JavaDD: a Declarative Debugger for Java

Hani Z. Girgis Bharat Jayaraman

Department of Computer Science and Engineering
University at Buffalo, the State University of New York
{hzgirgis,bharat}@cse.buffalo.edu

ABSTRACT

This paper presents a declarative approach to the debugging of object-oriented programs and illustrates the methodology through an extension of a novel interactive visualization system for Java developed in our previous research. Unlike traditional “procedural” debugging, we use the term “declarative debugging” to refer to a flexible set of queries on individual execution states and also over the entire history of execution (or portion of the history). Examples include queries to find all values assigned to a variable over its life-time; which variable has a certain value; the calling sequence that results in a certain outcome; whether a certain statement was executed; etc. These queries were arrived at by a systematic study of errors in object-oriented programs in our previous research. Our proposed system, *JavaDD*, maintains the execution history as a relational database of salient events, such as method call/return, thread start/end, variable assignment, etc. An important property of our approach is that these queries can be posed interactively (at any step of execution), and there is no need to develop a compiler to instrument the source code, as in related research projects. Furthermore, we also sketch a visual interface so that both queries and answers can be composed using inituitive object and sequence diagrams. We believe such an approach is a significant contribution to the art of program debugging. We present the architecture of JavaDD, a detailed catalog of our queries and their translation, and several examples illustrating the approach. We also compare our approach related research efforts in the area of query-based analysis of object-oriented programs.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging - Debugging aids - Tracing.

General Terms

Design, Human Factors, Languages

Keywords

Bug patterns, Java, query-based debugging, execution history, object/sequence diagrams, relational databases, visual interfaces

1. INTRODUCTION

Traditional debugging is a “procedural” process in that the programmer has to proceed in step-by-step and object-by-object in order to uncover the cause of a bug. To facilitate this process, programming environments have evolved considerably in the past three decades in order to provide increasingly better tools and techniques: break points, source code stepping, back traces, object inspectors, variable spying, watch points, etc. Modern IDEs, such as Eclipse, NetBeans, and Visual Studio, also provide similar capabilities. In recent years, there have been richer execution environments, such as the JIVE system for Java [11][11][12][13], BlueJ [22], jGRASP [16], and MVT [24]. Still, the debugging paradigm for object-oriented languages such as Java is fundamentally procedural in nature.

In this paper, we propose a declarative approach to the debugging of object-oriented programs, and we illustrate our methodology through a system called *JavaDD*, for *Java Declarative Debugger*. This work extends our earlier research on the JIVE system for Java, which supports forward and reverse program stepping, and visualizing execution states and histories using enhanced object diagrams and time-sequence diagrams at varying levels of granularity. The term ‘declarative’ contrasts from ‘procedural’ in that the former focuses on ‘what’ while the latter focuses on ‘how’. We use the term ‘declarative debugging’ to refer to a broad set of queries over the current execution state as well as the history of execution. Declarative debugging complements procedural debugging, and we expect that both forms of debugging will be used in the general case.

To illustrate declarative debugging, note that a crucial aspect of program understanding is knowing 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. As another example, consider a parse tree composed of, say, 1000 nodes. While searching for nodes that satisfy some criteria, an exception is thrown. Debugging this program via traditional debuggers is tedious since there are 1000 node instances and possibly tens of thousands of method calls. Therefore, single-stepping is not a viable technique. Inserting break points is also not helpful, since the code that is responsible for the search is recursive. On the other hand, using declarative debugging we can isolate the bug via two queries: first, find the environment (object and method call instances) where the exception is thrown; and, second, query the object instance when the exception is thrown. These two queries are used in many different debugging scenarios. For example, instead of searching for the environment of a thrown exception, the programmer can ask for the environment of a member field assignment or a method call.

In keeping with the JIVE philosophy for program visualization, we propose two broad categories of declarative debugging queries in this paper: (i) queries over individual execution states, where each state consists of the set of active objects and outstanding method invocations, and (ii) queries over the entire history of execution, or a subset of the history. Examples of (i) include queries about an object or a variable and its value, call chains, etc. Examples of (ii) include queries about all values assigned to a variable over its history, whether a particular statement was ever executed or not; etc. The need to query sub-histories is especially useful when debugging interactive programs with GUIs, since each interaction episode corresponds to a well-defined sub-history. We arrived at our queries by a study of the types of errors that arise in object-oriented programs [12][13]. Our proposed system has the ability to filter system objects so that a programmer may focus on the objects explicitly created by his or her program.

This paper also describes the architecture of *JavaDD* along with a detailed catalog of queries and their translation in terms of a deductive database system built in Prolog. We log the history of salient events (method call, return, assignment, object creation, etc) during the execution of a Java program using the JPDA interface (Java Platform Debugger Architecture). Such an approach allows us to avoid instrumenting the source program with debugging commands or queries, as in other related approaches [14][23][25], and enables a completely modular approach to declarative debugging. It is important to also note that our declarative debugger is interactive, and hence queries can be posed at any point during program execution. Our approach to recording the history of changes is incremental in nature, i.e., when a variable is assigned, we save only the previous value assigned to the variable. Thus, queries about previous execution states involve some state reconstruction. The benefit of this approach is that forward execution incurs no appreciable slowdown.

To complete the paper, we also sketch a visual interface so that both queries and answers can be composed in terms of a framework of object and time-sequence diagrams , source code and program output. In this way, a programmer is relieved of the burden of composing queries in an unfamiliar query language. Thus the contributions of our paper are: (i) a methodology for a Java declarative debugger without requiring source code instrumentation and using off-the-shelf Java compiler and JVM, (ii) the provision of queries over individual states and the history of execution, (iii) a higher-level visual interface for queries and answers. Taken together, we believe that this proposed approach of declarative debugging is a significant contribution to field of object-oriented program debugging.

The remainder of the paper is organized as follows. Section 2 surveys closely related research and compares them with our work. Section 3 presents the architecture of the Java DD system, including the event log language. Section 4 presents our declarative queries for debugging. Section 5 provides an evaluation of our approach, by first presenting an empirical survey of errors in object-oriented programs followed by two short case studies. Section 6 sketches the visual query interface for presenting and observing answers. Section 7 presents conclusions and areas of further research.

2. RELATED WORK

Three recent research projects employ the concept of declarative queries for program analysis [23], [25], and [14]. An important difference in our approach is our emphasis on interactive debugging and without having to develop a compiler to perform source code instrumentation or modify the JVM:

1. Lencevicius et al [23] proposed a query-based debugger to understand object relationships. Their query language is expressed in the same language as the target OO language (Self), and thus a programmer does not need to learn a new language. Queries consist of a search domain and a constraint. Both the compiler and the underlying virtual machine need to be modified to realize the query semantics. In query-based debugging of [23], a query evolution can result in side effects on program state, and might introduce bugs and hence their result may not reflect the actual state of the program. On the positive side, this approach can provide incremental delivery of results, a feature that is useful in dealing with queries that takes considerable time to find all answers.
2. Recently, PQL (Program Query Language) was developed by Martin et al [25] to query over program execution for finding errors and security flaws in programs. Queries may formulate application-specific code patterns that may result in vulnerabilities at run-time. The queries translated to Datalog (which is essentially declarative Prolog without functors), and provides the ability to take an action once a match found. A combination of static and dynamic analysis is performed to answer queries. The PQL compiler generates code that is weaved into the target application and matches against a history of relevant events at execution time. A number of interesting security violations are found by this technique.
3. Goldsmith et al [14] proposed the PTQL (Program Trace Query Language) as a relational query language designed to query program trace. Similar in goals with PQL, PTQL employs an SQL-like query language. Partiqle compiles the PTQL queries into instrumentation in a given Java program. PTQL queries can be used to specify what to be recorded during program execution, and hence this technique can be effective with programs that generate many irrelevant events.

WhyLine [21] is an interrogative debugger for the Alice programming environment. It allows the user to ask why or why didn't a given event occur. The WhyLine gives the answer in the form of an execution path that leads or was supposed to lead to the execution of the given event. The path is annotated with control flow information. The 'why' and 'why not' questions reduce to the question whether an execution path leading to the given event exists or not. In our work, we provide queries on call-chains, call-trees, and sub-histories in order to extract similar information as in WhyLine.

Pauw et al. [31] proposed a catalog of views to describe patterns in program behavior and provided the Object Visualizer system to dynamically visualize object oriented programs via the proposed views. The proposed views are macroscopic views that help in understanding the overall program execution; however, the debugging process requires microscopic views that provide more details as well. The catalog presented in [31] is composed of seven views and can be summarized as follows.

1. Allocation matrix displays the instantiation relation among objects i.e. which objects result in the instantiation of other objects.
2. Class-time chart groups method call according to their class in relation to time.
3. Functions-instances matrix groups method calls according to their instance .
4. Histogram of instances groups objects by class.
5. Inter-Class call cluster shows the coupling among classes.
6. Inter-class call matrix provides an overall view of object interactions grouped by class.
7. Intra-class call matrix groups for each object calls to methods defined in its own class.

The absence of a methodology for querying Java program execution was recognized in our earlier work [12] on the JIVE system. JIVE visualizes the entire history of execution, and provides three views: the time-sequence diagram which depicts the interaction among objects in relation to time; the object diagram which shows the state of objects; and the source code view that highlights the executed line. We have proposed the following seven queries: value of a variable; the variable that holds a certain value or range of values; arguments to a method call; the number of times a method is called, a statement is executed, or a class is instantiated; and the last activity of an object. However, this set provides a limited support for practical debugging. Faced with the lack of methodology of querying the execution history, an empirical study has been conducted to understand the difference in visualization of correct and erroneous programs [13]. This study provided a set of queries which have been formulated in the methodology presented in this paper.

Rosenblum [34] investigated why programming with assertions was not widespread industry wise. Two reasons were found. First, assertion preprocessors did not work well with the available development tools and that did not meet the need of the "average software developer." The second reason was the lack of studies on types of assertions that can guard effectively against known error patterns. Rosenblum proposed a method of programming with assertions based on an empirical study of software interface faults of C programs by Perry and Evangelist [32]. He proposed an assertion to guard against each kind of errors. His research contributed a classification of assertions under two main categories: assertions related to functions interfaces and assertions related to functions implementation.

Query-based debugging and programming with assertions or design-by-contract (DBC), in general, are both concerned with finding and signaling erroneous program behavior. Our approach is similar to Rosenblum's in two aspects. Both approaches are based on empirical studies of software errors. Rosenblum's method of programming with assertion and our method of query-based debugging aim to make the technique easy enough so that an "average developer" can use it by providing a catalog of commonly used assertions and queries respectively. Our work differs in that it is concerned with object oriented programs while Rosenblum work was done on C programs.

Gamma et al [8] have proposed a collection of design patterns that aimed at showing how to design object oriented programs. Fowler et al [7] have proposed a catalog of re-factoring object oriented programs. Providing a collection of re-factoring techniques in OO programs opened the possibility for automatic re-factoring tools. Meyers [29] and Bloch [5] presented a collection of best practices in programming in C++ and Java respectively. Several static analysis tools [35][17] exist to enforce best practices as outlined by Meyers and Bloch. Following the same path a collection of debugging queries will contribute to the rise of automated query-based (declarative) debuggers.

3. JAVA DD ARCHITECTURE

One of the potential obstacles in declarative debugging is that an execution history of large-scale software that runs for some time may have millions of events that demand large-scale database server. Consider a scenario when a developer is running an IDE, such as Eclipse, and has started a debugger based on the JavaDD framework. The JavaDD will invoke a JVM and interact with the database server that is running on the same machine. At this point the IDE, the JVM, and the database server are all competing for memory. A centralized debugger based on JavaDD would not be practical solution in that scenario, and hence we propose a distributed architecture. The distributed architecture opens the opportunity to provide debugging and analysis services.

3.1 System Tiers and Components

We have implemented a prototype of the JavaDD framework, and Figure 1 shows the main tiers and components of this framework. The architecture of JavaDD is a composed of four tiers. The first tier consists of three components: The JPDA, Prolog Beans server and relations specification. JPDA the Java Platform Debugger Architecture [18] is designed as a distributed system that can interface with a JVM running on the same machine or a different machine. JPDA has an event-based architecture. Prolog Beans [33] is a Prolog server that can be interfaced with Java or .Net. The client-server architecture of Prolog Beans allows the server to be a component of a distributed system. Prolog Beans was designed to handle large applications.

The second tier is composed of two components: the Logger and the Query Manager. Once the Logger receives a Java program it starts a JVM and subscribes for the desired events with the JPDA. It is also possible (but not implemented in the current prototype) that the Logger interacts with an already running JVM. Query Manager is responsible for constructing Prolog goals (queries) based on the invoked method and the passed arguments and sending the constructed goals to the Prolog Beans server. Once Query Manger receives answers, it forwards them back to the Tools Interface. In the case when the Query Manager receives a request to add a composite query, the query has to go through some security checks. Then the query is tested against standard execution history. The user is notified by an error message or the results of the saved composite query.

The third tier is composed form only one component: the Tools Interface which is a façade for the JavaDD Framework. The rational behind this design is that the framework is under continuous development and evaluation; therefore, a specific implementation may need to be replaced by another efficient implementation. Another example, we may wish to switch from Prolog Beans to a SQL server. The fourth tier has only one component: the User Interface that interacts with the Tools interface.

Figure 1

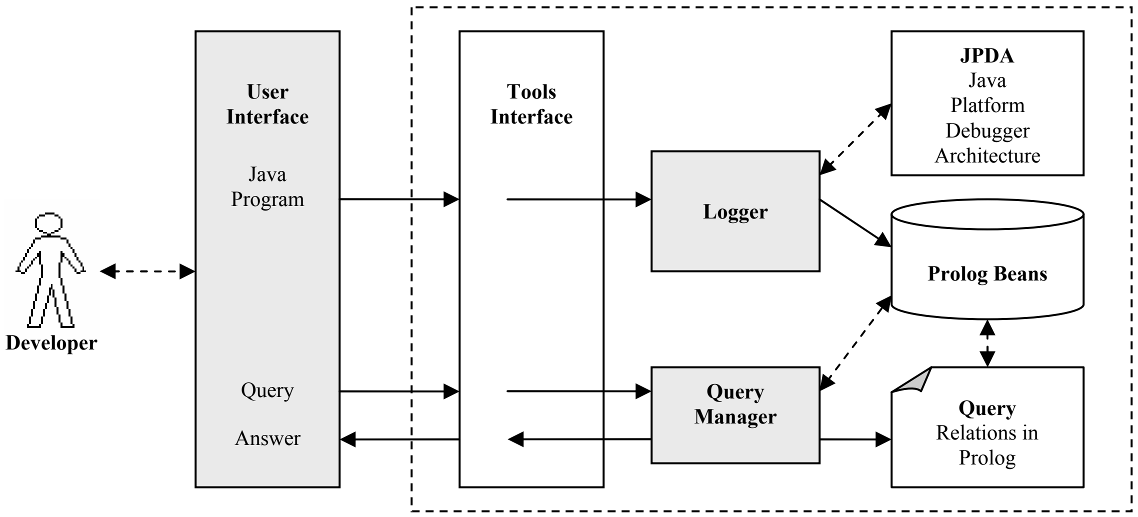


Figure 2: JEL BNF grammar

```
<events>            ::= event*
<event>             ::= event '('<id> , <execution-event> ')' '.'
<execution-event>   ::= <class-prepare>  |  <method-call>     |  <method-exit>  |  <set-field>
                      | <get-field>      |  <data-structure>   |  <exception>
                      | <step>           |  <thread-start>     |  <thread-death>
<info>              ::= <location> ,  <thread>
<method-call>       ::= methodcall   '(' <info>   , (<instance> | <class>), <name>, <arguments> ')'
<method-exit>       ::= methodexit   '(' <info>   , (<instance> | <class>), <name>, <value>')'
<set-field>         ::= setfield     '(' <info>   , (<instance> | <class>), <name>, <value>')'
<get-field>         ::= getfield     '(' <info>   , (<instance> | <class>), <name>  ')'
<data-structure>    ::= datastructure'(' <info>   , <contents>')'
<exception>         ::= exception    '(' <info>   , <instance> , <message> , ( <location> | uncaught ) ')'
<step>              ::= step         '(' <info>   , <local-variable-list> ')'
<class-prepare>     ::= memberfields '(' <thread> , <class> , <member-fields> ')'
<thread-start>      ::= threadstart  '(' <thread> , <thread-group>  ')'
<thread-death>      ::= threaddeath  '(' <thread> , <thread-group>  ')'
```

3.2 JEL: Java Events Log

JEL extends the work that JIVE and JyLog [19] have implemented similar recording techniques based on logging in XML. JEL describes the basic events in program execution history and can be easily modified to include a sophisticated description of static and dynamic information about a given program. The implemented prototype supports the description of 10 events: method call, method exit, set field, get field, data structure contents, exception, step, thread start, thread end, and member fields. Figure 2 shows part of the BNF grammar of JEL. Appendix A gives a full execution history in JEL for a binary search tree program. The remainder of this section gives more details about individual events. Each event has a unique id in addition to other specific information; objects are identified by their class and a unique id.

- **Method call event** records information about a method call. The event describes the source code location of the first executable line of the method body, the thread within which the invocation occurred, the class or the instance that this method was invoked on, method name, and method arguments.
- **Method exit event** records similar information as method call event. Method exit event records the returned value of the method call instead of the arguments.
- **Set Field event** records information about the thread where this event occurred, the source location where the field was set to a new value, the instance or the class

where this field is assigned to a new value and the new value.

- **Get Field event** records similar information as the set field event without providing the value of the field.
- **Data Structure event** is recorded after a method entry event and method exit event, and also after a set field event if the type of the field being assigned a new value is a data structure, and at user defined break points. The data structure can be array or a Collection instance. The event describes the source code information of the event targeted the recording of the data structure.
- **Step event** describes the thread and the source code location information in addition to names and values of visible local variables in each step. The program control flow can be inferred form step events.
- **Exception event** records information about the thread in which this exception is thrown or caught, the source code location, the exception instance, the exception message, and the catch location if it is caught or the uncaught keyword other wise.
- **Thread Start and Thread Death events** record the starting or the ending of a thread. The thread group is also recorded.
- **Class prepare event** records information regarding member fields and methods of a given class. Currently only recording member fields is implemented.

4. DECLARATIVE DEBUGGING

Declarative debugging is based upon a query catalog and execution history, which is recorded as a database. The database is populated by entries corresponding to execution events which are specified by JEL. Figure 3 gives the basic relations defined in the database schema.

Relation	Fields
methodcall	location, thread, instance-class, name, arguments
methodexit	location, thread, instance-class, name, returned-value
setfield	location, thread, instance-class, name, value
getfield	location, thread, instance-class, name
datastructure	location, thread, contents
exception	location, thread, instance, message, caught-uncaught
step	location, thread, variables
classprepare	thread, class, member-fields
threadstart	thread, thread-group
threaddeath	thread, thread-group

Figure 3

The query catalog given in Table 2 in Appendix A can be organized under three categories: queries on specific events, queries on execution history, and query management. Section 4.1 discusses queries on specific events. There are five kinds of queries over the execution history and are illustrated in section 4.2. Query management techniques are discussed in section 4.3. Table 2 in Appendix A gives an overview of the query catalog.

4.1 Queries on Specific Events

Query Where an Event Occurred. In object-oriented programming, execution events occur within an environment. An environment is an instance object and an instance method invocation or else it is a class and a static method invocation. This environment represents the enclosing environment for an event. The instance or the class is referred to as the enclosing instance or enclosing class and the method is referred to as the enclosing method for the event. The enclosing environment for a given event can be obtained by the where query given in Figure 4 for the following five events: set field event, get field event, method call event, object instantiation event which is recorded as method call event to the method <init> and exception event.

```
/*  Find the enclosing environment: the enclosing method  */
/*  and the enclosing class or instance for a given event      */

where( Event,  event( id(CallID) , EnclosingEnvironment)):-
        event( id( ID ) , Event ),
        enclosing_method( id(ID) , id( CallID ) , _),
        event( id(CallID) , EnclosingEnvironment).
```

Figure 4: The ‘ where’ query

Example 1: Where did a field assignment occur?

Consider the WhereExample program given in Figure 5. Class WhereExample has a static member field d which is assigned in method m and in the constructor of the HelperClass. The programmer wishes to know where the member field d is assigned. Q1 in Figure 6 shows how to query about the enclosing environment in which the member field d was assigned to any value. Results are returned in the Environment variable which is the enclosing method call event. A1 gives the answer to Q1 indicating that the assignment occurred in a call to method m in instance of WhereExample whose unique id is 417 and in the constructor of an instance of HelperClass whose id is 425. In some debugging scenarios, the question where a field was assigned a specific value arises. Q2 asks where field d was assigned to an instance of Double whose int value is 22.56.

```
public class WhereExample {
   public static Double d;
   public WhereExample() { m(); }
   public void m(){ d = new Double(22.56); }
   public static void main(String[] args) {
     WhereExample example = new WhereExample();
     HelperClass  helper  = new HelperClass();
   }
}
class HelperClass{
   public HelperClass(){ WhereExample.d = new Double(100.00); }
}
```

Figure 5

```
Q1  | ?-  where( setfield( _ , _ , classname( 'WhereExample') ,
                          name( 'd' ) , _ ),
              Environment).

A1  Environment =
    event( id(7),
            methodcall( location('WhereExample.java',4) ,
                          thread(main) ,
                          instance('WhereExample',417) ,
                          name(m) ,
                          arguments([ ])))
    Environment =
    event(id(19),
          methodcall( location('WhereExample.java',11) ,
                        thread(main) ,
                        instance('HelperClass',425) ,
                        name('<init>') ,
                        arguments([ ])))
Q2  | ?- where( setfield( _ , _ , classname( 'WhereExample') ,
                          name( 'd'),
                          value(instance( 'java.lang.Double' , _,
                                value('22.56’)))),
              Environment).

A2  Environment =
    event( id(7),
            methodcall( location('WhereExample.java',4) ,
                          thread(main) ,
                          instance('WhereExample',417) ,
                          name(m) ,
                          arguments([ ])))
```

Figure 6

Query the State of an Object. Querying the state of an object is concerned with the encapsulation aspect of object-oriented programming. The state of an object is captured in the values of its member fields and public and protected member fields of its super classes. Querying the state of an object helps in verifying class invariant. Case study 2: The Double Descent gives an example where investigating the state of an object revealed the reason for the program to throw a class cast exception.

Query To Validate Contracts. In design-by-contract [27][28][30] the client has to meet preconditions or specific requirements in order to be able to call a certain method. These requirements are usually constraints on the arguments and the state. Our method generalizes the requirement to be imposed on any execution event and not only on method calls as in DBC. Thus those requirements are candidate queries. There are three factors that can affect the execution of a given event within the enclosing method. These factors can be considered as the requirement for an event.

1. Arguments values
2. The returned value of all preceding method calls to a given event within the same enclosing method
3. Local variable values before the execution of the event

Analogously, the post-condition in DBC is the effect that the called method promises upon it is completion. Our methodology generalizes this idea to all executed events. The effect of the execution of an event on the enclosing method can appear in the following three areas.

1. The returned value of the enclosing method
2. Methods that have been called after the execution of the event within the same enclosing method
3. Local variables values after the execution of the event

DBC fails to specify directly that some other methods need to be called before or after a given method. Having recorded the execution history it is possible to inspect whether a certain method(s) has been called before or after a given event. Case study in section 5.2 discusses the application of these queries.

Group Method Calls according to Call Chain. Compared with the traditional procedural paradigm, the object-oriented paradigm engenders the use of many small methods and greater method interaction. Thus, posing queries regarding the interaction between objects is essential in the debugging process and in the understanding of object oriented programs in general. A method call can be viewed as a message whose content is the passed arguments. Each message has a response which is the returned value or void. A message can have no response if it exits abnormally, i.e. throws an exception. Call chain can serve as a proof of certain execution or as a way to inspect argument values propagated through the chain of calls. Case study 1: The Traveling Null Pointer discussed in section 5.2, shows how the call chain query is effective in locating a null pointer. Figure 7 illustrate the call_chain relation in Prolog.

```
/* list of id's presenting a call chain to event ID starting from Start */

call_chain( id(ID) , id(Start) , Out ):-
           call_chain_helper( id(ID) , id(Start) , [ ] , Out).

/* There two base cases: first, when search go less than the lower bound */
/* Second base case occurs when encounter main method.                  */
/* One recursive case to search for the rest of the call chain          */

call_chain_helper( id(ID) , id(Start), [Last|PathRest] ,  [Last|PathRest] ):-
    ID =< Start, !.
call_chain_helper( id(MainID) , id(_)    , Path, [MainID|Path] ):-
    event( id(MainID) ,
         methodcall(_ , _ , _ , name('main'),
                 arguments( [value( instance( 'java.lang.String[ ]', _ ))]))).
call_chain_helper( id(ID) , id(Start) , PathSoFar , Path ):-
    enclosing_method(id(ID) , id(Enclosing) , id(_)),
    call_chain_helper( id(Enclosing) , id(Start) ,[ID|PathSoFar] , Path).
```

Figure 7

4.2 Queries over the Execution History

Execution History Subset. In most cases the erroneous code is contained in a small segment of the source code and the erroneous behavior is contained in a small segment of the execution history. Consider a program which has a call chain of, say, 10 method calls leading to the execution of a certain event. And the erroneous behavior is suspected to be in the last three method calls of the call chain. A query over a domain of events starting at the eighth call and ending in the 10th call is more focused than a query over the entire call chain.

Comparing Histories. Eisenstadt [6] describes the "Dump & Diff" as a technique to locate errors. This technique works as follows. The output of print statements is saved to two text files corresponding to two different executions; the two files are then compared using a source-compare "diff" utility, which highlights the difference between the two outputs. This technique can be adapted to query multiple execution histories and to compare the results of multiple queries over the same execution history. Comparative queries can be helpful to see the difference between data structure contents, and call chains and much more. Comparative queries are known to be helpful in isolating errors related to software maintenance.

Call Tree. Grouping method calls according to a call tree is motivated by the same reasons to group method calls according to a call chain. However a call tree depicts a different kind of interaction among objects. Method calls that are involved in a call tree collaborate in achieving one task those method are not necessarily dependant on each other. Each method can be independent of the other methods unlike method calls in a call chain which are dependent on each other, i.e. the called method depends on the caller.

Gathering Data. Eisenstadt [6] in his study on how bug were found in 51 cases gathered from professional programmers found that in 27 cases, the bug was found by gathering data regarding the execution of the program. Novice programmers use the output print statement to gather data regarding some variables at certain points in the program execution. Traditional debuggers provide a watch on a field or a variable to allow the programmer to see how a variable or a field changes; however, it does not provide a full history of values assigned to that variable or the watched field. An advantage of our technique over the traditional debuggers is that the programmer is not responsible for gathering the data, and neither is he responsible for stepping over the program to observe the changes in the value of a variable or a field. Our methodology recognizes the importance of the data gathering phase in the debugging process, and extends to gather data regarding the following

1. Member field value history
2. Local variable value history (understating loop execution)
3. History of arguments of method calls
4. History of return value of method calls
5. Contents of data structure.
6. History of contents of data structure.
7. All Class instances and their states (under standing user defined data structures)
8. Thread status such: running and exited threads

Example 2: History of arguments to a tail recursive factorial.

Figure 8 shows an implementation of the factorial functions using the accumulator passing style. Q1 in Figure 9 shows how to pose the arguments_history query over the entire 34 events recorded for that program. The query can be refined to be posed on a specific domain of events rather than the entire history. In the answer given A1, the id preceding the arguments is the event id for that specific method call so the programmer can further investigate the execution of any call.

```
public int factorial(int n, int accum)
{
      if(n == 0) return accum;
      else return factorial(n-1 , n * accum);
}
```

Figure 8

```
| ?- arguments_history(id(0) ,
                        id(34) ,
                        methodcall( _ , _ ,_ , name('factorial') , _ ) ,
                        History).
History = [
            [ id(7),   arguments([ value(5), value(1) ]) ],
            [ id(9),   arguments([ value(4), value(5) ]) ],
            [ id(11), arguments([ value(3), value(20) ]) ],
            [ id(13), arguments([ value(2), value(60) ]) ],
            [ id(15), arguments([ value(1), value(120) ]) ],
            [ id(17), arguments([ value(0), value(120) ]) ]
          ]
```

Figure 9

Query about Statement Execution. One of the most recurring questions in the debugging process is whether a certain statement has been executed or not. Novice programmers find the answer for such a question by inserting multiple print statements in their code. Advanced developer would insert break points using a traditional debugger to verify whether a given statement has been executed or not. The answer to this question is either yes or no. We propose the following seven queries:

1. Was a given conditional statement executed?
2. Was a given method called?
3. Was a member field assigned?
4. Is there an instance of a specific class?
5. Was a specific exception caught?
6. Is a given thread still running?
7. Has a given thread exited?

4.3 Query Management

Compose and Save Queries. The ability to compose queries provides a way to adapt queries to recurring bug patterns as well as to the individual needs of the developer. Also it allows advanced user to add highly specialized queries. The idea is similar to the idea behind the Emacs system that allows the user to add macros dynamically to add functionally to the system. Composed queries guarantee the flexibility and extendibility of our framework. Allowing the user to add queries dynamically results in a general purpose dynamic analysis tool since there is no requirement on the contents of the macro besides it must be valid Prolog code.

Save Queries Answers. Calculating a query on a program history that has a million or more events is costly and time demanding. In many debugging scenarios the programmer may go back to examine the results of previous queries or would like to compare them. Recomputing a query on such execution history is wasteful; therefore, queries and their answers should be saved so it is convenient for the programmer to examine the previous results and perform comparisons.

5. EVALUTATION

5.1 Errors in Object Oriented Programs

Our studies started by building a database of incorrect Java programs and the corrected programs. Programs used in our research are gathered from [1][2] [5][26], the Java language specification and some errors produced by the first author, students and colleagues. The database is organized according to the Knuth classification [20] modified to cover object oriented and concurrent programming errors. Errors are classified into eleven categories:

1 OO programming related: OO
2 Concurrent programming : C
3 Design anomalies: DA
4 Algorithmic anomalies: A
5 Blunders: B
6 Data disasters: D
7 Forgetfulness: F
8 Language lossage: L
9 Mismatches: M
10 Robustness: R
11 Typographic trivia: T

The set of queries was used to debug 24 programs gathered from different resources. Table 2 in appendix A gives detailed explanation of each query. We have recorded the main queries that were used to locate the source of the erroneous behavior in each program. Table 1 shows these results. The second column gives the category and subcategory of the erroneous program, the third column lists queries used to isolate the bug. In some case more than one query could have been used to debug the program, in those cases we list queries separated by "or" otherwise queries are separated by comma.

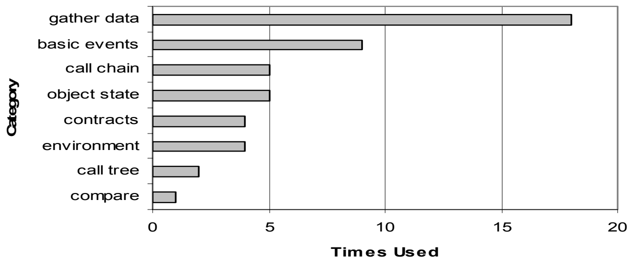
Figure 10 show a graph where the 32 queries were plotted versus the number of their usage in the experiment. We have found that 11 queries were not used, 12 queries were used only once, two queries were used exactly two times and seven queries were used more than two times. The erroneous programs used in this study do not capture all possible debugging scenarios; therefore; in order to conclude the usefulness of the unused 11 queries, the experiment need to be repeated on a wider range of erroneous programs. Gathering such programs is a difficult task since programmers rarely document the errors they make. The following seven queries were used the most:

1. Q5: the enclosing environment for an exception.
2. Q6: object state
3. Q14: local variable history
4. Q16: call chain
5. Q20: history of arguments of method calls
6. Q26: whether a given conditional statement executed or not.
7. Q27: whether a given method called or not

To validate the usage of queries according to their categories, results of the experiment are restructured according to the category rather to individual queries. In another words, queries are counted according to the category they belong to. Figure 11 shows a graph the represents the number of times a category was used. Eisenstadt [6] had found that data gathering is the most predominant technique used in debugging. Our results agree with Eisenstadt's finding. The query management category is not considered in this study due to the size and the simplicity of the subject erroneous programs.

	Error Pattern	Query
E1	C-The Orphaned Thread	Q23, Q24
E2	OO-The Split Cleaner	Q16, Q17, Q20
E3	OO-The Imposter Type	Q26
E4	OO-The Broken Dispatch	Q15, Q16
E5	OO-The Liar View	Q13 or Q17 or Q27
E6	OO-The Double Descent	Q5, Q6
E7	OO-The Null Flag	Q5, Q16
E8	OO-The Dangling Composite	Q9
E9	OO-The Run-On Initialization	Q5, Q6, Q16, Q30
E10	OO-The Traveling Null Pointer	Q5, Q10, Q11, Q16
E11	OO-Sibling objects blunder	Q6, Q20
E12	OO-Language Lossage	Q27
E13	A-Logic	Q20 , Q7
E14	A-Off-By-One	Q14
E15	B-Expression	Q6, Q20
E16	B-Expression	Q18, Q26, Q27
E17	B-Method	Q20, Q21
E18	B-Var	Q20
E19	B-Var	Q14
E20	D-Index	Q14
E21	D-Limit	Q14, Q22
E22	F-Init	Q6, Q29
E23	F-Location	Q14, Q26
E24	F-Location	Q18, Q27

Table 1



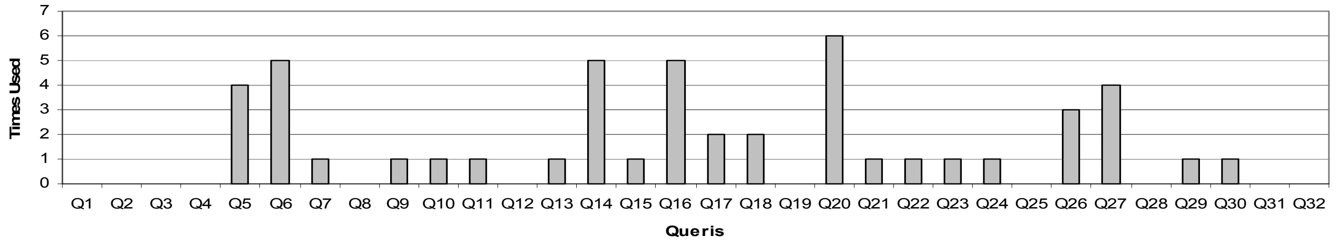


Figure 10

Figure 11

5.2 Case Studies

Case study 1: The Traveling Null Pointer

The Traveling Null Pointer bug pattern can be described as follows. A method call incorrectly returns a null pointer and the client of that method propagates the null pointer through a call chain, and finally a null pointer exception is thrown when the client code of the last call in the chain tries to de-reference the null pointer. In other words, the code that originates the null pointer and the code that de-references that pointer are far apart spatially and temporally. Figure 12 illustrate the Traveling null pointer bug pattern with Java code. The instance method "doSomeThing" in "FarAWayClass" returns a null pointer due to erroneous conditions. When this program is executed it reports a null pointer exception at line 14.

```
1   public class TheTravelingNullPointer {
2      public TheTravelingNullPointer() {
3                m1();
4      }
5      public void m1(){
6                FarAWayClass o = new FarAWayClass();
7                String result = o.doSomeThing();
8                m2(result);
9      }
10     public void m2(String result){
11               mN(result);
12     }
13     public void mN(String result){
14               if(result.equals("some result"))
15                         System.out.println("some result");
16               else
17                         System.out.println("other result");
18     }
19     public static void main(String[] args) {
20               new TheTravelingNullPointer();
21     }
22  }
23  class FarAWayClass{
24     public String doSomeThing(){
25               // some code that result in returning a null
26               return null;
27     }
28  }
```

Figure 12

Figure 13 shows the three queries used in debugging the Traveling Null Pointer program and their answers. Q1 query regarding the environment where the exception is throw. A1 indicates that the enclosing method is mN whose single argument is null. The current question is where the null pointer originated and the call to the enclosing method occurred at event id 25. Q2 enquires full detail call chain starting from event id 0 up to event id 25. A2 shows that method m1 called method m2 that called method mN. This initial call to main and the constructor is omitted for simplicity of presentation.

By investigating the arguments passed to m2 it is clear that it is also null. Method m2 is called from m1. Method m1 is call at event id 8. When looking at the source code of method m1, the programmer concludes that the local variable "result" holds a null value since it is passed as the argument to m2. As this point all figures point to the method o.doSomeThing as the error source. To make sure that o.doSomeThing is returning null pointer, Q3 asks regarding all methods that were called before m2 in the same enclosing environment. A5 confirms that o.doSomeThing has returned null pointer. Now the origin of the null pointer has been located and isolated that is the beginning of the fixing stage.

```
Q1   where(exception( location( 'TheTravelingNullPointer.java', 14),
                         _ , _ , _ , uncaught),
            Environment).

A1   Environment =
     event(id(25),
           methodcall(location('TheTravelingNullPointer.java',14),
                         thread(main),
                         instance('TheTravelingNullPointer',429),
                         name(mN),
                         arguments( [ value( null ) ] ) ) )

Q2   full_detail_call_chain( id(25) , id(0) , EventListOut ).

A2   event(id(25),
            methodcall( location( 'TheTravelingNullPointer.java',14),
                          thread('main'),
                          instance( 'TheTravelingNullPointer',429),
                          name('mN'),
                          arguments( [ value(null) ] )))

     event(id(23),
            methodcall( location( 'TheTravelingNullPointer.java',11),
                          thread('main'),
                          instance( 'TheTravelingNullPointe'r,429),
                          name('m2'),
                          arguments( [ value(null) ] )))

     event(id(8),
            methodcall(location( 'TheTravelingNullPointer'.java,6),
                         thread('main'),
                         instance( 'TheTravelingNullPointer',429),
                         name('m1'),
                         arguments([ ])))

Q3   pre_called_methods(id(23) , OutList ).
A3   [event(id(17),
            methodcall( location('TheTravelingNullPointer.java',26),
                          thread('main'),
                          instance('FarAWayClass',431),
                          name('doSomeThing'),
                          arguments([ ]))),
     event(id(20),
            methodexit(location('TheTravelingNullPointer.java',26),
                         thread('main'),
                         instance('FarAWayClass',431),
                         name('doSomeThing'),
                            value( null ) )),]
```

Figure 13

Case study 2: The Double Descent

The Double Descent bug pattern [1] is an example of erroneous processing of composite data structure. The program in Figure 14 builds a binary tree from random numbers and check to see if two adjacent nodes have zeros. Tree is the super class of Leaf and Node and has two abstract methods: insert and hasAdjacentZeros which return true if the values stored in a node and one of its right or left children are zeros, otherwise it returns false. The for loop at lines 39-41 in the main method constructs a tree of 52 Node instances and one instance of Leaf. The values of 50 instances are random numbers. Running the program while commenting out line 42 may throw a class cast exception. Running the program while line 42 is commented in will increase the chance for the exception to be thrown.

The class cast exception error message indicates that class cast exception occurred while executing line 32 within the hasAdjacentZeros method body. The double descent program will succeed in some cases where two zeros are generated among the random values. On the other hand the program will fail when recursion reaches a child that is a Leaf instance. When the search has to go all the way down the tree, the exception will be thrown. Line 32 shows the code that result in throwing the exception. It is clear that it only considers the children to be instances of Node and it does not account for leaf instances.

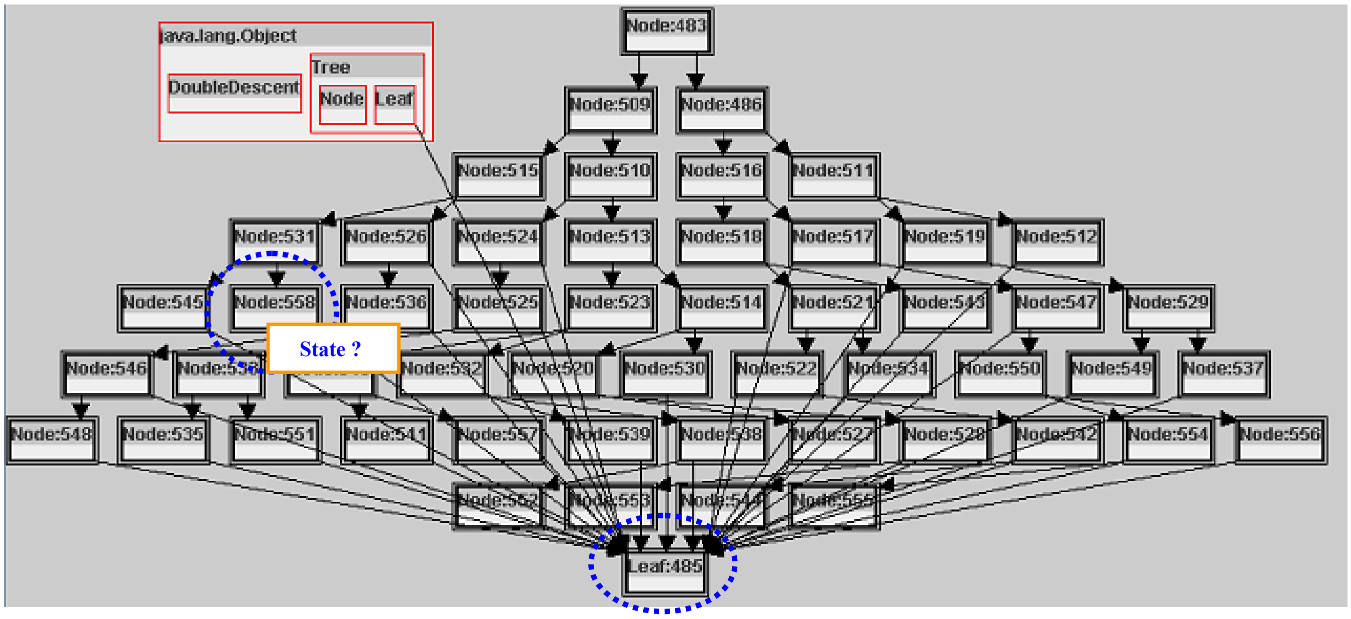
Query-based debugging can isolate this bug via 2 queries. Figure 15 shows the queries and their answers. The important answers are in bold and underlined. Q1 enquires about the environment where the exception occurred. A1 gives details about the enclosing method and the enclosing instance. The enclosing instance is a Node object its unique id is 558, and the enclosing method is hasAdjacentZeros that is defined in DoubleDescent.java at line 31. Q2 is a query about the state of Node object its unique id is 558 at event id 3246 that is the id of the call to the enclosing method. Investigating the object state shows that the right and left children are Leaf instances, and that accounts for the class cast exception. The id filed preceding the name in the instance filed is the id for the set-field event that assigned this filed the corresponding value.

```
Q1   | ?- where( exception( _ , _ ,
                     instance( 'java.lang.ClassCastException' , _ ) ,
                     _ ,
                     uncaught )  ,
             Environment).

A1   Environment =
     event(id(3246),
            methodcall(location('DoubleDescent.java',31),
                     thread(main),
                     instance('Node',558) ,
                     name(hasAdjacentZeros) ,
                     arguments([])))

Q2   | ?- object_state( id(3246), instance('Node',558), State).

A2   State =
      [
             instanceField( id(3210) ,
                          name(right),
                          value(instance('Leaf',485)) ),
             instanceField( id(3208),
                          name(left) ,
                          value(instance('Leaf',485)) ),
             instanceField( id(3206),
                          name(value),
                          value(0) )
     ]
```

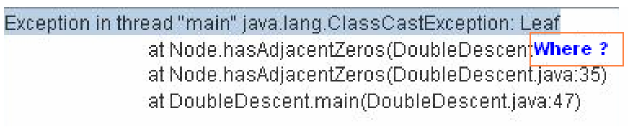
Figure 15

```
1   abstract class Tree {
2     public abstract boolean hasAdjacentZeros();
3     public abstract void    insert(int v);
4   }
5   class Leaf extends Tree  {
6     public static final Leaf ONLY = new Leaf();
7     private Leaf(){ }
8     public void insert(int v){
9       throw new UnsupportedOperationException( "Leaf.insert" );
10    }
11    public boolean hasAdjacentZeros(){ return false; }
12  }
13  class Node extends Tree  {
14    public int value;
15    public Tree left , right;
16    public Node(int v){
17       value = v;
18       left = Leaf.ONLY ;
19       right = Leaf.ONLY ;
20     }
21     public void insert(int v) {
22       if (value < v){
23          if (right instanceof Leaf) { right = new Node(v); }
24          else { right.insert(v); }
25       }else{
26          if (left instanceof Leaf) { left = new Node(v); }
27          else { left.insert(v); }
28       }
29     }
30     public boolean hasAdjacentZeros(){
31       return value == 0 &&
32       (( (Node) left).value  == 0 ||  ( (Node)right).value == 0)
33        || left.hasAdjacentZeros() || right.hasAdjacentZeros();
34    }
35  }
36  public class DoubleDescent {
37    public static void main(String argv[ ]){
38       Tree tree = new Node(500);
39       for(int i = 0 ; i < 4 ; i++){
40         tree.insert((int)(Math.random() * 1000000));
41       }
42       tree.insert(0);
43       tree.hasAdjacentZeros();
44    }
45  }
```

Figure 14

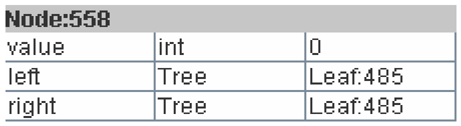


Object diagrams of the Double Descent program.



```
Exception in thread "main" java.lang.ClassCastException: Leaf
        at Node.hasAdjacentZeros(DoubleDescen Where ?
        at Node.hasAdjacentZeros(DoubleDescent.java:35)
        at DoubleDescent.main(DoubleDescent.java:47)
```

output console



Node:558		
value	int	0
left	Tree	Leaf:485
right	Tree	Leaf:485

state of object Node:558

Figure 16

6. TOWARDS VISUAL QUERIES

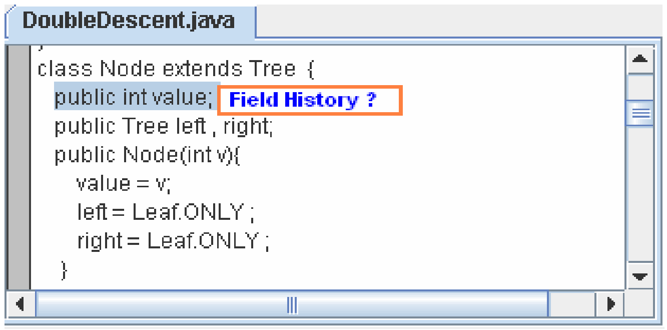
Our goal to make query-based debugging accessible to the "average programmer" will not be fulfilled without defining an easy-to-use interface that allows the user to pose higher level declarative queries. This section gives the final picture of the proposed declarative debugging method. Currently a project to integrate JIVE and the JavaDD framework is under active development at the University at Buffalo. JIVE design is based on the following seven criteria for effective visualization of object oriented programs [11]

1. Depict objects as environment of method execution.
2. Provide multiple views of the execution state.
3. Visualize history of execution and method interaction.
4. Support forwards and backwards execution of programs
5. Support queries on the runtime state.
6. Produce clear and legible drawings.
7. Uses exiting Java technologies.

The JIVE system provides multiple views: the object states at different levels of granularity, a sequence diagram to capture the history of execution and source code viewer to show the code being executed at the current point. In the remainder of this section we will show two examples on how to use JIVE as an interface to JavaDD. Screen shots presented in this section are taken using JIVE and manually edited to illustrate the declarative debugging technique.

The Double Descent Revisited. The bug in the Double Descent program discussed in 5.2 can by found in two simple steps. The user just highlights the exception in the program output window and right clicks on the highlighted text, then selects the desired query as shown the lower left corner of Figure 16. The information obtained form the visual interface is then translated to Prolog query by JavaDD. Query answer is translated back to a visual answer by JIVE. The upper part of the figure indicates that the exception has occurred in object Node:558 which is surrounded by a circle. From the visualization it is clear that Node:558's children are all leafs. The state of the Node 558 can be further inspected by right click on the contour diagram of the node and select a query from the list. The visual answer is given in the right left corner of the figure. The sate of Node:558 just before the exception is thrown is 0 as a value and the right and left children are leafs.

Consider a smaller version of the Double Descent program which builds a binary tree of 5 nodes that stores random generated numbers. The programmer wishes to know the values inserted in the tree. Figure 17 how such query can be posed declaratively. The user first highlights the member field "value", then right clicks on the highlighted area and selects the field-history query form the list. The visual query is translated to a field history query on the entire history starting from event 0 and ending in event 136, also since the query is posed on the source



```
DoubleDescent.java

class Node extends Tree {
   public int value;  Field History ?
   public Tree left , right;
   public Node(int v){
      value = v;
      left = Leaf.ONLY ;
      right = Leaf.ONLY ;
   }
```

```
?- field_history( id(0) , id(136) ,
                  instance('Node' , _ ),
                  name( 'value'),
                  HistoryList).

HistoryList = [
        instanceField( id(11),   name(value), value(500) )
        instanceField( id(37),   name(value), value(458) )
        instanceField( id(61),   name(value), value(29) )
        instanceField( id(85),   name(value), value(544) )
        instanceField( id(112),  name(value), value(333) )
]
```

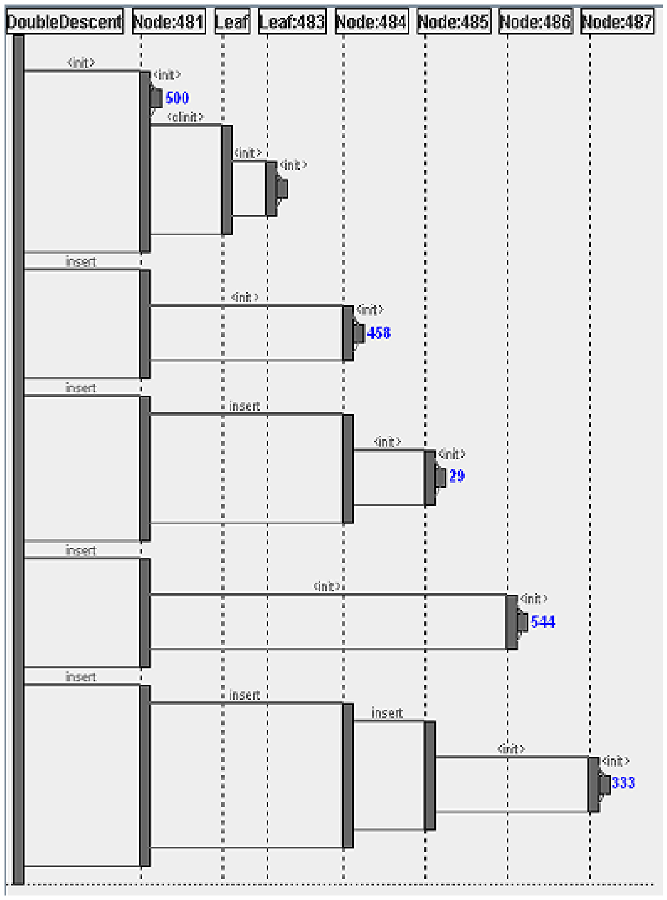


Figure 17

code level the instance id is ignored so the query will be posed on all class instances. The lower part of the figure shows how the answer is translated to visual notation. The sequence diagram shows not only the values of the member filed, but also when this value was assigned. The horizontal axis list all objects according to the order of their instantiation; therefore, the environment where the assignments occurred can be inferred from the sequence diagram.

7. CONCLUSIONS AND FUTURE WORK

We believe that our proposed paradigm of declarative debugging is a simple and effective method for debugging. As noted earlier, we believe that both procedural and declarative forms of debugging are useful. We presented many examples to show when declarative debugging is effective in eliciting information about object states and execution histories. Our proposed query catalog is based upon an extensive study of errors in object oriented programs. The visual interface for queries in the JavaDD framework frees the programmer from being burdened by using an unfamiliar textual query language, and provides many features that facilitate the debugging process. JavaDD can be easily interfaced from traditional debugger and as well as other visualization tools.

We have only sketched the visual interface for posing queries, and our current work is devoted to a full exploration of the visual interface. We are also applying our declarative debugger to larger programs, in order to gain a better understanding of the methodology and its potential limitations. We are also exploring the performance characteristics of the declarative debugger both in terms of the space and time needed for various types of queries.

An interesting dimension of future work is to explore the coupling among queries. For example, if Qi and Qj are coupled, then whenever Qi is used there is a good likelihood that Qj is also used. The coupling suggests composing Qi and Qj into one query. Another dimension to investigate is the relation between a set of queries and a recurring bug pattern. If such relation exists then combined with dynamic and static analysis there is a possibility for an expert system that can help in isolating bug patterns automatically.

ACKNOWLEDGMENTS

The JavaDD framework builds upon the extensive capabilities developed in the JIVE framework which was developed by Paul Gestwicki at the University at Buffalo.

8. REFERENCES

[1] Allen, E. *Bug Patterns in Java*. Apress, 2002.

[2] Bar, A. *Find The Bug*. Addison Wesley, 2005.

[3] Bederson, B. B., Grosjean, J., and Meyer, J. Toolkit Design for Interactive Structured Graphics. *IEEE Transactions on Software Engineering*, 30 (8), pages 535-546. 2004.

[4] Blackwell, et al. Cognitive Dimensions of Notations: Design Tools for Cognitive Technology. In *Proceedings of the 4th International Conference on Cognitive Technology: Instruments of Mind*. Springer-Verlag, 2001.

[5] Bloch, J. *Effective Java: Programming Language Guide*. Addison Wesley 2001.

[6] Eisenstadt, M. Tales of Debugging From The Front Lines. *Empirical Studies of Programmers* V,1993.

[7] Fowler, et al. *Refactoring improving the design of existing code*, Addison Wesley, 1999.

[8] Gamma, et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley 1995.

[9] Gestwicki, P.V.. *Interactive Visualization of Object Oriented Programs*. Ph.D. Thesis, CSE Department, University at Buffalo, NY, 2005.

[10] Gestwicki, P.V., and Jayaraman, B. Interactive Visualization of Java Programs, *Proc. IEEE Symposium on Human-Centric Computing, Languages, and Environments*, pp. 226-235, Washington, DC, September 2002.

[11] Gestwicki, P.V. and Jayaraman, B. Methodology and Architecture of JIVE. In *Proceedings of ACM Symposium on Software Visualization*, pages 95-104, 2005.

[12] Girgis, et al. Visual Queries for Interactive Execution of Java Programs. Technical Report 2004-19, CSE Dept., University at Buffalo, December 2004

[13] Girgis, H., Jayaraman, B., and Gestwicki, P.V. Visualizing Errors in Object Oriented Programs. *In OOPSLA 2005 Conf. Companion,* pages 156 – 157, 2005.

[14] Goldsmith, Simon. O'Callahan, Robert. Aiken, Alex. Relational Queries Over Program Traces. In *Proceedings of object oriented programming systems languages and applications*, pages 385-402, 2005.

[15] Hajiyev, et al. CodeQuest: Querying Source Code with DataLog. *In OOPSLA 2005 Conf. Companion*, pages 102 – 103, 2005.

[16] Hendrix, T., et al. An Extensible Framework for Providing Dynamic Data Structure Visualizations in a Light-weight IDE. In *Proceedings ACM SIGCSE Conference,* 2004.

[17] Hovemeyer, D. and Pugh, W.. Finding Bug is Easy. *ACM SIGPLAN Notices*: 39, 12: 92-106, 2004.

[18] JPDA: http://java.sun.com/j2se/1.5.0/docs/guide/jpda/index.html

[19] Jylog: http://sourceforge.net/projects/jylog/

[20] Knuth, D.E.. The Errors of TeX. *Software—Practice & Experience*, 19(7): 607-685, 1989.

[21] Ko, A. and Myers, B.. Designing the Whyline: A Debugging Interface for Asking Questions about Program Behavior. In *Proceedings of the ACM Conference on Human factors in Computing Systems*, pages 151-158, 2004.

[22] M. Kolling and J. Rosenberg. BlueJ: The Interactive Java Environment, In *Proceedings of ACM SIGCSE Conference*, New Orleans, LA, 1999

[23] Lencevicius R. Hölzle, U., and Singh, A. Query-based debugging of object-oriented programs. In *Proceedings of ACM Conference on Object Oriented Programming Systems Languages and Applications*, pages 304-317, 1997.

[24] Lönnberg, J. et al. MVT: A Visual Testing for Software. In *Proceedings of AVI '04: Automated Visual Interfaces*, pp. 385 – 388, ACM Press.

[25] Martin, M., Livshits, B., and Lam, M. Finding Application Errors and Security Flaws Using PQL: a Program Query Language. In *Proceedings of the ACM Conference on Object Oriented Programming Systems Languages and Applications*, pages 365-383, 2005.

[26] Metzger, R.C. *Debugging by Thinking: A Multidisciplinary Approach*. Elsevier Digital Press, 2004.

[27] Meyer, B. *Object-oriented Software Construction*, 2nd ed. Prentice Hall, 1997.

[28] Meyer, B. *Eiffel: The language*. Prentice Hall. 1992.

[29] Meyers, Scott. *Effective C++: 50 Specific Ways to Improve Your Programs and Design*. Addison Wesley 1997.

[30] Mitchell, R. McKim J. *Design by Contract by Example*. Addison-Wesley 2002

[31] Pauw, W.D., Helm, R. Kimelman, D, and Vlissides, J. Visualizing the behavior of object-oriented systems. In proceedings of object oriented programming systems languages and applications, pages 326-337, 1993.

[32] Perry, D. E. and Evangelist, W.M. An Empirical Study of Software Interface Faults --- An Update. In *Proceedings of the Twentieth Annual Hawaii International Conference on Systems Sciences*, January 1987, Volume II, pages 113-126

[33] PrologBeans: http://www.sics.se/sicstus/docs/latest/html/sicstus/PrologBeans.html#PrologBeans

[34] Rosenblum, D.. Towards a Method of Programming with Assertions. In *Proceedings of the 14th International Conference on Software Engineering*, pages 92 – 104, 1992

[35] Rutar, N., Almazan, C., and Foster, J. A Comparison of Bug Finding Tools for Java. In *Proceedings of the 15th International Symposium on Software Reliability Engineering*, pages 245 – 256. 2004.

A. APPENDIX

```
event( id( 0 ) , threadstart( thread( 'Signal Dispatcher' ) , threadgroup( 'system' ))).
event( id( 1 ) , threadstart( thread( 'main' ) , threadgroup( 'main' ))).
event( id( 2 ) , memberfields( thread( 'main' ) , classname( 'DoubleDescent') , [  ] )).
event( id( 3 ) , methodcall( location( 'DoubleDescent.java' , 46 ) , thread( 'main' ) , classname( 'DoubleDescent') , name( 'main' ) , arguments([ value(
                        instance( 'java.lang.String[]' , 459 ))  ]) )).
event( id( 4 ) , datastructure( instance( 'java.lang.String[]' , 459 ) , contents([ ]))).
event( id( 5 ) , memberfields( thread( 'main' ) , classname( 'Tree') , [  ] )).
event( id( 6 ) , memberfields( thread( 'main' ) , classname( 'Node') , [  instacefield( 'value' ), instacefield( 'left' ), instacefield( 'right' )  ] )).
event( id( 7 ) , methodcall( location( 'DoubleDescent.java' , 16 ) , thread( 'main' ) , instance( 'Node' , 474 ) , name( '<init>' ) , arguments([ value( 500)  ]) )).
event( id( 8 ) , methodcall( location( 'DoubleDescent.java' , 1 ) , thread( 'main' ) , instance( 'Node' , 474 ) , name( '<init>' ) , arguments([ ]) )).
event( id( 9 ) , methodexit( location( 'DoubleDescent.java' , 1 ) , thread( 'main' ) , instance( 'Node' , 474 ) , name( '<init>' ) , 'void' )).
event( id( 10 ) , step( location( 'DoubleDescent.java' , 17 ) , thread( 'main' ) , [ ])).
event( id( 11 ) , setfield( location( 'DoubleDescent.java' , 17 ) , thread( 'main' ) , instance( 'Node' , 474 ) , name( 'value' ) , value( 500))).
event( id( 12 ) , step( location( 'DoubleDescent.java' , 18 ) , thread( 'main' ) , [ ])).
event( id( 13 ) , memberfields( thread( 'main' ) , classname( 'Leaf') , [  classfield( 'ONLY' )  ] )).
event( id( 14 ) , methodcall( location( 'DoubleDescent.java' , 6 ) , thread( 'main' ) , classname( 'Leaf') , name( '<clinit>' ) , arguments([ ]) )).
event( id( 15 ) , methodcall( location( 'DoubleDescent.java' , 7 ) , thread( 'main' ) , instance( 'Leaf' , 476 ) , name( '<init>' ) , arguments([ ]) )).
event( id( 16 ) , methodcall( location( 'DoubleDescent.java' , 1 ) , thread( 'main' ) , instance( 'Leaf' , 476 ) , name( '<init>' ) , arguments([ ]) )).
event( id( 17 ) , methodexit( location( 'DoubleDescent.java' , 1 ) , thread( 'main' ) , instance( 'Leaf' , 476 ) , name( '<init>' ) , 'void' )).
event( id( 18 ) , methodexit( location( 'DoubleDescent.java' , 7 ) , thread( 'main' ) , instance( 'Leaf' , 476 ) , name( '<init>' ) , 'void' )).
event( id( 19 ) , setfield( location( 'DoubleDescent.java' , 6 ) , thread( 'main' ) , classname( 'Leaf') , name( 'ONLY' ) , value( instance( 'Leaf' , 476 )))).
event( id( 20 ) , methodexit( location( 'DoubleDescent.java' , 6 ) , thread( 'main' ) , classname( 'Leaf') , name( '<clinit>' ) , 'void' )).
event( id( 21 ) , step( location( 'DoubleDescent.java' , 18 ) , thread( 'main' ) , [ ])).
event( id( 22 ) , setfield( location( 'DoubleDescent.java' , 18 ) , thread( 'main' ) , instance( 'Node' , 474 ) , name( 'left' ) , value( instance( 'Leaf' , 476 )))).
event( id( 23 ) , step( location( 'DoubleDescent.java' , 19 ) , thread( 'main' ) , [ ])).
event( id( 24 ) , setfield( location( 'DoubleDescent.java' , 19 ) , thread( 'main' ) , instance( 'Node' , 474 ) , name( 'right' ) , value( instance( 'Leaf' , 476 )))).
event( id( 25 ) , step( location( 'DoubleDescent.java' , 20 ) , thread( 'main' ) , [ ])).
event( id( 26 ) , methodexit( location( 'DoubleDescent.java' , 20 ) , thread( 'main' ) , instance( 'Node' , 474 ) , name( '<init>' ) , 'void' )).
event( id( 27 ) , step( location( 'DoubleDescent.java' , 46 ) , thread( 'main' ) , [ ])).
event( id( 28 ) , step( location( 'DoubleDescent.java' , 47 ) , thread( 'main' ) , [ localvariable( name('tree') , value( instance( 'Node' , 474 )))  ])).
event( id( 29 ) , methodcall( location( 'DoubleDescent.java' , 22 ) , thread( 'main' ) , instance( 'Node' , 474 ) , name( 'insert' ) , arguments([ value( 400)  ]) )).
event( id( 30 ) , step( location( 'DoubleDescent.java' , 30 ) , thread( 'main' ) , [ ])).
event( id( 31 ) , step( location( 'DoubleDescent.java' , 31 ) , thread( 'main' ) , [ ])).
event( id( 32 ) , methodcall( location( 'DoubleDescent.java' , 16 ) , thread( 'main' ) , instance( 'Node' , 477 ) , name( '<init>' ) , arguments([ value( 400)  ]) )).
event( id( 33 ) , methodcall( location( 'DoubleDescent.java' , 1 ) , thread( 'main' ) , instance( 'Node' , 477 ) , name( '<init>' ) , arguments([ ]) )).
event( id( 34 ) , methodexit( location( 'DoubleDescent.java' , 1 ) , thread( 'main' ) , instance( 'Node' , 477 ) , name( '<init>' ) , 'void' )).
event( id( 35 ) , step( location( 'DoubleDescent.java' , 17 ) , thread( 'main' ) , [ ])).
event( id( 36 ) , setfield( location( 'DoubleDescent.java' , 17 ) , thread( 'main' ) , instance( 'Node' , 477 ) , name( 'value' ) , value( 400))).
event( id( 37 ) , step( location( 'DoubleDescent.java' , 18 ) , thread( 'main' ) , [ ])).
event( id( 38 ) , setfield( location( 'DoubleDescent.java' , 18 ) , thread( 'main' ) , instance( 'Node' , 477 ) , name( 'left' ) , value( instance( 'Leaf' , 476 )))).
event( id( 39 ) , step( location( 'DoubleDescent.java' , 19 ) , thread( 'main' ) , [ ])).
event( id( 40 ) , setfield( location( 'DoubleDescent.java' , 19 ) , thread( 'main' ) , instance( 'Node' , 477 ) , name( 'right' ) , value( instance( 'Leaf' , 476 )))).
event( id( 41 ) , step( location( 'DoubleDescent.java' , 20 ) , thread( 'main' ) , [ ])).
event( id( 42 ) , methodexit( location( 'DoubleDescent.java' , 20 ) , thread( 'main' ) , instance( 'Node' , 477 ) , name( '<init>' ) , 'void' )).
event( id( 43 ) , step( location( 'DoubleDescent.java' , 31 ) , thread( 'main' ) , [ ])).
event( id( 44 ) , setfield( location( 'DoubleDescent.java' , 31 ) , thread( 'main' ) , instance( 'Node' , 474 ) , name( 'left' ) , value( instance( 'Node' , 477 )))).
event( id( 45 ) , step( location( 'DoubleDescent.java' , 34 ) , thread( 'main' ) , [ ])).
event( id( 46 ) , step( location( 'DoubleDescent.java' , 37 ) , thread( 'main' ) , [ ])).
event( id( 47 ) , methodexit( location( 'DoubleDescent.java' , 37 ) , thread( 'main' ) , instance( 'Node' , 474 ) , name( 'insert' ) , 'void' )).
event( id( 48 ) , step( location( 'DoubleDescent.java' , 47 ) , thread( 'main' ) , [ localvariable( name('tree') , value( instance( 'Node' , 474 )))  ])).
event( id( 49 ) , step( location( 'DoubleDescent.java' , 49 ) , thread( 'main' ) , [ localvariable( name('tree') , value( instance( 'Node' , 474 )))  ])).
event( id( 50 ) , datastructure( instance( 'java.lang.String[]' , 459 ) , contents([ ]))).
event( id( 51 ) , methodexit( location( 'DoubleDescent.java' , 49 ) , thread( 'main' ) , classname( 'DoubleDescent') , name( 'main' ) , 'void' )).
event( id( 52 ) , threaddeath( thread( 'main' ) , threadgroup( 'main' ))).
event( id( 53 ) , threadstart( thread( 'DestroyJavaVM' ) , threadgroup( 'main' ))).
event( id( 54 ) , threaddeath( thread( 'DestroyJavaVM' ) , threadgroup( 'main' ))).
```

Figure 18: JEL Example for the Double Descent program.
The binary tree has two nodes which stores 500 and 400.

	Query	Purpose	Domain	Range
Q1	Where a member field is set to a new value?	To know the environment where the event occurred	The Entire History	Class and a static method call or an object and instance method call
Q2	Where a member field is accessed?			
Q3	Where a member method is called?			
Q4	Where an object is instantiated?			
Q5	Where an exception is thrown or caught?			
Q6	Object state	To know member fields values	History between 2 event ids specified by the user or between the id of the object instantiation event and id specified by the user	Set of triplets of Field value , filed name and event id when the field was set to a new value
Q7	Contents of data structure.	To know the values stored in an array or a collection instance	Specific event	List of pairs. Each pair has an index and value.
Q8	Method arguments	To know the arguments of a given method call		List of values
Q9	Value returned from a method call	To know the returned value from a method call		Void or a value
Q10	Local variable value prior to the execution of an event	To know the value of a local variable just before the execution of an event	Execution history between a specific event and the event of the enclosing method call	Local variable name and its value
Q11	The pre-called methods	To group methods that were called before the execution of an event within the same environment		A set of pairs. Each pair has a method call event and a method exit event
Q12	Local variable value after the execution of an event	To know the value of a local variable just after the execution of an event	Execution history between a specific event and the event when the enclosing method returns	Local variable name and its value
Q13	The post-called methods	To group methods that were called after the execution of an event within the same environment		A set of pairs. Each pair has a method call event and a method exit event
Q14	Local variable value history	To understand loop execution	Execution history between the event of a enclosing method call and event when the enclosing method exists.	A set of values
Q15	Compare	To know the difference between multiple similar query answers such data structure contents, call chain, and field history	Multiple similar query answers	Set of the elements that differ
Q16	Method call chain	To group method calls according to call chain	History between 2 event ids specified by the user	A set of method call events
Q17	Methods in a call tree	To group method calls according to a call tree		A set of pairs. Each pair has a method call event and a method exit event
Q18	Member field value history	To know values assigned to a member filed		Set of pairs. Each pair has the field value and event id where the field was set to that value
Q19	History of contents of data structure.	To know the contents of data structure at different points		A set of sets of pairs. Each pair has index and value
Q20	History of method arguments	To group calls to the same method. Useful in understanding recursive methods.		Set of pairs. Each pair has the event id and a list of arguments
Q21	History of method return value			Set of pairs. Each pair has the event id and a value
Q22	Class instances and their states.	To investigate the state of a group of instances of the same class. Useful in inspecting a user defined data structure		Set of object state. Each object state is a set of triplets of member field name, value and event id where the assignment occurred
Q23	Running Treads	To know the running threads		Set of pairs. Each pair has thread name and thread group
Q24	Exited Treads	To know the exited threads		
Q25	Suspended Threads	To know the suspended threads		
Q26	Was a given conditional statement executed?	To know whether such event occurred or not		event Id where it occurred or No other wise
Q27	Was a given method called?			
Q28	Was a member field assigned?			
Q29	Is there an instance of a specific class?			
Q30	Was a specific exception caught?			
Q31	Is a given thread still running?			
Q32	Has a given thread exited?			

Table 2