ISSN:0975-9646

# Construction of Software Model Graph and Analysing Object-Oriented Program(C#) Using Abstract Syntax Tree Method

Appala Srinuvasu Muttipati[#1], Poosapati Padmaja[*2]

[#]*Research Scholar, Department of Computer Science and Engineering, GITAM University Visakhapatnam, India*

[*]*Associate Professor, Department of Information Technology, GITAM University Visakhapatnam, India*

*Abstract*— **Software maintenance is one of the key exercises in any software engineering process in which source code examination assumes a critical part. Because of the high cost of maintenance, it has turnout to be very important to deliver high-quality software. Over the long haul, various investigations have been performed on source code to focus intricacy. This paper proposes an approach for construction of Software Model Graph and analyzing source code using Abstract syntax tree. With the assistance of our approach, source code analysis, can be recover the high-level structure of a software framework directly from its source code them produce a software model graph. Coupling, Common design structures, Refractor design structures can be distinguished regardless of the fact that their source code is altered. These reproduced structures can be consolidated into a single library entity, to be utilized productively in different parts of the current project or in future projects. This approach will likewise stay away from conflicting bug fixes..**

*Keywords*— **Source code Analysis, Abstract Syntax Tree, Software Model Graph, Coupling**

## I. Introduction

In many software engineering disciplines, source code analysis represents a fundamental and preliminary step required to perform activities such as software maintenance and program transformation. Programming building design additionally assumes a key part in refactoring procedures, which constitute the receptive part of upkeep assignments. Refactoring enhances the inward design structure of software by keeping the generation of low-quality products. Recognizing these types of identical non- standard design patterns and common design defects could offer a momentous advantage in terms of reducing the cost of maintenance; the reason is that the most commonly-used structures in software design are the best places to search for refactoring opportunities that they influence various parts of the design. For example, non-standard structures that are like to design patterns may be modified to comply with standard forms, and common design defects can be immediately distinguished, which permits them to be fixed in multiple areas at once. Likewise, frequent repeated indistinguishable design structures are usually the most reusable parts of the design; these parts can provide good candidates for additional use in future design.

Another source of the clones is the reproduced code because of copy past activities. Basically, developers modify these repeated parts independently to permit their source code to change, yet the design remains same [1, 2]. The code quality could diminish if developers apply a bug fix to one structure yet neglect to apply the same amendment to its duplicates.

### A. Unified Modeling Language

Unified Modeling Language (UML) represents a unification of the concepts and notations exhibited by the three amigos Grady Booch, Jim Rumbaugh, and Ivar Jacobson [3]. The objective is for UML to become a common language for creating models of object oriented programming. The two noteworthy parts of the UML: a Meta-Model and a Notation. The meta-model is a description of UML in UML. It clarifies the objects, attributes, and relationships necessary to represent the concepts of UML within a software application. The UML notation is rich and full bodied. It is included two noteworthy subdivisions. There is a notation for modeling the static components of an outline, for example, classes, attributes, and relationships. There is additionally a notation for modeling the dynamic components of an outline, for example, objects, messages, and finite state machines. For example consider an object oriented program (java, C#) analyze that source code and transform into UML class diagram shown in Fig.1 based on this construct a software model graph is shown in section B.

### B. Existing Software Model Graph

The purpose of a UML class diagram is to depict the classes within a model. In an object oriented application, classes have attributes (member variables), operations (member functions) and relationships (association, aggregation, inheritance, dependence and composite) with other classes. The UML class diagram can depict all these things quite easily. In most of the papers construction of software model graph is based on class diagram, where each class is represented as a vertices/ node and labeled C. The relations between the classes are considered as edges. Relationships {association, generalization, dependence, aggregation and realization} which are assigned with a unique prime number {2, 3, 5, 7, 11, and 13} and unique numbers label as edge weight. For example X and Y have the relations association and aggregation then edge weight will be the 2*7=14. Figure 2 shows the construction of Software model graph from the UML class diagram.

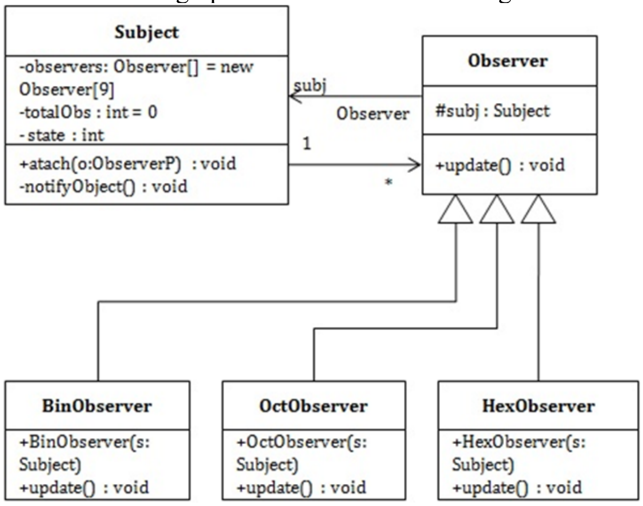


Fig. 1 UML class diagram of Observer pattern implementation

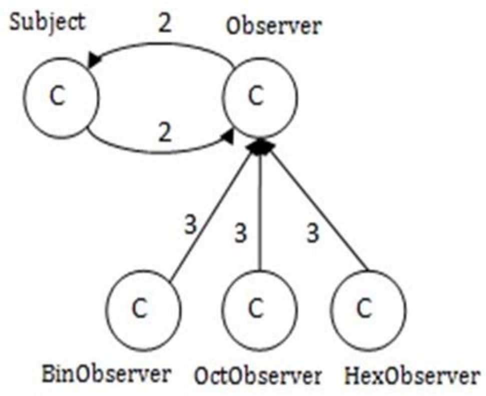


Fig. 2 UML based Software Model Graph for Observer pattern

## C. Cohesion and Coupling

The term coupling is used to gauge the relative interdependency between different classes as one class has the connected with another class. While on the other hand cohesion is defined as the strength of the attributes inside the class which implies how the attributes are connected inside the class. Coupling is constantly correlated with cohesion in such a way as if coupling is high then cohesion is low and vice versa [4]. One can say that a class is highly coupled or many dependent with other classes, if there are many associations and loosely coupled or some dependent with other classes if there is a less associations. The coupling is decided at the designing phase of the framework, it depends on the interface complexity of the classes. Therefore, the coupling is a degree at which a class is associated with other classes in the system.

Let us now illustrate the cohesive class which can perform a solitary undertaking inside of the software procedure. It obliges little cooperation with different procedures that are utilized as a part of different parts of a program. Cohesion gives the strength to the bond between attributes of a class and it is an idea through which catch the intra-module with cohesion. Therefore, cohesion is used to decide how closely or tightly bound the internal attributes of a class to each other. Cohesion gives a proposal to the designer about whether the different attributes of a class have a place together in the same class. Thus, the coupling and cohesion are associated with one other; hence the Fig. 3 demonstrates the general representation of coupling and cohesion.

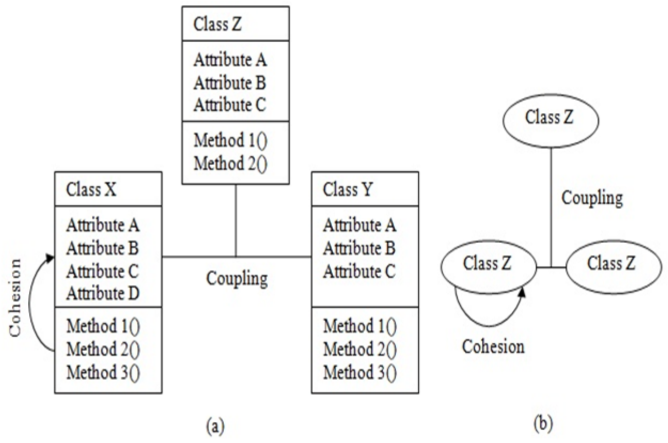


Fig. 3 a) General representation of cohesion and coupling b) software model graph representation of cohesion and coupling

## D. Paper Contributions

This paper proposes a software model graph, by use of the Nfactory libraries (open source) to extract the information at various levels in order to generate the abstract syntax tree from object oriented language (C#). The use of generating software model graph is to provide in-depth information regarding the interactions between the classes (vertices / nodes). The communication message of the nodes are shown with the edge weight in model graph, which helps developers in Refactoring, reused designs, common design structures, circular dependency structures and pattern design. All those factors can observe by utilizing Software model graph

## E. Paper Organization

The rest of the paper is organized as follows: section 2 presents the related work, section 3 discusses the proposed approach used to generate the syntax or parse tree from the C# source code and then build software model graph. Section 4 talks about the implementation details of the software model graph, section 5 discusses the outcome of the project. Finally, section 6 draws the conclusion and the future work.

## II. Related Work

Jeffrey L et al. [5] proposed a six grammar annotations - omission, labeling, Boolean access, list formation, inlining, and super class development that permit an abstract syntax for a language to be characterized based on its concrete syntax. An annotated grammar, generate both a parser and a rewritable AST. Concretizing the AST permits it to protect the designing of the original code even after rewriting. Moreover, an AST generator based on our method can couple the produced parser/AST-builder with a pseudo-preprocessor to permit representation and control of pre-processed code. In Ludwig, AST generator has been implemented and has been used to generate the rewritable

AST in a refactoring; the annotated grammar was more than a request of extent smaller than the generated code, and the overhead of concretizing ASTs was very reasonable.

Andrew Yahin et al [6] proposed Clone detection using abstract syntax trees. 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.

Pavitdeep singh et al [7] proposed a software quality tool for measuring the different code metrics for C# source code using Abstract syntax tree. Nfactory libraries are used to generating abstract syntax tree of the source code.

Harjot Singhvirdi and Balraj Singh [8] proposed different types of coupling i.e. static and dynamic coupling. These metrics performed under the different environments and calculate the mean and the standard. The value of standard deviation is useful in judging the representativeness of the mean and quality of software system.

## III. Proposed Approach

The proposed approach consists of various steps from C# source code to syntax tree creation. Once the syntax parse is generated it is resolved to using the Type system to generate the semantic tree, which is further utilized to construct the Software model graph refer to Fig.4.

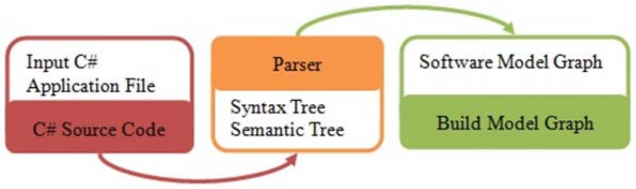
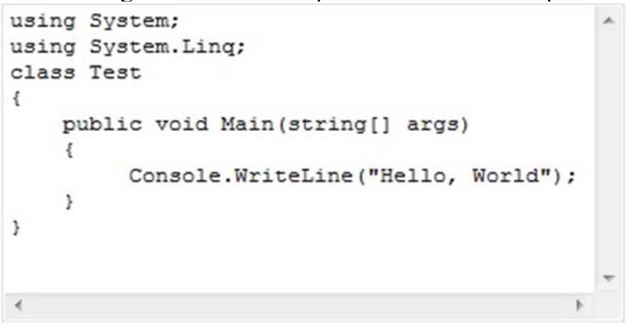


Fig. 4 Approach for constructing Software Model Graph

```
Algorithm: Constructing a software model graph
Input: C# Source Code
Output: Directed graph (Software Model Graph)
Step 1: Source code (object oriented code) as input
Step 2: Source code samples are passed into the language
        parse as input.
Step 3: Parser analyses the base class of the syntax tree as
        the AST (Abstract Syntax Tree)
Step 4: AST method is used to determine the semantics of
        any node classes within the syntax tree.
Step 5: generating software model graph
        a. Class ← node / vertex
        b. Relationship ← edge
        c. Labeling the node/vertices and placing the edge
           weights to edges.
Step 6: The output is Directed graph of Software model
        graph
```

### A. *C# Source Code*

The proposed system at first takes a single file as input and afterward peruses all the tasks inside of that single file (Solution file) and afterwards parses the task files to discover all the source files inside of the activities. During Amid traversal of different files present it will likewise shift through the files which are checked for prohibition during parsing. When all the obliged files are read by the system they are passed to the language parser for syntax tree creation. Fig.5 shows the sample source code example.

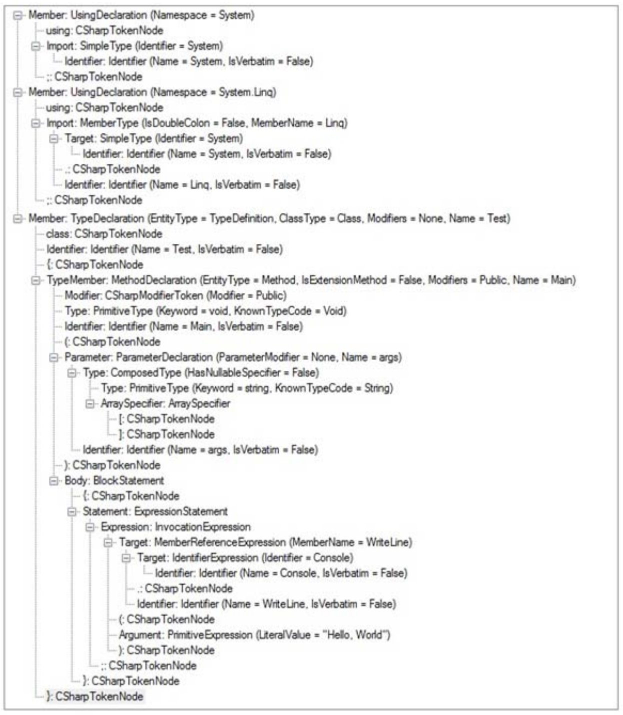


```
using System;
using System.Linq;
class Test
{
    public void Main(string[] args)
    {
        Console.WriteLine("Hello, World");
    }
}
```

Fig.5 Source Code Example

### B. *Language parser*

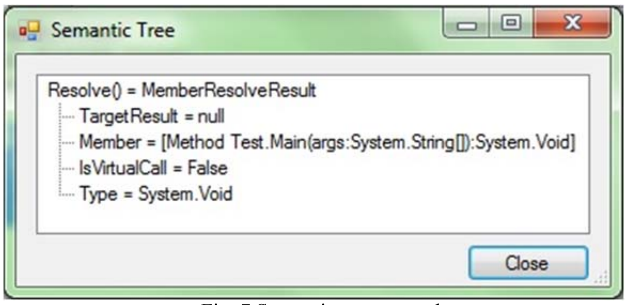
*1) Syntax Tree:* C# source code is just a string. Parsing the string into a syntax tree informs that it is an invocation expression, which has a member reference as target. The syntax tree is shown in Fig.6. A syntax tree doesn’t give the complete information regarding object. Some object most in all likelihood is a case technique, and some object seems to be a local variable, parameter or a field of the current class. It might be that some object could be a class name.



```
⊟ Member: UsingDeclaration (Namespace = System)
    using: CSharpTokenNode
  ⊟ Import: SimpleType (Identifier = System)
      Identifier: Identifier (Name = System, IsVerbatim = False)
    ;: CSharpTokenNode
⊟ Member: UsingDeclaration (Namespace = System.Linq)
    using: CSharpTokenNode
  ⊟ Import: MemberType (IsDoubleColon = False, MemberName = Linq)
    ⊟ Target: SimpleType (Identifier = System)
        Identifier: Identifier (Name = System, IsVerbatim = False)
      .: CSharpTokenNode
      Identifier: Identifier (Name = Linq, IsVerbatim = False)
    ;: CSharpTokenNode
⊟ Member: TypeDeclaration (EntityType = TypeDefinition, ClassType = Class, Modifiers = None, Name = Test)
    class: CSharpTokenNode
    Identifier: Identifier (Name = Test, IsVerbatim = False)
    {: CSharpTokenNode
  ⊟ TypeMember: MethodDeclaration (EntityType = Method, IsExtensionMethod = False, Modifiers = Public, Name = Main)
      Modifier: CSharpModifierToken (Modifier = Public)
      Type: PrimitiveType (Keyword = void, KnownTypeCode = Void)
      Identifier: Identifier (Name = Main, IsVerbatim = False)
      (: CSharpTokenNode
    ⊟ Parameter: ParameterDeclaration (ParameterModifier = None, Name = args)
      ⊟ Type: ComposedType (HasNullableSpecifier = False)
          Type: PrimitiveType (Keyword = string, KnownTypeCode = String)
        ⊟ ArraySpecifier: ArraySpecifier
            [: CSharpTokenNode
            ]: CSharpTokenNode
        Identifier: Identifier (Name = args, IsVerbatim = False)
      ): CSharpTokenNode
    ⊟ Body: BlockStatement
        {: CSharpTokenNode
      ⊟ Statement: ExpressionStatement
        ⊟ Expression: InvocationExpression
          ⊟ Target: MemberReferenceExpression (MemberName = WriteLine)
            ⊟ Target: IdentifierExpression (Identifier = Console)
                Identifier: Identifier (Name = Console, IsVerbatim = False)
              .: CSharpTokenNode
              Identifier: Identifier (Name = WriteLine, IsVerbatim = False)
            (: CSharpTokenNode
            Argument: PrimitiveExpression (LiteralValue = "Hello, World")
            ): CSharpTokenNode
          ;: CSharpTokenNode
        }: CSharpTokenNode
    }: CSharpTokenNode
```

Fig. 6 Syntax tree example

*2) Semantic Tree:* The semantic tree gives the information with respect to these attributes. The semantic tree constructed from the above syntax tree shown in Fig.7.



```
Resolve() = MemberResolveResult
  TargetResult = null
  Member = [Method Test.Main(args:System.String[]):System.Void]
  IsVirtualCall = False
  Type = System.Void
```

Fig .7 Semantic tree example

## C. Software Model Graph

The constructed software model graph represents the high - level view of software architecture as a simple directed and labeled graph (G). 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).

**Graph:** Let G = (N, E, $L_n$, $L_e$, n, e) be a directed software model graph, where N is a set of vertices, $E \subseteq N \times N$ is a set of edges, $L_n$ is a set of labels for the vertices, $L_e$ is a set of labels for the edges, n: $N \rightarrow L_n$ is a function that assigns a label to the vertices, e: $E \rightarrow L_e$ is a function that assigns a label to the edges.

Relation types in the software model graph are based on UML-like [27] 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 I LABELS OF NODES AND EDGES IN SMG

| **Node Label** | **Node Type** |
|---|---|
| C | Class |
| I | Interface |
| A | Abstract class |
| **Edge Label** | **Relation Type (Edges are directed from A to B)** |
| X | source class extends target class |
| I | source class implements target class |
| A | source class has field type of target class |
| T | source class uses target class in generic type declaration |
| L | source class method has a local parameter of target class |
| P | source class uses target class in its methods parameter |
| R | source class has methods has been return type of target class |
| M | source class has method call to target class |
| F | source class access field of target class |
| C | source class creates target class |
| O | source class overrides methods of Class B |

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 $\{M\} \cup \{P\} = \{M,P\}$. In our approach for detecting identical design-level clones, the edges are compared during their set of the labels in such a way that, when comparing two non- empty edge label sets, '$L_u$' and '$L_n$' are considered to be equal if and only if '$L_u$' $\subseteq$ '$L_n$' and '$L_n$' $\subseteq$ '$L_u$'. Fig. 1 demonstrates the UML class diagram of an observer design pattern example, and Fig.8 represents the related software graph model that we constructed. Fig.8 shows that the software model graph includes additional information compared to the UML class diagram, such as the "type field" (A) , "method call" (M), "override" (O), "methods parameter" (P), and "extend" (X) relations.

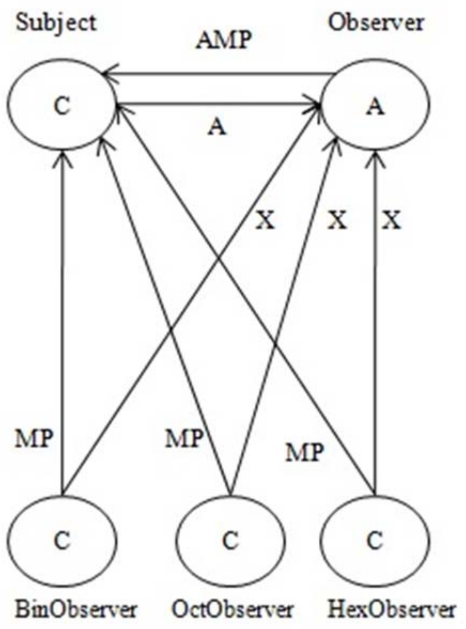


Fig. 8 Software model graph of example observer pattern

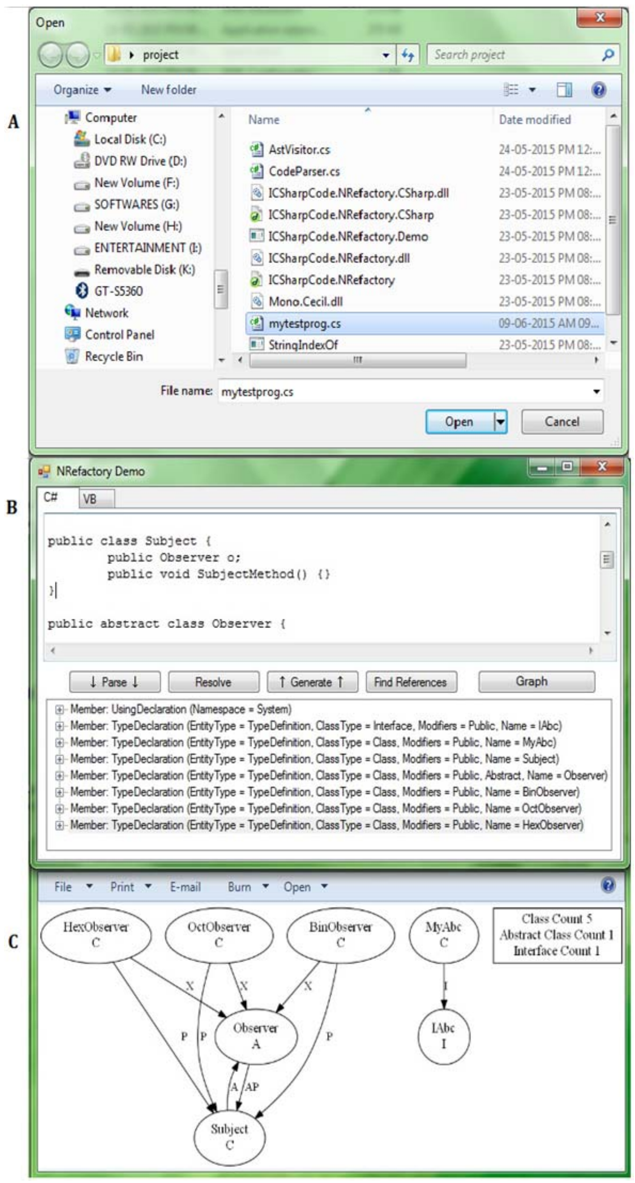
## D. Degree of Coupling

The degree of coupling [9] is calculated as the ratio of number of message received to the number of message sending. The degree of coupling is given in equation 1. Where DC is degree of coupling, MRC is message received coupling and MSC is message sender coupling. The MRC measures the complexity of the message received by the classes, as MRC is the number of messages received by a class from the other classes. The MSC is the number of message sender coupling among the objects of the classes; it is low level coupling that is achieved through the communication between the components.

$$\text{Degree of Coupling (DC)} = \frac{\text{MRC}}{\text{MSC}} \qquad (1)$$

## IV. RESULTS AND DISCUSSION

The results are analysed from sample source code. The outcome of software model graph is shown in Fig.9 which consists of 7 classes, one abstract class, one interface class and five normal classes. The directed edge weight is referred as communication message between the various classes (nodes/ vertices) as described in Table I. Fig.10 show common design structures that are identified manually from Fig. 9(C). Fig.11a) gives information about BinObserver class. Here BinObserver class is communicating to subject class with two different message, one is method (M) other is parameter (P) and to observer class with Extend (X). It performed coupling is MSC. Similarly rest of the classes shown in Fig. 11 (b) (c) (d) (e). In Fig. 11(d) communication of MSC there is multiple labels (AMP) on edge which refers three different communications are performing. Table II describes about the total numbers of MSC and MRC of various classes and their degree of coupling.



| Name | Date modified |
|---|---|
| AstVisitor.cs | 24-05-2015 PM 12:... |
| CodeParser.cs | 24-05-2015 PM 12:... |
| ICSharpCode.NRefactory.CSharp.dll | 23-05-2015 PM 08:... |
| ICSharpCode.NRefactory.CSharp | 23-05-2015 PM 08:... |
| ICSharpCode.NRefactory.Demo | 23-05-2015 PM 08:... |
| ICSharpCode.NRefactory.dll | 23-05-2015 PM 08:... |
| ICSharpCode.NRefactory | 23-05-2015 PM 08:... |
| Mono.Cecil.dll | 23-05-2015 PM 08:... |
| mytestprog.cs | 09-06-2015 AM 09... |
| StringIndexOf | 23-05-2015 PM 08:... |

File name: mytestprog.cs

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

public abstract class Observer {
```

```
Member: UsingDeclaration (Namespace = System)
Member: TypeDeclaration (EntityType = TypeDefinition, ClassType = Interface, Modifiers = Public, Name = IAbc)
Member: TypeDeclaration (EntityType = TypeDefinition, ClassType = Class, Modifiers = Public, Name = MyAbc)
Member: TypeDeclaration (EntityType = TypeDefinition, ClassType = Class, Modifiers = Public, Name = Subject)
Member: TypeDeclaration (EntityType = TypeDefinition, ClassType = Class, Modifiers = Public, Abstract, Name = Observer)
Member: TypeDeclaration (EntityType = TypeDefinition, ClassType = Class, Modifiers = Public, Name = BinObserver)
Member: TypeDeclaration (EntityType = TypeDefinition, ClassType = Class, Modifiers = Public, Name = OctObserver)
Member: TypeDeclaration (EntityType = TypeDefinition, ClassType = Class, Modifiers = Public, Name = HexObserver)
```

Class Count 5
Abstract Class Count 1
Interface Count 1

Fig. 9(A) Selection of C# source file (B) Abstract Syntax tree, Semantic (C) Software model graph

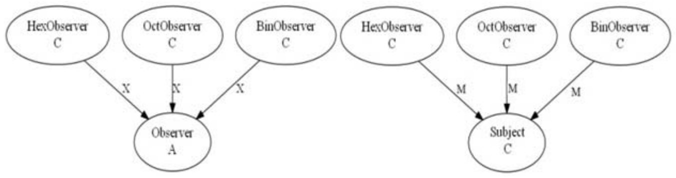


Fig. 10 Common design structures

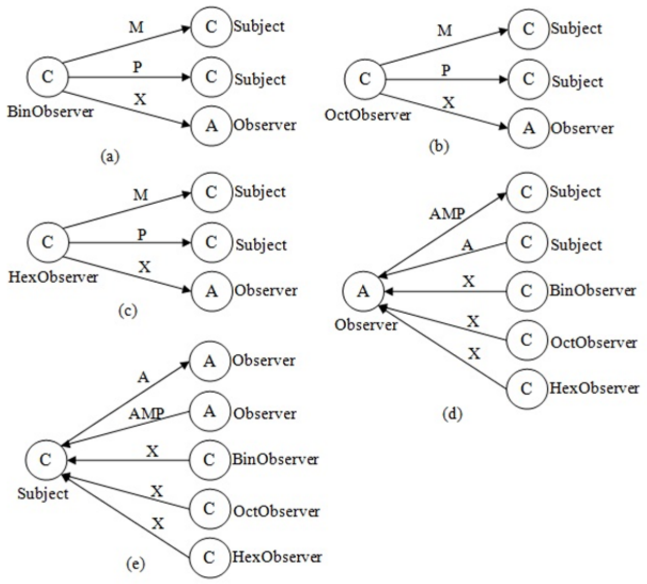


Fig. 11 Degree of Coupling

TABLE III

COUPLING METRICS

| Class | Object Oriented | | |
|---|---|---|---|
| | MSC | MRC | DC |
| BinObserver | 3 | 0 | 0/3 |
| OctObserver | 3 | 0 | 0/3 |
| HexObserver | 3 | 0 | 0/3 |
| Observer | 3 | 4 | 4/3= 1.3 |
| Subject | 3 | 4 | 4/3= 1.3 |

## V. CONCLUSION AND FUTURE WORK

In this paper an approach is proposed to generate a software model graph and to analyse the source code using abstract syntax tree method. In suggested approach, the solution file for the C# application is taken as an input to the system. It loads the file into memory and reads one by one to load all the source code in order to construct the syntax tree. Once the creation of abstract syntax tree is completed, it is ready for analysis. The Nfactor library [10] is utilized to generate syntax tree. Generated syntax tree is used for finding refactoring of similar source code and for finding a patterns design. The construction of software model graph provides in-depth information about a system. Based on SMG, common design structures, coupling and substructures can be found in result section. These structured graphs can help develop in understanding the architecture of the object oriented system.

The future enhancement for this work is to apply graph partitioning technique on software model graph to identify specific-domain structures, commonly used design structures; copy-past activity and design patterns of object oriented systems. These structures can assist for software developers to improve the quality and design of the software.

## REFERENCES

[1] Z. Li, S. Lu, S. Myagmar, Y. Zhou, "CP-Miner: Finding copy–paste and related bugs in large scale software code," *IEEE Trans. Softw. Eng*, vol. 32, pp. 172-192, 2006.

[2] P. Gandhi and P. K. Bhatia, "Optimization of Object- Oriented Design Using Coupling Metrics," *International Journal of Computer Applications*, Vol. 27(10), 2011, pp. 41-44.

[3] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*, Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1999.

[4] Vipin Saxena and Santosh Kumar, "Impact of Coupling and Cohesion in Object-Oriented Technology," *Journal of Software Engineering and Applications,* vol. 5, pp. 671-676, 2012.

[5] Jeffrey L. Overbey and Ralph E. Johnson, "Generating Rewritable Abstract Syntax Trees," *Software Language Engineering, Lecture Notes in Computer Scienc*. Springer- Verlag Berlin Heidelberg, vol. 5452, pp 114-133, 2009.

[6] Ira D. Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier, "Clone Detection Using Abstract Syntax Trees," in *Proceedings of the International Conference on Software Maintenance*, 1998, IEEE Computer Society, Washington, DC, USA , p. 368.

[7] Pavitdeep Singh, Satwinder Singh, and Jatinder Kaur, "Tool for generating code metrics for C# source code using abstract syntax tree technique." *SIGSOFT Softw. Eng*. Notes 38, vol. 5, p.1-6, August 2013. DOI=10.1145/2507288.2507312 http://doi.acm.org/10.1145/2507288.2507312

[8] Harjot Singh Virdi and Balraj Singh, "Study of the Different Types of Coupling Present in the Software Code," *International Journal of computer Science and Information Technology,* vol. 3(3), pp. 4153-4156, 2012.

[9] Jehad Al Dallal, "Identifying refactoring opportunities in object-oriented code: A systematic literature review," *Information and software Technology,* vol. 58, pp 231-249, 2015.

[10] Daniel Grunwald, Using NRefactory for analyzing C# code (http://www.codeproject.com/Articles/408663/Using-NRefactory-for-analyzing-Csharp-code )