Design and development of dependency analysis tool (DA-OOP) for an object oriented programme

Ratneshwer*

Department of Computer Science, MMV,
Banaras Hindu University,
Uttar Pradesh 221005, India
Email: ratnesh@bhu.ac.in
*Corresponding author

Guru Prasad Bhandari and
Kul Bahadur Chhetri

Department of Computer Science,
Banaras Hindu University,
Uttar Pradesh 221005, India
Email: Guru.bhandari@gmail.com
Email: kb.chhetri123@gmail.com

Abstract: This work presents 'design and development of a dependency analysis tool (DA-OOP) for an object oriented programme'. The proposed tool is capable of supporting generation of different dependency views of an object oriented programme. Dependency analysis of conventional software use traditional techniques of programme dependency representation. As far as OOP software is concerned, its specific features like classes, objects, inheritance relationships, encapsulation, polymorphism, overloading etc. should also be considered. An object oriented programme may observe dependencies among namespaces, classes, functions and variables. The main contribution of this work is to develop a dependency analysis tool for an object-oriented programme that will extract all possible dependencies of an OOP programme. The proposed tool 'DA-OOP' depicts the dependency information in form of text view, matrix view and graph view of an object oriented programme. The outcomes of the tool may be efficiently utilised in testing and maintenance of an object oriented programme.

Keywords: object oriented programme; dependency analysis; class graph; inter-class dependency; intra-class dependency.

Reference to this paper should be made as follows: Ratneshwer, Bhandari, G.P. and Chhetri, K.B. (2015) 'Design and development of dependency analysis tool (DA-OOP) for an object oriented programme', *Int. J. Software Engineering, Technology and Applications*, Vol. 1, No. 1, pp.102–117.

Biographical notes: Ratneshwer is serving as an Assistant Professor in Department of Computer Science (MMV), Banaras Hindu University, India. He has received his PhD in Computer Engineering from Department of Computer Engineering, IIT-BHU Varanasi. He is actively involved in teaching and research for last eight years. His active research area is software engineering especially component-based software engineering.

Guru Prasad Bhandari is a student of Master of Computer Application course at Department of Computer Science, Banaras Hindu University, India. His research interest is software engineering.

Kul Bahadur Chhetri is a student of Master of Computer Application course at Department of Computer Science, Banaras Hindu University, India. His research interest is software engineering.

1 Introduction

A software dependency analysis tool enhances the understandability of software by providing various dependency relationships. These extracted dependency information can be used in the testing and maintenance of the software and also in the software reengineering. Researchers have proposed several dependency analysis tools (Kittilä, 2008; Stafford et al., 1997; Richardson et al., 1992; Najumudheen et al., 2010) for an object oriented programme. Considerable amount of work have been done in the area of representing programmes into dependency graphs but it requires further extension on already existing techniques. Most of the research efforts focus on the expression-oriented or statement-oriented dependency. It is very cumbersome task to observe the dependency, of an object oriented programme, *statement by statement* as the size of graph would be very large and it would be difficult to analyse such information. This problem would be more significant in case of large size programmes. An object-oriented system consists of a number of messages between its constituent objects, so that dependency representation should be capable of depicting inter-procedural dependencies, along with the other concepts of the paradigm namely, polymorphism, dynamic binding, classes and objects and inheritance (Malloy et al., 1994). Effective dependence analysis of an OOP requires a different approach. It may be a feasible idea to consider class-oriented and function-oriented dependency for better system understandability. Such information may be useful to understand existing programmes, written in object-oriented programming languages, in a concise manner.

A *DA-OOP* (dependency analysis for an object-oriented programme) tool has been developed that captures the various dependencies in an object oriented programme. The tool 'DA-OOP' extract information from the implementation details (source-code) of an object oriented programme. This tool has dependency representation in three forms i.e., text view, graph view and matrix view, if it is applicable. Different concepts of the object oriented paradigm are represented, including abstraction, polymorphism and class inheritance. Message exchange among objects also taken into consideration and dependencies are represented in a very concise manner. A demonstration of the tool is given here that take an object oriented programme (developed in C#) as input and extract various types of dependency information in text, graph and matrix view. The division of the output representations into different views increases the clarity and gives the user an opportunity to select the required representation that actually requires for the purpose. In this paper, we have limit our discussion only for dependency analysis of object oriented programmes and did not include component based and service based software.

The rest of this paper is organised as follows:

Section 2 consist the related work, in brief, in the field of object oriented dependency analysis tool development. Section 3 introduces types of dependency in an object oriented programme. Design description of DA-OOP tool has been given in Section 4. Implementation details are described in Section 5. Various user interfaces are given in Section 6. Section 7 concludes the work.

2 Related work

In this section, a brief review of existing efforts regarding the tool development for dependency extraction especially in context of an object oriented programme is given. In literature several efforts have been mentioned. Several commercial tools are available to analyse object oriented programmes. These efforts are summarised here.

NDepend (2014) is a static analysis tool that supports architecture-level dependency analysis to get clear idea about cross-dependencies between objects, level of association between them. NDepend also supports code metrics and dependency matrix, declarative code rule and comparison of two versions. Linos and Courtois (1994) designed and implemented a software tool for understanding and re-engineering C++ programmes called *OO!CARE* (object-oriented computer-aided re-engineering) tool written in C a++. This tool considers the programme dependency, polymorphism dependency, message-pass dependency and implicit programme dependency. This model includes hierarchical displays for presenting class inheritance, control-flow programme dependencies and file dependencies. ProDAG (Richardson et al., 1992) is an implementation-level dependence analysis tool for Ada and C++ programmes. PRoDAG is an analysis toolset that provides an application programmatic interface for programme dependence analysis, direct and indirect data dependence, direct and indirect strong control dependence, direct and indirect weak control dependence, strong syntactic dependence and weak syntactic dependence. Stafford et al. (1997) developed an architecture level dependence analysis technique, called *chaining* and implementing the technique in a tool called *Aladdin.* In *Aladdin*, the representation consists of a set of cells, where each cell represents the set of relationships that could exist between a given pair of architectural elements. This set is queried in order to construct chains of dependent elements. Najumudheen et al. (2010) proposed a dependence-based representation to test coverage analysis of object-oriented programmes named *call-based object-oriented system dependence graph (COSDG)*. COSDG represents dependence details, call graph details and inheritance details, which includes control dependence, data dependence and membership dependence. Ferrante et al. (1987) presented programme dependence graph (PDG) that creates the data and control dependency explicitly where data dependency represents the data flow relationships and control dependency represents control flow relationships of a programme. PDG helps to vectorisation, node splitting, code motion and loop fusion that permits incremental optimisation by optimising transformations in compiler optimisation. Linos (1995) presented a tool called PolyCARE (polyparadigmatic computer-aided re-engineering), supports programmes written in multiple programming languages (i.e., both object-oriented and procedural languages). It maintains a repository with control and data flow programme dependencies. Visualisation of such dependencies, transformation tools between different representations and graphical abstraction techniques are the main features of PolyCARE. The user interface is

done through specially designed windows. Each window in PolyCARE is equipped with a group of typical operations for manipulating graphs or text.

It can be observed that various tools are available for dependency analysis for an object oriented programme. The present work extends the above contributions further by adding some newer features. In this present work we have categorised the dependency information in two ways that is micro view and macro view. Micro view shows the relationship between objects where as macro view clearly displays which object function calling which other object functions. Similarly it also displays inter object dependency and intra object dependency. The main purpose of *DA-OOP* is to help maintainer to visualise the dependency of block of code from different prospective that could allow easy debugging of the programme and code refactoring.

3 Dependency in an object oriented programme

An object-oriented system is composed of a collection of communicating objects that cooperate with one another to achieve some desired goals. Similar objects form classes, which provide the static description of the properties and behaviours that their instances will have. Therefore, extracting, analysing and modelling classes/objects and their relationships is of key importance in acquiring in-depth understanding of object-oriented software systems (Dong and Godfrey, 2007). Various types of dependencies, in an object oriented programme, are proposed in the literature. Here we have briefly mentioned some dependency types that we have deal within our work.

3.1 Class dependencies

Class dependencies are dependencies among classes due to inheritance. Class dependency can either be implicit or explicit on the basis of either inheriting from the inbuilt classes or user-defined type. This dependency represents super class/parent class to sub-class/child class relationship in inheritance hierarchy.

3.2 Interface dependencies

Interface dependencies are dependencies among interfaces and classes. While one class uses interface or interface uses another interface then there would be interface dependency. Class A cannot carry out its work without some implementation of interface I. Therefore, whenever a class depends on an interface, that class also depends on an implementation (Ross, 2014).

3.3 Method/functional dependencies

Method or field dependencies are dependencies on concrete methods or fields of an object. It does not matter what the class of the object is, or what interfaces it implements, as long as it has a method or field of the required type.

3.4 Compile-time and runtime dependencies

A dependency that can be resolved at compile time is a compile-time dependency. A dependency that cannot be resolved until runtime is a runtime dependency. Compile-time dependencies tend to be easier for developers to see than runtime dependencies, but sometimes runtime dependencies can be more flexible (Ross, 2014).

3.5 Interclass dependency

One class calls an object of another class to use its properties if its properties are defined as public access qualifier. Inter-class relationship presents class to class reference relationship. Inter-class dependency can be viewed into two aspects: micro view and macro view. Object declaration of one class may occur in every other class hence micro view show this relationship. Macro view describes relationships of one class to each and every function of other classes if this class calls the function declared in another class.

3.6 Intra-class dependency

There may be relationship between a function of a class to another function of the same class. If a function calls to another function defined at the same class then intra-class dependency occurs. Intra-class dependency graph does not show any relationship among classes.

3.7 Control and data dependency

Control dependence is a situation in which a programme's instruction executes if the previous instruction evaluates in a way that allows its execution. A statement S2 is control dependent on S1 if and only if S2's execution is conditionally guarded by S1 (Beck and Cunningham, 1989). Data dependence arises from two statements which access or modify the same resource. A statement S2 is flow dependent on S1 if and only if S1 modifies a resource that S2 reads and S1 precedes S2 in execution (Cunningham, 1999). Data dependency concerns with the dependency among data values by checking its output parameters and returned value. Programme dependency is the composite form of both control dependency and data dependency.

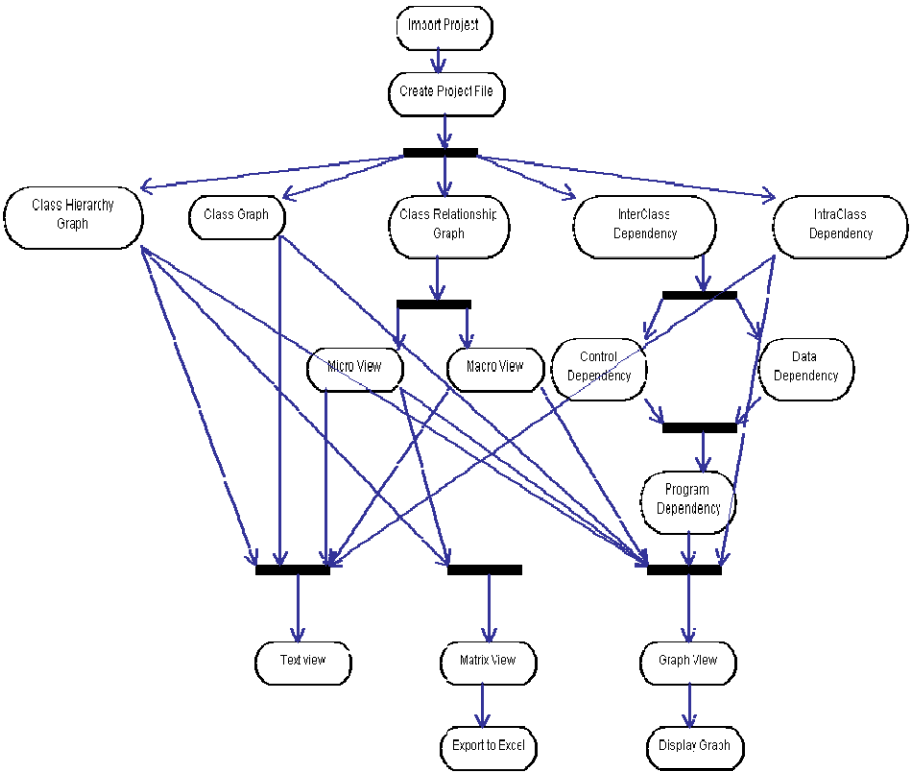
4 Design of the proposed tool *'DA-OOP'*

The developed tool '*DA-OOP*' takes an object oriented programme as input and generates following dependency representations: class hierarchy graph (CHG), class to function relationship graph, class relationship (micro view and macro view), control dependency graph, data dependency graph and fan-in and fan-out graph of an object. A design description of the tool '*DA-OOP*' has been shown here. Unified modelling language (UML) has been used for making design of different activities in the system.

4.1 Activity diagram

Figure 1 shows an activity diagram for '*DA-OOP*'. First one has to import the project source code for which dependency information has to be generated. Then project source code converted into a single project file. Then the tool generate various dependency information such as CHG, class graph, class relationship graph, interclass dependency graph and intra-class dependency graph. Two aspects are displayed as – micro view and macro view of class relationship graph after class relationship graph. There is a facility that the generated results can be imported into an excel file for storage purpose.

Figure 1 Activity diagram of the tool 'DA-OOP' (see online version for colours)

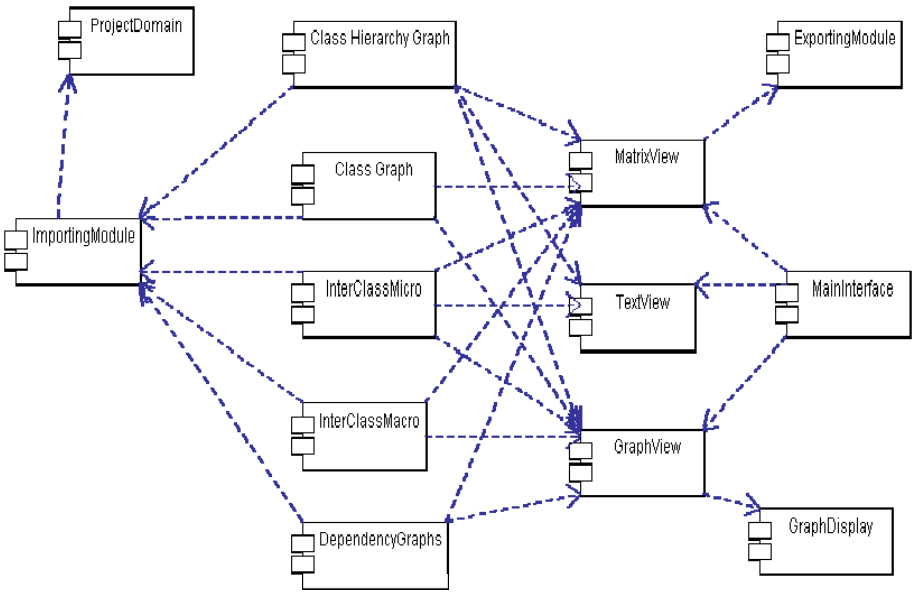


4.2 Component diagram

This view provides an opportunity to map classes onto implementation components and nodes (Rumbaugh et al., 1999). Figure 2 shows a component diagram of '*DA-OOP*'. To import the project *ProjectDomain* component has been used. *ImportingModule* gets imported project and create project file for that particular project. Various graph models are getting project file to analyse the project from the importing *Module* component and generating their output through *MatrixView, TextView* and *GraphView* components. *MatrixView* component is also linked with *ExportingModule* component to export the matrix version into excel and *GraphView* with *GraphDisplay* component to display graph

version of dependency as image form. All three components are using *mainInterface* component.

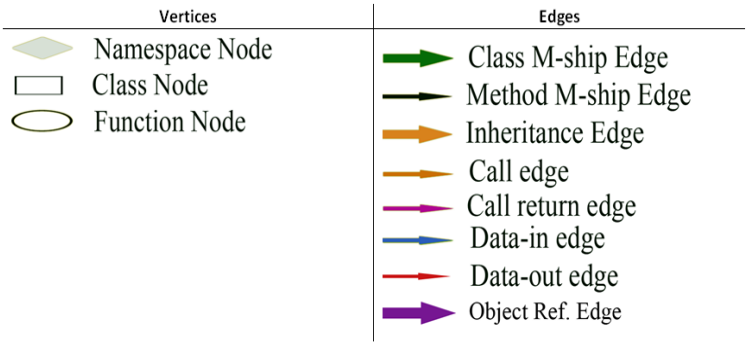
Figure 2 Component diagram of '*DA-OOP*' (see online version for colours)



5 Description of the tool '*DA-OOP*'

In OOP, different types of dependencies among block of code can occur if one block use the properties of another block or share data variables. In fact, testing, debugging, code impact analysis may be achieved with the help of different views of dependencies such as text view, matrix view and graph view. We have taken small sample code to illustrate the '*DA-OOP*' tool. This is just the example code but our tool is capable to handle large size C# projects also. In Figure 3, we have mentioned the notations used for generated dependency graphs.

Figure 3 Notations used for graphical representations (see online version for colours)



We have demonstrated the features of '*DA-OOP*' with pictorial representation of the graphs representing the sample code given as example. The graphical representation of different vertices and edges are shown in the Figure 3. Diamond, rectangle and oval shape are used for namespace node, class node and function node respectively. Different edges are representing different type of relationships on the basis of their colour and emphasise. Below is the sample code that we have used for the purpose of demonstration.

Figure 4 Sample programme written in C#

```
using System;
namespace sampleProgram
{
   public class parent
   {
      public void methodOne()
      {
         //body of this method
      }
      public void methodTwo()
      {
         //body of this method
      }
   }
   public class childA : parent
   {
      public int methodAa(int a, int b)
      {
         int sum;
         sum = a + b;
         return sum;
      }
      public void methodAb(int sm)
      {
         System.Console.Out("result=" +
sum);
      }
   }
public class ChildB : parent
   {
      public void methodBa(int r)
      {
         System.Console.Out("result=" +
r);
      }
      public int methodBb(int a,int b,int
c)
      {
         int result = a * b * c;
         methodBa(retsult);
           return result;
      }
   }
   public class mainClass
   {
      public void main()
      {
         childA objA = new childA();
         int s = objA.methodAa(3,5);
         objA.methodAb(s);
         childB objC = new childB();
         objC.methodBb();
      }
   }
}
```

5.1 CHG (class hierarchy graph)

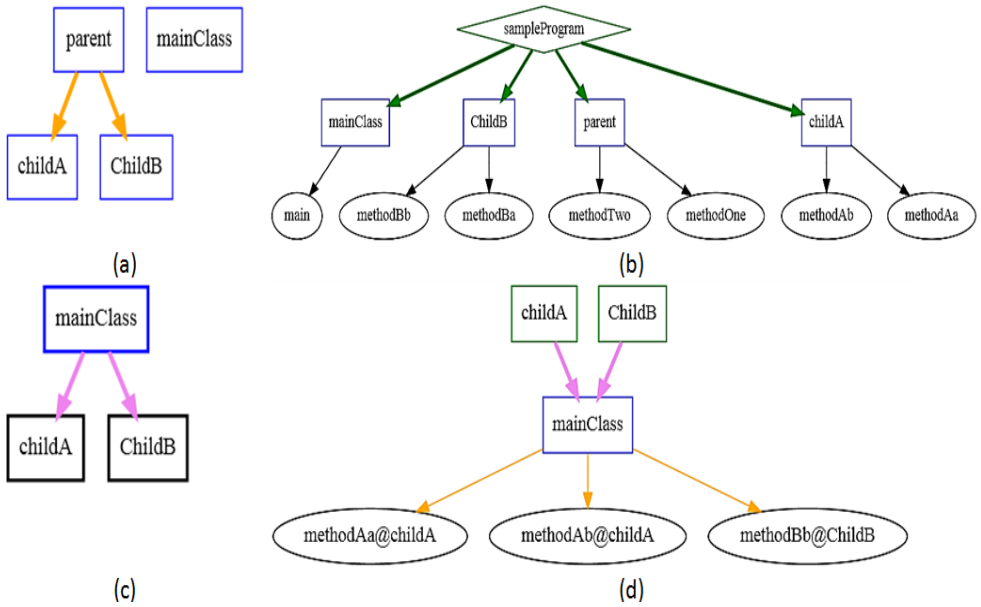
It presents super class/parent class to sub-class/child class relationships and by checking whether the class is being inherited by other class or not. It has three views – text view, matrix view and graph view. Text view just shows super to sub-class relationship by arrow pointer like as child A → parent ; means child A is inherited by parent class parent. In matrix view, we have mentioned M*M matrix to hold the relationship of each and every class to any other classes. If a class inherits any other class, then matrix entry is represented by '1', where row represents the child class and column represents the parent class and 0 means no inheritance relationship. Graph-Viz tool (Graphviz, 2014) has been

used to display the graph view into browser. Inheritance edge goes from super class to child class e.g., child A and child B classes are inheriting parent class hence inheritance edge is pointing to child A and child B classes coming from parent.

5.2 Class graph

Class call graph shows graph of all the namespaces and classes with their corresponding methods. It has two views – text view and graph view. It depicts all the namespace, classes and their corresponding functions by using class membership edges and method membership edges among namespace node, class node and method node. Figure 5(b) demonstrates the class graph of sample code given in Figure 4.

Figure 5 Several graph view of the sample code, (a) CHG (b) class graph (c) class relationship micro view (d) class relationship macro view (see online version for colours)



5.3 Class relationships (micro view)

Class relationships are of two types, one is micro view which is collapsed view and another is macro view which is expanded form. Micro view depends on whether a class has created an object of another class for further use or not. Macro view checks which class is creating an object of another class and as well as which function has been called.

Class micro view presents relationship among classes on the basis of whether the class creates an object of another class or not. This relationship has also three views – text view, matrix view and graph view. *Calling class – called class → object created* is the format of text view of class relationships micro view. For example *mainClass –childA → objchildA* where *mainClass* creates an object *objchild A* of *child A*. Matrix view has m*m matrix where m is the number of all the classes in the project. Cell value '1' means row class is creating an object of column class. Object reference edge

has been used to show relationship between calling class to called class. Class nodes and inheritance edges are used to demonstrate class micro view in Figure 5(c) where inheritance edge points to *child A* and *child B* from *mainclass* that means *mainClass* has created an object of that two classes.

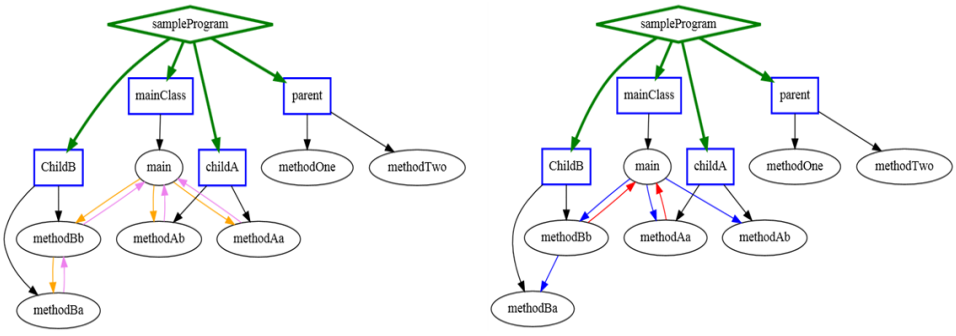
5.4 Class relationship (macro view) graph

If one class creates an object of another class and uses its behaviour of that class then only class relationship macro view exists. This feature has two views text and graph view. Figure 5(d) depicts class relationship macro view graph of sample code shown in Figure 4. For example *method a* of *childA* class has been used in *mainClass* by creating an object of *childA*.

5.5 Control dependency graph

Control dependency analysis or control-flow analysis describes the flow of code execution. Control enters into the block through the first statement as entry point and remains there until the last statement does not complete its execution. This has been shown with the help of different edges and nodes. Figure 6(a) exhibits the control-flow of the object oriented programme given in Figure 4 that includes all the namespaces, classes and functions with different edges – class membership edge from namespace to class, function membership edge from class to function. Orange colour edge shows entry-control whereas violet colour edge shows exit-control from the block. For e.g., *main()*defined inside *mainClass* class, is calling *methodAa()* defined inside *childA()* hence shown by *entry-control edge*, is going from *main()* and returning back by violet line from *medhodAa()* to *main()*that means control returns back from *methodAa()* to *main()* after completion of the last statement of *methodAa()*.

Figure 6 Demonstration of (a) control dependency and (b) data dependency (see online version for colours)



(a) (b)

5.6 Data dependency graph

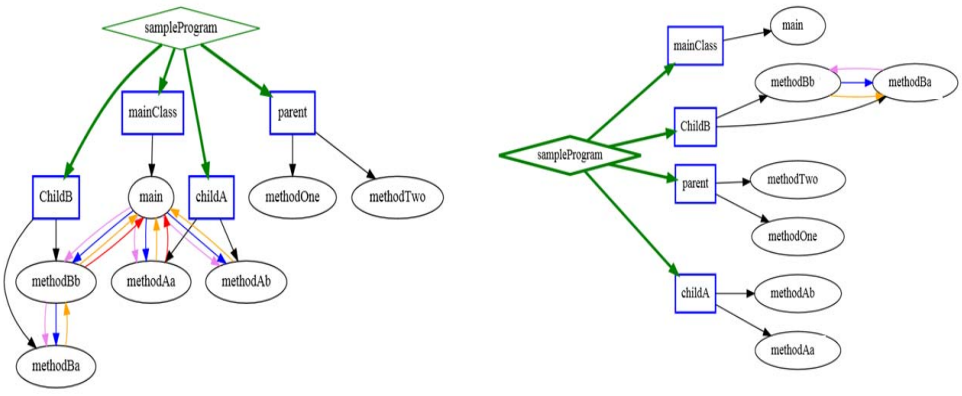
The simple techniques here that has been implemented to check the data dependency between calling function to called function is just by checking the input parameters of the

function and output parameter or checking whether the called function returns value or not. If called function gets the values from calling function as its input parameter(s), we are sure that called function is data dependent on calling function and if called function return value back to the calling function then calling function is data dependent on called function because here calling function is waiting for the data from called function. Data dependency has text view and graph view. Figure 6(b) demonstrates the data dependency graph of the sample code shown in Figure 4 where blue colour arrow shows the data-in edge from calling function to called function and red colour arrow shows the data-out edge that means called function returns the data value after execution of statements inside it to the calling function. For e.g., *main()* of *mainClass* is calling function *methodAa()* of *childA*. *main()* gets values from *methodAa()* hence it is shown by blue colour. And at last *methodAa()* sends the return value or data to the *main()* shown by red colour arrow or *data-out edge.*

5.7 Programme dependency graph

DA-OOP is capable to exhibits the text view and graph view of programme dependency. Different edges and nodes are subjected to their special meaning shown in Figure 3. Figure 7(a) exhibits the programme dependency graph of the sample code shown in Figure 4 where blue colour arrow shows the *data-in edge* from calling function to called function and red colour arrow shows the *data-out edge* that means called function returns the data value after execution of to the calling function. Programme dependency graph includes all the namespaces, classes and functions with different edges – *class membership edge* from *namespace node* to *class node, function membership edge* from class to function. Orange colour edge shows *entry-control* whereas violet colour edge shows *exit-control* from the block For e.g., *main()* of *mainClass* is calling function *methodAa()* of *childA*. *main()* gets values from *methodAa()* hence it is shown by blue colour edge. And at last *methodAa()* sends the return value or data to the *main()* shown by red colour arrow or *data-out edge* and also shown control dependency by *entry-control edge,* is going from *main()* and returning back by violet line from *medhodAa()* to *main()* that means control returns after last instruction.

Figure 7 Demonstration of (a) programme dependency graph and (b) intra class dependency graph (see online version for colours)



(a) (b)

5.8 Intra-class dependency graph

Dependency among functions within a class is shown in intra-class dependency graph. It has also text view and graph view. Functions of one class are only connected to the functions of the same class if one function invokes another function. In Figure 7(b), dependency among functions within a class has been shown in intra-class dependency graph of the sample code given in Figure 4. While a function of class calls to another function of the same class then there would be intra-class dependency which is shown by the *call edges* and *call-return edges*. For example: *methodBb* of *childB* is calling shown by *call edge* to *methodBa* inside the same class and returning back from *methodBa* to *methodBb* after last statement execution.

5.9 History record

DA-OOP creates the history record of already opened project as repository. It maintains file of the analysed project as a copy of that project in text form. In the future, maintainer or tester will be able to check the dependency from the history project list even he lost the project source code.

6 User Interface and output

In this section, the main interface of '*DA-OOP*' is presented. Windows Form design has been used to design the interface and the outputs are shown in the browser. Forms have button controls to perform the desired operation on clicking event. Label controls are used to support the user to understand the output. The concept has been implemented using C#.Net 2012, Graph-Viz tool, Internet Explorer and MS-DOS. Regular expression and file handling have been used in the project. User has to import the source code from importing form before analysing the code.

6.1 Main user Interface

Figure 8 shows the main user interface, includes all the forms and controls inside it. Project title is shown in the title bar.

Menu bar is located just below the title bar that includes all the options of the system but additional options are shown in left side of this interface. Notations are given in the centre of the interface left to work-flow picture. Various relationships are listed on the left of the interface with their corresponding options on button controls. Icon menu bar holds the icons of link of main features of the system to make the system more users friendly.

User are also allowed to select project from the list of already analysed or opened project in this *DA-OOP* from the second tab 'select from collection' tab option which is next to 'import new project'. Project title will be shown in the centre after selecting the project package and list of files which are included in that project will also be shown as list in the centre list box control. User can check whether the project uploaded is that he wants to analyse or not by clicking button load. Figure 9 shows project import interface of DA-OOP.

Figure 8 Main interface of '*DA-OOP*' (see online version for colours)

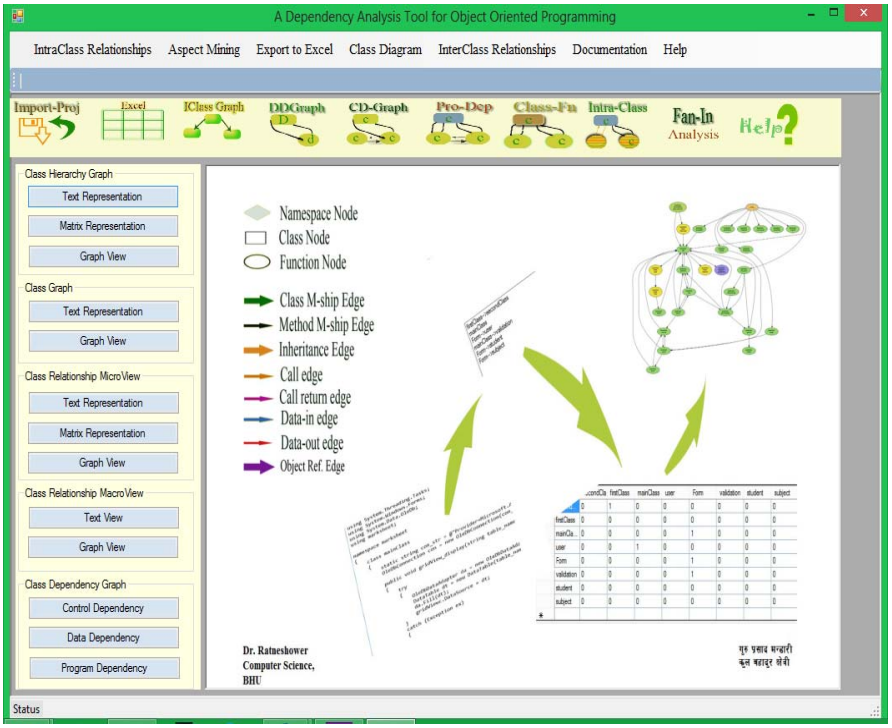


Figure 9 Project importing form of DA-OOP (see online version for colours)

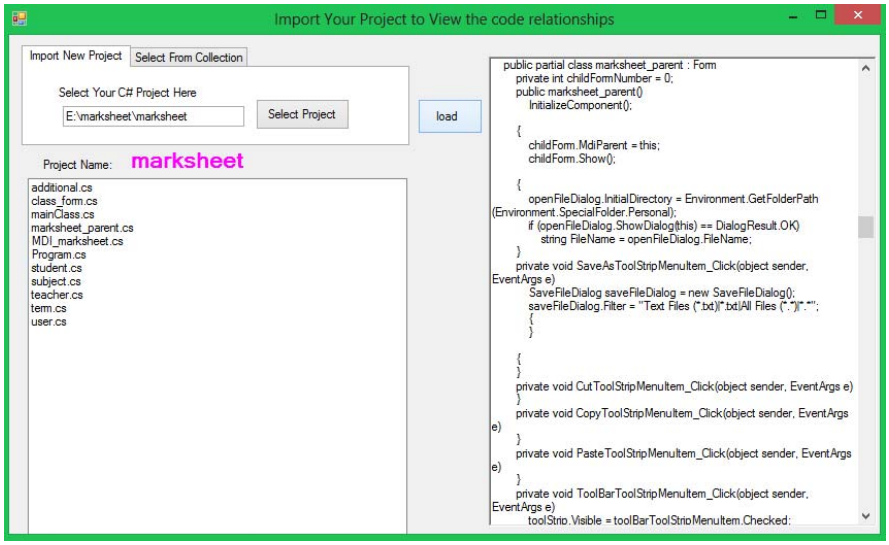


Figure 10 Sample interface of text and matrix view of CHG of DA-OOP (see online version for colours)

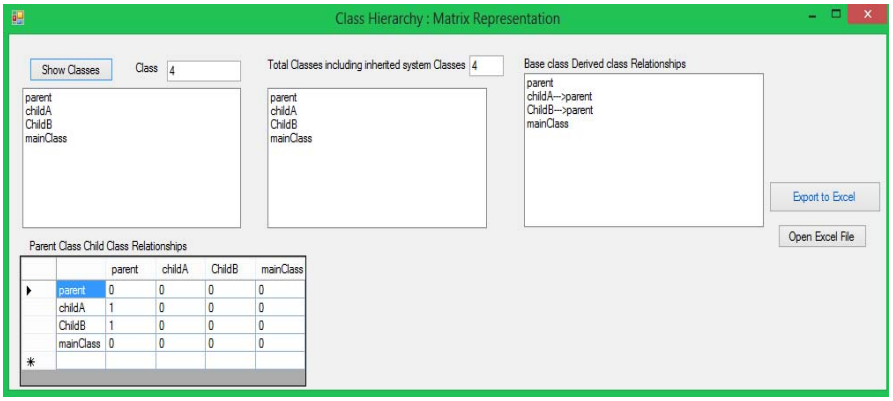
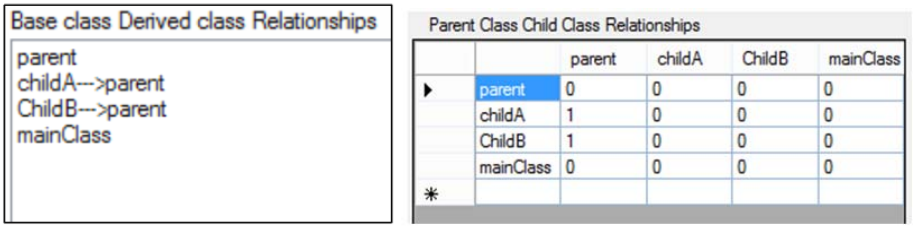
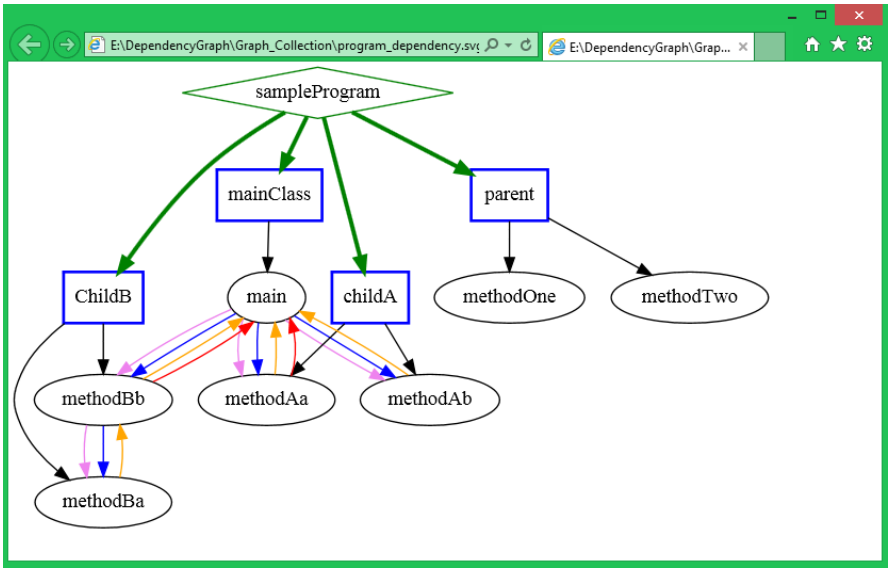


Figure 11 (a) Text representation and (b) Matrix representation of CHG of DA-OOP (see online version for colours)



(a) (b)

Figure 12 Graph view of programme dependency by DA-OOP (see online version for colours)



At least two forms are presented for representation of one graph. Here we have presented only text, matrix and graph view of class hierarchy relationships as example. Figure 9 shows sample text view and matrix view form of DA-OOP. Figure 10(a) shows text view and 10(b) shows matrix view of CHG of DA-OOP separately. Figure 11 presents the graph view of dependency graph which is shown in browser using Graph-Viz tool.

7 Conclusions

A tool named as 'DA-OOP' has been designed and developed which will eventually be useful to analyse the dependency among code blocks for object oriented programme maintainer. All project parameters such as *CHG, class graph, class relationship micro view and macro view, Control dependency, data dependency, programme or system dependency, interclass relationships* and *intra class relationships* with user friendly interface are shown precisely in a tool. The hierarchical success tools for identification of dependency are developed but *DA-OOP* has tried to include all the possible types of dependencies of an object oriented programme.

Further supportive research is required to analyse of various object oriented features and uncertainties. In future, we will try to include all possible dependencies of component based programming, service-oriented architecture and abstract-oriented architecture as well. We will try to make this tool interoperable for any programming language and portable for any platform.

References

Beck, K. and Cunningham, W. (1989) 'A Laboratory for teaching object-oriented thinking', *Object-Oriented Programming Languages and Systems Conference Proceedings*, October, Also SIGPLAN Notices, Vol. 24, No. 10.

Cunningham, H.C. (1999) 'Object-oriented design and programming', *Lecture Notes, CSci 581-01: Special Topics in Computer Science Object-Oriented Design and Programming Spring Semester*, Department of Computer and Information Science The University of Mississippi.

Dong, X. and Godfrey, M. (2007) 'System-level usage dependency analysis of object-oriented systems', *Proceedings of 23rd IEEE International Conference on Software Maintenance (ICSM 2007)*, 2–5 October 2007, Paris, France, IEEE 2007, ICSM 2007, pp.375–384.

Ferrante, J., Ottenstein, K.J. and Warren, J.D. (1987) 'The program dependence graph and its use in optimization', *ACM Transactions on Programming Languages and Systems*, Vol. 9, No. 3, pp.319–349.

Graphviz (2014) *Graph Visualization Software*, Web Article [online] http://www.graphviz.org/ (accessed 1 April 2014).

Kittilä, K. (2008) *Analysing and Managing Software Dependencies with a Dependency Structure Matrix Tool*, Master Thesis, Department of Information Processing Science, University of Oulu.

Linos, P. (1995) 'PolyCARE: a tool for reengineering multi-language program interactions', *Proceedings of 1st IEEE International Conference on Engineering Complex Systems*, Ft. Lauderdale, FL, 6–11 November, pp.338–341.

Linos, P.K. and Courois, V. (1994) 'A tool for understanding object-oriented program dependencies', *Proceedings of the IEEE Third Workshop on Program Comprehension*, 14–15 November, Washington, DC., pp.20–27.

Malloy, B., McGregor, J.D., Krishnaswamy, A. and Medikonda, M. (1994) *An Extensible Program Representation for Object-Oriented Software*, Technical report, Clemson University.

Najumudheen, E.S.F., Mall, R. and Samanata, D. (2010) 'A dependence representation for coverage testing of object-oriented programs', *Journal of Object Technology*, Vol. 9, No. 4, pp.1–23.

NDepend (2014) Web Article [online] http://www.ndepend.com/Features.aspx (accessed 15 March 2014).

Richardson, D.J., O'Mally, T.O., Moore, C.T. and Aha, S.L. (1992) 'Developing and integrating ProDog in the Arcadia environment', *Proceedings of ACM SIFSOFT 92: Fifth Symposium on Software Development Environments*, Washington, D.C., December, pp.109–119.

Ross, A. (2014) *Understanding Dependencies*, Web Article [online] http://tutorials.jenkov.com/ood/understanding-dependencies.html (accessed 15 March 2014).

Rumbaugh, J., Jacobson, I. and Booch, G. (1999) *The Unified Modeling Language Reference Manual*, December 1998, Addison-Wesley, Massachusetts, USA.

Stafford, J., Richardson, D. and Wolf, A. (1997) *Aladdin: A Tool for Architecture-Level Dependence Analysis of Software Systems*, Technical Report CU-CS-858-98, Department of Computer Science, University of Colorado.