Copyright 2001 IEEE. Published in the Proceedings of IWPC2001, IEEE Computer Society Press, Los Alamitos, CA, pp. 293 - 299. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works, must be obtained from the IEEE. Contact: Manager, Copyrights and Permissions / IEEE Service Center / 445 Hoes Lane / P.O. Box 1331 / Piscataway, NJ 08855-1331, USA.

Hidden Dependencies in Program Comprehension and Change Propagation

Zhifeng Yu, Václav Rajlich
Department of Computer Science
Wayne State University
Detroit, MI 48202 USA
{zny,rajlich}@cs.wayne.edu

Abstract

Large software systems are difficult to understand and maintain. Program dependency analysis plays a key role in both understanding and maintenance. This paper discusses hidden dependencies among software components that make both understanding and maintenance hard. Hidden dependency is a relationship between two seemingly independent components and it is caused by a data flow inside of a third software components. The paper uses Abstract System Dependence Graphs to define hidden dependencies. It discusses the impact of hidden dependencies on the process of change propagation and also discusses an algorithm that warns about possible presence of hidden dependencies.

1. Introduction

Software change is the basic process of both software evolution and software servicing of the staged life cycle model [19]. Program comprehension is a prerequisite of change. After making a change in the software, the programmer is required to identify all the consequences of the change and reintroduce consistency into the software.

Many software maintenance tools depend on tracing dependencies within programs [22]. Program dependency analysis plays a fundamental role in program comprehension, change propagation, or impact analysis [2]. All software components have to be comprehended in their context, and the context is defined by component dependencies. Software maintainers need to trace those system dependencies and make corresponding changes to guarantee that change has been propagated correctly and the software is again consistent.

This paper introduces a notion of "hidden dependencies" among software components. Hidden dependencies contradict a popular opinion expressed in [9, p109], "if a class A is unaware of the existence of class B, it is also unconcerned about any change to B". We consider hidden dependencies to be design faults that violate this rule and we treat them as antipatterns [5] to be discovered and eliminated whenever practical. In order to discover hidden dependencies, we use the results of the standard program analysis.

An example of a hidden dependence is given in section 2. Section 3 defines the Abstract System Dependence Graph (ASDG) for object oriented programs. Section 4 defines hidden dependencies and analyzes their impact on program comprehension and change propagation. It also introduces an algorithm that warns about possible presence of hidden dependencies. Section 5 contains related work. Finally conclusions are given in section 6.

2. Example of Hidden Dependence: Session Manager

In this section, we give an example of hidden dependence among objects. It is a fragment of a widely used Java application. The fragment consists of three classes collaborating to manage a session, including the saving and retrieval of each individual application status, as shown in Figure 1.

"SessionContext" is a wrapper class for application context. It contains the state of application and is shared by other applications. It calls methods setContext() and getContext() of Class "SessionManager" to update or delete its own context or retrieve other application contexts. To support undo, class "Log" keeps log of transactions. For example, if the application wants to update its context, the "SessionManager" will call Log.setOperationCode() to record the update, then back up the current context and finally update the application context as required. If undo is requested, the "sessionManager" will call Log.getOperationCode() to get the code of last transaction and reverse the previous transaction.

Since the transaction code is of type int, there is a transaction code protocol and it is assumed that both

0-7695-1131-7/01 $10.00 © 2001 IEEE

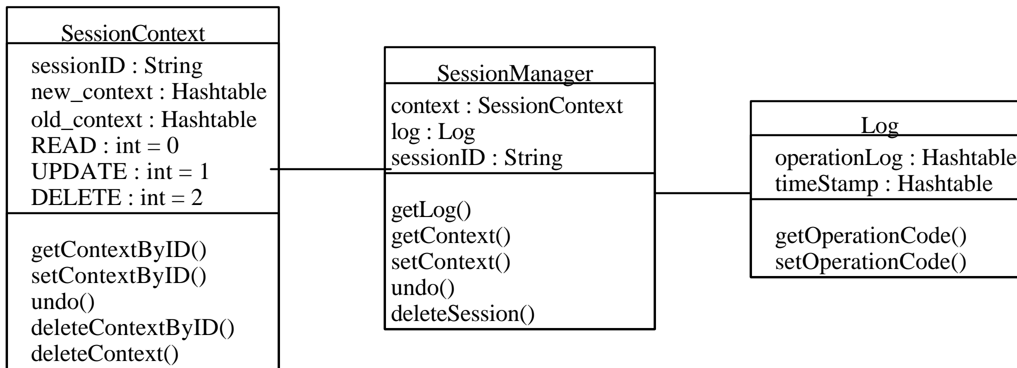

Figure 1. Session Manager class diagram

"SessionContext" and "Log" will understand it. For instance, "0" is the code of transaction "READ".

The classes "SessionContext" and "Log" are completely independent each other. All measures of [3] also indicate that the coupling between "SessionContext" and "Log" is zero. However an instance of "Log" and an instance of "SessionContext" inside class "SessionManger" are linked by a data flow inside the method undo(), as seen in the following fragment of the code:

```
public class SessionManager {
      String sessionID;
      Log log = null;
      SessionContext context = null;
      void undo(String applicationID) {
            context.undo(applicationID,
              getLog().getOperaCode(applicationID));
      }
      . . .
   }
```

Here, the getLog() is called to retrieve the relevant instance of "Log". The getOperaCode() then returns the code of last operation recorded by Log. The return value should be understood correctly by instance context" of "SessionContext" in order to undo the last transaction.

It is very possible that the original protocol may be broken during program change. For example, if we want

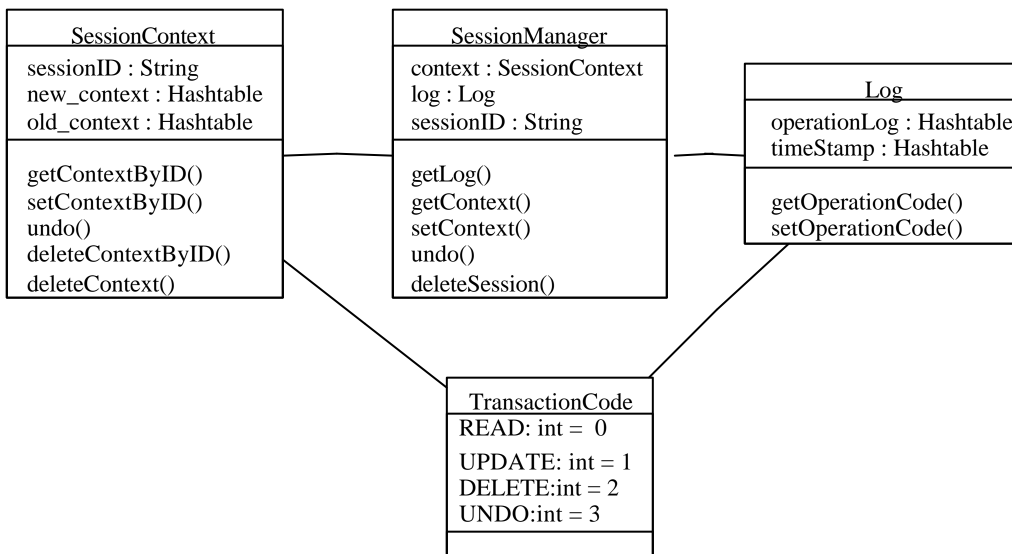

Figure 2 Introduction of class "TransactionCode"

0-7695-1131-7/01 $10.00 © 2001 IEEE

to introduce a new transaction “undo” and have a capability to undo this transaction in the future, the “SessionContext” will have a new entry “UNDO” as transaction code. This change does not have any syntactical impact on “SessionManager” at all, therefore the programmer may stop the change propagation here because “Log” and “SessionContext” are seemingly independent of each other. However, the original protocol is now broken, as “Log” cannot understand the “UNDO” transaction code. To keep the system working, new code has to be added to “Log”.

This type of dependence is caused by a missing encapsulation. It could be eliminated by the introduction of a new class “TransactionCode” as in Figure 2. Then any change of the transaction coding will localize in this class and propagate properly to both “SessionContext” and “Log”.

However, not all object-oriented programs are well designed. Sometimes encapsulation is missed or done improperly. Missing or improper encapsulation spreads the data or attributes of a concept among several objects and results in a hidden dependency between those objects. In spite of the fact that some of those objects are seemingly independent of each other, the missing encapsulation causes dependencies that may come as a surprise to the programmer. We call them “hidden dependencies”.

3. Abstract System Dependence Graph

In order to define hidden dependencies, we define Abstract System Dependence Graph (ASDG) [6]. The ASDG represents the program at the level of software components and it captures issues related to the program design. It is derived from more detailed System Dependence Graph.

3.1. System Dependence Graph

System Dependence Graph (SDG) represents programs on the granularity level of statements [7,8,12,13]. Its vertices are classes, class members, and program statements. The edges represent control and data dependencies.

Data flows are defined in the following way [15]: A variable v defined at statement s_0 of a program p is data flow dependent on a variable u, if and only if u may affect the value of v at s_0. For any statement “s”, we have DEF(s)={variables defined by statement s} and USE(s)={variables used in statement s}. A dataflow edge

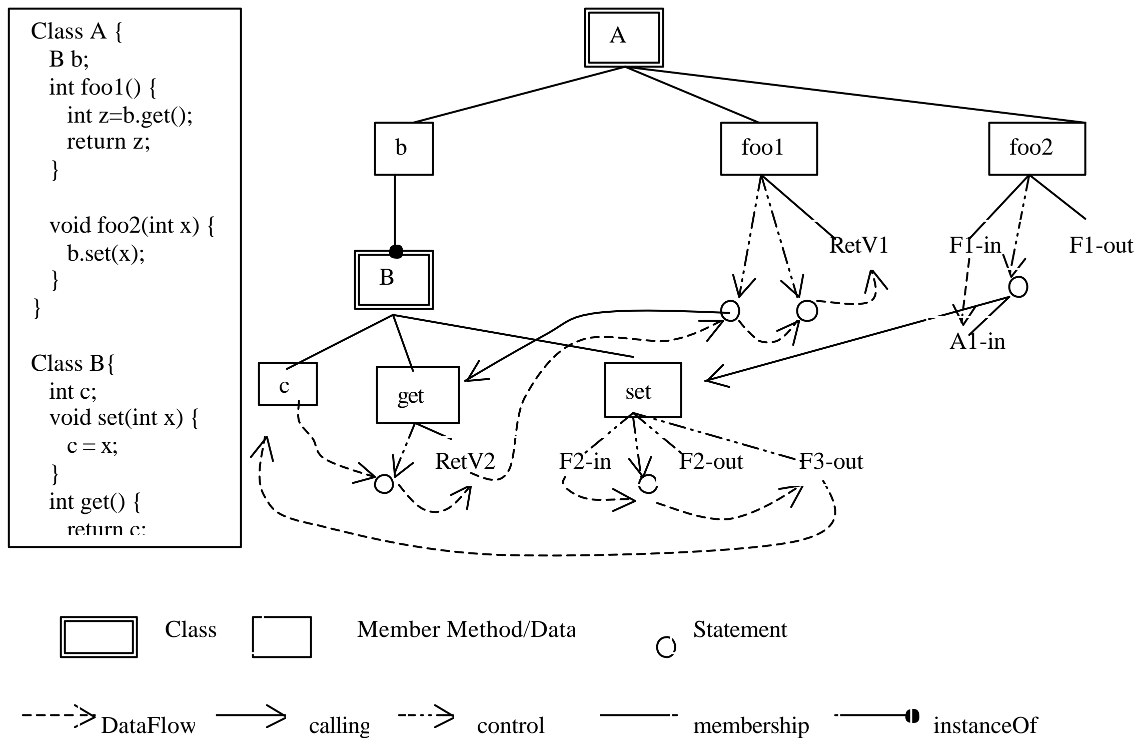

Figure 3 Example of System Dependence Graph

0-7695-1131-7/01 $10.00 © 2001 IEEE

$v_1 \rightarrow v_2$ exists if one of the following conditions holds:

Type 1: if both of v_1 and v2 are statements, then

$$DEF(v_1) \cap USE(v_2) \neq \emptyset;$$

Type 2: if v1 is a variable and v2 is a statement, then

$$v_1 \in USE(v_2);$$

Type 3: if v1 is a statement and v2 is variable, then

$$v_2 \in DEF(v_1);$$

To model parameter passing, SDG associates each procedure with a Formal_in vertex for each formal parameter of the method, and a Formal_out vertex for each formal parameter that may be modified by this method. Each calling site has an Actual_in vertex for each actual parameter and an Actual_out vertex for each actual parameter that may be modified by the methods.

At procedure entries and call sites, global variables are treated as parameters. A call edge connects a call vertex to the entry vertex of the called procedure dependence graph.

Figure 3 shows the construction of SDG for the code sample. In it, RetV1 is a return value for A.foo1(), RetV2 is a return value for B.get(), etc.

SDG focuses on granularity of statements, but this level of detail is insufficient for most of the program comprehension tasks. Also, the SDG of a large program is too complex. Therefore the maintainers use dependencies among higher granularity units and the Abstract System Dependence Graph (ASDG) is more helpful [6].

3.2. Abstract System Dependence Graph

The ASDG is derived from SDG and represents program dependence among the classes and class members. The vertices in the ASDG are classes, variables, and methods. The edge connects two vertexes if there is a relationship of composition (membership), method calling, data flow, inheritance, or instance. ASDG does not contain control flow edges.

Since the vertexes in the ASDG do not include statements, the following process converts SDG into ASDG:

(1) For a statement which is not a calling site:

a) if USE(s)=∅ or DEF(s)=∅, remove statement s and both of the incoming and outgoing edges;
b) Otherwise, make data flow edges from the vertexes in USE(s) to the vertexes in DEF(s), and remove the statement s and all of the edges associated with it.

(2) For the statement which is a calling site:

a) Make data flow edges from the variable vertexes in USE(s) to the variable vertexes in DEF(s), if any.
b) The SDG builds the summary edges among Actual_in and Actual_out vertexes. ASDG needs further abstraction on the transitive flow by:
   i. Remove the Actual_in, Actual_out vertex and calling statements by constructing new summary edges. The summary edges model the transitive data flow across the procedure call. For example, the F1_In→A1_in→F2_In will become F1_In→F2_In, and A1_in is removed.
   ii. Make a calling edge directly from the calling method vertex to the called method vetex, and remove the calling statement vertex.

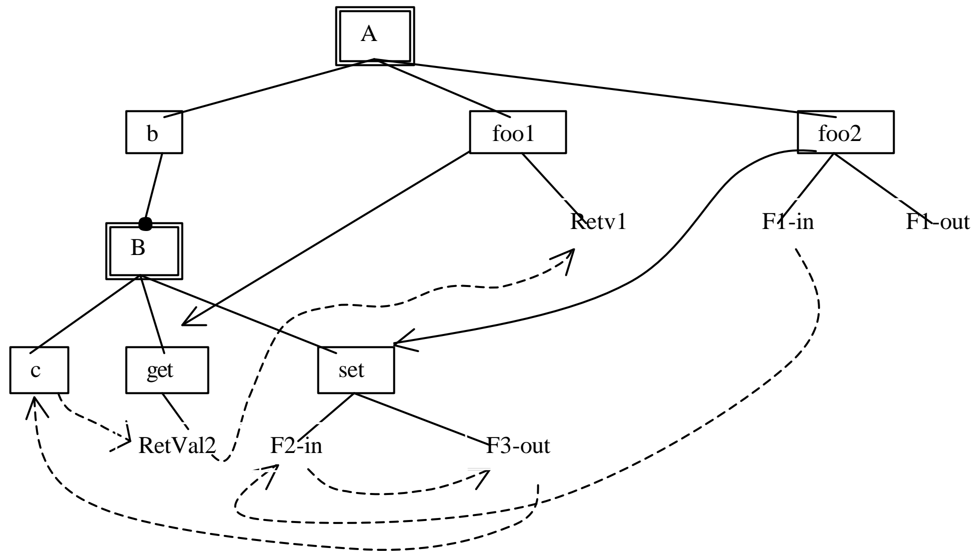

Figure 4 Abstract System Dependence Graph

Figure 4 shows the construction of Abstract System Dependence Graph for the sample code from Figure 3, and a thick dashed edge is used to represent the data flow that is a part of the hidden dependency.

4. Hidden Dependencies

Informally, we say a data flow dependence between Class A and B is a hidden dependence, if the following conditions hold:

1) Class A and B are not explicitly dependent each other, i.e., in the ASDG class A and class B are not neighbors, and hence they fulfill the criterion of [9, p109], "class A is unaware of the existence of class B".

2) However, there is the third class C, which is dependent on both class A and B, and there is data flow inside the class C that occurs between the instance of class A and instance of class B.

The Figure 5 is an Abstract System Dependence Graph for class "SessionManager" after we applied the procedure defined in section 3.2 to the SDG. The figure indicates the existence of a hidden dependence between class "Log" and class "SessionContex". There is a data flow starting from "operationLog", a variable of "Log", to "RetV2" and "F3_in" and finally reach the "new_context" and "old_context" of class "SessionContext", however there is no direct relationship between these two classes syntactically.

4.1. Hidden Dependencies and Impact Analysis

As described in [2], the impact analysis uses the dependency information to produce the change impact sets. Similarly, [18] models the change propagation as a process that keeps track of inconsistent program dependencies and proposes the locations where the subsequent changes are to be made.

Case study of [3] indicates that 50% to 60% of the dependencies are investigated in vain, but at the same time a significant number of dependencies are missed. We believe that hidden dependencies are often among the missed dependencies.

Taking the original "SessionManager" for example, before knowing the hidden data flow dependence, we may think the change on "SessionContext" will propagate to "SessionManager" and stop there. If we do so, the "Log" is missed in the estimated impact set and the software is left in inconsistent state.

4.2. Hidden Dependencies as Anti-Pattern

Hidden dependencies complicate software comprehension and evolution, therefore they should be

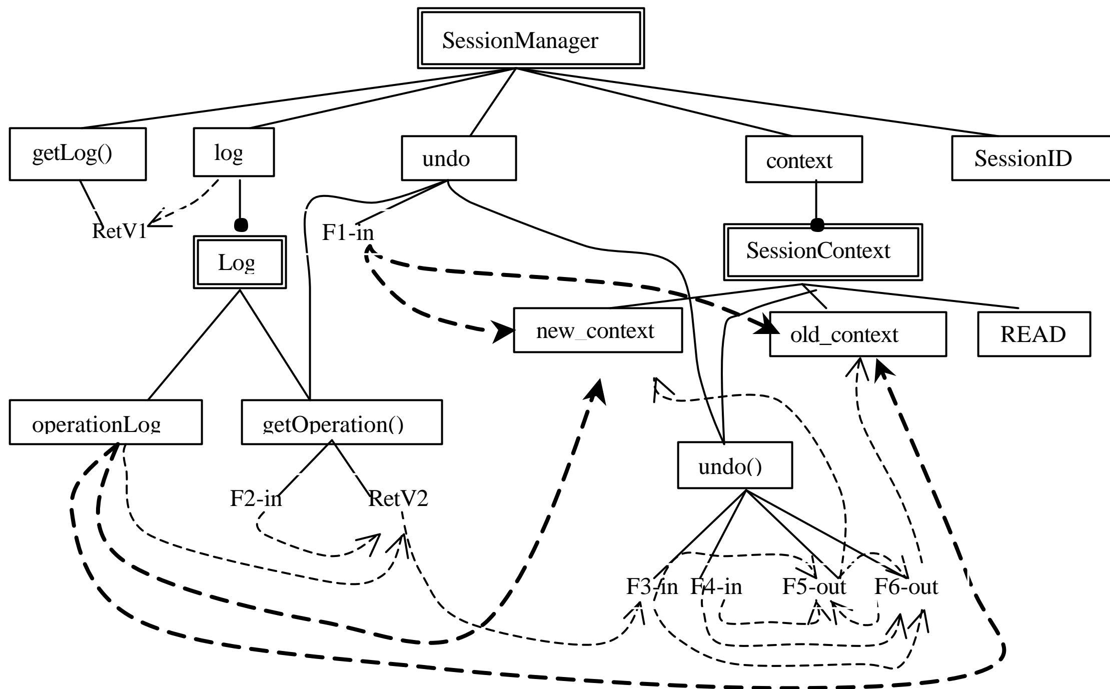

Figure 5 Abstract System Dependence Graph for class "SessionManager"

avoided. They fit with the notion of antipattern. According to [5, p7], "an antipattern is a literary form that describes a commonly occurring solution to a problem that generates decidedly negative consequences".

The cause of hidden dependence anti-pattern is the lack of abstraction and encapsulation. The "SessionManager" example is a typical case of lack of encapsulation.

Since hidden dependencies have a negative impact, we present a method of detecting them. For an ASDG, we define a class dependence relationship $<A, B>$, if at least one dependence edge exists between class A and class B and the former depends on the latter.

For any single class A, D(A) is a collection of dependencies associated with class A, defined as $D(A)=\{<A,B>, <B,A> \mid <A,B> \text{ or } <B,A> \text{ is in ASDG}\}$.

DFD(A) is the collection of data flow dependencies occurring inside of class A, defined as $DFD(A) = \{<B, C>, <C, B> \mid$ there is data flow dependence between instance of class B and instance of class C within class A, $A \neq B$, $A \neq C$, and $B \neq C$ $\}$.

Assuming HD(S) is the set of potential hidden dependencies for system S, HD(S) will be calculated as following:

$HD(S) = \emptyset$;
for each class $C \in S$ {
    for each $< C_1, C_2> \in DFD(C)$ {
        if $\{< C_1, C_2>, <C_2, C_1> \} \cap (D(C_1) \cup D(C_2)) = \emptyset$
            $HD(S) = HD(S) \cup \{< C_1, C_2>\}$;
    }
}

The HD(S) is a potential hidden dependence set. More knowledge of semantics of the classes involved is required to determine which of them are actual hidden dependencies.

A solution for the hidden dependence antipattern is to ensure proper design that decreases the number of hidden dependencies. Program concepts should be localized and encapsulated within a single class in order not to disperse their attributes among several classes and cause implicit dependencies. The component that provides a service and the component that is served should interpret the service in an identical way.

5. Related Work

In the literature, there are analysis algorithms and tools for procedural programs [7, 15, 16] and object oriented programs [1,10, 12, 14, 17]. A difficult problem in data-flow analysis is pointer aliasing. Although the Java eliminates the pointers, the objects passed by reference still introduce the aliasing problem. The techniques for handling parameters passed by reference and for dealing with aliasing are discussed in [8]. Landi described the pointer aliasing problem in [11] and Tonella also presented an approach to interprocedural analysis in [21].

Another problem is how to understand class interactions and establish a set of change propagation rules. Several techniques or models for object oriented software maintenance were presented, like "Change Assistant" of [17] that uses cross-reference information. Potential propagation rules are introduced by [1]. Also of interest is data model used in OO!CARE [14]. Change propagation model of [18] manages the change based on the dependence analysis.

Other researchers found that some classes may be coupled via uncommon mechanisms and some existing dependencies may not be detected by static analysis alone. The research of coupling measurement [3,4] suggests ordering of impact analysis by the strength of coupling. The case study in [3] shows that a significant number of ripple effects are not covered by selecting highly coupled classes only, and some existing dependencies may not be detected by static analysis alone.

Abstract system dependence graph and its role in program comprehension was discussed in [6].

Antipatterns [5] describe commonly occurring solutions to a problem that generate decidedly negative consequences. The hidden dependence of this paper is an important antipattern.

6. Conclusion and Future Work

This paper presents hidden dependencies among components caused by internal data flows inside other components. The occurrence of a hidden dependence is caused by a lack of proper abstraction and encapsulation.

Our future work will be devoted to further investigation of hidden dependencies caused by other object-oriented features like inheritance. In an empirical work, we will explore a question how common the hidden dependencies are in actual systems. We are also planning to work on software restructuring and refactoring that removes hidden dependencies.

References

[1] S. Barros, Th. Bodhuin, A. Escudie, P. Queille, F.Voidrot. "Supporting Imapct Analysis: a Semi-Automated Techniques and Associated Tool", *Proc. of Intl. Conf. on Software Maintenance*, Opio, France, October, 1995, pp 42-51.

[2] S. A. Bohner, R. S. Arnold, "An Introduction to Software Change Impact Analysis", *Software Change Impact Analysis*, IEEE Computer Society Press, ISBN 0818673842, 1996, pp.1-28.

[3] L.Briand, J.Wust, and H. Lounis. "Using coupling for Impact Analysis in Object-Oriented Systems", *Proc. of Intl. Conf. on Software Maintenance*, Oxford (UK), Sep 1999, pp475-482.
[4] L. C. Briand, J. W. Daly, J. K. Wust. "A Unified Framework for Coupling Measurement in Object-Oriented Systems", *IEEE Transactions on Software Engineering*, 25(1), 1999, pp91-121.
[5] W. J. Brown, R. C. Malveau, W. H. Brown, H. W. McCormick, T. J. Mowbray, *Antipatterns: Refactoring Software, Architectures, and Projects in Crisis*, John Wiley & Sons, ISBN: 0471197130, 1998
[6] K. Chen, V. Rajlich, "Case Study of Feature Location Using Dependence Graph", *Proc. of Int. Workshop of Program Comprehension,* Los Alamitos, CA, 2000, pp. 241-249.
[7] J. Ferrante, K. Ottenstein, J. Warren, "The Program Dependence Graph and its use in optimization", *ACM Transactions of Programming Language and Systems* 9(3), July 1987, pp. 319-349.
[8] S. Horwitz, T. Reps, D. Binkley, "Interprocedural slicing using dependence graphs", *ACM Transactions on Programming Languages and Systems*, 12(1), Jan. 1990, pp.26-60.
[9] C. S. Horstmann, G. Cornell. *Core Java 1.1: Volume 1-Fundamentals*, Prentice Hall, ISBN 0137669577 1997, p. 109.
[10] Y. K. Jang, H. S. Chae, Y. R. Kwon, D. H. Bae, "Change Impact Analysis for A Class Hierarchy", *Conf. on Asia Pacific Software Engineering*, Taibei, Taiwan, 1998, pp. 304-313.
[11] W. Landi, B. G. Ryder, S. Zhang. "Inter-procedural modification side effect analysis with pointer aliasing", *Proc. of SIGPLAN's93 Conf. on Program Language Design and Implementation*, June, 1993, pp. 56-67.
[12] L. Larsen, M. J. Harrold, "Slicing Object-Oriented Software", *Proc. of Int. Conf. of Software Engineering*, 1996, pp. 495-505.
[13] D. Liang, M. J. Harrold, "Slicing Objects Using System Dependence Graphs", *Proc. of the Intl. Conf. on Software Maintenance*, Bethesda, MD, November 1998, pp. 358-367.
[14] P.K.Linos, V.Courtois, "A Tool for Understanding Object-Oriented Program Dependencies", *IEEE Transaction of Software Maintenance,* 1994, pp. 20-27.
[15] P.E.Livadas, P.K.Roy, "Program Dependence Analysis", *Proc. of Int. Conf. on Software Maintenance*, 1992, pp. 356-365.
[16] J. P. Loyall, S.A. Mathisen. "Using Dependence Analysis to Support the Software Maintenance Process", *Proc. of Int. Conf. on Software Maintenance*, Calif., 1993 pp. 282-291.
[17] M. Platoff, M. Wagner, J. Camaratta, "An Integrated Program Representation and Toolkit for the Maintenance of C Programs", *Proc. of the Int. Conf. on Software Maintenance*, Calif. 1991, pp. 129-137.
[18] V. Rajlich, "A Model for Change Propagation Based on Graph Rewriting", *Proc. of the Int. Conf. on Software Maintenance*, Bari, 1997, pp. 84-91
[19] V. Rajlich, K. Bennet, "A Staged Model for the Software Life Cycle", *IEEE Computer,* 33(7), July, 2000, pp. 66-71.
[20] P. Tonella, G. Antoniol, R. Fiutem, E. Merlo, "Flow Insensitive C++ Pointers and Polymorphism Analysis and its Application to Slicing", *Proc. of the Int. Conf. of Software Engineering*, Boston, 1997, pp. 433-443.
[21] P.Tonella, G. Antoniol, R. Fiutem, E. Merlo, "Variable Precision reaching Definitions Analysis for Software maintenance", *Proc. of the First Euromicro Conf. on Software Maintenance and Reengineering*, Berlin, 1997, pp. 60-67.
[22] N. Wilde, R. Huitt, "Maintenance Support for Object-Oriented Programs", *IEEE Transaction of Software Engineering*, 18(12), December, 1992, pp. 1038-1044.
[23] N. Wilde, P. Matthews, R. Huitt, "Maintaining Object-Oriented Software", *IEEE Journal of Software*, January, 1993, pp. 75-80.
[24] N. Wilde, S. W. Dietrich, F. W. Calliss, "Designing Knowledge-Base Tools for Program Comprehension: A Comparison of EDATS and IMCA", *SERC-TR-79-F*, Software Engineering Research Center, University of Florida, CSE-301, FL 32611, December 1995.
[25] S.S. Yau, J. S. Collofello, t. MacGregor, "Ripple Affect Analysis for Software Maintenance", *Proc.COMPSAC-78*, IEEE Computer Society, 1978, pp. 60-65.