http://ansinet.com/itj

ITJ

ISSN 1812-5638

INFORMATION TECHNOLOGY JOURNAL

ANSInet
Asian Network for Scientific Information
308 Lasani Town, Sargodha Road, Faisalabad - Pakistan

Information Technology Journal 5 (5): 876-883, 2006
ISSN 1812-5638

Algorithms for Defect Detection in Object Oriented Programs

S. Sarala and S. Valli
Department of Computer Science and Engineering,
College of Engineering Guindy, Anna University, Chennai-25, India

Abstract: Defects are any condition which causes malfunctioning or which prevents the attainment of expected or previously specified results. Defects, which lead to logical error are a burden for the user or programmer. Also, the compiler is not equipped to track such defects. A piece of code can be tested to increase confidence by exposing potential flaws or deviations from user's requirements. In this study algorithms are developed to automatically detect defects in C++ Programs. The algorithm checks the data type of the actual parameters and formal parameters for a exact match. If a match doesn't occur, the tool reports this situation. Defect results due to omission or mismanipulation. This study checks the correctness of the program when operator [] is overloaded. In the context of inheritance when virtual function is used, it has been observed that expected results are not achieved under certain circumstances. Algorithms have been developed to handle this situation. Also the working of function templates in the context of character input and also the working of the program in the context of exception handling is tackled.

Key words: Static testing, logical and execution errors, rule, regular expression, defect detection

INTRODUCTION

Software testing ensures the quality of the code. The objective of testing is to authenticate incorrectness and succeed when an error is detected (Beizer, 1990; Myers and Glenford, 1979). Testing validates whether the observed behavior conforms to the specifications. In particular, it checks whether the implemented functions are as intended (Patton, 2001). Code verification or checking is heavily manual, error-prone and time consuming (Edward, 1995). To overcome these problems, static testing techniques have been proposed. Unit testing is applied to check interfaces with respect to whether the parameters are passed in correct order, the number of formal parameters are equal to the number of actual arguments, incorrect variable usage and inconsistent data type. The usage of testing tools relieves the burden of the programmer and makes them less error-prone and increases testing efficiency and effectiveness. This work automates static testing to detect defects in object-oriented source code. The defects in C++ source code are detected in this study.

The static checking tools looks at the source code in text format and determines possible issues. The static checking tools looks for unreachable code, unused variables (So *et al*., 2002), uninitialised variables, impossible paths, buffer overflows and memory leaks. Code reviews, inspections, tool-assisted analysis are static testing methods. Static testing reveals possible errors, misunderstandings, missed codes, ignored or forgotten codes. It can test for all possible inputs and array boundaries. These tests help in fixing possible logical errors. Generally, logical errors are not found by the compiler, which may lead to program termination and might never be discovered. It is difficult to find, fix and it causes inconvenience, financial losses or disasters too.

This study focuses on detecting errors, which goes unnoticed by the compiler and results in execution error or produces incorrect results. Flex rules are developed in this work to detect defects which lead to logical error or execution error. Object-oriented testing is tedious and time consuming. Therefore, a tool support is very much essential.

Block of statements or code, a method or a complete class, a unit or a piece of code is tested for finding defects. Object-oriented software testing has to tackle problems introduced by the features such as encapsulation, inheritance and polymorphism (Younessi, 2002). The interaction between two or more objects is implicit in the code. This makes it difficult to understand object interactions and to test such interactions. A class can be inherited by subclass (Booch, 1991). A method that is tested to be correct in the context of the base class does not guarantee that it will work correctly in the context of the derived class. These features make testing difficult because the control flow of the object-oriented program is less transparent.

Corresponding Author: S. Sarala, Department of Computer Science and Engineering, College of Engineering Guindy, Anna University, Chennai-25, India Tel: 9111052611

CODING FAULTS IN OBJECT ORIENTED PROGRAMS

The work of Binder 1999 and Porwal and served as the motivation for this work. Binder (1999) has detected various coding faults, which causes failures. For instance, parameters in function call could have a coding bug, overlapping or conflicting functions, the wrong method called due to coding error or unexpected runtime binding, wrong parameters or incorrect parameter values. Coding errors may be in the form of misspelling and misnaming (Porwal and Gurusaran, 2001).

C++ Soft Bench Code Advisor is an automated error detection tool for the C++ language. Code Advisor is an automated rule checker. Compilers do not find errors in constructs that are legal but unlikely to be what the programmer intended. Code Advisor uses detailed semantic information available in the Soft Bench static database to detect high-level problems not typically found by the compilers. When it detects a rule violation, Code Advisor displays the line number in which the violation is encountered in an error browser. The static database contains information about global and local objects (Duesing and Diamant, 1997). It detects the errors in the context of virtual functions called from constructors, when iostream routines are mixed with studio routines. It also detects the local variables hiding the data members.

STATIC ANALYSIS

Self-checks are executable condition statements, which tests the internal state of the software (So *et al.*, 2002). It examines the state for anticipated erroneous conditions. By static analysis the authors identify unused variables, infinite loops, incorrect calculations due to wrong operators as performing multiplication instead of adding, incorrect variable usage such as substituting one array subscript with another, division by zero failures, incorrect formulated branch conditions, missing branches, missing threads, failure to update a value before its new value is needed, inconsistencies in the formal and actual parameters ordering, inconsistencies in the data structure in the case of linear linked list erroneously going circular.

Goichi *et al.* (2005) have developed algorithms and tools to detect buffer overflows, stack overflow by static analysis of C source code. They have analyzed the buffer overflow vulnerabilities that lead to serious damage during execution of the C code.

Evans and Larochelle (2002) have developed a lightweight static analysis tool, the Splint for ANSI C. The authors make use of annotations which is C comments followed by the delimiter '@' after '*'. Annotations are associated with function parameters and results, global variables and structure fields. The annotation /* @ not null @ */ when associated with parameter declaration, implies that the value passed for the parameter is not NULL. Splint would report a warning when the actual parameter is NULL. Splint checks whether the source code is consistent with the properties implied by annotations. Splint reports a warning when storage is not released. Buffer overflows in the case of stack and heap is also handled by Splint. Splint also detects the misuse of files as in the case of failing to close a FILE, failing to reset a read/write file between certain operations.

PERFORMANCE OF C COMPILER

Bailey and Davidson (2003) has automated testing of procedure-calling conventions. The authors were able to detect faults in C compilers. The authors use the signature of the procedure in their implementation for detecting faults. The authors define the procedure's signature as the procedure's name, the order and types of its arguments and the return type.

RESULTS DETECTION OF LOGICAL DEFECTS

Rules have been framed to detect the presence of defect which lead to logical error in 'if' condition, subscript operator, function call operator using flex.

Algorithm to validate IF statement: The syntax of 'if' taken into consideration is if (arg 1 op arg 2) where arg 1 and arg 2 are operands and op is a relational operator. The regular expressions (1) to (5) are used in formulating the rule to detect logical error in the if statement.

Regular Expressions to check Assignment operator: The Regular expression (1) identifies the presence of "if" and it matches white spaces and checks for '('.

"if"[\t]* "(" (1)

[a-z_0-9]* (2)

The regular expression (2) extracts arg 1 and arg 2; arg 1 can be a variable starting with an alphabet followed by any number of alphabet or digit or a constant made up of digit or digits.

"=" (3)

The regular expression (3) identifies the presence of assignment operator in the relational expression.

"=" (4)

Closing parenthesis is matched by using the regular expression (4).

A rule is framed using regular expressions (1) to (4). The rule is given by (5). If the rule matches the action part of the rule executes. The action of the rule displays that, Assignment operator is used instead of equality operator and also the line number in which the error is encountered is displayed. The variable yylineno of flex contains the line number in which the defect is encountered. Constructs are presented which was subjected to the rule for defect detection.

```
"if"[ \t]*"(" [a-z_0-9]* "=" [a-z_0-9]* ")"          (5)
{
printf("Assignment operator is used instead of
equality operator in line %d", yylineno);
}
```

Logical defect in the IF condition: The code fragment in Table 1 has a missing equality operator in the 'if' condition. When this code was analyzed by the rule, the absence of equality operator was indicated.

The regular expression (1) matched the pattern "if(" in Table 1. The first argument, namely i, was extracted by the regular expression (2). The regular expression (3) extracted the assignment operator and the second argument, variable j, was extracted by regular expression (2). The regular expression (4) matched the pattern ")". Since the rule matched the action part executed and displayed Assignment operator is used instead of equality operator. When the code in Table 1 is executed, it results in logical error. Always then block only executes.

When the code fragment in Table 2 was subjected to the rule, it validated the presence of equality operator and it ascertained that the code fragment is free from logical error.

When the construct in Table 3 was subjected to the rule, the regular expression (2) extracted the second operand, namely 5. The regular expression (2) matches any combination of alphabets, digits, alphabets and digits. Therefore, if the operand is a constant or a variable it will be matched by regular expression (2). The rule matched and the action executed indicating "Assignment operator is used instead of equality operator".

Algorithm to validate character input: The regular expressions (6), (2) and (3) are used in formulating the rule to detect the defect which leads to logical error when a value is assigned to a character variable. Another rule has also been formed using regular expressions (6), (2), (3) and (8).

Table 1: Construct with logical error in the if statement

```
if(i= j)
    cout << "here";
    else
    cout << "there";
```

Table 2: Construct free of logical error in the if statement

```
if(b = = a)
    cout << "the operands are equal" ;
    else
    cout << "the operands are unequal" ;
```

Table 3: Code fragment to illustrate usage of constant as argument

```
if(a = 5)
{
    …
}
```

Regular expressions to check the value of a char data type variable: The regular expression (6) finds 'char' data type and matches all white spaces. The regular expression (2) identifies the name of the character variable. Using regular expression (3) the assignment operator is identified. The regular expression (7) extracts the integer value assigned to the character variable.

"char"[\t]* (6)

[0-9]* (7)

[0-9]*[.]{1}[0-9]* (8)

A rule is framed using regular expressions (6), (2), (3) and (7). The rule is given in (9). If the rule matches, the action part gets executed. In the action part the message 'Missing character input' is displayed. The line number in which the defect is present is also displayed. The second rule checks the float value assigned to a character variable has given by (10). Constructs are presented which was subjected to the rule for defect detection.

```
"char"[ \t]*[a-z_0-9]* "=" [0-9]*                    (9)
{
printf("Missing character input in line %d",
yylineno);
}
```

```
"char"[\t]*[a-z_0-9]*"="[0-9]*[.]{1}[0-9]*           (10)
```

The pattern {1} in the regular expression (10) matches with only one occurrence of the predecessor pattern, namely '.'.

Logical error due to defect in integer value assigned to a character variable: The code fragment in Table 4 has an integer value assigned to a character variable. When this code was analyzed by the rule, "Missing character input" was indicated.

The regular expression (6) matched the pattern "char" in Table 4. Also it eliminated the white spaces following 'char' keyword. The variable name, namely a, was extracted by the regular expression (2). The regular expression (3) extracted the assignment operator. The regular expression (7) extracted the value, namely 10, assigned to the character variable. Since the rule (9) matched, the action part executed and displayed 'Missing character input in line 8.

When the code in Table 5 was subjected to the rule, the assignment of character value to the character variable was verified and hence the construct is free of defect was ascertained.

When the code in Table 6 was subjected to the rule (10), the regular expression (6) matched the data type 'char' and the white spaces. The regular expression (2) matched the character variable, namely, a. The regular expression (3) matched the assignment operator '='. The regular expression (8) matched the digit 5, the decimal point '.' and digit 7 after the decimal point. Since the rule (10) matched, the action corresponding to the rule executed. It displayed 'missing character input in line 8'.

Algorithm to detect missing function call operator: The presence of function call operator () in the context of member function is checked by the rule. The omission of the function call operator is not trapped by the compiler. The function call does not take place and the statement has no effect during execution in the absence of the function call operator. This is a logical error, since this behavior is not anticipated by the programmer.

Regular expression to detect missing function call operator: The regular expression (2) identifies the object and the function name.

"(" (11)

The regular expression (11) matches the '(' in the prototype.

[a-z_0-9]*[\t]* [a-z_0-9]* "(" (12)
{
The member function name is extracted and stored in array temp
}

Table 4: Construct with integer value assignment to a character variable

```
template<class T>
void fun(T a)
{
        cout << a;
}
void main()
{
        char a=10;
        fun(a);
}
```

Table 5: Construct with a character value assigned to a character variable

```
template<class T>
void fun(T a)
{
        cout << a;
}
void main()
{
        char a='9';
        fun(a);
}
```

Table 6: Construct with a float value assigned to a character variable

```
template<class T>
void fun(T a)
{
        cout << a;
}
void main()
{
        char a=5.7;
        fun(a);
}
```

A rule (12) is framed using regular expressions (2) and (11) to extract the member function name from the prototype. The call to member function is matched.

[.]{1} (13)

";" (14)

A rule (15) framed using regular expressions (2) and (13) and (14).

[a-z_0-9]+[.]{1} [a-z_0-9]+";" (15)
{
The member is extracted and stored in array temp1
}

'temp1' array's content is compared with 'temp' array. If there is a match the omission of function call operator is displayed.

Defect in function call operator: The code fragment in Table 7 has a missing function call operator. The rule detected the absence of missing function call operator.

The regular expression [a-z_0-9]* in rule (12) matched the return type, void, in Table 7. The regular expression [\t]* matched the white spaces in the function

Table 7: Construct with missing function call operator

```
class CX
{
private:
int x;
public:
void init(int j)
{
        x=j;
}
};
void main()
{
        CX y1;
        y1.init;
}
```

prototype. The regular expression [a-z_0-9]* after tab in (12) matched the function name 'init'. The regular expression (11) matched '('. Since the rule (12) matched, the action executed. The function name 'init' is stored in array 'temp' in the action part.

The regular expression (2) matched the object name, y The regular expression (13) matches only one membership operator. So the membership operator was matched by regular expression (13). The function name, init, was matched by regular expression (2). The delimiter was matched by the regular expression (14). Therefore, the rule (15) matched. The action part executed and extracted the member function name 'init'. The name was stored in array temp1. Since the contents of the array temp and temp1 matched, the message "Missing function call operator" was displayed.

DETECTION OF EXECUTION ERRORS

Rule has been framed to detect defects, which leads to execution error as well as logical error depending on the context in which it is used. The subscript operator leads to both logical as well as execution error depending on the usage.

Algorithm to validate subscript operator: The syntax of subscript operator taken into consideration is

a[i] = b[j] or a[i] = 6 (constant)

int|float|char|double [\t]*[a-z_0-9]*"["[0-9]*"]" (16)

"[" (17)

[\t]* (18)

[-]? (19)

"]" (20)

Regular expressions to check out of bound condition: The regular expression (2) extracts the array name. The regular expression (17) matches the "[" and if any spaces exist after the square brackets, the regular expression (18) matches them.

If a negative subscript is used, the regular expression (19) matches it. It allows only one minus sign to be present. The subscript is found by the regular expression (7). This represents one or more digits. The "]" in regular expression (20) matches the closing bracket. The assignment operator is matched by the regular expression (3). A constant on the right hand side is matched by the regular expression (7). The right hand side can even be a reference to an array element. The rule is given by (21). Code fragments are presented to illustrate logical and execution error caused by the subscript operator depending on its usage.

Logical defect in subscript operator:

```
[a-z_0-9]*|[a-z_0-9]*"["[\t]*[-]?[0-9]+"]""="[a-z_0-9]*|
        [a-z_0-9]*"["[\t]*[-]?[0-9]+"]"                    (21)
    {
    The subscript is checked with max
    }
```

The rules (16) and (21) are used in checking out of bound condition. When the code in Table 8 was subjected to the rule (16), the array name was stored as 'a' and the size of the array was stored as 2 in 'max'. The rule (21) matched the reference to array elements, namely a[0], a[1] and a[2]. The second operand of 'OR' operator before the assignment operator in (21) matched the array element reference. In each case the subscript value was checked with 'max'. Since it is within bounds, there is no defect.

When the code fragment in Table 8 is executed, ob[1] which is outside the bound of the array a, retrieves junk value. This is logical error. This defect was detected by the rule (21). The code in Table 8 works and assigns junk value. This defect is a burden for the programmer. The rule identifies the subscript which are out of bounds and displays the message. The rule identifies subscript out of bounds even in the context of subscript operator being overloaded. The use of subscript operator in Table 8 for assigning value leads to logical error whereas the use of subscript operator in Table 9 leads to execution error.

The rule (16) matches with array declaration. In the action part the array name and size of the array is stored. The reference in main() was matched by regular expression(2) and (3). The second operand of 'OR' operator to the right of '=' in (21) matched ob[4]. Since the rule (21) matched, the corresponding action executed.

Table 8: Code fragment to illustrate defect in array subscript operator

```
class atype
{
int a[2];
public:
atype(int i,int j, int k)
{
        a[0]=i;
        a[1]=j;
        a[2]=k;
}
int& operator[](int I)
{
        return a[i];
}
};
int main()
{
    atype ob(1,2,3);
    int j;
    j=ob[4];
    return 0;
}
```

Table 9: Code fragment to illustrate defect which leads to execution error

```
class atype
{
int a[2];
public:
atype(int i)
{
        a[-1]=i;
}
};
int main()
{
    atype aa(5);
    return 0;
}
```

The subscript, namely 4 was checked with max, which is 2. Since the subscript is greater than max, the rule displayed 'subscript is out of bound'.

When a negative subscript is used as given in Table 9, the presence of '-' in the rule (21) matched the minus sign. The rule detected the defect.

Algorithm to check defect in memory allocation: The syntax of dynamic memory allocation taken into consideration is op 1 = op 2, where op1 is the variable and op 2 is the 'new' keyword. The Regular expression (6), (22) and (2) are used in formulating the rule to detect the data members which needs to be allocated memory dynamically.

Regular expressions to check the presence of new operator to dynamically allocate memory: The regular expression (6) finds 'char' data type and matches all white spaces.

```
"*"                                    (22)
```

```
"new"                                  (23)
```

The regular expression (22) identifies the operator, which implies dynamic memory requirement. The regular expression (2) identifies the variable name, which needs to be dynamically allocated memory. If the rule as given by (24) matches, the action part gets executed. In the action part the variable name is extracted. This variable needs memory to be dynamically allocated. The variable name is stored in the array 'assignment'.

Another rule is used to validate the dynamic allocation of memory. Regular expressions (2), (3) and (23) are used in framing the rule which validates the presence of 'new' operator for allocating memory dynamically. The rule is given in (25) and if the rule matches, the action part gets executed. In the action part the variable which is assigned memory dynamically is extracted and stored in an array temp. The content of assignment array is compared with temp contents. If the assignment array content does not match with the temp array's contents, the message 'Missing new operator' is displayed.

```
"char"[ \t]*"*"[a-z_0-9]*                          (24)
{
store the variable in assignment array.
}
```

```
[ \t]*[a-z_0-9]* [\t]*"=new"                       (25)
{
store the variable in temp array.
}
```

Execution error due to missing new operator: The code fragment in Table 10 has a missing new operator. When this code was analyzed by the rule, the missing new operator was found.

The regular expression (6) matched the pattern "char" in Table 10. Also it eliminated the white spaces following 'char' keyword. The regular expression (22) extracted the '*' operator which implies that the member needs to be dynamically allocated memory. The regular expression (2) extracted the data member, namely 'str'. Since the rule (24) matched, the action executed. In the 'assignment' array str was stored, the variable which needs to be allocated memory dynamically. The pattern, which was matched by the rule, will be contained in 'yytext' of lex. This is an array. The contents of yytext is processed in the action part.

The regular expression (2) extracted s, from Table 10 for which memory is dynamically allocated. The regular expression (3) matched '=' in the same statement and regular expression (23) extracted the 'new' operator. Since the rule (22) matched the action part got executed. The variable is stored in array 'temp'. The 'assignment' array's contents was compared with the contents of array temp.

Table 10: Code fragment with missing new operator

```
class string
{
char  *str;
int len;
public:
        string(char *s, int a)
        {
        s=new char[10];
        strcpy(str,"hello");
        }
};
```

Since the content of assignment array is not found in temp array, 'Missing new operator' is displayed.

Algorithm to detect defects in exception handling constructs: The developed rule checks whether every exception thrown is handled. If an exception is unhandled, it leads to execution error.

Regular expressions to check missing exception handlers: The regular expression (26) matches the exception handler which handles any exception thrown. If there is a integer exception handler, the rule (27) sets the enumerated data type a to true. The presence of float exception handler is matched by the rule (28), the corresponding action part assigns 'yes' to b. The presence of string exception handler is matched by the rule (29), the action part assigns 'yes1' to the enumerated data type b1. The presence of integer exception is matched by the rule (30), the action part assigns INT to the enumerated data type typeThrown. The presence of float exception is matched by the rule (31). The action part assigns FLOAT to the enumerated data type typeThrown1. The presence of string exception is matched by the rule (32), the action part assigns STRING to the enumerated data type typeThrown2.

```
"catch(...)" {                                  (26)
exit(0);   }

"catch(int" {                                   (27)
a = true;      }

"catch(float" {                                 (28)
b = yes;        }

"catch(char"[ \t]*"*" {                         (29)
b1 = yes1;    }

"throw"[ \t]*[0-9]* {                           (30)
typeThrown = INT;   }

"throw"[ \t]*[0-9]*[.]+[0-9]* {                 (31)
typeThrown1 = FLOAT;  }

"throw"[\t]*[\"]{1}[a-z]* {                     (32)
typeThrown2 = STRING;    }
```

```
if (a==1&& typeThrown == INT ) {                (33)
printf("integer exception not handled");
}

if (b= =1&& typeThrown1 = = FLOAT) {            (34)
printf("float exception not handled");
}

if (b1==1 && typeThrown2 == STRING) {(35)
printf("string exception not handled");
}
```

In method main(), the conditions (33)-(35) test the missing handlers and reports the same. If the handler is omitted, it results in execution error.

A construct free of defects: The code fragment in Table 11 has the generic exception handler. All the exceptions will be handled by the generic handler. The rule (26) matched catch (…) in Table 11. The action part terminated, due to exit(0). This ensures that the construct in Table 11 is free from defect.

When the code in Table 12 was subjected to the rules, the rule (30) matched and assigned INT to typeThrown. The condition (33) matched since the enumerated member a, had the default value. It displayed "integer exception not handled".

Table 11: Code fragment with abnormal exception in throw

```
class sample
{
public:
void fun1()
        {
        throw 10;
        }
void fun2()
        {
        throw 10.9;
        }
void fun3()
        {
        throw "one";
        }
};
int main()
        {
        try
        {
                sample s;
                s.fun1();
                s.fun2();
        }
        catch(...)
        {
        }
}
```

Table 12: Construct with omitted integer exception handler

```
void fun1()
{
        throw 10;
}
```

CONCLUSIONS

The proposed algorithms are used to detect flaws in C++ source code which leads to logical error or execution error and improves the quality of the code by automatically detecting defects which goes unnoticed by the compiler. Moreover, to detect flaws caused by the omission of code or wrong implementation in the context of function templates, member functions, exception handling, dynamic memory allocation, relational expressions and array subscripting. Further it is extended to detect defects in C# programs caused by typographical mistakes and omission of characters which results in execution error. By using this approach, the defect rate for the users of the class is reduced. The benefit is to relieve the users from the burden of detecting these defects.

REFERENCES

Bailey, M.W. and J.W. Davidson, 2003. Automatic detection and diagnosis of faults in generated code for procedure calls. IEEE Trans. Software Eng., 29: 1031-1042.

Beizer, B., 1990. Software-Testing Techniques, Van Nostrand Reinhold, New York.

Binder, R.V., 1999. Testing object-oriented systems-models, patterns and tools, Addison Wesley, pp: 640-641.

Booch, G., 1994. Object oriented analysis and design with applications. Benjamin/Cummings, Redwood City, California.

Duesing, T.J. and J.R. Diamant, 1997. Code advisor: Rule-based C++ defect detection using a static database, Hewlett-Packard J., 48: 19-21.

Edward, K., 1995. Software Testing in the Real World. Addison-Wesley.

Evans, D. and D. Larochelle, 2002. Improving security using extensible lightweight static analysis. In IEEE Software, 19: 42-51.

Goichi, N., M. Kyoko and M. Ichiro, 2005. Buffer-overflow detection in c program by static detection. J. National Institute of Information and Communication Technology, 52: 35-41.

Myers and J. Glenford, 1979. The Art of Software Testing. John-Wiley and Sons.

Patton, R., 2001. Software Testing. Chapter 9, SAMS Techmedia, pp: 149.

Porwal, R. and Gurusaran, 2004. An Experimental evaluation of weak-branch criterion for class testing, J. Sys. Software, 70: 209-224.

So, S.S., S.D. Cha, T.J. Shimeall and Y.R. Kwon, 2002. An empirical evaluation of six methods to detect faults in software. Software Testing, Verification and Reliability, 12: 155-171.

Younessi, H., 2002. Object-Oriented Defect Management of Software. Prentice Hall, USA.