BLACK BOX AND WHITE BOX TESTING TECHNIQUES –A LITERATURE REVIEW

Srinivas Nidhra[1] and Jagruthi Dondeti[2]

[1]School of Computing, Blekinge Institute of Technology, Karlskrona, Sweden
nidhra.srinivas@gmail.com
[2]Jawaharlal Nehru Technological University, Hyderabad, Andhra Pradesh, India
jagruthi.sri@gmail.com

ABSTRACT

There are several methods for automatic test case generation has been proposed in the past. But most of these techniques are structural testing techniques that require the understanding of the internal working of the program. There is less practical coverage of all testing techniques together. In this paper we conducted a literature study on all testing techniques together that are related to both Black and White box testing techniques, moreover we assume a case situation of Insurance premium calculation for driver and we derive the test cases and test data for white box testing methods such as Branch testing, Statement testing, Condition Coverage testing, multiple condition coverage testing, in the similar way we derive the test cases and test data for the black box testing methods such as: Equivalence partitioning and Boundary value analysis.

The overall aim of this literature study is to clearly explain different testing techniques along with a case situation and their advantages.

KEYWORDS

Software testing, Functional testing, Structural testing, test cases, black box testing, white box testing, Testing techniques.

1. INTRODUCTION

Software testing is a most often used technique for verifying and validating the quality of software [1]. Software testing is the procedure of executing a program or system with the intent of finding faults [10]. It is measured to be labour intensive and expensive, which accounts for > 50 % of the total cost of software development [11] [2]. Software testing is a significant activity of the software development life cycle (SDLC). It helps in developing the confidence of a developer that a program does what it is intended to do so. In other words, we can say it's a process of executing a program with intends to find errors (Biswal et al. 2010) [3]. In the language of Verification and Validation (V&V), black box testing is often used for validation (i.e. are we building the right software?) and white box testing is often used for verification (i.e. are we building the software right?) [2-4][11]. This study emphasizes the need to investigate various testing techniques in software testing field; we have conducted a literature review to obtain the reviews from state-of-art.

2. RELATED WORK-TAXONOMY OF TESTING TECHNIQUES

Traditionally Software testing techniques can be broadly classified into black-box testing and white-box testing [5] [12]. Black box testing is also called as **functional testing,** a functional testing technique that designs test cases based on the information from the specification [5]. With black box testing, the software tester should not (or does not) have access to the internal source code itself. Black box testing not concern with the internal mechanisms of a system; these are focus solely on the outputs generated in response to selected inputs and execution conditions [5]. The code is purely

DOI : 10.5121/ijesa.2012.2204

considered to be a "big black box" to the tester who can't see inside the box. The software tester knows only that information can be input into the black box, and the black box will send something back out. This can be done purely based on the requirement specification knowledge; the tester knows what to expect the black box to send out and tests to make sure the black box sends out what it's supposed to send out [6].

On the other side white box testing is also called as structural testing or glass box testing, structural testing technique that designs test cases based on the information derived from source code [5]. The white box tester (most often the developer of the code) knows what the code looks like and writes test cases by executing methods with certain parameters [5]. White box testing is concern with the internal mechanism of a systems, it mainly focus on control flow or data flow of a programs [1] [5] [18].

White-box and black-box testing are considered corresponding to each other. Many researchers underline that, to test software more correctly, it is essential to cover both specification and code actions [5] [4] [13].

Software testing is a vast area that mainly consists of different technical and non-technical areas, such as Requirements Specifications, maintenance, process, design and implementation, and management issues in software engineering. Our study focuses on the state of the art in describing different testing techniques, before stepping into any detail of the maturation study of these techniques, let us have a brief look at some technical concepts that are relative to our work.

2.1. The Testing Spectrum

Software testing is involved in each stage of software life cycle, but the way of testing conducted at each stage of software development is different in nature and it has different objectives.

Unit testing is a code based testing which is performed by developers, this testing is mainly done to test each and individual units separately. This unit testing can be done for small units of code or generally no larger than a class. [13].
Integration testing validates that two or more units or other integrations work together properly, and inclines to focus on the interfaces specified in low-level design [13].
System testing reveals that the system works end-to-end in a production-like location to provide the business functions specified in the high-level design [13].
Acceptance testing is conducted by business owners, the purpose of acceptance testing is to test whether the system does in fact, meet their business requirements [13].
Regression Testing is the testing of software after changes has been made; this testing is done to make sure that the reliability of each software release, testing after changes has been made to ensure that changes did not introduce any new errors into the system [13].
Alpha Testing Usually in the existence of the developer at the developer's site will be done.
Beta Testing Done at the customer's site with no developer in site.
Functional Testing is done for a finished application; this testing is to verify that it provides all of the behaviors required of it [13].

Table1. Testing Spectrum

Testing Type	Opacity	Specification	Who will do this testing?	General Scope
Unit	White Box Testing	Low-Level Design Actual Code structure	Generally Programmers who write code they test	For small unit of code generally no larger than a class
Integration	White & Black Box Testing	Low and High Level Design	Generally Programmers who write code they test	For multiple classes

Functional	Black Box Testing	High Level Design	Independent Testers will Test	For Entire product
System	Black Box Testing	Requirements Analysis phase	Independent Testers will Test	For Entire product in representative environments
Acceptance	Black Box Testing	Requirements Analysis Phase	Customers Side	Entire product in customer's environment
Beta	Black Box Testing	Client Adhoc Request	Customers Side	Entire product in customer's environment
Regression	Black & White Box Testing	Changed Documentation High-Level Design	Generally Programmers or independent Testers	This can be for any of the above

2.2. Functional Technique and Structural Technique

The information flow of testing techniques is shown in Figure 1. Moreover testing involves the outline of proper inputs, implementation of the software over the input, and the analysis of the output. The "Software Configuration" contains requirements specification, design specification, source code, and so on. The "Test Configuration" includes test cases, test plan and procedures, and testing tools [6] [7]. Grounded on the testing information flow, a **testing technique** stipulates the strategy used in testing to select input test cases and analyse test results. Various techniques reveal different quality aspects of a software system, and there are two major categories of testing techniques, functional and structural [6] [7].

Functional Testing: the software program or system under test is observed as a "**black box**". The choice of test cases for functional testing is based on the **requirement** or **design specification** of the software entity under test. Examples of expected results sometimes are called **test oracles**, include requirement/design specifications, hand calculated values, and simulated results. Functional testing mainly focus on external **behaviour** of the software entity [6] [7].

Structural Testing: the software entity is viewed as a "**white box**". The selection of test cases is based on the implementation of the software entity. The goal of selecting such test cases is to cause the execution of specific spots in the software entity, such as specific statements, program branches or paths. The expected results are evaluated on a set of coverage criteria. Examples of coverage criteria include path coverage, branch coverage, and data-flow coverage. Structural testing highlights on the internal structure of the software entity [18].

2.3. Scope of the Study

In articles [6] [7] [29] author's provided the different testing technique but not able to explain elaborately and not covered all testing techniques together, In this paper, we focus on all Functional and Structural Testing Techniques and we examine different examples for each and every testing technique, and we consider a case of driver's Insurance premium calculation, and we provide a detail study of both black and white box testing techniques.

- Scope covers the following testing techniques as shown in Figure1

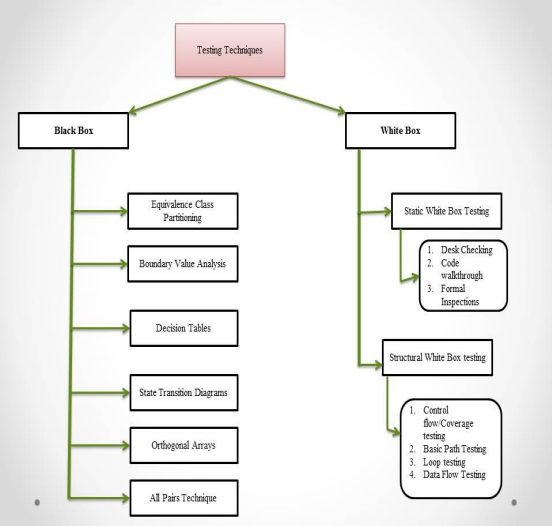

Figure1. Testing Techniques

2.4 Research Questions:

RQ1: What are the different Black box and White box testing techniques?
RQ 1.1: Usefulness of testing techniques?

3. RESEARCH METHODOLOGY

A. Literature Review

The literature review was performed according to the guidelines proposed by B. Kitchenham's [30] which were adopted for searching different testing techniques from literature review. Over all 29 articles was found those are most relevant to our literature study. Among those articles 5 are grey literature articles and 4 are book related information. Over all we find 12 Journals and 8 conference papers. The database was taken in such a manner that it covers most of the journals as well as conference papers. All the articles are related to testing techniques. In 29 articles authors speak about testing techniques in one another manner. The majority of articles mainly focus on case studies, theoretical reports, literature study, experience reports and field studies.

3.1 Data Retrieval-Quantitative Study

In order to retrieve the data from literature studies we framed the search terms

Search

We used the following Boolean search string to ensure we captured a wide variety of papers related to black and white box testing techniques in software testing. Over all we cover three databases and we got total set of 1954 articles without any refinement. We limit language as English.

Search terms:

For IEEE and Engineering village we use the following search terms separately for black box and white box testing techniques.

- IEEE: "Black box" and "Software testing" Results 197
  "White box" and "software testing" Results 90
- Engineering Village: "Black box" and "Software testing" Results 410
  "White box" and "software testing" Results 191
- Science Direct : (Black box) AND (test*) Results 627
  (White box) AND (test*) Results 415

Note: here we did further refinement and consider only software and computer science papers. Moreover we considered a few papers from Google scholar and those are purely Grey literature papers.

Selection Criteria:

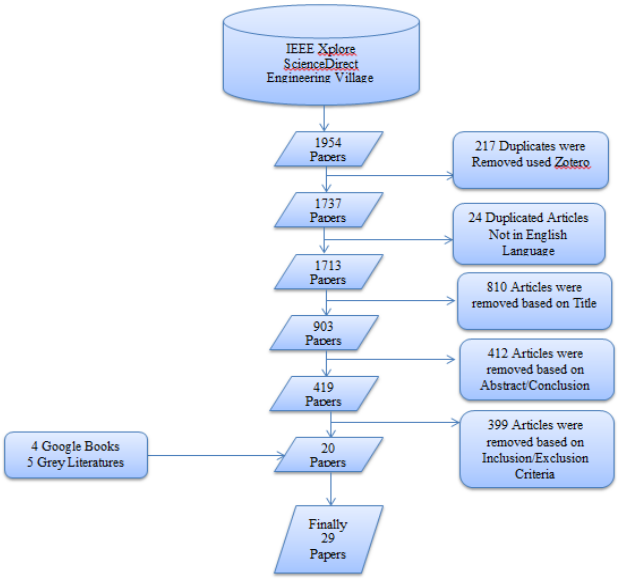

Figure2. Selection Criteria

4. DIFFERENT TESTING TECHNIQUES

4.1 Qualitative data collection

Functional Testing: the software program or system under test is viewed as a "**black box**". Black Box Testing: it is testing based on the requirements specifications and there is no need to examining the code in black box testing. This is purely done based on customers view point only tester knows the set of inputs and predictable outputs. Black box testing is done on the completely finished product [6] [7].

Why Black Box Testing:

Black box testing plays a significant role in software testing, it aid in overall functionality validation of the system. Black box testing is done based on customers' requirements-so any incomplete or unpredictable requirements can be easily identified and it can be addressed later. Black box testing is done based on end user perspective. The main importance of black box testing it handles both valid and invalid inputs from customer's perspective **[7].**

When Black Box Testing:

Black box testing is done from beginning of the software project life cycle. All the testing team members need to be involved from beginning of the project. During black box testing testers need to be involved from customers' requirements gathering and analysis phase. In the design phase test data and test scenarios need to be prepared **[7].**

Advantages:

The main advantage of black box testing is that, testers no need to have knowledge on specific programming language, not only programming language but also knowledge on implementation. In black box testing both programmers and testers are independent of each other. Another advantage is that testing is done from user's point of view. The significant advantage of black box testing is that it helps to expose any ambiguities or inconsistencies in the requirements specifications.

Black box testing techniques are [29]:

- Equivalence Class Partitioning
- Boundary Value Analysis
- Decision Tables
- State Transition Diagrams (or) State Transition Diagrams
- Orthogonal Arrays
- All Pairs Technique

4.1.1 Equivalence Class Partitioning

Equivalence class testing is based upon the assumption that a program's input and output domains can be partitioned into a finite number of (valid and invalid) classes such that all cases in a single partition exercise the same functionality or exhibit the same behaviour [20] [29]. The partitioning is done such that the program behaves in a similar way to every input value belonging to an equivalence class. Test cases are designed to test the input or output domain partitions. Equivalence class is determined by examining and analysing the input data range. Only one test case from each partition is required, which reduces the number of test cases necessary to achieve functional coverage [20] [29]. The success of this approach depends upon the tester being able to identify partitions of the input and output spaces for which, in reality, cause distinct sequences of program source code to be executed [l].

Equivalence Class Partitioning- Test Cases [20]**:**

The Steps for creating test cases are as follows

a. define the equivalence classes .

b. Write the initial test case that cover as many as valid equivalence classes as possible.

c. Continue writing test cases until all of the valid equivalence classes have been included.

d. finally write one test case for each invalid class.

Equivalence Class Partitioning- Advantages:

a. It eradicates the need for exhaustive testing, which is not feasible.

b. One of the advantages of equivalence class portioning is; it enables a tester to cover a large domain of inputs or outputs with a smaller subset selected from an equivalence class.

c. It allows a tester to select a subset of test inputs with a high chance of identifying a defect.

Equivalence Class Partitioning- Limitations:

a. One of the limitations of this technique is that it makes the assumption that the data in the same equivalence class is processed in the same way be the system **[7]** [20].

b. Equivalence partitioning is not a stand-alone method to determine test case. It has to be supplemented by ***boundary value analysis*** **[7]** [20].

4.1.2 Boundary Value Analysis

Boundary value analysis is performed by creating tests that exercise the edges of the input and output classes identified in the specification. Test cases can be derived from the **'boundaries'** of equivalence classes. Typically programming errors occur at the boundaries of equivalence classes are known as "**Boundary Value Analysis"**. Generally some time programmers fail to check special processing required especially at boundaries of equivalence classes. A general example is programmers may improperly use < instead of <=. The choices of boundary values include above, below and on the boundary of the class **[7]** [20] [29].

Boundary Value Analysis- Limitations [20]**:**

a. One of the limitations of boundary value analysis is it cannot be used for Boolean and logical variables.

b. Cannot estimate boundary analysis for some cases such as countries.

c. Not that useful for strongly-typed languages.

Table2. For Appendix-A Deriving Equivalence Class Partitioning and Boundary Value Analysis for Driver insurance premium

Conditions Parameter	Value	Test case1	Test case2	Test case3	Test case4	Test case5	Test case6
Age Valid Group	Integer 1-65	20	30	47	-	-	-
Age Invalid Group	Invalid Integer <1, >70				-2	70	12.1
	Decimal Number						
	Calculation Operators +,-,*,\,%				\	+	%
	(!,$,^,#,@, other special symbols				!	#	@
Gender Valid Group	String="female" or "male"	Female	Male	female			
Gender Invalid Group	a-e, A-E,G-L, g-l, n-z, N-Z, 0-9				K	L	M
Married Valid	True or False	T	F	T			
Invalid Married					&	$	@

4.1.3 Decision Tables

Decision tables are human readable rules used to express the test experts or design experts knowledge in a compact form [14]. Decision Tables can be used when the outcome or the logic involved in the program is based on a set of decisions and rules which need to be followed. Decision table mainly consists of four areas called the condition stub, the condition entry, the action stub and finally action entry [2] [19] [29].

Decision Tables-Approach [29]:

The Steps for using Decision Table testing are as given below:

Step1: Analyse the given test inputs or requirements and list out the various conditions in the decision table.

Step2: Calculate the number of possible combinations (Rules).

Step3: Fill Columns of the decision table with all possible combinations (Rules).

Step4: Find out Cases where the values assumed by a variable are immaterial for a given combination. Fill the same by "Don't care" Symbol.

Step5: For each of the combination of values, find out the action or expected result.

Step6: Create at least one Test case for each rule. If the rules are binary, a single test for each combination is probably sufficient. Else if a condition is a range of values, consider testing at both the low and high end of range.

Example:

Consider bank software responsible for debiting from an account. The relevant conditions and actions are:

C1: The account number is correct

C2: The Signature matches

C3: There is enough money in the account

A1: Give money

A2: Give Statement indicating insufficient funds

A3: Call vigilance to check for fraud!

For the above situation the decision table for bank software consists of:

Input:

C1: Account No: Correct, Incorrect
C2: Signature: Match, not match
C3: Enough money: Yes, No
Outputs:-
A1: Give Money
A2: Give Statement of insufficient funds
A3: Call vigilance

Now Rules are:
A1 when correct account no,
A2 when signature matched
A3 when sufficient money

Table3. Decision table for Bank account

Condition Entries	Account No.	Signature	Money	Actions
1	Correct	Match	Yes	A1
2	Correct	Match	No	A2
3	Correct	Not match	Yes	?
4	Correct	Not match	No	?
5	Incorrect	Match	Yes	A3
6	Incorrect	Match	No	?
7	Incorrect	Not match	Yes	?
8	Incorrect	Not match	No	?

? : There are no rules define for rest of the possibilities

4.1.4 State Transition Diagrams (or) State Graphs

State Graph is an excellent tool to capture certain types of system requirements and document internal system design. When a system must remember what happened before or when valid and invalid orders of operation exists, and then state transition testing could be used. This state graphs are used when system moves from one state to another state. State graphs are represented with symbols, circle is used to represent state, arrows are used to represent transition, and event is represented by label on the transition. Thus from the starting state to the end state the various transition and routes are represented in the form of a transition diagram as mentioned [21][22] [28].

Example:

Here we have to model the starship Enterprise. It has three impulse drive settings: drive (d), neutral (n) and reserve (r) . The ship itself has three possible states: such as moving forward (F), moving backward (B) and stopped (S). The combinations of which impulse thrusters are firing and how the ship moves create nine states:

.dF	.nF	.rF
.dS	.nS	.rS
.dB	.nB	.rB

The impulse driver thruster control requires that you go through neutral to get to drive or reverse. All thrusters are turned off in neutral: d< >n< >r. The possible inputs are: d>d, r>r, n>n, d>n, n>d, n>r, and r>n.

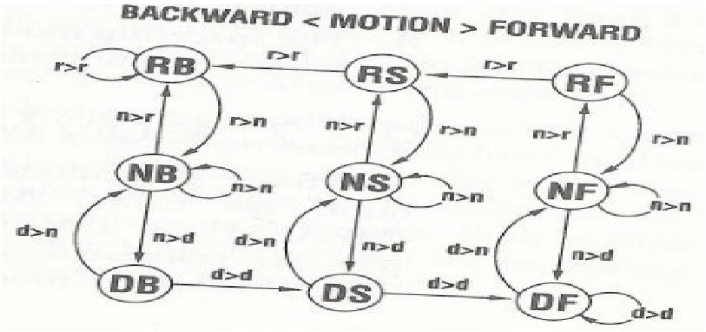

Figure3. State graph

Table4. State Table

STATE	r>r	r>n	n>n	n>r	n>d	d>d	d>n	r>d	d>r
RB	RB	NB							
RS	RB	NS							
RF	RS	NF							
NB			NB	RB	DB				
NS			NS	RS	DS				
NF			NF	RF	DF				
DB						DS	NB		
DS						DF	NS		
DF						DF	NF		

4.1.5 Orthogonal Arrays

Orthogonal Array Testing Strategy (OATS) is a systematical, statistical way of testing pair-wise interactions by deriving suitable small set of test cases from a large number of scenarios. The testing strategy can be used to reduce the number of test combinations and provide maximum coverage with a minimum number of test cases. OATS utilizes an array of values representing variable factors that are combined pair-wise rather than representing all combinations of factors and levels.

Example for OATS:

Here we have considered 3 parameters named as – A, B, and C and it has positive values as 1, 2 and 3. Testing all combinations of the 3 parameters would involve executing a total of 27 test cases. Generally while programming works start a fault will mostly occurs for two parameters, not for three. In this case the fault may occur for each of the 3 test cases as: A=1, B=1, C=1, A=1, B=1, C=2, and A=1, B=1, C=3. The usefully of OATS is no need to run all the 27 test cases, only 9 test cases are enough to test. The 9 scenarios outlined in Table5 (on the next screen) address all possible pairs within the three parameters.

Table5. Pair-wise Combination of Parameters- Sample array

	A	B	C
1	1	1	3
2	1	2	2
3	1	3	1
4	2	1	2
5	2	2	1
6	2	3	3
7	3	1	1
8	3	2	3
9	3	3	2

All possible pairwise combinations between parameters A and B, B and C, and C and A are displayed in Table 5 here nine scenarios provide the same coverage as executing all 27 scenarios. This same concept is applied to more complex scenarios where testing an application might require 10,000+ test cases and utilizing OATS, it can be reduced significantly in the number of test scenarios, such as down to 1,000 or less test cases to execute.

4.1.6 All Pairs Testing

This is an accepted technique for verifying a finite number of parameters with a finite number of values and keeping the number of test cases reasonable.

4.2 Structural Testing:

Structural Testing: the software unit is seen as a "**white box**". The choice of test cases is grounded on the **implementation** of the software entity. Design test cases that test the internal functioning of the software from the developer's perspective, white box testing mainly focus on internal logic and structure of the code. White-box is done when the programmer has techniques full knowledge on the program structure. With this technique it is possible to test every branch and decision in the program. When the internal structure is known it is interesting to look at different coverage criteria. One of the crucial one is decision coverage. The test is precise only if the tester recognizes what the program is supposed to do. The tester can then see if the program separates from its intended goal [6] [18] [29].

Why and When White-Box Testing:

White box testing is mainly used for detecting logical errors in the program code. It is used for debugging a code, finding random typographical errors, and uncovering incorrect programming assumptions [29].

White box testing is done at low level design and implementable code. It can be applied at all levels of system development especially Unit, system and integration testing. White box testing can be used for other development artefacts like requirements analysis, designing and test cases [18].

White box testing techniques are:

1. **Static white box testing**
   a. Desk checking
   b. Code walkthrough
   c. Formal Inspections

2. **Structural White box testing**
   a. Control flow/ Coverage testing
   b. Basic path testing
   c. Loop testing
   d. Data flow testing

4.2.1 Static white box testing

Static white box testing which involves only the source code of the product and not the binaries or executable, static white box testing will be done before the code is executed or completed. For static white box testing only selected peoples are involved to find out the defects in the code. The main aim of the static testing is to check whether the code is according to the Functional requirements, design, coding standards, all functionalities covered and error handling [18] [29].

4.2.1.1 Static white box testing: Desk checking

Desk checking is the primary testing done on the code. Static checking will be done by programmers before compiled or executed, if any error is find it is going to check by author and he will correct the code, in this process the code is compared with requirements specification or design to see that the designed code is according to client adhoc requests [29].

Advantages of Desk Checking:

In this process the authors who have knowledge in the programming language very well will be involved in desk checking testing. This can be done very quickly without much dependency on other developers or testers. The main advantages are defects detected in this stage are easily located and correct at same time.

4.2.1.2 Static white box testing: Code walkthrough

This testing is also known as technical code walkthrough, in this testing process a group of technical people go through the code. This is one type of semi-formal review technique. In Code walkthrough process a high level employees involved such as technical leads, database administrators and one or more peers. The people who involved in this technical code walkthrough they raise questions on code to author, in this process author explains the logic and if there is any mistake in the logic, the code is corrected immediately [29].

Advantages of Code walkthrough:

The main advantage of code walkthrough is that as a group of technical leads who have experience in programming look through the code, so the defects that are related to database or code can be easily identified. More over this process aid to ensure that program follows the proper coding standards [29].

4.2.1.3 Static white box testing: Formal Inspections or Fagan Inspection process.

Inspection is a formal, efficient and economical method of finding errors in design and code [15]. It's a formal review and aimed at detecting all faults, violations and other side effects. According to M. E. Fagan "A defect is an instance in which a requirement is not satisfied" [16]. Fagan inspection process is a structured process of finding defects in the provided source code.
Fagan inspection consists of following phases [17].

Planning: In planning phase Moderator arrange the availability of the right participants and arrange suitable meeting place and time.
Overview: All inspection participants are given documentation of design where it includes overall view of design and even detailed design in specific areas like paths, logic of code and so on.
Preparation: Using design documentation we tried to understand the design and its logic. Depending on historical inspections and its ranking or errors we tried to increase the error detection, so that more fruitful areas can be concentrated.
Inspection: Inspection meeting involves the process in which the code is inspected and defects are found. Defects are noted down and handed to the author.
Rework: Rework is done to fix the defect. The code is corrected by the author.
Follow up: Follow up is done by the moderator to ensure that the defect has been fixed correctly.

Table6. Minutes of Meeting format during Inspection process

No	Date of Meeting	Actions Items	Inspections Meeting		Reviews	
			Chaired by	Attended by	Start Time	End Time
1	12-02-10	Review	Moderator	Team	13:00	15:00
2	16-02-10	Review	Moderator	Team	10:00	12:00
3	19-02-10	Review	Moderator	Team	15:00	17:00
4	26-02-10	Review	Moderator	Team	13:00	15:00
5	28-02-10	Review	Moderator	Team	15:00	17:00

Example of Inspection Check list after finding Defects in a code:

Table7. Check list of Fagan Inspection process

Assigned Date	09/02/10	Finished Date	04/03/10	
Submitted to course coordinator			Wasif Afzal	
Subject	Inspection Summary Report for Code Review			
Priority	High:15	Low: 22	Medium : 40	
Number of Faults Found			77	
Re-Inspection Required (Y/N)			N	
Total Number of Inspection meetings			5	
Total Inspection time (Hrs)			10	
Moderator	Author	Recorder	Reader	Inspector
Srinivas	Maanasa	Matthias	Aparna	Sudhakar

4.2.2 Structural white box testing

Structural testing
Testing take into account the code, code structure, internal design and how they are coded.
Commonly used techniques for structural testing are [25] [29]:

1. **Control Flow/Coverage Testing**
   a. Statement coverage
   b. Branch coverage
   c. Decision/Condition Coverage
   d. Function Coverage

2. **Basis Path Testing**
   a. Flow Graph Notation
   b. Cyclomatic Complexity
   c. Deriving Test Cases
   d. Graph Matrices

3. **Loop Testing**
   a. Simple Loops
   b. Nested loops
   c. Concatenated loops
   d. Unstructured loop

4. **Data Flow testing**

4.2.2.1 Coverage Testing- Statement Coverage

In a Statement Testing each node or statements are traversed at least once, statement testing also known as node coverage [23] [24] [29].

Example: For *Appendix-A* here we provided the Flow chat for driver's insurance premium calculation and deriving the test cases for statement coverage. Each node is traversed at least once in this case:

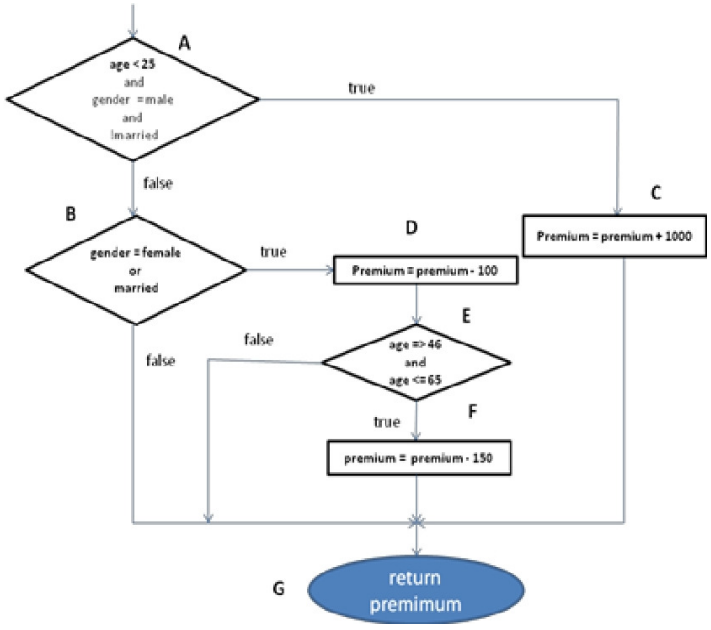

Figure4. Flow chat for drivers insurance premium calculation

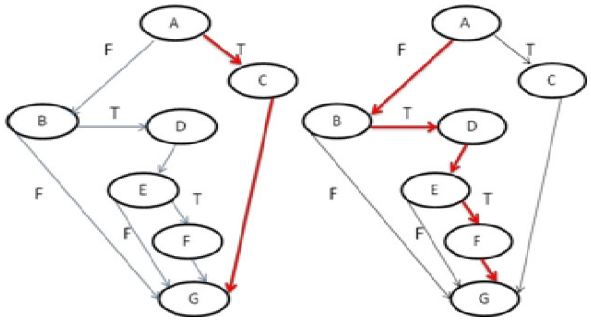

Figure5. Flow Graph for Statement Coverage

Note: Statement coverage must satisfy each node or statements are traversed at least once TC1: {age=20, gender="male", married = false}
Path P1: {A, C, G}
TC2: {age=60, gender="male", married=true}
Path P2: {A, B, D, E, F, G}.

Table8. Test cases for Statement Coverage

TNo	Condition	Input/Test Data	Expected Result
1	(age < 25) && (gender.equals("male")) && (!married)	age=20 gender="male" married=false	1300
2	(married) || gender.equals("female") (age>=46 && age<=65)	age=60 gender="male", married=true	50

4.2.2.2 Coverage Testing- Branch Coverage

In a branch testing each edge is traversed at least once. The outcome possibilities are at least true and false. The decision coverage or branch coverage is also known as Edge coverage [6].
Decision Coverage: For each decision, decision coverage measures the percentage of the total number of paths traversed through the decision point in the test. If each possible path has been traversed in a decision point, it achieves full coverage [6] [23] [24] [29].

From the Figure6, we are drawing the flow graph for Branch coverage for Appendix -A

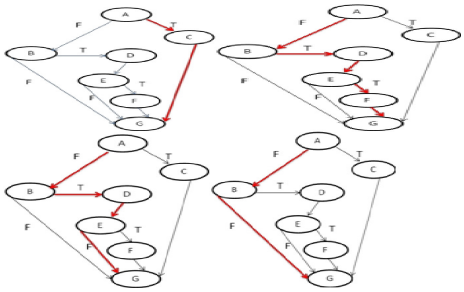

Figure6. Flow Graph for Branch Coverage

The Path for Branch coverage is:
TC1: {age=20, gender="male", married = false}
Branch Coverage Path P1: {A, C, G}
TC2: {age=60, gender="male", married=true}
Branch Coverage PathP2: {A, B, D, E, F, G}
TC3: {age = 35, gender = „female", married=false}
Branch Coverage Path P3: {A, B, D, E, G}
TC4: {age = 35, gender="male", married=false}
Branch Coverage Path P4: {A, B, G}.

Tno:	Condition	Input/Test Date	Expected Result
1	(age<25) && (gender.equals ("male")) && (!married)	Age=20, gender="male" , married = false	1300
2	(Married) || gender.equals ("female") (age>=46 && age<=65)	Age =60, gender="male", married = true	50
3	(married) || gender.equals ("female")	Age=35, gender='female', married=false	200
4	(married) || gender.equals("female")	Age=35, gender="male", married=false	300

Table 9: Test cases for Branch Coverage

4.2.2.3 Coverage Testing- Decision/Condition Coverage

Here condition is to verify that all condition expression within each branch will be tested (the true and the false condition of each sub-expression within the decision branch must be tested at least one time) [6] [18] [29].

Condition: A condition is a Boolean valued expression that cannot be broken down into simpler Boolean expressions [2]. A decision is often composed of several Boolean conditions [6].
From the Figure7, we are drawing the Control flow graph for Conditional coverage

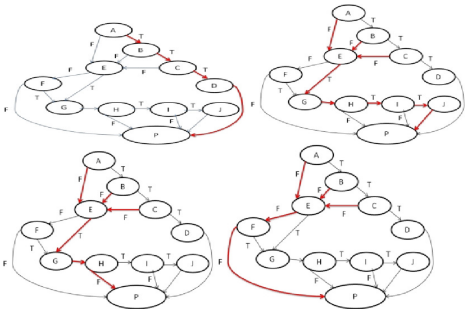

Figure7. Flow Graph for Condition Coverage

Deriving the test cases for pseudo code shown in Appendix A, the below Table 10-12 shows the test cases for each condition for condition coverage.

Table10. Conditional test cases for condition if (Age>=46 and age<=65)

TC No	Input/Test data	Age>=46	Age<=65	Result
1	Age=50	T	T	T
2	Age=30	F	T	F
3	Age=69	T	F	F
4	Age=40	F	F	F

Table11. Conditional test cases for condition if (married || gender.equals("female")

TCNo	Input/test data	gender.equals ("female")	married	Result
1	gender="female" married=true	T	T	T
2	gender="male", married=true	F	T	T
3	gender="female" married=false	T	F	T
4	gender="male", married=false	F	F	F

Table12. Conditional test cases for condition if(age<25 && gender.equals("male")&& !married)

TC No	Input/ test data	Age < 25	gender = Male	!married	Results
1	age=20, gender="male" married=false	T	T	T	T
2	age=41, gender="female", married=true	F	F	F	F
3	age=24, gender="male", married=true	T	T	F	F
4	age = 17, gender = ,,female" married=false	T	F	T	F
5	age=27, gender="male", ,married=false	F	T	T	F
6	age=15, gender="female", married= true	T	F	F	F
7	age=60, gender="female", married=false	F	F	T	F
8	age=46, gender="male", married=true	F	T	F	F

TC1 :{ age =20, gender="male", married=false}
Condition coverage path P1 :{ A, B, C, D, P}
TC2: {age=60, gender="male" married=true}
Condition coverage path P2: {A, E, G, H, I, J, P}
TC3: {age=35, gender="female", married=false}
Condition coverage path P3: {A, E, H, P}
TC4: {age=35, gender="male", married= false}
Condition coverage path P4: {A, E, F, P}

4.2.2.4 Coverage Testing- Function Coverage

In Function coverage, most programs are realized by calling a set of functions; in this requirements of a product are mapped to functions during the design phase. Each function is the smallest logical unit that does a specific functionality; there could be functions for computing the average of 10 numbers, inserting a row into the database, calculating the premium etc.
Tests are written to exercise each of the different functions in the code [26] [27].

4.2.2.5 Basic Path Testing- Flow Graph Notation

A control flow graph (CFG) is a directed graph that consists of two types: node and control flow. (1) Node: expressed by a labeled circle, representing one or more statements, decision condition, procedures of program, or convergence of two or more nodes. (2) Control flow: expressed by arc with arrow or line, can be called an edge, representing the program control flow. In a CFG, a node including condition is called a predicate node, and edges from the predicate node must converge at a certain node. Area defined by edges and nodes is referred to as region [8].
On a flow Graph:

a. In flow graphs the symbol arrows called as Edges that represent the flow of control
b. Circles are called as nodes, which represent one or more actions.
c. Areas bounded by edges and nodes called regions
d. A predicate node is a node containing a condition

Any procedural design can be translated into a flow graph.

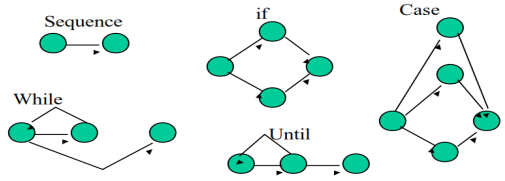

Figure8. Different control Flow Graph Notations

4.2.2.6 Basic Path Testing- Cyclomatic Complexity

The notion of cyclomatic complexity was presented by McCabe. Cyclomatic complexity is software metric that delivers a quantitative degree of the logical difficulty of a program. Cyclomatic Complexity **(CYC)** is derived as the number of edges of the program's control-flow graph minus the number of its nodes plus two times the number of its linked components. Cyclomatic complexity purely depend on the Control Flow Graph (CFG) of the program to be tested [5-9].

McCabe was also given calculation formula of complexity of a program structure.
V (G) = e – n + 2.

An alternate way of computing the cyclomatic complexity of a program from an inspection of its control flow graph is as follows:

V(G) = total number of non-overlapping bounded region + 1
i.e. V(G) =P+1
where P is the number of binary decision predicates.

Steps to arrive at Cyclomatic Complexity:
1. Draw a corresponding flow graph
2. Determine Cyclomatic complexity
3. Determine independent paths
4. Prepare test cases

Example of Cyclomatic complexity: Consider a Pseudo Code
1. do while not eof
2. Read Record
3. if record field 1 = 0
4. then process record
5. store in buffer;
6. increment counter
7. else if record field 2 = 0
8. then reset counter
9. else process record
10. store in file
11. end if
12. endif
13. enddo

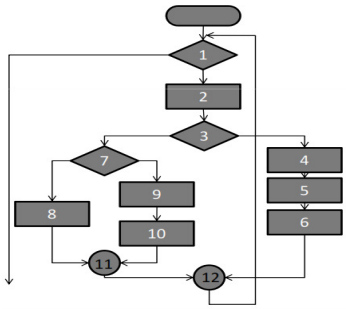

Figure9. Flow Chart for Pseudo Code

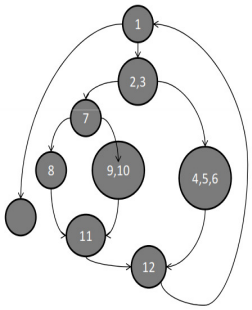

Figure 10. CYC for Pseudo Code

Determining the cyclomatic complexity for Figure 10
Path 1: 1-13
Path 2: 1-2-3-7-8-11-12-1-13
Path 3: 1-2-3-9-10-11-12-1-13
Path 4: 1-2-3-4-5-6-12-1-13

Is the path:
1-2-3-4-5-6-12-1-2-3-7-8-11-12-1-13 Independent?

4.2.2.7 Basic Path Testing- Deriving Test Cases

In this testing we have to use the design or code for drawing the corresponding control flow graphs, and have to determine the Cyclomatic complexity of the resultant flow graph V (G), after this we have to find the linearly independent paths. Finally prepare the test cases that will force execution of each path in the basis set [5] [6] [8] [9] [29].

For example if we have six independent paths, then we should have six test cases.
For each test case we need to define the input conduction and expected output.

4.2.2.8 Basic Path Testing- Graph Matrices

Graph matrices are used for derivation of flow graph and determination of a set of basis paths [29].

Software tools to do this can use a graph matrix

Graph matrix:

a. Is square with #sides equal to #nodes
b. Rows and columns correspond to the nodes.
c. Entries correspond to the edges.

1. Can associate a number with each edge entry.
2. Use a value of 1 to calculate the Cyclomatic Complexity
   a. For each row, sum column values and subtract 1
   b. Sum these totals and add 1
3. Interesting link weights are
   a. Probability that a link (edge) will be executed
   b. Processing time for traversal of a link
   c. Memory required during traversal of a link
   d. Resources required during traversal of a link

4.2.2.9 Loop Testing

1. Errors often occur near the beginnings and ends of the loop; in loop testing path has to cover at least once.
   a. Selects test paths according to the location of definitions and use of variables.
2. Test for loop (iterations)
   a. Loop testing
   b. Loop fundamental to many algorithms.
   c. Can define loops as simple, concatenated, nested , and unstructured

Simple Loops [29]:
Simple loops of size n:
Skip loop entirely;
Only one passes through loop;
Two passes through loop;
M passes through loop where, m<n.
(n-1), n and (n+1) passes through the loop,
Where n is the maximum number of allowable passes through the loop [29].
A typical Simple Testing loop is shown in figure11

Nested Testing [29]:
In this loop the number of probable tests increases as the number of levels of nesting grows.
Start with inner loop. Set all other loops to minimum values.
Conduct simple loop testing on inner loop.
Work outwards.
Continue until all loops are tested.
A typical Nested Testing loop is shown in figure11
Concatenated loop [29]:
If independent loops, use simple loop testing.
If dependent, treat as nested loops. A typical concatenated loop is shown in figure11
Unstructured loops [29]:
Don't test-redesign. A typical unstructured loop is shown in figure11

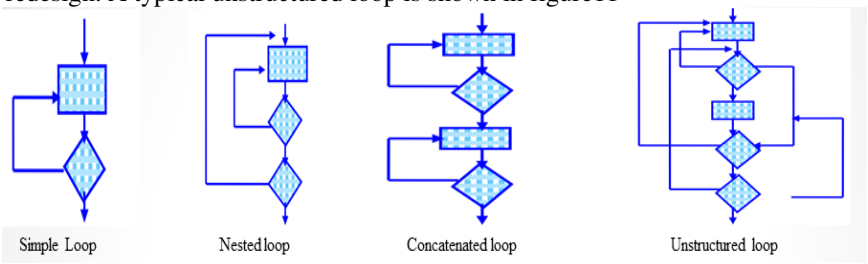

Figure11. Different Loop Testing's

4.2.2.10 Data Flow Testing

1. Data flow testing looks at the lifecycle of a particular piece of data (i.e. a variable) in an application [29].
2. Variables that contain data are created, used and killed (destroyed)
3. Concerned with the flow of data in the program
4. By looking at patterns of data usage, risky areas of code can be found and more test cases can be applied.
5. Dataflow testing uses control flow graphs to explore the unreasonable things that can happen to data.
6. Data can be used in 2 ways- Defined and used.

Data Flow Testing_ Technique [29]**:**

Data can be defined.

Example of defined data (Def)

```
Int x;
X= a+b;
Scanf(&x, &y);
X [i-1] = a+b;
```

Data can be used in a variable for performing some computations

Example of used data (Use)

```
A=X+2; (Data In X is being used for calculations)
Printf("value of x = ", x);
If(X<10)
```

Select paths through the program's control flow and test the status of data in each of these paths.
Pick enough paths to ensure that every data object has been initialized prior to use or all defined objects have been used for something.
All the def criteria (for definitions of all variables) must be exercised
All the use criteria of all variable definitions must be covered.

5. RESULTS

The data obtained from the literature are studied both quantitatively and qualitatively; shown in section 3 and section 4. Quantitative data mainly focus on number of articles obtained, their categorization, section criteria and articles based on years Etc. Whereas qualitative data mainly focus on testing techniques and their methods, models, advantages, case situations etc. We carefully retrieved the qualitative data from 29 articles and we find different testing techniques and we provided a few examples and case situations to explain in brief manner. From section 4 the qualitative data reveals that there are different testing techniques exists, from 29 articles each and every testing technique is considered carefully and we examine them and we proposed in this article. A few articles results are not so relevant to our study nevertheless we modified those data according to our studies and included those papers.

6. CONCLUSION AND FUTURE WORK

In this paper we proposed both black box and white box testing techniques. A few cases and examples are considered outside of this study, those cases and examples are only used to provide a clear explanation regarding testing techniques. In this study we cover almost all testing techniques related to both black box and white box, nonetheless our study has few limitations we don't not validate these techniques from industrial perspectives, we considered only from literature perspectives i.e. from state-of-art, our future work is to check the usability and usefulness of each and every technique from state-of-practice.

ACKNOWLEDGEMENTS

Heartfelt appreciations to Nidhra Bikshamaiah, Nidhra Kamalamma, Dondeti Nageswar rao, Dondeti Prasanna and all our family members for always being our side with love and affection and inspire us for higher education. And we would like to thanks Wasif Afzal, for his support to this study.

REFERENCES

[1] D. Shao, S. Khurshid, and D. E. Perry, "A Case for White-box Testing Using Declarative Specifications Poster Abstract," in Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION, 2007. TAICPART-MUTATION 2007, 2007, p. 137.

[2] M. Sharma and B. S. Chandra, "Automatic Generation of Test Suites from Decision Table - Theory and Implementation," in Software Engineering Advances (ICSEA), 2010 Fifth International Conference on, 2010, pp. 459 –464.

[3] M. R. Keyvanpour, H. Homayouni, and H. Shirazee, "Automatic Software Test Case Generation," Journal of Software Engineering, vol. 5, no. 3, pp. 91–101, Mar. 2011.

[4] M. Shaw, "What makes good research in software engineering?," International Journal on Software Tools for Technology Transfer (STTT), vol. 4, no. 1, pp. 1–7, 2002.

[5] H. Liu and H. B. Kuan Tan, "Covering code behavior on input validation in functional testing," Information and Software Technology, vol. 51, no. 2, pp. 546–553, Feb. 2009.

[6] P. Mitra, S. Chatterjee, and N. Ali, "Graphical analysis of MC/DC using automated software testing," in Electronics Computer Technology (ICECT), 2011 3rd International Conference on, 2011, vol. 3, pp. 145 –149.

[7] T. Murnane and K. Reed, "On the effectiveness of mutation analysis as a black box testing technique," in Software Engineering Conference, 2001. Proceedings. 2001 Australian, 2001, pp. 12 – 20.

[8] Z. Zhonglin and M. Lingxia, "An improved method of acquiring basis path for software testing," in Computer Science and Education (ICCSE), 2010 5th International Conference on, 2010, pp. 1891 – 1894.

[9] F. Lammermann, A. Baresel, and J. Wegener, "Evaluating evolutionary testability for structure-oriented testing with software measurements," Applied Soft Computing, vol. 8, no. 2, pp. 1018–1028, Mar. 2008.

[10] G. J. Myers, T. Badgett, T. M. Thomas, and C. Sandler, The Art of Software Testing. John Wiley & Sons, 2004.

[11] "Jess,"http://www.jessrules.com/, Accessed July 16, 2010

[12] J. H. Hayes and A. J. Offutt, "Increased software reliability through input validation analysis and testing," in Software Reliability Engineering, 1999. Proceedings. 10th International Symposium on, 1999, pp. 199 –209.

[13] P. Jorgensen, Software testing: a craftman's approach, CRC Press, 2002. p. 359.

[14] B. Beizer, Software Testing Techniques. Dreamtech Press, 2002.

[15] S. R. Rakitin, Software Verification and Validation for Practitioners and Managers. Artech House, 2001.

[16] T. Gilb, D. Graham, and S. Finzi, Software inspection. Addison-Wesley, 1993.

[17] M. E. Fagan, "Design and code inspections to reduce errors in program development," IBM Systems Journal, vol. 15, no. 3, pp. 182 –211, 1976.

[18] F. Saglietti, N. Oster, and F. Pinte, "White and grey-box verification and validation approaches for safety- and security-critical software systems," Information Security Technical Report, vol. 13, no. 1, pp. 10–16, 2008.

[19] S. Noikajana and T. Suwannasart, "Web Service Test Case Generation Based on Decision Table (Short Paper)," in Quality Software, 2008. QSIC '08. The Eighth International Conference on, 2008, pp. 321 –326.

[20] T. Murnane, K. Reed, and R. Hall, "On the Learnability of Two Representations of Equivalence Partitioning and Boundary Value Analysis," in Software Engineering Conference, 2007. ASWEC 2007. 18th Australian, 2007, pp. 274 –283.

[21] G. Scollo and S. Zecchini, "Architectural Unit Testing," Electronic Notes in Theoretical Computer Science, vol. 111, no. 0, pp. 27–52, Jan. 2005.

[22] L. Ran, C. Dyreson, A. Andrews, R. Bryce, and C. Mallery, "Building test cases and oracles to automate the testing of web database applications," Information and Software Technology, vol. 51, no. 2, pp. 460–477, Feb. 2009.
[23] S. Liu and Y. Chen, "A relation-based method combining functional and structural testing for test case generation," Journal of Systems and Software, vol. 81, no. 2, pp. 234–248, Feb. 2008.
[24] P. G. Frankl and E. J. Weyuker, "Testing software to detect and reduce risk," Journal of Systems and Software, vol. 53, no. 3, pp. 275–286, Sep. 2000.
[25] A. S. Boujarwah and K. Saleh, "Compiler test case generation methods: a survey and assessment," Information and Software Technology, vol. 39, no. 9, pp. 617–625, 1997.
[26] Y.-D. Lin, C.-H. Chou, Y.-C. Lai, T.-Y. Huang, S. Chung, J.-T. Hung, and F. C. Lin, "Test coverage optimization for large code problems," Journal of Systems and Software, vol. 85, no. 1, pp. 16–27, Jan. 2012.
[27] Y.-C. Huang, K.-L. Peng, and C.-Y. Huang, "A history-based cost-cognizant test case prioritization technique in regression testing," Journal of Systems and Software, vol. 85, no. 3, pp. 626–637, Mar. 2012.
[28] Kansomkeat, Supaporn, Rivepiboon, and Wanchai, "Automated-generating test case using UML statechart diagrams," Proceedings of the 2003 annual research conference of the South African institute of computer scientists and information technologists on Enablement through technology ,2003.
[29] http://www.internetjournals.net/journals/tir/2009/January/Paper%2006.pdf
[30] B. Kitchenham and S. Charters, "Guidelines for performing systematic literature reviews in software engineering," Version, vol. 2, 2007, pp. 2007–01.

APPENDIX-A

Assumption study for: Equivalence class partitioning, Boundary value analysis, Statement Coverage, Branch Coverage, Decision/Condition coverage.

1. The average cost of an insurance premium for drivers is €300, however, this premium can increase or decrease depending on three factors: Age, gender and marital status. Drivers that are below the age of 25, male and single face an additional premium increase of €1000. If a driver outside of this bracket is married or female their premium reduces by €100, and if there are aged between 46 and 65 inclusive, their premium goes down by another €150.

Pseudo-Code:

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

public class demo {
public static void main(String[] args)throws
IOException {
int age = 0;
boolean married = false;
String gender = null;
BufferedReader br = new BufferedReader(new
InputStreamReader(System.in));
try {
System.out.println("enter age:");
age = Integer.parseInt(br.readLine());
System.out.println("eneter married :");
married = Boolean.parseBoolean(br.readLine());
System.out.println("enter gender :");
gender = br.readLine();
}catch(Throwable e) {
e.printStackTrace();
}
System.out.println("enter age:" +age);
System.out.println("eneter married :" +married);
```

```
System.out.println("enter gender :" +gender);
perdetails (age, gender, married);
}
private static int perdetails(int age, String gender,
boolean married) {
int  premimum = 300;
if((age < 25) && (gender.equals("male")) &&
(!married)){
premimum = premimum + 1000;
}
else{
if (married || gender.equals("female")){
premimum = premimum - 100;
if((age >= 46) && (age <= 65)){
premimum = premimum - 150;
}
}
}
System.out.println("premimum" + premimum);
return premimum;
}
}
```

Authors

Srinivas Nidhra

Srinivas Nidhra received Master's Degree in computer science from Blekinge Institute of Technology, Karlskrona, Sweden. His research interests include global software engineering, IT security, programming languages and software testing.

Jagruthi Dondeti

Jagruthi Dondeti received Bachelors of Technology in Computer Science Engineering from Jawaharlal Nehru Technological University, Hyderabad, India. Her research interests include IT Security emphasis on cryptography and Network Security.