DEBUGGER SOFTWARE

By

Bhaskar Arora (051276)

Rohit katiyar (051277)

Anshul Sharma (051287)

MAY-2009

Submitted in Partial Fulfillment of the Degree of Bachelors of Technology

DEPARTMENT OF COMPUTER SCIENCE ENGINEERING & INFORMATION TECHNOLOGY

JAYPEE UNIVERSITY OF INFORMATION TECHNOLOGY-WAKNAGHAT

CANDIDATE'S DECLARATION

I hereby certify that the work which is being presented in the thesis entitled "**Debugger Software**" by "**Anshul Sharma(051287), Bhaskar Arora(051276)** and **Rohit Katiyar(051277)** in partial fulfillment of requirements for the award of degree of B.Tech. (C.S.E.) submitted in the Department of (Computer Science Engineering & Information Technology) at JAYPEE UNIVERSITY OF INFORMATION TECHNOLOGY is an authentic record of my own work carried out during a period from 18-07-2008 to 12-05-2009 under the supervision of Ms. Maneesha Srivastav. The matter presented in this thesis has not been submitted by me in any other University / Institute for the award of B.Tech. Degree.

[illegible]	[illegible]	[illegible]
Anshul Sharma	Bhaskar Arora	Rohit Katiyar
(051287)	(051276)	(051277)

This is to certify that the above statement made by the candidates is correct to the best of my knowledge

[illegible]

Ms. Maneesha Srivastav

(Project supervisor)

Department of Computer Science and Engineering,

Jaypee University of Information Technology,

Waknaghat, Solan-173215, India

ACKNOWLEDGMENT

We feel great pleasure in expressing our regards and gratitude to Ms. Maneesha Srivastav, Lecturer Computer Science Department , Jaypee University of Information Technology for giving us an opportunity to prepare a project for our final year.

We are indebted to her for bequeathing us her vast knowledge and generous guidance at every stage of our project. Her advice, suggestions and encouragement led us through this project presentation.

[illegible]	[illegible]	Katiyar
Anshul Sharma	Bhaskar Arora	Rohit Katiyar
(051287)	(051276)	(051277)

TABLE OF CONTENTS

TABLE OF CONTENTS

TABLE OF CONTENTS

TOPIC **PAGE NO.**

TABLE OF CONTENTS

TABLE OF CONTENTS

LIST OF FIGURES

FIGURES **PAGE NO.**

ABSTRACT

A **debugger** is a computer program that is used to test and debug other programs. The code to be examined might alternatively be running on an instruction set simulator (ISS), a technique that allows great power in its ability to halt when specific conditions are encountered but which will typically be much slower than executing the code directly on the appropriate processor.

Here we develop a debugger in which, we uses the java predefined libraries and classes to define the header files and functions for the input program (must be written in java) to be debug provided with interactive graphical user interface (GUI) , our java debugger has following features.

- Create a new file using create tool to open a new window.
- Interactive graphical user interface to save, run and close.
- Write a program to find the errors.
- Save the program in the disk drive where we should our project module.
- Compile the program using the runnable tool to check out the errors.
- Show the errors in the debugger output with the exact location of bugs in the input program.
- Save the errors so that user can check these errors later.
- Close the user interface.

CHAPTER 1

INTRODUCTION

1.1 What is Debugging?

In computers, debugging is the process of locating and fixing or bypassing bugs (errors) in computer program code or the engineering of a hardware device. To *debug* a program or hardware device is to start with a problem, isolate the source of the problem, and then fix it. A user of a program that does not know how to fix the problem may learn enough about the problem to be able to avoid it until it is permanently fixed. When someone says they've debugged a program or "worked the bugs out" of a program, they imply that they fixed it so that the bugs no longer exist.

Debugging is a necessary process in almost any new software or hardware development process, whether a commercial product or an enterprise or personal application program. For complex products, debugging is done as the result of the unit test for the smallest unit of a system, again at component test when parts are brought together, again at system test when the product is used with other existing products, and again during customer beta test , when users try the product out in a real world situation. Because most computer programs and many programmed hardware devices contain thousands of lines of code, almost any new product is likely to contain a few bugs. Invariably, the bugs in the functions that get most use are found and fixed first. An early version of a program that has lots of bugs is referred to as "buggy."

Debugging tools (called *debuggers*) help identify coding errors at various development stages. Some programming language packages include a facility for checking the code for errors as it is being written.

1.1.1 Origin of Debugging

There is some controversy over the origin of the term "debugging."

The terms "bug" and "debugging" are both popularly attributed to Admiral Grace Hopper in the 1940s. While she was working on a Mark II Computer at Harvard University, her associates discovered a moth stuck in a relay and thereby impeding operation, whereupon she remarked that they were "debugging" the system. However the term "bug" in the meaning of technical error dates back at least to 1878 and Thomas Edison (see software bug for a full discussion), and "debugging" seems to have been used as a term in aeronautics before entering the world of computers. Indeed, in an interview Grace Hopper remarked that she was not coining the term. The moth fit the already existing terminology, so she saved it.

The Oxford English Dictionary entry for "debug" quotes the term "debugging" used in reference to airplane engine testing in a 1945 article in the Journal of the Royal Aeronautical Society, Hopper's bug was found 9 September 1947. The term was not adopted by computer programmers until the early 1950s. The seminal article by Gill in 1951 is the earliest in-depth discussion of programming errors, but it does not use the term "bug" or "debugging". In the ACM's digital library, the term "debugging" is first used in three papers from 1952 ACM National Meetings. Two of the three use the term in quotation marks. By 1963, "debugging" was a common enough term to be mentioned in passing without explanation on page 1 of the CTSS manual.[6]

Kidwell's article *Stalking the Elusive Computer Bug* discusses the etymology of "bug" and "debug" in greater detail.

1.1.2 Tools

Debugging is, in general, a lengthy and tiresome task. The debugging skill of the programmer is probably the biggest factor in the ability to debug a problem, but the difficulty of software debugging varies greatly with the programming language used and the available tools, such as *debuggers*. Debuggers are software tools which enable the programmer to monitor the execution of a program, stop it, re-start it, set breakpoints, change values in memory and even, in some cases, go back in time. The term *debugger* can also refer to the person who is doing the debugging.

Generally, high-level programming languages, such as Java, make debugging easier, because they have features such as exception handling that make real sources of erratic behaviour easier to spot. In lower-level programming languages such as C or assembly, bugs may cause silent problems such as memory corruption, and it is often difficult to see where the initial problem happened. In those cases, memory debugger tools may be needed.

In certain situations, general purpose software tools that are language specific in nature can be very useful. These take the form of *static code analysis tools*. These tools look for a very specific set of known problems, some common and some rare, within the source code. All such issues detected by these tools would rarely be picked up by a compiler or interpreter, thus they are not syntax checkers, but more semantic checkers. Some tools claim to be able to detect 300+ unique problems. Both commercial and free tools exist in various languages. These tools can be extremely useful when checking very large source trees, where it is impractical to do code walkthroughs. A typical example of a problem detected would be a variable dereference that occurs *before* the variable is assigned a value. Another example would be to perform strong type checking when the language does not require such. Thus, they are better at locating likely errors, versus actual errors. As a result, these tools have a reputation of false positives. The old Unix *lint* program is an early example.

For debugging electronic hardware (e.g., computer hardware) as well as low-level software (e.g., BIOSes, device drivers) and firmware, instruments such as oscilloscopes, logic analyzers or in-circuit emulators (ICEs) are often used, alone or in combination. An ICE may perform many of the typical software debugger's tasks on low-level software and firmware.

1.1.3 Debugging Process

Print debugging is the act of watching (live or recorded) trace statements, or print statements, that indicate the flow of execution of a process.

Often the first step in debugging is to attempt reproduce the problem. This can be a non-trivial task, for example in case of parallel processes or some unusual software bugs. Also specific user environment and usage history can make it difficult to reproduce the problem.

After the bug is reproduced, the input of the program needs to be simplified to make it easier to debug. For example, a bug in a compiler can make it crash when parsing some large source file. However, after simplification of the test case, only few lines from the original source file can be sufficient to reproduce the same crash. Such simplification can be made manually, using a divide-and-conquer approach. The programmer will try to remove some parts of original test case and check if the problem still exists. When debugging the problem in GUI, the programmer will try to skip some user interaction from the original problem description and check if remaining actions are sufficient for bug to appear. To automate test case simplification, *delta debugging* methods can be used.

After the test case is sufficiently simplified, a programmer can use a debugger to examine program states (values of variables, the call stack) and track down the origin of the problem. Alternatively tracing can be used. In simple case, tracing is just a few print statements, which output the values of variables in certain points of program execution.

Remote debugging is the process of debugging a program running on a system different than the debugger. To start remote debugging, debugger connects to a remote system over a network. Once connected, debugger can control the execution of the program on the remote system and retrieve information about its state.

Post-mortem debugging is the act of debugging the core dump of process. The dump of the process space may be obtained automatically by the system, or manually by the interactive user. Crash dumps (core dumps) are often generated after a process has terminated due to an unhandled exception.

1.1.4 Anti-Debugging

Anti-debugging is "the implementation of one or more techniques within computer code that hinders attempts at reverse engineering or debugging a target process". The types of technique are:

- **API-based:** check for the existence of a debugger using system information.
- **Exception-based:** check to see if exceptions are interfered with.
- **Process and thread blocks:** check whether process and thread blocks have been manipulated.
- **Modified code:** check for code modifications made by a debugger handling software breakpoints.
- **Hardware- and register-based:** check for hardware breakpoints and CPU registers.
- **Timing and latency:** check the time taken for the execution of instructions.

Debugging can be inhibited by using one or more of the above techniques. There are enough anti-debugging techniques available to sufficiently protect software against most threats.

1.2 Software Bug

A **software bug** is an error, flaw, mistake, failure, or fault in a computer program that prevents it from behaving as intended (e.g., producing an incorrect or unexpected result). Most bugs arise from mistakes and errors made by people in either a program's source code or its design, and a few are caused by compilers producing incorrect code. A program that contains a large number of bugs, and/or bugs that seriously interfere with its functionality, is said to be **buggy**. Reports detailing bugs in a program are commonly known as **bug reports**, fault reports, problem reports, trouble reports, change requests, and so forth.

1.3 Common types of computer bugs

➢ Conceptual error (code is syntactically correct, but the programmer or designer intended it to do something else)

Maths bugs

- Division by zero
- Arithmetic overflow or underflow
- Loss of arithmetic precision due to rounding or numerically unstable algorithms

Logic bugs

- Infinite loops and infinite recursion

Syntax bugs

- Use of the wrong operator, such as performing assignment instead of equality test. In simple cases often warned by the compiler; in many languages, deliberately guarded against by language syntax

Resource bugs

- Null pointer dereference
- Using an uninitialized variable
- Off by one error, counting one too many or too few when looping
- Access violations
- Resource leaks, where a finite system resource such as memory or file handles are exhausted by repeated allocation without release.
- Buffer overflow, in which a program tries to store data past the end of allocated storage. This may or may not lead to an access violation. These bugs can form a security vulnerability.
- Excessive recursion which though logically valid causes stack overflow

Co-programming bugs

- Deadlock
- Race condition
- Concurrency errors in Critical sections, Mutual exclusions and other features of concurrent processing. Time-of-check-to-time-of-use (TOCTTOU) is a form of unprotected critical section.

Teamworking bugs

- Unpropagated updates; e.g. programmer changes "myAdd" but forgets to change "mySubtract", which uses the same algorithm. These errors are mitigated by the Don't Repeat Yourself philosophy.

- Comments out of date or incorrect: many programmers assume the comments accurately describe the code
- Differences between documentation and the actual product

1.4 What is a debugger?

A **debugger** is a computer program that is used to test and debug other programs. The code to be examined might alternatively be running on an *instruction set simulator* (ISS), a technique that allows great power in its ability to halt when specific conditions are encountered but which will typically be much slower than executing the code directly on the appropriate processor.

When the program crashes, the debugger shows the position in the original code if it is a **source-level debugger** or **symbolic debugger**, commonly seen in integrated development environments. If it is a **low-level debugger** or a **machine-language debugger** it shows the line in the disassembly. (A "crash" happens when the program cannot continue because of a programming bug. For example, perhaps the program tried to use an instruction not available on the current version of the CPU or attempted access to unavailable or protected memory.)

Typically, debuggers also offer more sophisticated functions such as running a program step by step **(single-stepping)**, stopping **(breaking)** (pausing the program to examine the current state) at some kind of event by means of breakpoint, and tracking the values of some variables. Some debuggers have the ability to modify the state of the program while it is running, rather than merely to observe it.

The importance of a good debugger cannot be overstated. Indeed, the existence and quality of such a tool for a given language and platform can often be the deciding factor in its use, even if another language/platform is better-suited to the task. However, it is also important to note that software can (and often does) behave differently running under a debugger than normally, due to the inevitable changes the presence of a debugger will make to a software program's internal timing. As a result, even with a good debugging tool, it is often very difficult to track down runtime problems in complex multi-threaded or distributed systems.

The same functionality which makes a debugger useful for eliminating bugs allows it to be used as a software cracking tool to evade copy protection, digital rights management, and other software protection features.

Most mainstream debugging engines, such as gdb and dbx provide console-based command line interfaces. Debugger front-ends are popular extensions to debugger engines that provide IDE integration, animation, and visualization features.

1.5 Difference between debugger and compiler?

Compiler is a special software or set software that translates source code from one computer language to another. In most cases from high-level programming language to low-level programming language.

Debugger is another program that is used for testing and debugging purpose of other programs. Most of the time it is using to analyze and examine error conditions in application. It will be able to tell where exactly in your application error occurred, give you all needed addressed of variables, variable representation in code, stack trace and all other low level and some times high level information. Debugger allows you to run your code step by step, make breakpoints in application (to examine specific parts of code) and it can halt application after it crashed in order to examine the problem.

As you can see this is completely different software, but almost always used together. Most of the times Debugger and Compiler are integrated into IDE.

1.6 <u>Debugger front-end</u>

In computer programming, some of the most capable and popular debugger programs implement only a simple command line interface (CLI) — often to maximize portability and minimize resource consumption. However, many programmers find that debugging via a graphical user interface (GUI) is easier and more productive. This is the *raison d'être* for GUI **debugger front-ends**, which are programs that allow users to monitor and control subservient CLI-only debuggers via a more intuitive graphical interface. Some debugger front-ends are designed to be compatible with a variety of CLI-only debuggers, while others are targeted at one specific debugger.

1.7 <u>Debugger in parallel</u>

Developing concurrent applications is a difficult task. Beside the bugs a programmer can make in the sequential parts of his program, he can also produce bugs related to concurrency and thread synchronization, like deadlocks, livelocks, or not guaranteed mutual exclusion. One approach to prevent programmers from such bugs is formal verification, like model checking or theorem proving, combined with a formal system specification and refinement to obtain the executable program. However, in practice, this approach does not scale well for larger concurrent applications. The communication (and synchronization) part of the application may interfere with the computation. Often, it is even the case that concurrency is a means to express algorithms. In these cases, formal verification is often not applicable, e.g. because of infinite state spaces or state space explosion problems. As a consequence, validating an implementation is in many cases restricted to testing.

1.8 List of popular debuggers with features

1 DDD (Data Display Debugger)

- DDD has GUI front-end features such as viewing source texts and its interactive graphical data display, where data structures are displayed as graphs.
- A simple mouse click dereferences pointers or views structure contents, updated each time the program stops. Using DDD, you can reason about your application by watching its data, not just by viewing it execute lines of source code.
- DDD is used primarily on Unix systems, and its usefulness is complemented by many open source plug-ins available for it.

2 GNU Debugger (GDB)

- GDB offers extensive facilities for tracing and altering the execution of computer programs. The user can monitor and modify the values of programs' internal variables, and even call functions independently of the program's normal behavior.
- GDB target processors (as of 2003) include: Alpha, ARM, AVR, H8/300, System/370, System 390, X86 and its 64-bit extension X86-64, IA-64 "Itanium", Motorola 68000, MIPS, PA-RISC, PowerPC, SuperH, SPARC, and VAX.
- Lesser-known target processors supported in the standard release have included A29K, ARC, ETRAX CRIS, D10V, D30V, FR-30, FR-V, Intel i960, M32R, 68HC11, Motorola 88000, MCORE, MN10200, MN10300, NS32K, Stormy16, V850, and Z8000. (Newer releases will likely not support some of these.)
- GDB has compiled-in simulators for even lesser-known target processors such like M32R or V85.

3 Microsoft Visual Studio Debugger

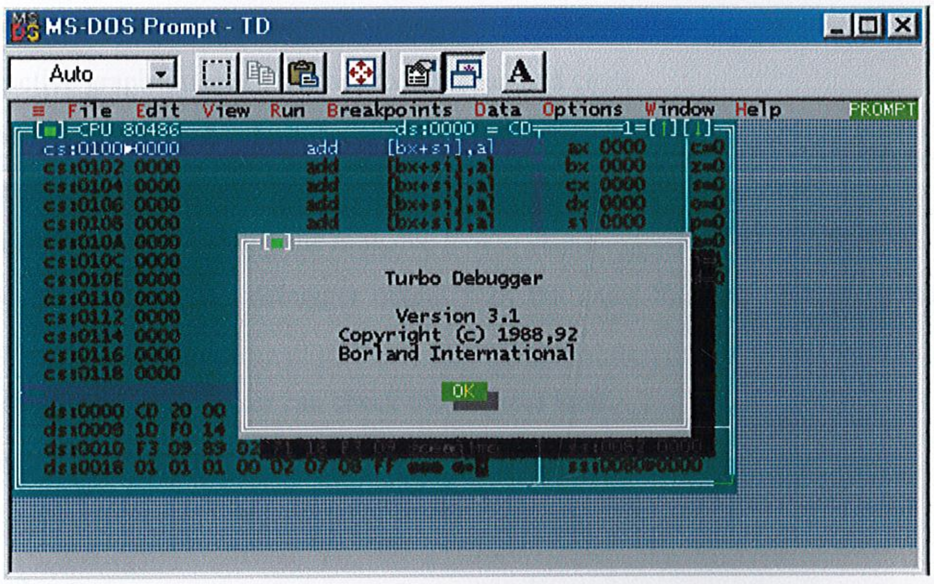
- Full symbol and source integration.
- Attaching and detaching to and from processes.
- Integrated debugging across programs written in both .NET and native Windows languages (calls from C# to C++, for example).
- Remote machine debugging.
- Full support for C++, including templates and the standard library
- Edit and continue support, enabling source code change and recompilation without having to restart the program (32 bit applications only).
- Local and remote debugging of SQL stored procedures on supported versions of Microsoft SQL Server

4 Eclipse

The Eclipse SDK includes the Eclipse Java Development Tools, offering an IDE with a built-in incremental Java compiler and a full model of the Java source files. This allows for advanced refactoring techniques and code analysis. The IDE also makes use of a *workspace*, in this case a set of metadata over a flat filespace allowing external file modifications as long as the corresponding workspace "resource" is refreshed afterwards. The Visual Editor project *(discontinued since June 30, 2006)* allows interfaces to be created interactively, thus allowing Eclipse to be used as a RAD tool.

5 Turbo Debugger

The final version of Turbo Debugger came with several versions of the debugger program: TD.EXE was the basic debugger; TD286.EXE ran in protected mode, and TD386.EXE was a virtual debugger which used the TDH386.SYS device driver to communicate with TD.EXE. The TDH386.SYS driver also added breakpoints supported in hardware by the 386 and later processors to all three debugger programs. The only real difference between TD386 and the other two debuggers was that TD386 allowed some extra breakpoints that the other debuggers did not (I/O access breaks, ranges greater than 16 bytes, and so on). There was also a debugger for Windows 3 (TDW.EXE). Remote debugging was supported.



```
[■]=CPU 80486==================ds:0000 = CD=========1=[↑][↓]=
 cs:0100►0000          add    [bx+si],al      ax 0000   c=0
 cs:0102 0000          add    [bx+si],al      bx 0000   z=0
 cs:0104 0000          add    [bx+si],al      cx 0000   s=0
 cs:0106 0000          add    [bx+si],al      dx 0000   o=0
 cs:0108 0000          add    [bx+si],al      si 0000   p=0
 cs:010A 0000
 cs:010C 0000
 cs:010E 0000
 cs:0110 0000
 cs:0112 0000
 cs:0114 0000
 cs:0116 0000
 cs:0118 0000

 ds:0000 CD 20 00
 ds:0008 10 F0 14
 ds:0010 F3 09 89 02
 ds:0018 01 01 01 00 02 07 08 FF                ss:0080 0000
```

```
Turbo Debugger

Version 3.1
Copyright (c) 1988,92
Borland International
```

(Fig1.6: Snapshot of existing turbo debugger).

1.9 Problem Statement

To design a debugger in java by using the JAVA predefined libraries in order to debug other java programs with the help of interactive graphical user interface (GUI).

1.10 Proposed Solution

A possible solution to the persisting problem is our debugger in which we uses the java predefined libraries and classes to define the header files and functions for the input program (must be written in java) to be debug provided with interactive graphical user interface (GUI) , our java debugger has following features.

- Create a new file using create tool to open a new window.
- Interactive graphical user interface to save, run and close.
- Write a program to find the errors.
- Save the program in the disk drive where we should our project module.
- Compile the program using the runnable tool to check out the errors.
- Show the errors in the debugger output with the exact location of bugs in the input program.
- Save the errors so that user can check these errors later.
- Close the user interface.

1.11 Chapter Layout

- In the Chapter 1 i.e. **Introduction**, it explains the introduction of debugging, debugger, features, type of errors, problem statement.
- In the Chapter 2 i.e. **Literature Survey**, it include the background and research papers which had been used to develop our debugger.
- In the Chapter 3 i.e. **Features** Incorporated, it explains all the features provided.
- In the Chapter 4 i.e. **Implementation**, it explains all the implemented features and
- In the Chapter 5 i.e. **Conclusion**, it concludes the work of the whole thesis.

CHAPTER 2

LITERATURE SURVEY

2.1 Interactive Program Debugging

The computer software industry is in a period of massive growth that shows no signs of diminishing as new markets are continuously identified and approached. Currently, the tools and techniques to aid in the development and debugging of software is extremely limited and has not kept pace with the needs of the software industry. Improvements have been made in the software design processes (through formal object oriented analysis and design techniques), performance tuning, and debugging of communication patterns. The basic debugging techniques and process, however, have remained largely the same with little advancement. What few tools and techniques are available to aid in the debugging process are rarely used by programmers due to high learning curves, slow generation of results, lack of perceived benefits, and the inability of the tool to grab the programmer's focus and maintain it. We are investigating debugging techniques based on interactive computational steering and visual representation geared towards improving the debugging cycle in cognitively based ways such that users will be attracted to the tools sufficiently to avail themselves of the new capabilities they provide.

Use of the visualization and data steering techniques aid users in identifying where errors are being generated or provide a verification mechanism for showing that the application does in fact work as specified under various configurations and exceptional conditions. The instruction level steering allows users to modify the behavior of the algorithm from within the debugging environment. Using fine grain instrumentation techniques provides data about the application and correspondingly capabilities in the debugging environment not otherwise available. With the instrumentation level steering we are providing capabilities that are generally not provided in even the best debugger.

2.2 JavaTA: A Logic-based Debugger for Java

This paper shows some of the benefits of applying logic programming techniques in the debugging of object-oriented programs. Debugging object-oriented programs has traditionally been a procedural process in that the programmer has to proceed step-by-step and object-by object in order to uncover the cause of an error. In this paper, we propose a logic-based approach to the debugging of object-oriented programs in which debugging data can be collected via higher level logical queries. We represent the salient events during the execution of a Java program by a logic database, and implement these queries as logic programs. Such an approach allows us to answer a number of useful and interesting queries about a Java program. To illustrate our approach, note that a crucial aspect of program understanding is observing how variables take on different values during execution. The use of print statements is the standard procedural way of eliciting this information.

This is a classic case of the need to query over execution history. Other examples include queries to find which variable has a certain value; the calling sequence that results in a certain outcome; whether a certain statement was executed; etc. We arrived at a set of queries by a study of the types of errors that arise in object-oriented programs.

They proposed two broad categories of queries in this paper: (i) queries over individual execution states and (ii) queries over the entire history of execution, or a subset of the history. Our proposed method recognizes the need to query subhistories; such capability is especially useful when debugging large scale software whose program trace is composed of millions of execution events. Our system has the ability to filter system objects so that a programmer may focus on the objects explicitly instantiated form user defined classes.

Thus the contributions of our paper are:

(1) logic-based approach to debugging object-oriented programs;

(2) the provision of queries over individual states and the history of execution;

(3) a prototype for a trace analysis for object oriented programs.

2.3 Tracking Down Software Bugs using Automatic Anomaly Detection

This paper introduces DIDUCE, a practical and effective tool that aids programmers in detecting complex program errors and identifying their root causes. By instrumenting a program and observing its behavior as it runs, DIDUCE dynamically formulates hypotheses of invariants obeyed by the program. DIDUCE hypothesizes the strictest invariants at the beginning, and gradually relaxes the hypothesis as violations are detected to allow for new behavior. The violations reported help users to catch software bugs as soon as they occur. They also give programmers new visibility into the behavior of the programs such as identifying rare corner cases in the program logic or even locating hidden errors that corrupt the program's results.

We tackle the problems of both detecting bugs and hunting down the root causes of bugs using the concept of dynamic invariant detection and checking. Most programs obey many invariants, many of which are not documented anywhere, and in fact, may not be known even to the original writers of the code. Explicitly specifying known program invariants, usually with the goal of documenting the programs, or of checking the invariants dynamically or statically is a tedious task. In addition, manual specification of invariants tends to capture only a few abstract, high-level invariants and a few implementation-specific, low-level invariants at key program points, usually reflecting the parts of the program that programmers tend to think and worry about the most.

We implemented the DIDUCE system for Java programs and applied it to four programs of significant size and complexity. DIDUCE succeeded in identifying the root causes of programming errors in each of the programs quickly and automatically. In particular, DIDUCE is effective in isolating a timing-dependent bug in a released JSSE (Java Secure Socket Extension) library, which would have taken an experienced programmer days to find. Our experience suggests that detecting and checking program invariants dynamically is a simple and effective methodology for debugging many different kinds of program errors across a wide variety of application domains.

2.4 Algorithmic Debugging of Java Programs

In this paper we propose applying the ideas of declarative debugging to the object-oriented language Java as an alternative to traditional trace debuggers used in imperative languages. The declarative debugger builds a suitable computation tree containing information about method invocations occurred during a wrong computation. The tree is then navigated, asking the user questions in order to compare the intended semantics of each method with its actual behavior until a wrong method is found out. The technique has been implemented in an available prototype. We comment the several new issues that arise when using this debugging technique, traditionally applied to declarative languages, to a completely different paradigm and propose several possible improvements and lines of future work.

While this view of the software as an assembly of components has been successful and it is widely used during the phases of design and implementation, it has had little influence on later phases of the software development cycle such as testing and debugging. Indeed, the debuggers usually included in the modern IDEs of languages such as Java, are sophisticated *tracers*, and they do not take advantage of the component relationships.

CHAPTER 3

FEATURES

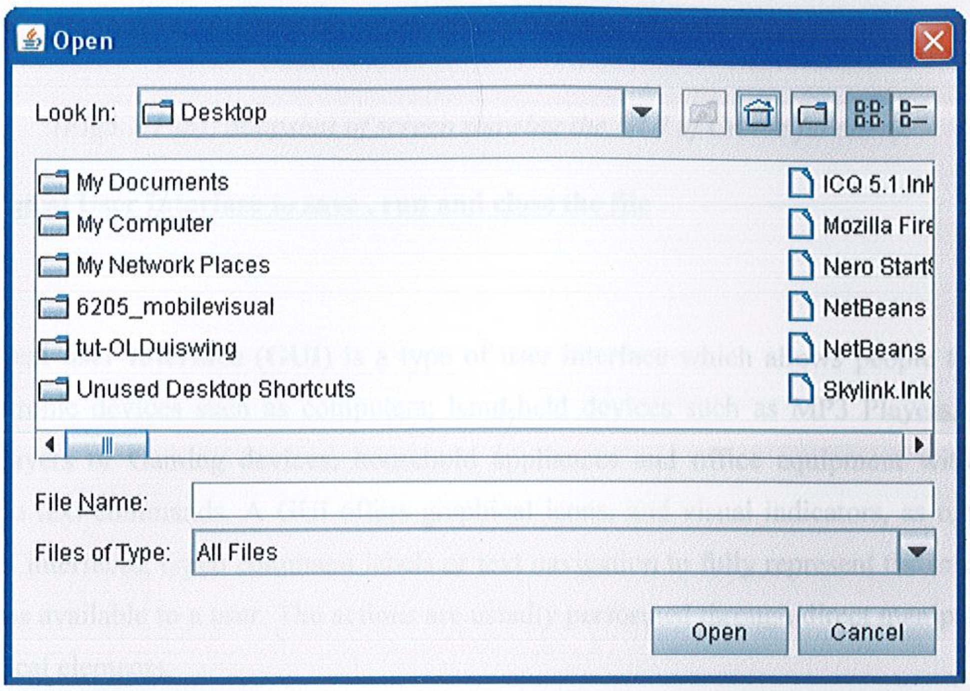
3.1 Create a New File

We used packages and libraries of JAVA platform to design the user interface of creating a new file. Create button helps to perform an action to open a new window of GUI used to write input program of infinite lines.

3.1.1 Create Action

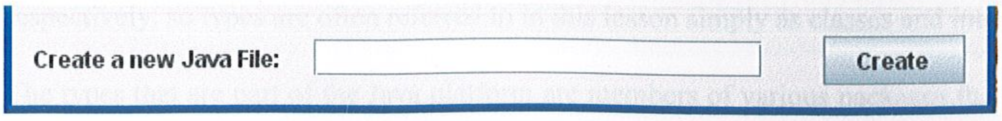
Action of creating a new file is done by create button. For creating and performing the action event of button we choose the File Choosers. File choosers provide a GUI for navigating the file system, and then either choosing a file or directory from a list, or entering the name of a file or directory. To display a file chooser, you usually use the JFileChooser.API to show a modal dialog containing the file chooser. Another way to present a file chooser is to add an instance of JFileChooser to a container.

A JFileChooser object only presents the GUI for choosing files. Your program is responsible for doing something with the chosen file, such as opening or saving it. The JFileChooser API makes it easy to bring up open and save dialogs. The type of look and feel determines what these standard dialogs look like and how they differ. In the Java look and feel, the save dialog looks the same as the open dialog, except for the title on the dialog's window and the text on the button that approves the operation. Here is a picture of a standard open dialog in the Java look and feel:



(Fig3.1.1 (a): Snapshot of directory select navigator).

Directory selector navigator is used to add an another text " create a new java file" and perform the action of the create button by the various inbuilt classes, libraries and functions like JLabel, JTextField Area, set text etc……



(Fig3.1.1 (b): Snapshot of screen showing the GUI of Create function).

3.2 Graphical User Interface to save , run and close the file

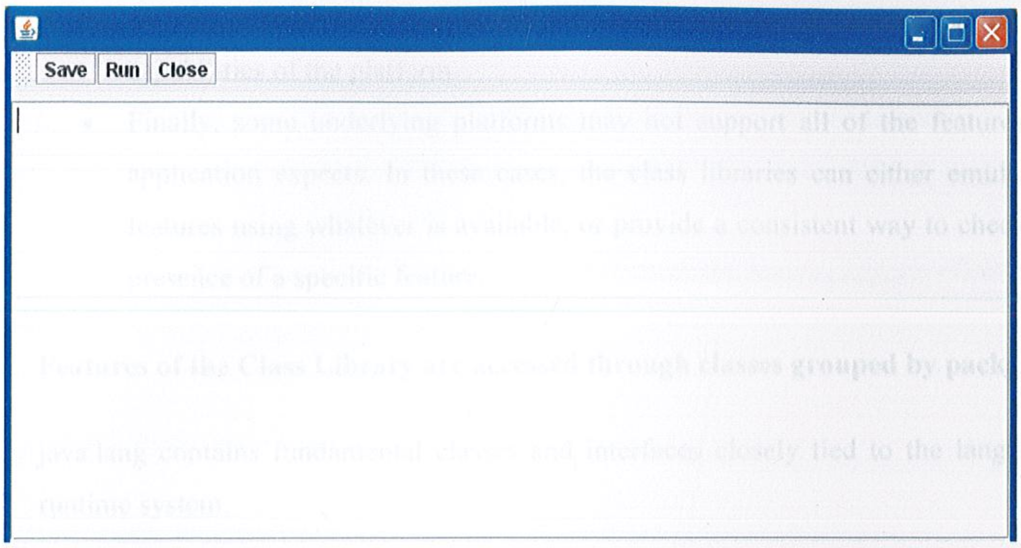
A **graphical user interface (GUI)** is a type of user interface which allows people to interact with electronic devices such as computers; hand-held devices such as MP3 Players, Portable Media Players or Gaming devices; household appliances and office equipment with images rather than text commands. A GUI offers graphical icons, and visual indicators, as opposed to text-based interfaces, typed command labels or text navigation to fully represent the information and actions available to a user. The actions are usually performed through direct manipulation of the graphical elements.

In our tools a GUI is implemented which contains the save, run and close button .For creating these buttons used various packages, libraries and functions. Save the incorrect program in the hard disk where we saved our project module by using the save button and after that for debug it by the runnable action to check out the bugs through run button. Close button is used to close GUI.

3.2.1 Packages

A package is a grouping of related types providing access protection and name space management. Note that types refers to classes, interfaces, enumerations, and annotation types. Enumerations and annotation types are special kinds of classes and interfaces, respectively, so types are often referred to in this lesson simply as classes and interfaces.

The types that are part of the Java platform are members of various packages that bundle classes by function: fundamental classes are in java.lang, classes for reading and writing (input and output) are in java.io, and so on. You can put your types in packages too.



(Fig3.2.1: Snapshot of screen showing the save, run and close button.)

3.2.2 <u>Libraries</u>

The **Java Class Library** is a set of dynamically loadable libraries that Java applications can call at runtime. Because the Java Platform is not dependent on any specific operating system, applications cannot rely on any of the existing libraries. Instead, the Java Platform provides a comprehensive set of standard class libraries containing much of the same reusable functions commonly found in modern operating systems.

The Java class libraries serve three purposes within the Java Platform:

- Like other standard code libraries, they provide the programmer a well-known set of functions to perform common tasks, such as maintaining lists of items or performing complex string parsing.
- In addition, the class libraries provide an abstract interface to tasks that would normally depend heavily on the hardware and operating system. Tasks such as network access and file access are often heavily dependent on the native capabilities of the platform.
- Finally, some underlying platforms may not support all of the features a Java application expects. In these cases, the class libraries can either emulate those features using whatever is available, or provide a consistent way to check for the presence of a specific feature.

Features of the Class Library are accessed through classes grouped by packages:

java.lang contains fundamental classes and interfaces closely tied to the language and runtime system.

- **I/O and networking:** access to the platform file system, and more generally to networks, is provided through the java.io, java.nio, and java.net packages.
- **Mathematics package:** java.math provides regular mathematical expressions, as well as arbitrary-precision decimals and integers numbers.
- **GUI and 2D Graphics:** the java.awt package supports basic GUI operations and binds to the underlying native system. It also contains the 2D Graphics API. The javax.swing package is built on AWT and provides a platform independent widget toolkit, as well as a Pluggable look and feel. It also deals with editable and non-editable text components.
- **Text:** the java.text package deals with text, dates, numbers, and messages.
- **Applets:** java.applet allows applications to be downloaded over a network and run within a guarded sandbox.
- **Java Beans:** java.beans provides ways to manipulate reusable components.

3.2.2.1 Abstract Window Tool Kit

The Abstract Window Toolkit (AWT) supports Graphical User Interface (GUI) programming. AWT features include:

- A rich set of user interface components.
- A robust event-handling model.
- Graphics and imaging tools, including shape, color, and font classes.
- Layout managers, for flexible window layouts that don't depend on a particular window size or screen resolution.
- Data transfer classes, for cut-and-paste through the native platform clipboard.

3.2.3 Classes and functions used in our tool

We used the atomic components to accept input from an end user.Thee various atomic Swing components are:

- **JFrame.**
- The **JFrame class** is a extension of the AWT Frame class. A frame is a top-level window that contains the title, border, minimize and maximize buttons. Cannot add components directly to JFrame. We need to add components to the content pane of the JFrame class.
- **JButton** Class.
- The **JButton class** enables to add a button to an applet or a panel.We can text or create an icon associated with a button.
- **JTextField Class.**
- The **JTextField class** creates a text field that enables to edit a single line of text.

- **JLabel Class**

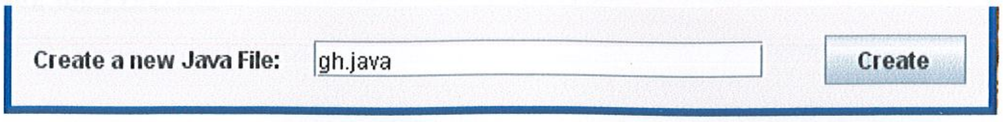
  The JLabel class is used to display text ans an image.By default, a label that displays text is left aligned and a label that displays image is horizontally centered.We can also change the alignment of a label.

3.3 Write a input source code to save and run it

The action performed by the create button, open a new GUI where we write our input source code or errroneous code for checking out the bugs. Program may be thousand lines of code and used to save in the hard disk by save button where we saved our project module and GUI also used to debug the errroneous code.

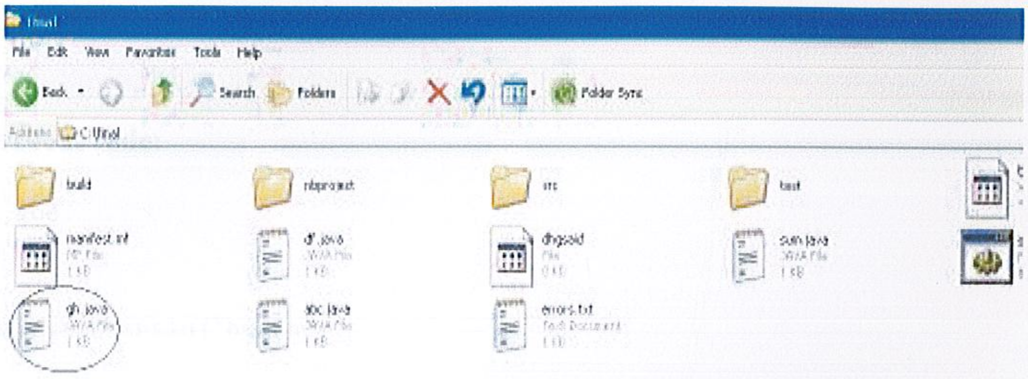
3.4 Save The Program In The disk-drive

A GUI is designed to save the file, debug the source code and close the user interface. Open a new window or GUI where write input program contains save, run and close button. Through the save button , save the file in the hard disk where we saved our project module. In project module folder, the file save as the same name as given in the last window like gh.java.



(Fig3.4 (a): Snapshot showing the screen of Create function loading gh.java file).

- gh.java file contains input source code program and save in the project module folder.



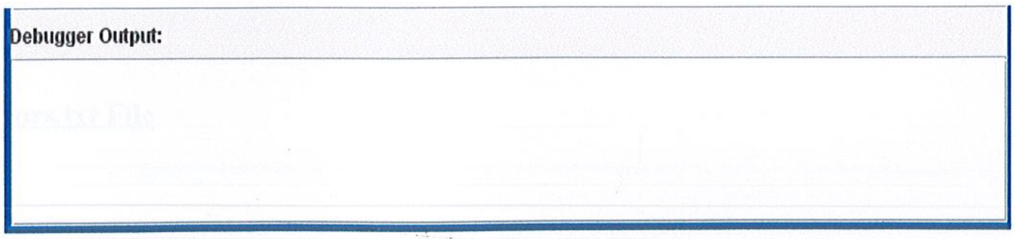
(Fig3.4 (b): Snapshot of screen showing the creation input program file in project folder).

3.5 Debug the Program

When we clicked on the run button it performs action event to debug the entire code which is used to show the errors or bugs in the debugger output screen and highlights which type of bugs takes place and determine how many bugs the errorneous code have.

Errroreneus code:

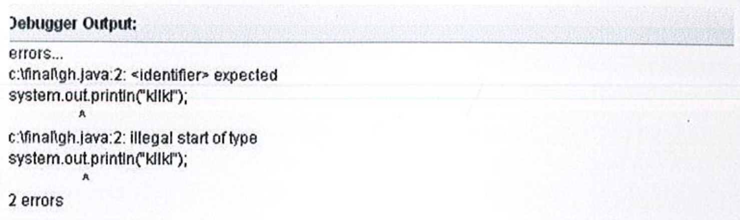
```
Class abc

{

system.out.println("hdasa")

}
```



(Fig3.5: Snapshot of debugger output).

3.6 <u>Show the Bugs in Debugger Output</u>

Showing of bugs in a debugger output screen to calculate type of bug in the lines and also determine quantity of bugs. In the screen it highlights the lines of the code where the possibility of bugs or errors takes place.



```
Debugger Output:
errors...
c:\final\gh.java:2: <identifier> expected
system.out.printin("kllkl");
              ^
c:\final\gh.java:2: illegal start of type
system.out.printin("kllkl");
               ^
2 errors
```

(Fig3.6: Snapshot of debugger output screen).

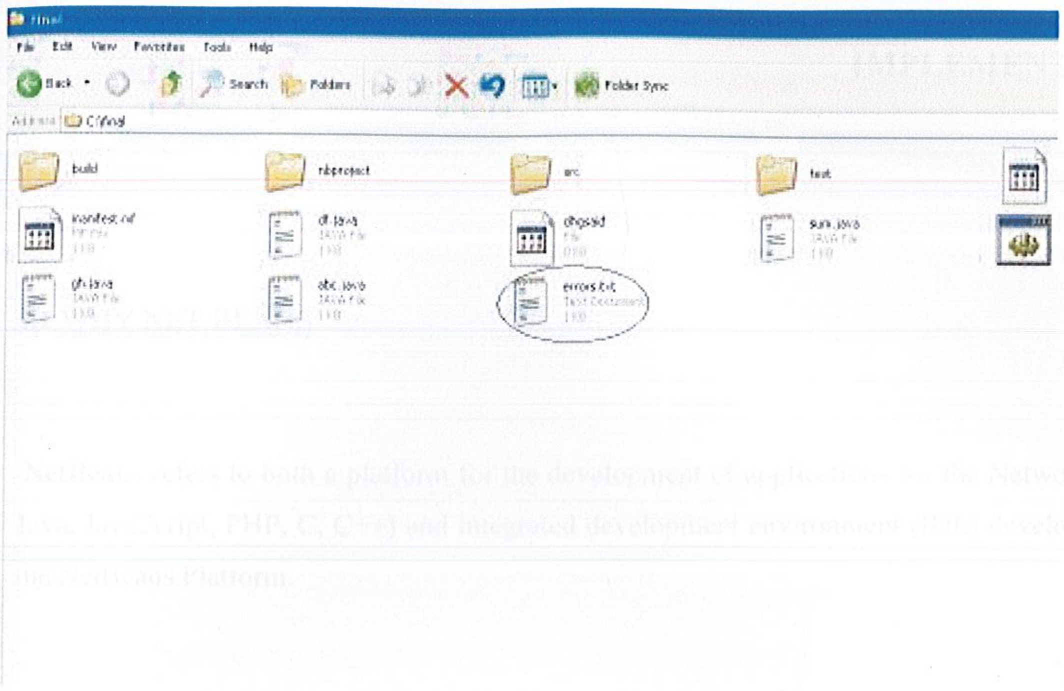
3.7 <u>Close the User Interface</u>

Close and exit the GUI by close button.

3.8 <u>Errors.txt File</u>

After showing of bugs in the screen, there is automatically create a file named is "errors.txt" and save in the project module.

- In errors.txt save the same contents of output screen.



(Fig3.8: Snapshot of screen showing errors.txt file).

CHAPTER 4
IMPLEMENTATION

4.1 WHY NET BEANS

NetBeans refers to both a platform for the development of applications for the Network (using Java, JavaScript, PHP, C, C++) and integrated development environment (IDE) developed using the NetBeans Platform.

The NetBeans Platform allows applications to be developed from a set of modular Software components called modules. A module is a Java archive file that contains Java classes written to interact with the NetBeans Open APIs and a manifest file that identifies it as a module. Applications built on modules can be extended by adding new modules. Since modules can be developed independently, applications based on the NetBeans platform can be extended by third party developers.

4.1.1 NET BEANS PLATFORM

The **NetBeans Platform** is a reusable framework for simplifying the development of other desktop applications. When an application based on the NetBeans Platform is run, the platform's Main class is executed. Available modules are located, placed in an in-memory registry, and the modules' startup tasks are executed. Generally, module's code is loaded into memory only as it is needed.

Applications can install modules dynamically. Any application can include the update center module to allow users of the application to download digitally-signed upgrades and new features directly into the running application Reinstalling an upgrade or a new release does not force users to download the entire application again.

The platform offers services common to desktop applications, allowing developers to focus on the logic specific to their application. Among the features of the platform are:

- User interface management (e.g. menus and toolbars)
- User settings management
- Storage management (saving and loading any kind of data
- Window management

4.1.2 <u>NETBEANS IDE</u>

The **NetBeans IDE** is an open-source integrated development environment written entirely in Java using the NetBeans Platform. NetBeans IDE supports development of all Java application types. Among other features are an Ant-based project system version control and refactoring.

4.1.3 NETBEANS INTEGRATED MODULES

4.1.3.1 GUI DESIGN TOOLS

The GUI design-tool enables developers to prototype and design Swing GUIs. By dragging and positioning GUI components. A graphical user interface builder or GUI builder, also known as GUI designer is a software development tool that simplifies the creation of GUIs by allowing the designer to arrange widgets using a drag-and-drop WYSIWYG editor.

User interfaces are commonly programmed using an event-driven architecture so GUI builders also simplify creating event-driven code. This supporting code connects widgets with the outgoing and incoming events that trigger the function providing the application log.

4.2 JAR FILE

A **JAR** file (or java archive) aggregates many files into one. Software developers generally use .jar files to distribute Java classes and associated metadata. JAR files build on the ZIP file format. Computer users can create or extract JAR files using the jar command that comes with the JDK. They can also use zip tools. A JAR file has an optional manifest file located in the path MANIFEST.MF. The entries in the manifest file determine how one can use the JAR file. JAR files intended to be executed as standalone programs will have one of their classes specified as the "main" class.

4.2.1 Packaging programs in jar file

Package Sealing: Packages stored in JAR files can be optionally sealed so that the package can enforce version consistency. Sealing a package within a JAR file means that all classes defined in that package must be found in the same JAR file.

Package Versioning*:* A JAR file can hold data about the files it contains, such as vendor and version information.

Portability*:* The mechanism for handling JAR files is a standard part of the Java platform's core API.

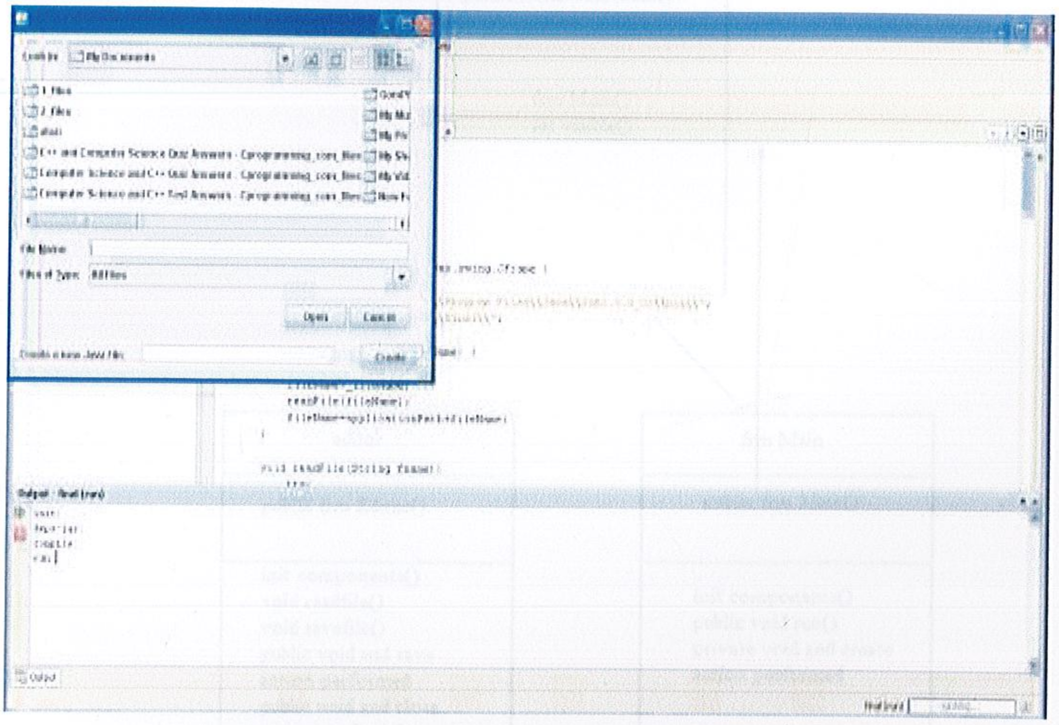
4.2.2 Manifest files

The manifest is a special file that can contain information about the files packaged in a JAR file. By tailoring this "meta" information that the manifest contains, you enable the JAR file to serve a variety of purposes.

These files can also include a class-path entry, which identifies other JAR files for loading with the JAR. This entry consists of a list of absolute or relative paths to other JAR files. Although intended to simplify JAR use, in practice it turns out to be notoriously brittle, as it depends on all the relevant JARs being in the exact locations specified when the entry-point JAR was built. To change versions or locations of libraries, a new manifest is needed.

4.3 Implementation of Features

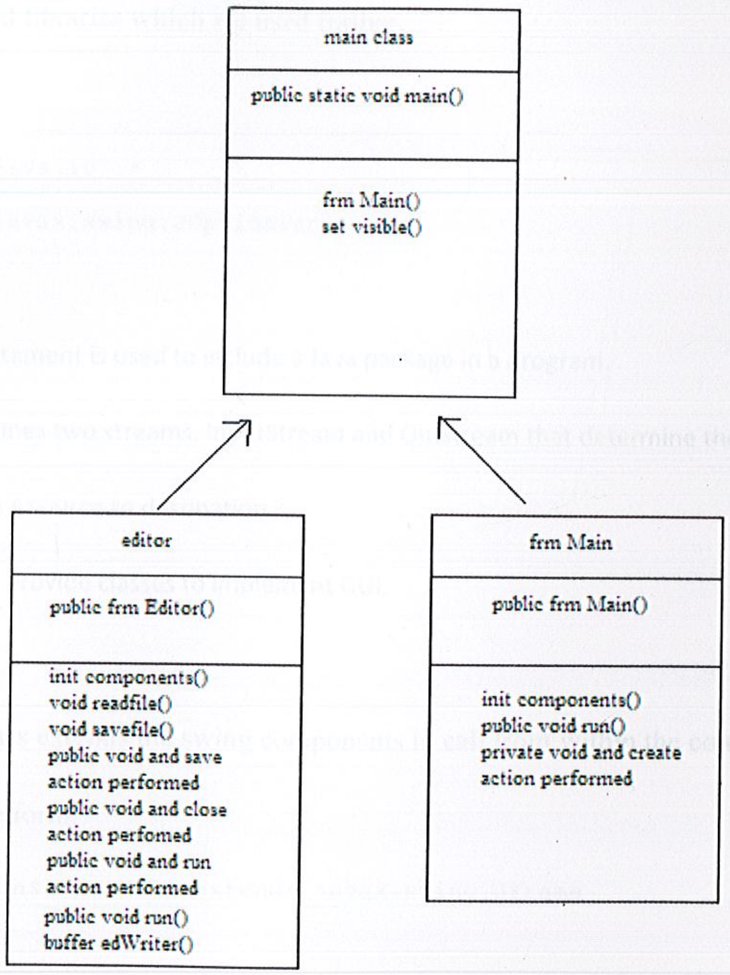
4.3.1 Create a New Java File



(Fig 4.3.1(a): snapshot of GUI to perform create action)

First we have to use 3 classes in java

- Main class
- Editor class
- frmmain class



(Fig 4.3.1(b): class diagram of our debugger)

In **Main class**, we use command line arguments

Public static void main()

In which we call the object of **frmMain()** to use the variables and functions of frmMain class.In frmMain()

For creating the new file, first of all we choose the define packages which contain the predefined libraries which we used further.

```
Import java.io .*

Import javax.swing.JOptionPane;
```

Import statement is used to include a Java package in a program.

Java.io defines two streams, InputStream and Outstream that determine the flow of

bytes from a source to destination

Java.swing provide classes to implement GUI.

frmMain class extends the swing components to call from within the constructor to

initialize the form.

```
public class frmMain extends javax.swing.JFrame

{

public frmMain() {

initComponents();

}
```

frmMain class extends JFrame class as extension of the AWT Frame class.frmMain() constructor is used to call init components() function for executing of components. A frame is top-level window that contains the title,border,minimize and maximize buttons.

Then after that define the init() method calls the rest of the life cycle methods in the chain.

In init method call

```
private void initComponents() {

        jFileChooser1 = new javax.swing.JFileChooser();
        jLabel1 = new javax.swing.JLabel();
        txtNewFile = new javax.swing.JTextField();
        cmdCreate = new javax.swing.JButton();
```

JFilechooser File choosers provide a GUI for navigating the file system, and then either choosing a file or directory from a list, or entering the name of a file or directory. To display a file chooser, you usually use the JFileChooser.

Add an additional feature create a java file by carious AWT components

AWT components that enables the end users to interact with applications created in java.

Used the components like

```
jLabel1 = new javax.swing.JLabel();
txtNewFile = new javax.swing.JTextField();
```

- Labels are used for displaying a single line of text in a container.
- JTextField class creates a text fiels that enables to edit a single line of text.

```
jLabel1.setText("Create a new Java File:");
```

here label is to " Create a new Java File

```
cmdCreate.setText("Create");
        cmdCreate.addActionListener(new java.awt.event.ActionListener() {
                public void actionPerformed(java.awt.event.ActionEvent evt)
{
                        cmdCreateActionPerformed(evt);
                }
        });
```

setText ("………") sets the required text to the text area.

cmdCreate.setText used to set a the text " create" in the button which is below the cancel option selected from navigator and after that perform the action by clicking on it.

If an application performs some action, based on the generation of the some action event then it implements the ActionListener interface.Each component,such as button that generates the action event,registers the ActionListener to receive the events by calling its addActionListener() method.

ActionListener interface defines the actionPerformed() to receive and process action event.

```
javax.swing.GroupLayout layout=new
javax.swing.GroupLayout(getContentPane());

getContentPane().setLayout(layout);
```

The layout managers are used to position the components, such as an applet, a panel or a frame in a container.The layout managers implement the java.awt.LayoutManager interface.A layout manager is an instance of the LayoutManager interface.All layout managers make use of the setLayout() method to set the layout of the container is set.

Basically it is used to set the position of label , textField and button " create" below the GUI navigator which we selected from JFileChooser.

```
private void cmdCreateActionPerformed(java.awt.event.ActionEvent
      evt){
      try{
        File f=new File(txtNewFile.getText());
        f.createNewFile();
        Editor frm=new Editor(txtNewFile.getText());
        frm.setVisible(true);
        }
        catch(IOException ex){
        JOptionPane.showMessageDialog(this, ex.getMessage());}
        }
```

try block encloses the statements that might raise an exception within it and defines the scope off the exception handlers associated with it. The catch block is used as an exception-handler. Enclose the code that want to monitor inside a try block to handle run-time.

In try block a f is a object called to call getText() to retrieve the string value when an action is performed. Editor class object is called here to read, write, save and run the file.

Create a new file which have the extension *. Java and when we clicked on the create button it automatically open a new GUI and simultaneously save the created file in the project module without any text.

4.3.2 <u>Graphical User Interface to save , run and close</u>

GUI is used to saving the file and debug the program code which we will write in the screen.

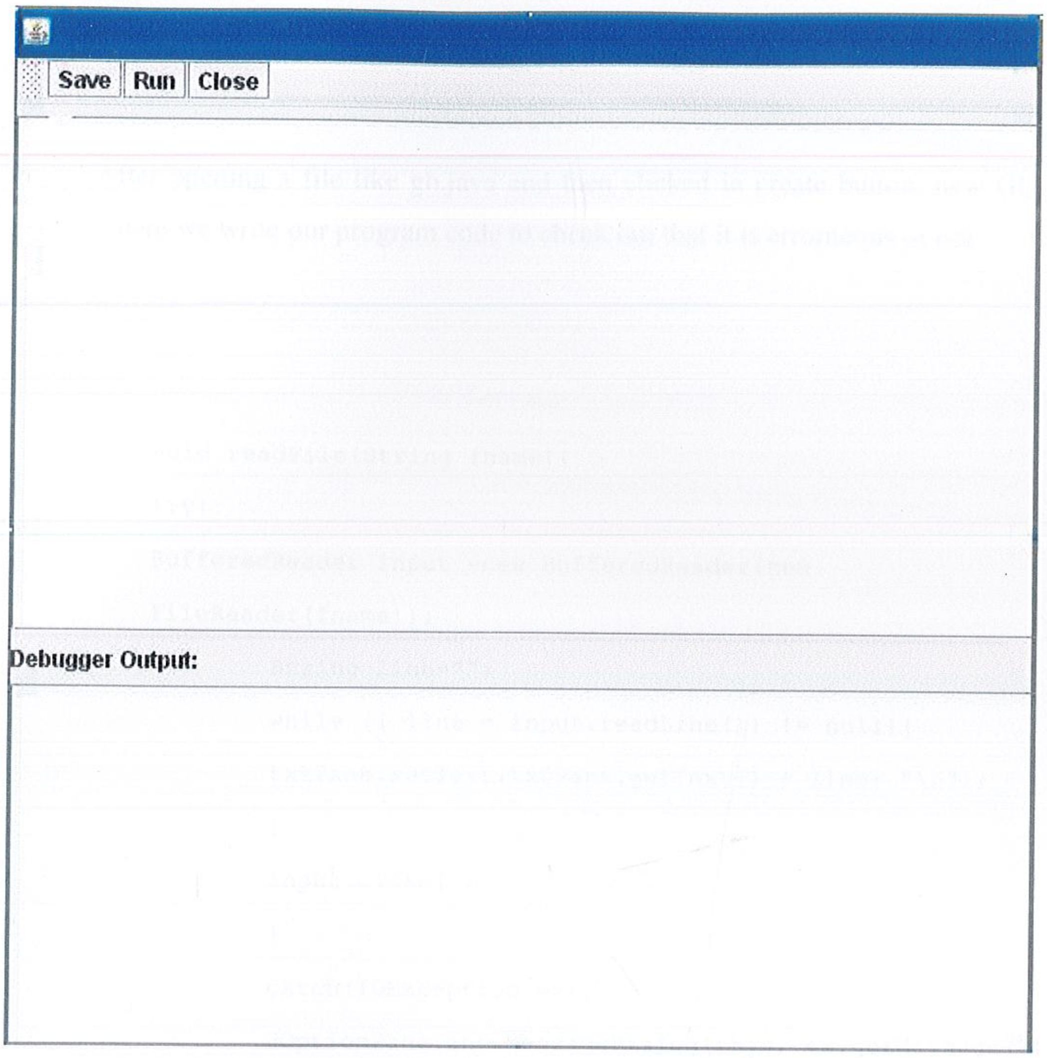
GUI is designed to initialize the init components and in init method() defines JLabel, JtextField and use the tool bar. Normally done the same work as we did in creating create button

```
private  void initComponents() {

        jToolBar1 = new javax.swing.JToolBar();
        cmdSave = new javax.swing.JButton();
        cmdRun = new javax.swing.JButton();
        cmdClose = new javax.swing.JButton();
        jScrollPane1 = new javax.swing.JScrollPane();
        txtPane = new javax.swing.JTextPane();
        jLabel1 = new javax.swing.JLabel();
```

And after that set the layout to locate the positions and call the actionListener to perform the action event.

```
cmdSave.setText("Save");
cmdSave.addActionListener(new java.awt.event.ActionListener()
{
public void actionPerformed(java.awt.event.ActionEvent evt) {
cmdSaveActionPerformed(evt);
}
});
jToolBar1.add(cmdSave);
```

Similarly, perform on the run and close button



(Fig 4.3.2 GUI of whole framework)

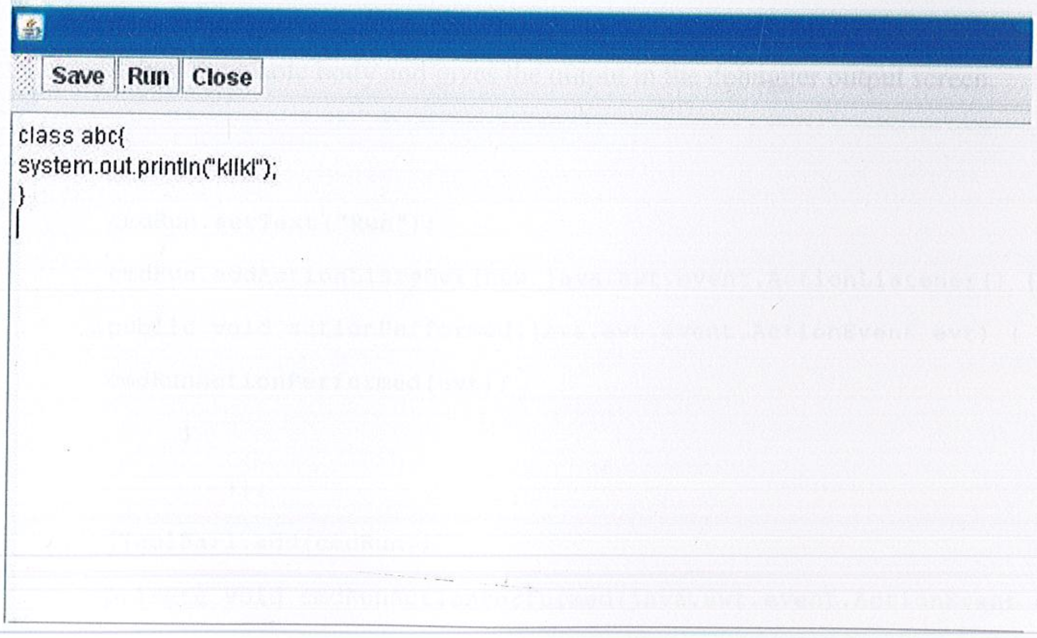
4.3 .3 **Write a input code in the GUI**

After opening a file like gh.java and then clicked in create button, new GUI is opened where we write our program code to check out that it is errorneous or not.

```
void readFile(String fname){
try{
BufferedReader input =new BufferedReader(new
FileReader(fname));
        String line="";
        while (( line = input.readLine()) != null){
        txtPane.setText(txtPane.getText() + line+ "\n");
        }
        input.close();
        }
        catch(IOException ex){
        JOptionPane.showMessageDialog(this, ex.getMessage());
    }
}
void saveFile(){
  try{
  Writer output=new BufferedWriter(new FileWriter(fileName));
   output.write(txtPane.getText());
   output.close();
   }
  catch(IOException ex)
```

```
        {
         JOptionPane.showMessageDialog(this, ex.getMessage());
        }
```

The BufferReader is used to read text from a character input stream and attaches a Buffer to the character input Streams. Buffers allo reading more than one Character at a time. The method input readLine() reads a text at a time. After reading call the BufferWriter calss which extend the Writer class ans writes text to a character output stream and allows more than one character at a time.



```
class abc{
system.out.println("klIkl");
}
```

(Fig 4.3.3 snapshot of input program)

4.3.4 save a written code

The written code is saved by the action performed by save button in an application path where we saved our project module.In project module, file is created when create action is performed but the text will be written in it when we perform the save action.

```
private void cmdSaveActionPerformed(java.awt.event.ActionEvent evt)
 {
saveFile();
  }
```

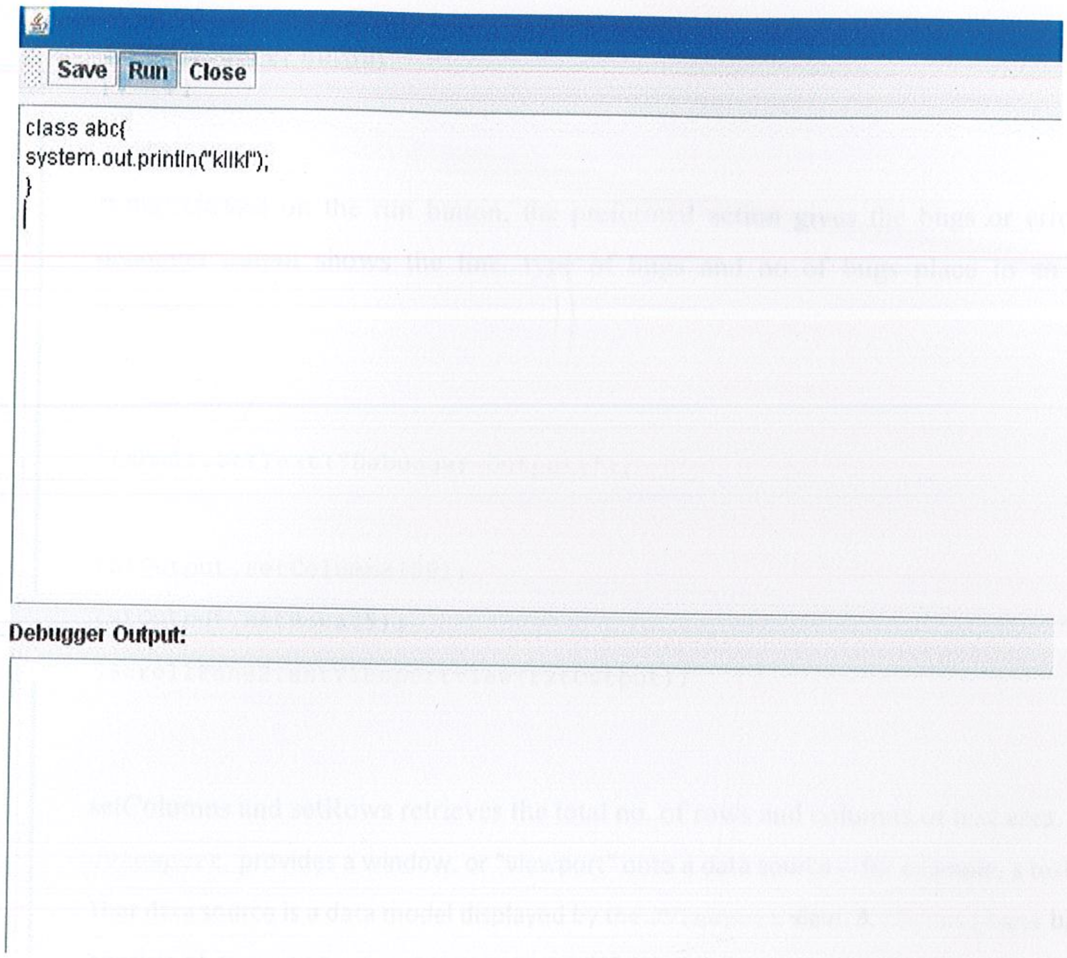
4.3.5 Debug the written code

Run the program code of thousand lines by the run button which perfrom the action to debug the whole body and gives the output in the debugger output screen.

```
cmdRun.setText("Run");
cmdRun.addActionListener(new java.awt.event.ActionListener() {
public void actionPerformed(java.awt.event.ActionEvent evt) {
cmdRunActionPerformed(evt);
     }
        });
jToolBar1.add(cmdRun);
private void cmdRunActionPerformed(java.awt.event.ActionEvent evt)
{
// TODO add your handling code here:
//compiling the file first
```

```
saveFile();

txtOutput.setText("");

try{

Writer output=new BufferedWriter(new FileWriter(applicationPath +
"setpath.bat"));

txtOutput.setText(txtOutput.getText() + "set path=\"" + javaInstallPath
+ "\"\n");

txtOutput.setText(txtOutput.getText() + "javac -Xstdout " +
applicationPath + "errors.txt  " + fileName + "\n");

txtOutput.setText(txtOutput.getText() + "java  -cp " + filePath + " " +
justFileName.replace(".java","") + "\n");

output.write(txtOutput.getText());

output.close();

Runtime.getRuntime().exec(applicationPath + "setpath.bat");

runner r=new runner();

r.start();
```

when we clicked on the run button it first loads application path plus filename after that look for setpath.bat.setpath.bat is javac.exe where to look for class files to import or java.exe where to find class files to interpret. Also the application path of errors.txt which contain the same content as when we debug the code.



```
class abc{
system.out.println("klikl");
}
```

(Fig 4.3.5: snapshot of running of input program through highlighting run button)

4.3.6 Debugger output

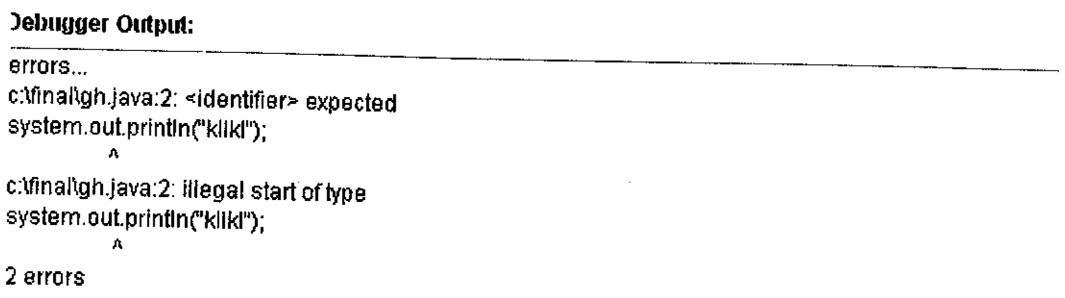
When clicked on the run button, the preformed action gives the bugs or errors in the debugger output shows the line, type of bugs and no of bugs place in an incorrect program.

```
jLabel1.setText("Debugger Output:");


txtOutput.setColumns(20);

txtOutput.setRows(5);

jScrollPane2.setViewportView(txtOutput);
```

setColumns and setRows retrieves the total no. of rows and columns of text area. The `JViewport` provides a window, or "viewport" onto a data source -- for example, a text file. That data source is a data model displayed by the `JViewport` view. A `JScrollPane` basically consists of `JScrollBars`, a `JViewport`, and the wiring between them, as shown in the diagram at right.

```
while(( line = input.readLine()) != null)
{
 errors+=line + "\n";
}
input.close();
if(errors.length()>0){
System.out.println("Errors!");
txtOutput.setText("Unable to compile..\n");
txtOutput.setText(txtOutput.getText() + errors);
return;
}
```

The above code is used to show errorneous lines of the input code in the debugger output screen.



```
Debugger Output:
errors...
c:\final\gh.java:2: <identifier> expected
system.out.println("klikl");
          ^
c:\final\gh.java:2: illegal start of type
system.out.println("klikl");
          ^
2 errors
```

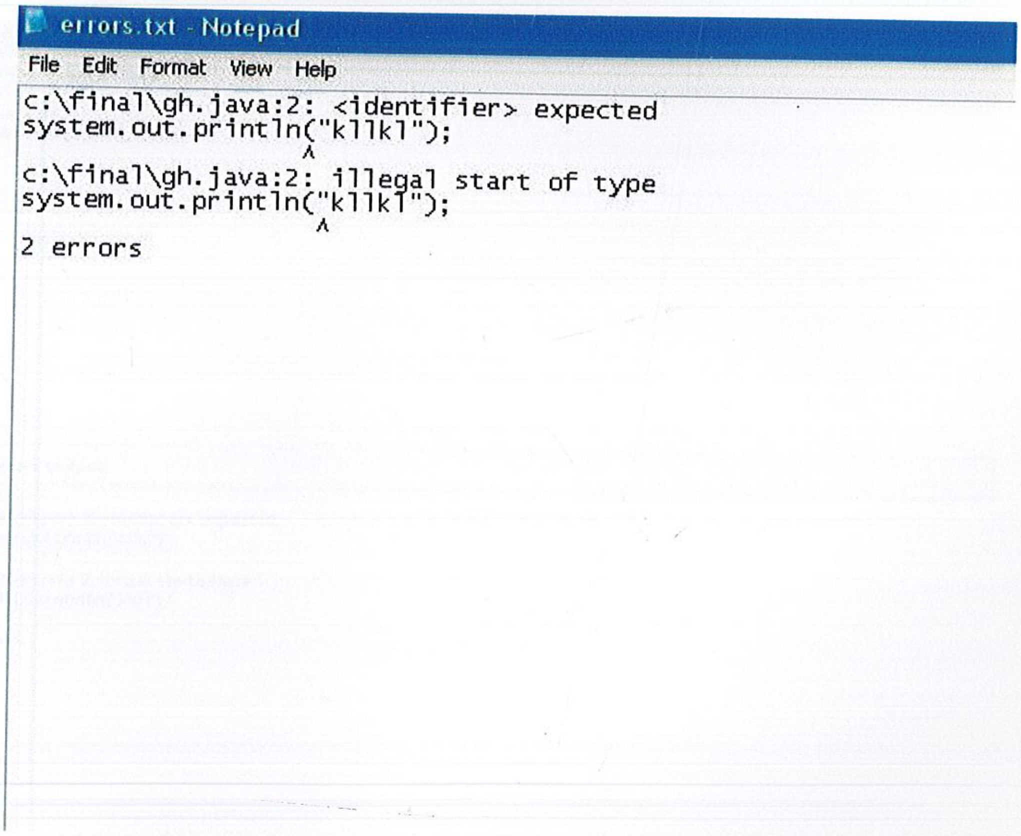
(Fig 4.3.6: snapshot of debugger output screen showing the bugs)

4.3.7 <u>Output content save in errors.txt</u>

In application path of the project module there is automatically exists a file named as errors.txt when we debug written code. In errors.txt contains the same content as the debugger output that types of errors and the no. of bugs. If we want to create a new file then the text of errors.txt will be deleted and contents of new file will exists.

```
File errFile=new File(applicationPath + "errors.txt");
```

```
if(errFile.exists())

{

errFile.delete();

}

errFile=new File(applicationPath + "errors_done.txt");

if(errFile.exists())

 {

  errFile.delete();

 }
```



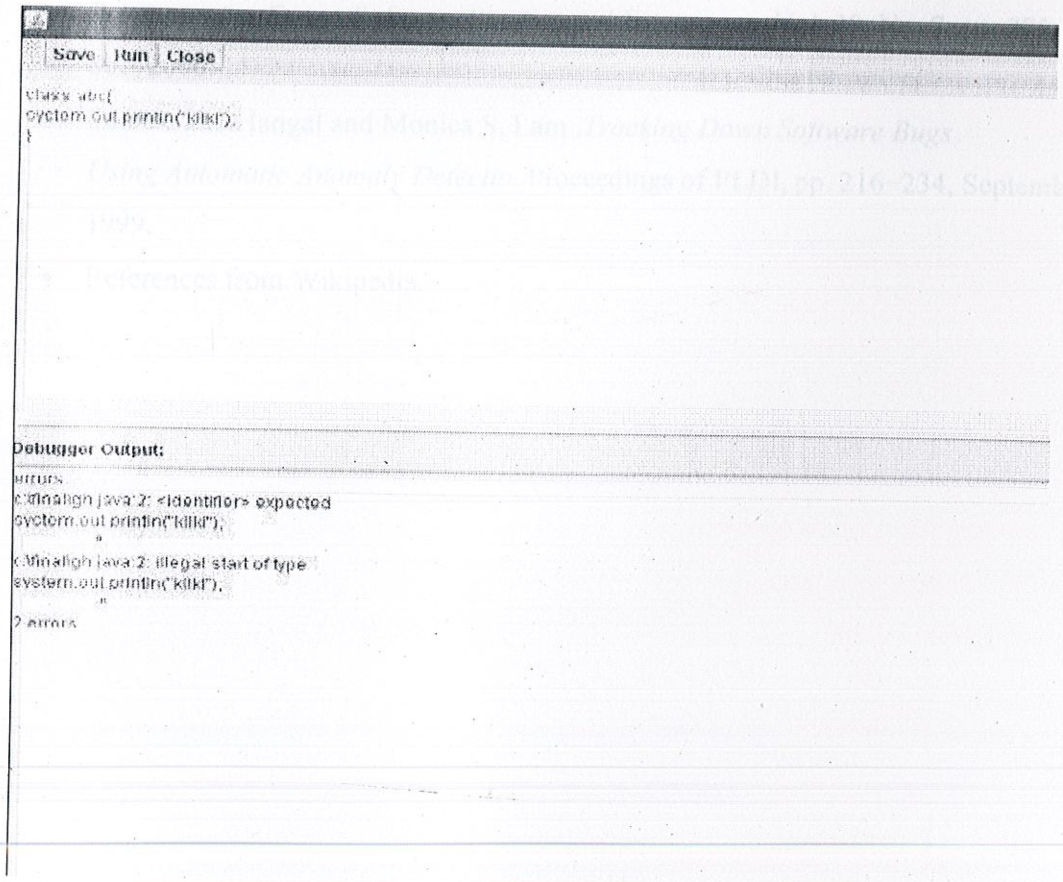
```
c:\final\gh.java:2: <identifier> expected
system.out.println("kllkl");
                        ^
c:\final\gh.java:2: illegal start of type
system.out.println("kllkl");
                         ^
2 errors
```

(Fig4.3.7: Snapshot showing errors.txt file)

CHAPTER 5

RESULT AND CONCLUSION

Our debugger is used to debug an erroronous java code and help us to determine the quantity of bugs and display expected and unexpected types of errors. Thus we conclude that this java debugger is used to debug other java programs to determine the no. of bugs and the type of bugs and work most probably as same as c debugger does.



```
class abc{
system.out.println("kllkl");
}
```

```
errors
c:\finaligh java:2: <identifier> expected
system.out.println("kllkl");
          ^
c:\finaligh java:2: illegal start of type
system.out.println("kllkl");
          ^
2 errors
```

(Fig5.1: Snapshot of screen showing the errors in the input program).

BIBLIOGRAPHY

- Hani Z. Girgis and Bharat Jayaraman. *JavaTA: A Logic-based Debugger for Java.* University at Buffalo,2007.
- Luke Tierney. *Threading and GUI Issues for R.* University of Minnesota March 5, 2001
- Robert F. Erbacher. *Interactive Program Debugging.* IEEE Software, Vol. 2, No. 1, 1985, pp. 2839.
- W. R. Bush, J. D. Pincus and D. J. Sielaff. A *Static Analyzer for Finding Dynamic Programming Errors.*Software Practice and Experience, Vol. 30, No. 7, pp. 775–802, 2000.
- Sudheendra Hangal and Monica S. Lam .*Tracking Down Software Bugs Using Automatic Anomaly Detectio*. Proceedings of PLDI, pp. 216–234, September 1999.
- References from Wikipedia.

CODE

➢ Main class

```
package finalpkg


public class Main {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {

        // TODO code application logic here
       /*  frm object is called of frmMain class */

           frmMain frm=new frmMain();

        frm.setVisible(true);
    }

}
```

➢ **Editor class**

```
package finalpkg;

import java.io.*;
import javax.swing.JOptionPane;

public class frmEditor extends javax.swing.JFrame{
    String fileName="";
   /* Installing java platform */
String javaInstallPath="c:\\Program
Files\\Java\\Jdk1.6.0_03\\bin\\";
   /* application path of project module */
String applicationPath="c:\\final\\";

/** Creates new form frmEditor */

public frmEditor(String _fileName)
{

/* initialize components
        initComponents();
        fileName=_fileName;
        readFile(fileName);
    }

void readFile(String fname)
{
try{

/*input object of BufferReader class is created and it buffers the text
written at the  command prompt*/

BufferedReader input =new BufferedReader(new FileReader(fname));
            String line="";

  /* readLine() method reads  a line of text into line */


while (( line = input.readLine()) != null){
               txtPane.setText(txtPane.getText() + line+ "\n");
    }
input.close();
        }

/* catch accepts the object ex of the Exception class that refres the
exception caught, as a parameter*/

catch(IOException ex){
JOptionPane.showMessageDialog(this, ex.getMessage());
}
}
```

```
/* writes text to a character output screen*/

void saveFile(){
try{

Writer output=new BufferedWriter(new FileWriter(fileName));
output.write(txtPane.getText());
output.close();
}
catch(IOException ex)
{
JOptionPane.showMessageDialog(this, ex.getMessage());
}
}


 /** This method is called from within the constructor to
     * initialize the form.
     * The content of this method is
     * always regenerated by the Form Editor.
     */


/* to add components such as buttons like save,run and close*/
        private void initComponents() {

        jToolBar1 = new javax.swing.JToolBar();
        cmdSave = new javax.swing.JButton();
        cmdRun = new javax.swing.JButton();
        cmdClose = new javax.swing.JButton();
        jScrollPane1 = new javax.swing.JScrollPane();
        txtPane = new javax.swing.JTextPane();
        jLabel1 = new javax.swing.JLabel();
        jScrollPane2 = new javax.swing.JScrollPane();
        txtOutput = new javax.swing.JTextArea();


setDefaultCloseOperation(javax.swing.WindowConstants.EXIT_ON_CLOSE);

jToolBar1.setRollover(true);

cmdSave.setText("Save");
cmdSave.setFocusable(false);

cmdSave.setHorizontalTextPosition(javax.swing.SwingConstants.CENTER);
 cmdSave.setVerticalTextPosition(javax.swing.SwingConstants.BOTTOM);


/* Save action perform*/

cmdSave.addActionListener(new java.awt.event.ActionListener() {
public void actionPerformed(java.awt.event.ActionEvent evt) {
cmdSaveActionPerformed(evt);
  }
  });
```

```
    jToolBar1.add(cmdSave);

   /* Set layout of run button*/

         cmdRun.setText("Run");
         cmdRun.setFocusable(false);

  cmdRun.setHorizontalTextPosition(javax.swing.SwingConstants.CENTER);

  cmdRun.setVerticalTextPosition(javax.swing.SwingConstants.BOTTOM);
  /* RunAction*/

  cmdRun.addActionListener(new java.awt.event.ActionListener() {
  public void actionPerformed(java.awt.event.ActionEvent evt) {
  cmdRunActionPerformed(evt);
   }
   });
   jToolBar1.add(cmdRun);

  cmdClose.setText("Close");
  cmdClose.setFocusable(false);

   /*Layout of close */

  cmdClose.setHorizontalTextPosition(javax.swing.SwingConstants.CENTER);

  cmdClose.setVerticalTextPosition(javax.swing.SwingConstants.BOTTOM);
  cmdClose.addActionListener(new java.awt.event.ActionListener()
  {
  public void actionPerformed(java.awt.event.ActionEvent evt)
  {
  cmdCloseActionPerformed(evt);
   }
     });
  jToolBar1.add(cmdClose);

  jScrollPane1.setViewportView(txtPane);

  /* implement  debugger output screen*/

        jLabel1.setText("Debugger Output:");

        txtOutput.setColumns(20);
        txtOutput.setRows(5);
        jScrollPane2.setViewportView(txtOutput);

      javax.swing.GroupLayout layout = new
      javax.swing.GroupLayout(getContentPane());
      getContentPane().setLayout(layout);
      layout.setHorizontalGroup(

layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
.addComponent(jToolBar1, javax.swing.GroupLayout.DEFAULT_SIZE, 649,
Short.MAX_VALUE)
.addComponent(jScrollPane1, javax.swing.GroupLayout.DEFAULT_SIZE, 649,
Short.MAX_VALUE)
.addGroup(layout.createSequentialGroup()
.addComponent(jLabel1)
.addContainerGap())
.addComponent(jScrollPane2, javax.swing.GroupLayout.DEFAULT_SIZE, 649,
Short.MAX_VALUE)
   );
layout.setVerticalGroup(layout.createParallelGroup(javax.swing.GroupLay
out.Alignment.LEADING)

.addGroup(layout.createSequentialGroup()
.addComponent(jToolBar1, javax.swing.GroupLayout.PREFERRED_SIZE, 25,
javax.swing.GroupLayout.PREFERRED_SIZE)

.addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.RELATED)
.addComponent(jScrollPane1, javax.swing.GroupLayout.PREFERRED_SIZE,
274, javax.swing.GroupLayout.PREFERRED_SIZE)

.addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.RELATED)
.addComponent(jLabel1)

.addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.RELATED)
.addComponent(jScrollPane2, javax.swing.GroupLayout.DEFAULT_SIZE, 99,
Short.MAX_VALUE))
        );


    }

private void cmdSaveActionPerformed(java.awt.event.ActionEvent evt) {
        //  handling code here:
        saveFile();
    }

private void cmdCloseActionPerformed(java.awt.event.ActionEvent evt) {
        // TODO add  handling code here:
        this.dispose();
    }

private void cmdRunActionPerformed(java.awt.event.ActionEvent evt) {
        // TODO add  handling code here:
        //compiling the file first
        saveFile();
        txtOutput.setText("");
        try{
String filePath=fileName.substring(0,fileName.lastIndexOf("\\")) +
"\\";
String justFileName=fileName.substring(fileName.lastIndexOf ("\\")+1);
Writer output=new BufferedWriter(new FileWriter(applicationPath +
"setpath.bat"));

/* existence of erros.txt*/
File errFile=new File(applicationPath + "errors.txt");

 if(errFile.exists()){

 errFile.delete();
  }
errFile=new File(applicationPath + "errors_done.txt");
```

```
if(errFile.exists()){
  errFile.delete();
        }

txtOutput.setText(txtOutput.getText() + "set path=\"" + javaInstallPath
+ "\"\n");

txtOutput.setText(txtOutput.getText() + "javac -Xstdout " +
applicationPath + "errors.txt  " + fileName + "\n");
txtOutput.setText(txtOutput.getText() + "java  -cp " + filePath + " " +
justFileName.replace(".java","") + "\n");
output.write(txtOutput.getText());
output.close();
Runtime.getRuntime().exec(applicationPath + "setpath.bat");
/* application path for execution of program*/

runner r=new runner();
r.start();



//output=new BufferedWriter(new FileWriter(applicationPath +
"setpath.bat"));
//String filePath=fileName.substring(0,fileName.lastIndexOf("\\")) +
"\\";
//String justFileName=fileName.substring(fileName.lastIndexOf("\\")+1);
          //txtOutput.setText(txtOutput.getText() + "java  -cp " +
filePath + " " + justFileName.replace(".java","") + "\n");
 //output.write(txtOutput.getText());
 //output.close();
//Runtime.getRuntime().exec(applicationPath + "setpath.bat");
        }
catch(Exception ex){

JOptionPane.showMessageDialog(this, ex.toString());
}
    }

/*  read errors in debugger output screen*/
    public void readErrors(){
        while(true){

            try{

BufferedReader input=new BufferedReader(new FileReader (applicationPath
+ "errors.txt"));
input.close();
input=new BufferedReader(new FileReader(applicationPath +
"errors.txt"));
input.close();
input=new BufferedReader(new FileReader(applicationPath +
"errors.txt"));

String errors="";
String line="";

while(( line = input.readLine()) != null){

errors+=line + "\n";
      }

input.close();

if(errors.length()>0){
   System.out.println("Errors!");
   txtOutput.setText("errors\n");
   txtOutput.setText(txtOutput.getText() + errors);
   return;
    }
    else
    {
    System.out.println("errors!");
    return;
     }
     }
     catch(Exception ex)
      {
      System.out.println(ex.getMessage());
      }
     }
   }

   class runner extends Thread{
   /*  Override  */
      public void run(){

      boolean compilationOver=false;
      while(!compilationOver)
      {
      try{
      File f=new File(applicationPath + "errors.txt");
      compilationOver=f.exists();
      }
      catch(Exception ex){
      System.out.println("trying!");
       }
          }
      System.out.println("Done!");
       try{
       this.sleep(5000);
           }
       catch(Exception ex){
        System.out.println(ex.getMessage());
                 }
      readErrors();
      }
      }
 /**
  * @param args the command line arguments
  */
public static void main(String args[]) {
```

```
            java.awt.EventQueue.invokeLater(new Runnable() {
                public void run() {
                    new frmEditor("").setVisible(true);
                }
            });
        }

        // Variables declaration
        private javax.swing.JButton cmdClose;
        private javax.swing.JButton cmdRun;
        private javax.swing.JButton cmdSave;
        private javax.swing.JLabel jLabel1;
        private javax.swing.JScrollPane jScrollPane1;
        private javax.swing.JScrollPane jScrollPane2;
        private javax.swing.JToolBar jToolBar1;
        private javax.swing.JTextArea txtOutput;
        private javax.swing.JTextPane txtPane;
        // End of variables declaration
    }
```

➢ **frmMain Class**

```
package finalpkg;
import java.io.*;
import javax.swing.JOptionPane;
public class frmMain extends javax.swing.JFrame

{

/** Creates new form frmMain */

 public frmMain() {
 initComponents();
    }

    /** This method is called from within the constructor to
     * initialize the form.
     * The content of this method is
     * always regenerated by the Form Editor.
     */


private void initComponents() {

        jFileChooser1 = new javax.swing.JFileChooser();
        jLabel1 = new javax.swing.JLabel();
        txtNewFile = new javax.swing.JTextField();
        cmdCreate = new javax.swing.JButton();


setDefaultCloseOperation(javax.swing.WindowConstants.EXIT_ON_CLOSE);

        jLabel1.setText("Create a new Java File:");


 cmdCreate.setText("Create");
 cmdCreate.addActionListener(new java.awt.event.ActionListener() {
public void actionPerformed(java.awt.event.ActionEvent evt) {
 cmdCreateActionPerformed(evt);
```

```
        }
            });

    javax.swing.GroupLayout layout = new
    javax.swing.GroupLayout(getContentPane());
            getContentPane().setLayout(layout);
            layout.setHorizontalGroup(

    layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
    .addComponent(jFileChooser1, javax.swing.GroupLayout.PREFERRED_ SIZE,
    javax.swing.GroupLayout.DEFAULT_SIZE,
    javax.swing.GroupLayout.PREFERRED_SIZE)
    .addGroup(layout.createSequentialGroup()

      .addContainerGap()
      .addComponent(jLabel1)

    .addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.UNRELATED)
    .addComponent(txtNewFile, javax.swing.GroupLayout.PREFERRED_SIZE, 238,
    javax.swing.GroupLayout.PREFERRED_SIZE)

    .addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.RELATED,
    32, Short.MAX_VALUE)
    .addComponent(cmdCreate, javax.swing.GroupLayout.PREFERRED_SIZE, 75,
    javax.swing.GroupLayout.PREFERRED_SIZE)
    .addContainerGap())
            );
            layout.setVerticalGroup(

    layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
    .addGroup(layout.createSequentialGroup()
    .addComponent(jFileChooser1, javax.swing.GroupLayout.PREFERRED_SIZE,
    javax.swing.GroupLayout.DEFAULT_SIZE,
    javax.swing.GroupLayout.PREFERRED_SIZE)

    .addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.RELATED)
```

```
 .addGroup(layout.createParallelGroup(javax.swing.GroupLayout.Alignment.
BASELINE)
 .addComponent(jLabel1)
 .addComponent(cmdCreate)
 .addComponent(txtNewFile, javax.swing.GroupLayout.PREFERRED_SIZE,
javax.swing.GroupLayout.DEFAULT_SIZE,
javax.swing.GroupLayout.PREFERRED_SIZE))
                    .addContainerGap(javax.swing.GroupLayout.DEFAULT_SIZE,
Short.MAX_VALUE))
         );

    pack();
    }

private void cmdCreateActionPerformed(java.awt.event.ActionEvent evt) {
        //  handling code here:
        try{
            File f=new File(txtNewFile.getText());
            f.createNewFile();
            frmEditor frm=new frmEditor(txtNewFile.getText());
            frm.setVisible(true);
        }
        catch(IOException ex){
            JOptionPane.showMessageDialog(this, ex.getMessage());
        }

    }

    /**
     * @param args the command line arguments
     */
    public static void main(String args[]) {
        java.awt.EventQueue.invokeLater(new Runnable() {
            public void run() {
                new frmMain().setVisible(true);
            }
        });
```

```
}

// Variables declaration
private javax.swing.JButton cmdCreate;
private javax.swing.JFileChooser jFileChooser1;
private javax.swing.JLabel jLabel1;
private javax.swing.JTextField txtNewFile;
// End of variables declaration
```