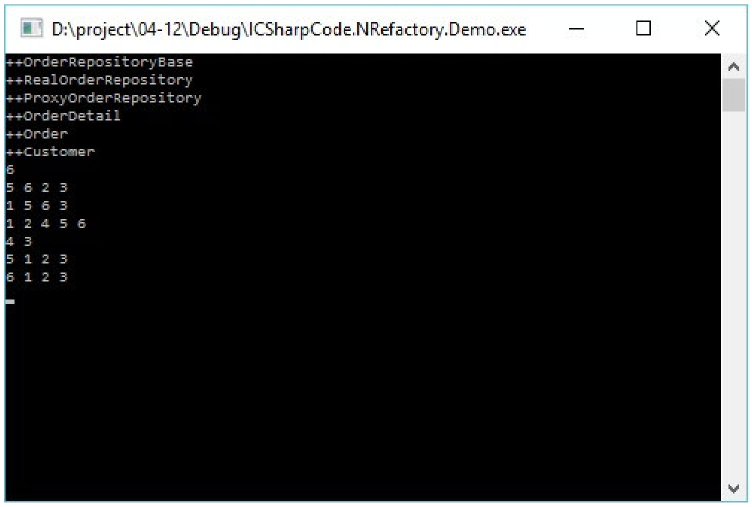


```
namespace ProxyPattern.Entities
{
    public class Customer
    {
        public int Id { get; set; }
        public string FirstName { get; set; }
        public string LastName { get; set; }
    }
}

namespace ProxyPattern.Entities
```

Found 1 references to ProxyPattern.RealSubject.RealOrderRepository

```
Attribute: AttributeSection (AttributeTarget = assembly)
Attribute: AttributeSection (AttributeTarget = assembly)
Member: NamespaceDeclaration (Name = ProxyPattern.Entities, FullName = ProxyPattern.Entities)
Member: NamespaceDeclaration (Name = ProxyPattern.Entities, FullName = ProxyPattern.Entities)
Member: NamespaceDeclaration (Name = ProxyPattern.Entities, FullName = ProxyPattern.Entities)
Member: NamespaceDeclaration (Name = ProxyPattern.Proxy, FullName = ProxyPattern.Proxy)
Member: NamespaceDeclaration (Name = ProxyPattern.RealSubject, FullName = ProxyPattern.RealSubject)
    namespace: CSharpTokenNode
    Identifier: Identifier (Name = ProxyPattern, IsVerbatim = False)
    .: CSharpTokenNode
    Identifier: Identifier (Name = RealSubject, IsVerbatim = False)
    {: CSharpTokenNode
    Member: TypeDeclaration (EntityType = TypeDefinition, ClassType = Class, Modifiers = Public, Name = RealOrderRepository)
    }: CSharpTokenNode
Member: NamespaceDeclaration (Name = ProxyPattern.SubjectBase, FullName = ProxyPattern.SubjectBase)
```

Figure 9. Find References



```
D:\project\04-12\Debug\ICSharpCode.NRefactory.Demo.exe
++OrderRepositoryBase
++RealOrderRepository
++ProxyOrderRepository
++OrderDetail
++Order
++Customer
6
5 6 2 3
1 5 6 3
1 2 4 5 6
4 3
5 1 2 3
6 1 2 3
```

Figure 10. Class names and adjacency nodes

In figure 6 we can observe the check box if the check box is selected it generates the direct label graph, if not it generated undirected label graph, which is shown in figure 11 and figure 12

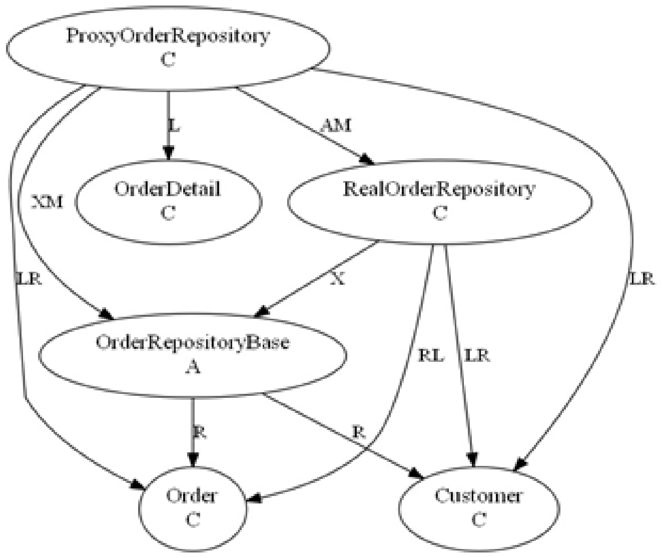


Figure 11. Directed Label graph

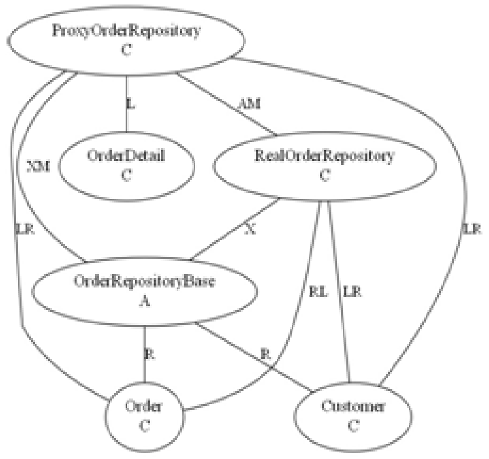
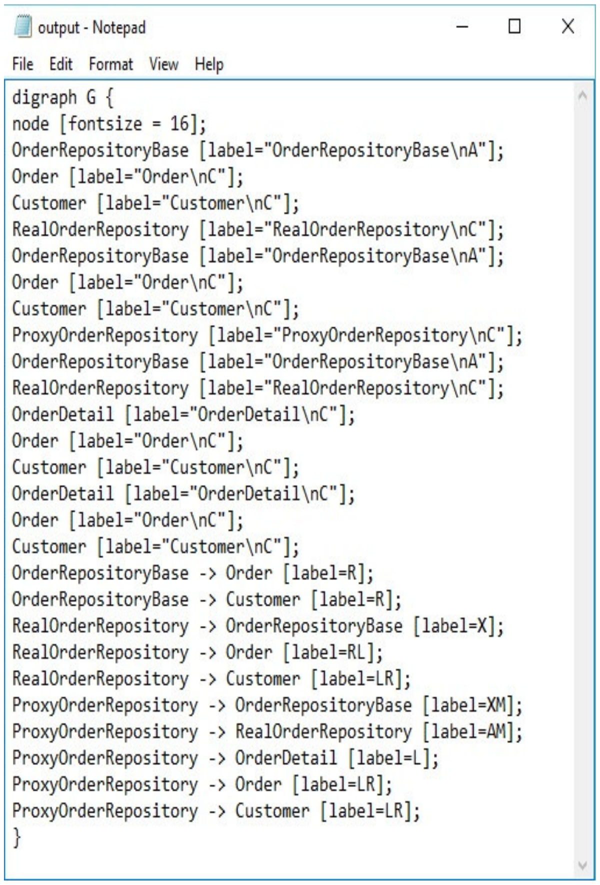


Figure 12. Undireected Label graph

Table 2: Nodes and edges in proxy

Node Label	Total number of node labels
C	5
I	0
A	1
Edge Label	**Total number of edge labels**
X	2
I	0
A	1
T	0
L	5
P	0
R	6
M	2
F	0
C	0
O	0

The above Table 2 gives the information about the total number of class labels and edge labels occur in the proxy pattern and Figure 13 shows the output with respective of class and their edge labels.



```
digraph G {
node [fontsize = 16];
OrderRepositoryBase [label="OrderRepositoryBase\nA"];
Order [label="Order\nC"];
Customer [label="Customer\nC"];
RealOrderRepository [label="RealOrderRepository\nC"];
OrderRepositoryBase [label="OrderRepositoryBase\nA"];
Order [label="Order\nC"];
Customer [label="Customer\nC"];
ProxyOrderRepository [label="ProxyOrderRepository\nC"];
OrderRepositoryBase [label="OrderRepositoryBase\nA"];
RealOrderRepository [label="RealOrderRepository\nC"];
OrderDetail [label="OrderDetail\nC"];
Order [label="Order\nC"];
Customer [label="Customer\nC"];
OrderDetail [label="OrderDetail\nC"];
Order [label="Order\nC"];
Customer [label="Customer\nC"];
OrderRepositoryBase -> Order [label=R];
OrderRepositoryBase -> Customer [label=R];
RealOrderRepository -> OrderRepositoryBase [label=X];
RealOrderRepository -> Order [label=RL];
RealOrderRepository -> Customer [label=LR];
ProxyOrderRepository -> OrderRepositoryBase [label=XM];
ProxyOrderRepository -> RealOrderRepository [label=AM];
ProxyOrderRepository -> OrderDetail [label=L];
ProxyOrderRepository -> Order [label=LR];
ProxyOrderRepository -> Customer [label=LR];
}
```

Figure 13. Output of the node label names and edge relations

Results and Discussion

The results are analyzed from proxy pattern source code. The outcome of class oriented model graph is shown in figure 11 and figure 12 which consists of 6 classes, five are classes and one is abstract classes and 16 relations between the classes is shown in Table 2. The tool will generate the directed as well as undirected label graphs for given project. Table 3, shows the information about Gifviewer and QLNET applications with total number of nodes/vertices and edges and their edge weights. The correctness of the tool is confirmed by a manual check process within various classes of the source code are considered and walking through the graph established the correctness of our tool. The tool yields correct results for all the applications developed with different .Net frameworks.

Table 3: Information about two real world applications

Name of the application	Number of vertices	Number of edges	Number of edge weight							
			X	I	A	L	P	R	M	O
GIFVIEWER	47	165	11	0	29	38	3	3	176	33
QLNET	735	899	461	49	255	753	234	256	210	50

Conclusion

In this paper presented an abstract syntax tree based class-oriented model graph tool which is specifically designed to cover the few communication messages in this construction. It takes the solution file for the C# application as an input to the system. It then loads all the projects files in memory. Project files are read one by one to load all the source code files in order to construct the syntax tree. Once the creation of the syntax trees are completed, it is ready for the analysis. Graph is generated at the class and relations. The class-oriented model graph generated via this tool is quite satisfactory. Moreover, user can save the data to notepad using the export functionality for future work and research purpose. Future enhancement for this work is to find the different types of the software structures are included in a particular object-oriented system by applying a clustering approach.

References

[1] Booch, J. Rumbaugh and I. Jacobson, Unified Modeling Language User Guide, 2nd ed., Addison-Wesley Professional, 2005.

[2] Thomas Kuhn, Abstract Syntax Tree, Olivier Thomann. Made available under the EPL v1.0 November 20, 2006. http://www.eclipse.org/articles/Article-JavaCodeManipulation_AST/

[3] Baxter, A. Yahin, L. Moura, M. S. Anna and L. Bier, “Clone Detection Using Abstract Syntax Trees,” In Proceeding of the international Conference on Software Maintenance, Washington, DC, USA: IEEE Computer Society, 1998, pp. 368.

[4] Using NRefactory for analyzing C# code http://www.codeproject.com/Articles/408663/Using-NRefactory-for-analyzing-Csharp-code

[5] Roslyn, https://roslyn.codeplex.com/

[6] P. Singh, S. Singh and J. Kaur, “Tool for generating code metrics for C# source code using abstract syntax tree technique. SIGSOFT Software Engineering. Notes 38, 2013, vol. 5, pp.1-6.

[7] Boost C++ Libraries, http://www.boost.org/doc/libs/1_60_0/libs/graph/doc/table_of_contents.html

[8] Graphviz-Graph Visualization software, http://www.graphviz.org/