Design and implementation of a web based PL/SQL debugger using Oracle's debug API

A L V A R O   M A Y O R G A

Master of Science Thesis
Stockholm, Sweden 2005

IMIT/LECS-2005-42

Design and implementation of a web based PL/SQL debugger using Oracle's debug API

-

Alvaro Mayorga
alvaro@kth.se

May 2005

Industry advisor at Corus Technologies: Johan Palm.

Examiner and academic supervisor at KTH: Vladimir Vlassov

Department of Microelectronics and Information Technology

Royal Institute of Technology, Stockholm

Abstract

Corus Technologies is a company that has designed and developed Corus QL (Corus QuickLink), a powerful and flexible integration tool aimed for system integration and application integration. This integration tool generates PL/SQL and Java code based on a general data model. Additions and modifications to the generated code sometimes need to be done; therefore it is important to be able to debug these programs within the framework of Corus QL.

The objective of this Master Thesis is to investigate whether the implementation of a PL/SQL debugger with a front end run on a web browser and a tailor-made PL/SQL package based on Oracle's DBMS_DEBUG debug API is realizable or not. A market survey of PL/SQL debuggers and a study of DBMS_DEBUG were carried out and their results noted. A package has been implemented on top of DBMS_DEBUG and it has the requirement that this PL/SQL package has to be able to be used as a command-line debugger, directly from a SQL client. In addition, the implemented package must be able to be used as a debug API for Java applications.

The difficulties encountered and their alternative solutions and decisions that were made during the execution of this project are described and discussed in this report. Decisions taken regarding system architecture, application and control models, GUI design, strategies and technologies utilized to carry out this project are presented as well as a number of conclusions obtained and guidelines for future work.

Keywords: PL/SQL debugger, Corus Technologies, web based, servlet application, CORUS_DEBUG, iCODE.

Preface

This Master's Thesis is a result of my Degree Project performed at Corus Technologies in Stockholm, Sweden between June and November 2000. It was reviewed on spring 2005 introducing some changes and improvements. It also concludes my Master of Science in Electrical Engineering at the Royal Institute of Technology (KTH), in Stockholm.

Corus Technologies is a company that has designed and developed Corus QL (Corus QuickLink), a powerful and flexible integration tool aimed for system integration and application integration.

Using a developing tool, which is part of the Corus integration system, PL/SQL and Java code is generated based on a general data model. Some additions and modifications to this code need sometimes to be done. For this purpose it is of great importance to be able to debug this generated code within the framework of the Corus system integration.

The conclusions and results obtained in this Master Thesis before the revision constitute the base of a PL/SQL debugger developed and integrated in Corus QL in year 2001.

Acknowledgments

I want to thank Johan Palm at Corus Technologies, for the initial input and guidance to start with the execution of this project. I also want to express my gratitude to Vladimir Vlassov, associate professor at IMIT for his guidance and motivation to carry out this project to its end despite that it sometimes appeared to be too overwhelming.

Table of contents

Table of figures

1 Introduction

This report is the result of a Master's thesis that presents a Degree project for Master of Science in Electrical Engineering at the Royal Institute of Technology, Stockholm, at the Department of Microelectronics and Information Technology. The degree project was performed at Corus Technologies' Developing Department in Stockholm, Sweden

1.1 Company profile and motivation

Corus Technologies is a company working with the development and sale of Corus QuickLink[1] (QL). Corus QL is an integration tool aimed for system integration and application integration respectively known nowadays as Inter Enterprise Integration (IEI) and Enterprise Application Integration (EAI). Corus QL has been designed and implemented to make information exchange possible between any kind and any number of computer systems over a network.

Corus Technologies has developed a tool for system integration that describes a general data model for the systems to be integrated and generates all the necessary code to perform the integration. The objects generated are database tables, triggers, views, indexes and stored procedures and functions building PL/SQL packages. Eventually some parts of these packages need to be added or modified. Therefore, it is of great importance that the code generated can be debugged within the framework of the application. There are several reasons for why third-party debuggers are not always suitable. For instance, they require client installation, do not support the needs of Corus completely, increase the cost for the end customers and some of them, at the date of start of this project, lacked the functionality of getting and setting global package variables, which is of great importance for Corus.

Consequently, the benefits of a web based debugger built in the environment of Corus QL are:

- Client installation is not necessary.
- Reduced costs for the Corus' Technologies customers, which in its turn may lead to more customers.
- The generated code can be debugged within the same environment it was generated.
- Access and manipulation of package variables which are important when systems are integrated with Corus QL.
- A built in debugger in the same environment can easily be improved or modified.

1.2 Specification of the project

A market survey of the existing Oracle's PL/SQL debugger tools on the market will be carried out. Then the Oracle's DBMS_DEBUG package will be studied in detail in order to decide the functionality to be implemented in the final application. As a part goal a callable Oracle package should be created on top of DBMS_DEBUG package. Furthermore, a web-based application able to be run and be used within Corus QL's

[1] Corus QuickLink is the former Corus Application Linking System (ALS).

framework should be designed and implemented utilizing Java servlets Technology and possibly JavaScript scripts. Finally, the web based application will be integrated into Corus QL's framework.

The package to be implemented on top of DBMS_DEBUG is a key part for the entire project. One of the objectives of this package is to abstract and hide the details that are not necessary for the end user. The user will be liberated of the task of declaring and initializing variables of the data types defined in DBMS_DEBUG before being able to call the procedures and functions of DBMS_DEBUG. Example of these variables and record fields in DBMS_DEBUG are the values for namespaces, lib unit types, frame numbers, current user name, break flags, info requested, program name, etc. One other example is reading, traversing and displaying PL/SQL tables containing, for instance, breakpoints information. Another objective is that this package will provide such transparency from DBMS_DEBUG and be implemented in such way that the users shall be able to use the package as a command-line debugger.

The integration of the web based application and the PL/SQL package might implicate the creation, declaration and use of specific user-defined data types and tables in order to be integrated into the core of Corus QL. Each integrated system created with Corus QL is built by generated PL/SQL and Java code; this implies that the web based application and the package will also be generated and perhaps with some adaptations for each client. The web based debugger will be added as a new tool or a new feature in one of the existing tools in Corus QL. Both the web based application and the PL/SQL package will need a number of adjustments and changes in order to cooperate without interfering with the rest of the integrated system generated by Corus QL.

1.3 Goal (The precise goal of this project)

Within the frame of this project the following points need to be fulfilled or completed in a satisfactory manner for Corus Technologies, as well as fulfill the requirements for a Master thesis project from the Department of Microelectronics and Information Technology in Stockholm:

1. A market survey should be done to examine the functionality of third-party Oracle PL/SQL debuggers on the market. Principally Quest Software's TOAD debugger tool and Compuware's Xpediter/SQL Debugger tool will be studied and tested.

2. A detailed study of Oracle's DBMS_DEBUG package should be done and decisions regarding the functionality to be implemented taken.

3. A Java servlet should be created to implement the API created in the preceding part goal.

4. A complete web application using the servlet/servlets made in the previous part goal and this application should provide a state-of-the-art GUI.

5. Finally, both the PL/SQL package created and the web application should be integrated into Corus' framework. This will be done depending on the remaining amount of time after the previous part goals are accomplished.

6. Furthermore, a report regarding the decided architecture and implementation should be written for internal use at Corus Technol§ogies.

These points represent part goals of the project. There is though a central goal and that can be formally formulated as follows:

The main goal of this project is to verify that the entire chain is realizable, from the database using a tailor-made debug API on top of DBMS_DEBUG, via servlets and eventually JavaScript scripts, to the graphical representation.

1.4 Practical information about the project

This project was performed at Corus Technologies, Stockholm, Sweden, between June 2000 and December 2000. In spring 2005, the implemented package was reviewed with some changes in the implementation resulting in what is referred to as *review version*. The web based application was also reviewed and updated according to the reviewed PL/SQL package. This project was though first presented at KTH in May 2005 due to different reasons as the complexity and size of the project, lack of technical information available about the API provided by Oracle and due to working commitments. The results of the first version of this project and the first draft of this report were the base of a new version of the PL/SQL package that has been integrated in Corus QL in 2001.

The persons involved with this project were:
My advisor at Corus Technologies, Johan Palm who has been working with the development of Corus QL since the foundations of the company in 1997,
The examiner of this thesis project and my advisor at the Department of Microelectronics and Information Technology in KTH was acting associate professor Vladimir Vlassov.

1.5 Structure of the thesis

Chapter 2. Gives a brief description of the theoretical background necessary for the completion of this project. A debugger's construction in general, SQL, Oracle's PL/SQL, Oracle's DBMS_DEBUG package, database and Java applications models and Java servlets technology are addressed in this chapter.

Chapter 3. The design, strategic decisions, models, methods and technologies utilized in the realization of the project, the development of the callable package API and the final web application are explained and discussed in this chapter.

Chapter 4. Presents an evaluation of the project, the use cases and evaluation model utilized and discusses the obtained evaluation results.

Chapter 5. Discusses the result of the project in general, the functionality and capability of the CORUS_DEBUG API, the reasons of some decisions made in the project and of some limitations of the web based debugger. It also compares the functionality between the web application and the studied commercial debuggers.

Chapter 6. Summaries and concludes the project and makes an evaluation between the initial goals and results obtained.

Chapter 7. Addresses some ideas for future work.

2 Background

This Chapter gives a brief introduction of what a debugger is, the different kinds of debuggers and how they work. It also provides a short overview of the main features and components of a debugger, how a debugger usually is constructed, relational databases and more specifically Oracle's Database, SQL and Oracle's PL/SQL, some of Java-programming technologies and techniques used in this project. This theoretical background is necessary to understand and assimilate how the final application and the callable package have been implemented, as well as why some decisions and strategies have been taken.

2.1 Related work

In this Chapter the theoretical material necessary to know and understand this project will be exposed. Some literature and link references are given whenever necessary.

2.1.1 Debugging principles, features and techniques

A debugger definition, different types of debuggers, why and when one needs a debugger, how debuggers work, the required components for building a remote debugger and the main and most common debugging facilities are assessed in this chapter.

2.1.1.1 What is a debugger and what is a debuggee?

A fundamental concept and a natural start point in the context of developing a debugger is the basic definition of a debugger. Even though there is no official definition of a debugger (in the meaning valid for this report), summarizing different definitions from [11], [21], [26], [35] [36] and other sources and debug experience, an own definition is presented:

> A debugger is a programming tool, a program itself, developed for the purpose of helping the programmer to locate run-time programming or logic errors, also known as 'bugs'.

When using a debugger the flow of the program execution can be followed and the program execution stopped at certain points (strategically chosen) in order to inspect the state of the program and of its components, which constitute the program context information. Executing and stopping a program, and inspecting the state of the program allow the programmer to see whether the program is behaving as expected or not. When malfunction of a program is noticed, the features and functions of a debugger provide with help to deduce or discover the reason and location of the program's malfunction.

When using a debugging tool there are indeed two programs that will be executed, one of them is the debugger itself, usually just addressed as the *debugger*; and the other one is the target program to be debugged, usually called the *debuggee*.

2.1.1.2 Why and when are debuggers needed?

The motivation of why a debugger is needed for Corus QL is presented in Chapter 1.1, but not why debuggers in general are needed, this is the subject of this Chapter.

In the programming process, a number of assumptions are made. Often it happens that not all of the logical assumptions can be made at the modeling or at the specification phase in a software development project, and not all of the made assumptions are always correct [14]. These assumptions can for instance refer to data format, interactivity with a user or other programs, communication mechanism with other programs, etc. Therefore, programmers spend a lot of time debugging their programs, analyzing and tracking the cause of why a program or part of a program doesn't work as intended and expected.
The use of a debugger (a debugging tool) is not always necessary to debug, but nevertheless is much more efficient. One alternative with limited efficiency is for example, to have messages printed or logged that show the state of the program. Soon, this debugging method is not useful, for example, when the program is a multithreaded or concurrent program. Another example is when simply the program is too large or complicated as explained in [11], or when the program crashes without output. Therefore, both standalone and integrated debuggers (presented in Chapter 2.1.1.4) are very popular, necessary and used for experienced programmers working with very complex software development.

2.1.1.3 Methods for debugging programs

Methods of debugging neither refer to specific manners, nor to some specific order in which to execute some debugging functions when using a debugger; these are rather called debugging *strategies*. These debugging strategies vary depending on the type of debugger used and the nature of the program being debugged.

What methods of debugging specify in general are the methods, techniques or tools (a debugger for example) used in order to trace and locate the cause of why a program is not working as expected or as intended. Besides using a debugger that is called *interactive source-level debugging*, there are other techniques used to debug as e.g. using print statements, log files, trace files, having programs that make functions call on termination or that just log the state of the program.

2.1.1.4 Types of debuggers.

Rosenberg claims in [11], pp.11-12 that *"Debugging is a very general activity, but each specific application and bug require a special use of a debugger to locate and eliminate the fault quickly and decisively"*. This claim of Rosenberg reminds us of this project's background which is the need of Corus QL to develop a built-in debugger for PL/SQL within its framework. What type of debugger the one in this project may be and other types of debugger are listed below.

There are many types of debuggers, but they can be grouped in:

- Source-level (symbolic). Maps underlying machine representation back to source code.
- Machine-level debuggers. Lack mapping between machine code and source code.

- Stand-alone debuggers. A program exclusively dedicated to debugging.
- Debuggers integrated in development environments.
- OS Kernel debuggers. Focused on operating systems components.
- Application-level debuggers. Focused on user-written applications.
- Application-specific debuggers. General purpose and high level debuggers.
- In-circuit emulation debuggers. Specifically for in-circuit emulators, see [23].
- Local debuggers. Both debugger and debuggee processes are local.
- Remote debuggers. The debuggee is remotely located.
- Mixed debuggers. Where interpreted and compiled code can be handled.
- Debuggers for single-threaded programs.
- Debuggers for multi-threaded programs.
- Multi-language debuggers. When more than one programming language can be debugged with the same debugger.

A brief description of most of the types of debuggers listed above is found in [11] pp.12-19.

The complete name of the type of debugger of this project is a *locally and remotely run interactive single-threaded source-level application GUI debugger*. But usually it is just referred to as a *source-level symbolic GUI debugger.*

2.1.1.5 How debuggers work

When studying debuggers in general a question is naturally raised, *how do debuggers work?* This is a very tricky and difficult question to answer because on one hand there are different types of debuggers and on the other hand the question doesn't specify what level of the debugger it refers to. In addition, there is the fact that even debuggers and their algorithms, functionality and complexity develop in time, as well as programming languages and the programs to be debugged do [11], [14].

First of all we should remind ourselves of the definition presented earlier in Chapter 2.1.1.1 that a debugger is a program itself. As other programming tools a debugger can be integrated in a complex graphical environment, but this is not the issue at this level. We will study a debugger at the lower levels as system signals, system and functions or procedure calls, the memory stack, etc.

When debugging a program (or multiple programs) it is usually necessary to have two (at least) sessions (or processes) in the runtime environment, whether it is a binary file to be executed or byte-code to be interpreted. One of these sessions, the target program to be studied or debugged, is the *debuggee*. The other session, the one that controls the debuggee session, is the *debugger session.* To make possible controlling the debuggee there are two mechanisms used. One mechanism is attaching the debugger process to the process to be debugged. The other mechanism is letting the debugger process start a new process, a child process to run the program to be debugged. The attaching mechanism is the most popular [11].

Since debugging is the act of using a programming tool in some predetermined or strategic manner, the most suitable way to explain the functionality and construction of a debugger is by going through a number of steps during a debugging session and explaining the different steps in it. A brief step by step explanation of a typical debugging session follows here:

1. First of all, in order to make a program *debuggable* it must be *debug compiled* (compiled in debug mode or with a debug option flag), usually done by compiling the source code with a debug option. Debug compiling implicates that the compiler inserts special instructions in the generated machine code that will make it possible for the OS to provide the debugger with execution control over the debuggee and to see the context information about the debuggee and the CPU.

2. Once the target program is *debug compiled* it can be run. There are two modalities to do that, one is when the debugger attaches itself a running process (that will be the debuggee) and the other is when the debugger starts a child process to run the target program for debugging.

3. Usually the next step is to browse the source code of the debuggee; therefore, it is very useful if the debugger has a code browsing utility. The code browsing is to choose some strategic line or lines of code where the user wants the debuggee to stop in order to examine the state of the program (meaning the value of its variable and objects, memory allocated, etc) and/or the CPU and its registers.

4. Setting a (source-level or instruction-level) breakpoint implies that the debugger inserts a special instruction in the executable image text of the debuggee that causes it to halt when this instruction is executed. This instruction is usually a general interrupt instruction or a special dedicated breakpoint instruction. This and other instructions depend on the definition of the CPU architecture.
   From the point of view of the user, there are two kinds of breakpoints *conditional* and *unconditional*. The conditional breakpoints will stop the execution of the debuggee when a condition is satisfied (for instance when a specific exception is raised) and the unconditional ones will stop at the specific line of code each time it is next to be executed.

5. Once breakpoints has been set on different lines in the source code, besides deleting them, debuggers usually also provide the possibility of modifying them in different ways (enabling, disabling, changing conditions, etc.). For more details see [11], pp.107-133.

6. Running the program until the breakpoint previously set is usually the next measure taken. It can be either executing the debuggee line by line or using some function that runs the program until either an arbitrary or a specific breakpoint is hit.

The execution of the debuggee stops and the programmer has some alternatives for the next step.

7. One alternative, is to examine the back trace stack in order to see what functions, programs, or procedures have been called an in which order. The back trace stack is constructed by examining one or more hardware registers, which gives the debugger enough data for this task. These CPU's registers provide a stack pointer that points to a location in memory for the current executing instruction that becomes the return address when a new routine is called. For more details see [11], pp.135-149.

8. Another alternative is to examine the current value of certain variable or objects in the program. This implies that the debugger access the variables, objects and even functions belonging to the debuggee in registers generated by the compiler, which are known as *symbol tables*. These symbol tables are not the same as the one used by the compiler. Another intricate issue that is handled by a debugger is *scope resolution*, which is even more complicated for distributed, multi-threaded and remote debugging. For more details see [11], pp.151-172.

9. One following action the programmer can take is to change the contents of a data item. For this purpose a debugger must have an interpreter that handle expression with a similar or equal syntax to the language of the debuggee. The main difference in this interpreter is that it does not allocate an own storage for variables, but use the variables of the debuggee.

10. Another strategy or functionality is the *watchpoints*. Some variable can be set as watchpoints in order to make the debugger automatically read the values of these variables. It is necessary for this functionality that the debugger has a built-in polling function that reads the value for the watchpoints variables at least each time the debuggee stops. Important here is that the debugger does not 'lie' to the user showing bugs (or changes in variable values that appear to be bugs) where they don't exist. This is a risk that is common in debuggers [11] p. 157. When the user thinks he has discovered a bug, it is in fact due to compiler-debug information, or other issues out of the control of the developer of the debugger.

Hopefully the problem or *bug* has been found by the programmer, otherwise new breakpoints and/or watchpoints can be set, more variable values, and stack information can be read and possibly the debuggee executed some more times.

11. Furthermore, in the case of remote debugging there are other steps when setting up the communication, the communication mechanisms used between the hosts involved in the debugging session, how the different memory addressing and the remote memory stacks has to be handled by the debugger. But sometimes this is taken care of by other applications at lower level.

The scope of this thesis neither permits nor needs to go deeper into the details of the construction of a debugger at a lower level. The API provided by Oracle abstracts most of the functionality required and presented in the steps above. The interested of deepening in the construction of debuggers are strongly suggested to consult [11] and [14].

2.1.1.6 Basic principles of debugger design and development

- An interpretation of *The Heisenberg uncertainty principle* (found at [18]) within the context of debugging (and testing) is that a *debugger or other debugging tools should neither change nor affect the behavior or outcome of the debuggee*. This principle is though violated in a lot of ways in software debugging, as for instance, by the fact that the debugger and debuggee share memory and are controlled by the same operating system [11] p.7. Both programs affect each other because of changed conditions in execution scheduling, context switch, delay times, etc. Another aspect pointed by Telles in [14] is that compiling in debug mode may introduce some behavior in the program that doesn't appear in the release version of the program.

- *Truthful information must always be provided during debugging.* This is a principle originally stated by Zellweger in 1984 in [17] and discussed by Rosenberg in [11]. This principle is to always provide truthful information in a debugging process. Any misinformation will mislead the user when testing theories of the location or cause of bugs and the effect can be devastating. This principle is often violated by optimizing compilers and even by modern compilers that "always do a certain amount of register allocation optimization", as discussed in [HDW, p. 9].

- *Program context information*: constitutes the most important information that a programmer can retrieve during a debugging process. It can include different types of information as source code, stack back-trace, variable values, thread information, memory and CPU registers, counters, and more. Locating the bug and how it manifests is essential to remove the bugs in a program. That is why it is important to have a debugger that provides good and reliable program context information, since bugs may manifest in other places than where they are located. Seeing the source code that maps to where the program is stopped at each moment is fundamental to start with bug location, followed by the stack back trace and accessing variable values. Without all of these collaborating features it is hard to locate the bug, and how it is caused.

- Another principle presented by Rosenberg in [11] states that "*System developments occur long before any corresponding strong debugging support for the new systems developments is available"*. He discusses the importance of the applications developers to push not only for technologies they need, but also for debuggers they will need to debug their ever increasingly complicated applications.

2.1.1.7 Principles for building remote debuggers

A remote debugger is a debugger that is not run in the same physical place as the target system. The main reason for constructing remote debuggers is to debug software on a system that for some reasons cannot locally run a functional debugger. Though, this is not the case in this project, instead it is the need to debug software located at a physical remote place. In this case the strategy of remote logging can be used.

The necessary components for a remote debugger are a client, a server, a communication mechanism and a debugger protocol. The client for this project (where the only programming languages that can be used are Java, PL/SQL and JavaScript, and HTML as markup language) could be a Java application, a Java applet or a dynamic HTML page run in a web browser. The restriction of the Corus QL's framework reduces the client to be an HTML page or a set of HTML pages, probably reinforced with JavaScript scripts.

What could be seen as the server side of the application will then properly be a Java servlet or a set of collaborating servlets and classes stored in a web server. But this is only the server side of the web based part of the application. On the bottom of the application lais a tailor-made debug API specially designed and constructed to communicate with Java applications. This debug API is the application component that is the real server side of the application.

The communication protocol in this project will probably be HTTP or HTTPS and depends on the communication mechanism that probably will be a TCP connection over the Internet. The debugger protocol, if any, will be decided during the implementation phase.

2.1.1.8 Functions provided by a debugger

Tracking and controlling the execution of a program, reading and manipulating the target system memory, e.g. variables and constants, constitute the main and basic functions a debugger must offer. Some standard functions are described below.

- **Process attaching or process creation**. In order to debug a program the debugger must be able to attach to a running process or to create an own child process in which the target program can be run.

- **Run or execute**. Perhaps the fundamental function to build in a debugger. Permit the ability of starting and executing a target program from the debugger.

- **Source code browsing**. In order to find out *where* the bug(s) is/are located it is of extreme importance that the user is provided with a source browsing functionality. With this function the user can see the mapping between the source and where the debuggee has been stopped.

- **Step or next**. Among all the facilities and functions a debugger can offer this is the basic one. Step or next allows the user to advance in the execution of the target program (debuggee) line by line, command instruction by command instruction, and sometimes even at a such low-level as by machine instruction.

- **Step- or trace-over, -out, -in**. These functions are more refined and/or advanced variations of single step. They permit the user to jump to and over break points, to execute all the way out of the scope of a procedure, method, or function and finally to step within the source code of a program all the way until the main function or the beginning of the program.

- **Get or read**. This function allows reading the contents of the memory at different points of time during the program execution. This information is invaluable, because it helps the user to locate *where* the bug is located.

- **Set, write or assign**. This function allows the user to arbitrarily assign strategic values in the target system memory, for example changing the value of a variable. This ability of assigning different values during the program execution allows the user to study the different behaviours and results that a program may have depending on the changes made on the variables.

- **Set and Remove a break point.** This function allows the user to set a mark, usually known as *breakpoint*, at a specific line of code or at a specific program instruction where the execution of the program will be stopped. This is one of the most used functions as the programs written are large and not only composed of a couple of lines, but usually of many lines of code and often distributed over different modules or packages located in different physical files.

- **Enable and disable break points.** Sometimes it is more adequate disabling a set break point than removing it, and enabling a breakpoint instead of setting a new one. This depends on the type of debugger and how it is implemented.

- **Watch point or data break point:** A watch point is used to stop execution whenever the value of a variable or an expression changes, without having to predict a particular place in your program where this may happen. This is an advanced function harder to implement.

- **Stack trace or back trace.** Because during the execution of a program the execution cursor jumps back and forth between different methods, procedures, functions and packages or modules, it is of big help to the user to know in which part of the program the execution cursor is, and which is the current valid scope. *Execution cursor* is a logical point that shows exactly where the program executed is.

2.1.1.9 Architecture and implementation of a debugger

A typical debugger implementation has an underlying architecture similar to the one illustrated in Figure 2.1, which is a modification of the one presented by Rosenberg in [11], p22. In Figure 2.1 the three inner-most rings represent the CPU, the operating system running on the CPU and a debug API that permits access and control at the OS level. The second outer-most ring represents the core of the debugger, depicting the use of the functionality provided by the OS debug API. The outer-most ring represents graphical representations of the debugger.

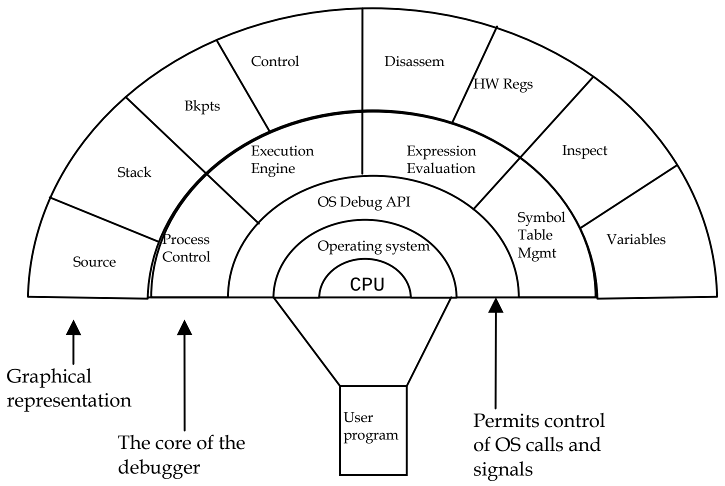


Figure 2.1. Typical debugger architecture. Based on Figure 2.1 from [11], p.22.

Many of the fields in Figure 2.1 are not addressed in this project, either because of the types of debugger that will be implemented, or the type of programs that will be debugged. A more detailed presentation of debugger architecture, functionality, and implementations details, as data types used at lower levels and algorithms, are well described in [11].

2.1.2 Relational Databases, SQL and Oracle's PL/SQL

This chapter briefly presents fundamental theory for the understanding of the project and its realization.

2.1.2.1 Relational databases.

Although the history of the origin and developing of relational databases is quite exciting and interesting it is left out of this report taking in account that it is in somehow out of the scope and lack relevance for the core of the project. A definition is presented instead, and since there is no official definition but different definitions from many books and article writers, as in [3], p 98, one own definition had to be done.

A *relational database* is a collection of stored operational data, perceived by its users as in the form of tables and often related to each other by some key fields, and that allows to its users to update, insert, and retrieve the data stored in these tables.

Nowadays, the majority of databases are relational databases and almost all of them use a de facto standard relational database query language, namely SQL.

2.1.2.2 SQL

SQL that stands for *Structured Query Language*, is an ANSI and ISO standard, and is the de facto standard database query language. It has many dialects but the current standard is *SQL-92* and there are two almost identical versions of it: ANSI X3.135-1992, "Database Language SQL" and ISO/IEC 9075:1992, "Information Technology --- Database Languages --- SQL" [13].

Although SQL is not a general purpose programming language it provides the necessary tools to create and maintain database objects and to provide security restrictions in a relational database. Three different parts of SQL take care of these functionalities:

- Data Definition Language (DDL) – to create all the objects defined in a database, to modify its structure after its creation and to destroy objects that no longer are needed.
  Ex:
  ```
  CREATE TABLE EMPLOYEES (
                EMP_ID           INTEGER             NOT NULL,
                EMP_FST_NAME  CHARACTER (15),
                EMP_LST_NAME  CHARACTER (15),
                EMP_DEPT         CHARACTER (15),
                EMP_SALARY      INTEGER );
  ```
- Data Manipulation Language (DML) – provides the user with the necessary tools to insert, change and retrieve data from the database, just as exactly as he wants.
  Ex:
  ```
  UPDATE EMPLOYEES
    SET SALARY = SALARY * 1.04,
          WHERE EMP_FST_NAME = 'John'
          AND     EMP_LST_NAME = 'Doe';
  ```

- Data Control Language (DCL) – provides with the necessary tools that, if properly used, will prevent many of the typical security problems in a system.
  Ex:
  ```
  GRANT SELECT, INSERT, UPDATE, DELETE
    ON EMPLOYEES
    TO PERSONNEL_MANAGER;
  ```

Some people state that with SQL's DDL, DML and DCL a developer can build robust applications of any required size, complexity and kind [4]. This is not entirely true because since SQL is a non-procedural programming language, there are some tasks, which are not possible to do just by using SQL [13]. For example: if a table needs to be traversed and to execute an insert or an update for each row depending on some condition, then at least two SQL statements are needed, but it can be done with just a call to a PL/SQL program unit. Another example is the execution of more than just two SQL statements with conditional parameters and where the parameters are probably obtained by executing other PL/SQL programs; in this case the need of a procedural language is more evident. Therefore SQL needs to be supplemented by a procedural programming language. Here is where Oracle's PL/SQL makes its entrance.

2.1.2.3 Oracle's PL/SQL

Oracle's relational database was in fact the first one in the world to support SQL, and is now the world's leading supplier of software for information management. Oracle also developed PL/SQL that stands for Procedural Language/SQL.

> *PL/SQL*, that stands for *Procedural Language/SQL*, is a procedural programming language that combines the power and flexibility of SQL with the constructs found in procedural languages as variables, types, control structures, procedures and functions [13].

Oracle supports the ANSI SQL standard "Database Language SQL", most known as SQL92 or SQL2, defined in the document X3.135-1995. Oracle's PL/SQL has become a sophisticated language with many features and with, by Oracle, added functionality in the so called *built-in packages*. These built-in packages increase the functionality of an already very robust, flexible and powerful programming language even more.

2.1.2.4 Features of Oracle's PL/SQL

PL/SQL has many different features and capabilities. They would be illustrated by example but since this report is not supposed to be an introduction to PL/SQL we will limit us to just a couple of examples for the most relevant features that are good to know for this project. From this point SQL will refer to Oracle's PL/SQL.

Block Structure. The basic unit in PL/SQL is a *block*. It begins with the keywords BEGIN and END. PL/SQL programs are built up by blocks, which can be nested within other blocks. A block in PL/SQL has three sections a declarative (started with DECLARE), an executable (started with BEGIN) and an exception-handling section (started with EXCEPTION). Only the executable section is required.

Error-handling. The runtime-errors encountered in a program are captured and handled in the exception-handling section of the block.

Variables and types. *Variables,* which are how information is transmitted between PL/SQL and the database, are a storage location that can be read from and assigned from the program. Each variable is associated with a specific predefined or user-defined *type* that tells what kind of information this variable can hold. An important and powerful capability provided by PL/SQL, is that – besides types as the database columns (ex: VARCHAR2(25)) and additional types (ex: BOOLEAN) – it also supports user-defined types such as tables and records.

index-by tables or PL/SQL tables. A type of special interest and that can't be found in SQL is *pl/sql tables*, or also known as *index-by tables.* They are one-dimensioned, unbounded sparse collections of homogeneous elements or records.

Looping constructs. A *loop* permits us to execute a sequence of statements repeatedly until some condition is satisfied. PL/SQL provides us with different kinds of loops: *while*-loop, the numeric for-loop, *repeat-until emulated* loop and the *cursor* loop.

Conditional and sequential control. As expected, traditional constructs for conditional control as *if ... then ...* or *if ...then...elsif ... then... else...* are provided. For sequential control there is the *goto* statement that permits to jump to a labeled part of the code that must in the same scope as the goto statement. However, the *case* statement is unfortunately not provided.

The features mentioned above are basic ones, and are what every programmer surely expects to find in a procedural language. But there are in fact advanced features that give the power, robustness and flexibility that characterize PL/SQL and some of them will be discussed now. All of them are presented and exemplified more extensively in [4], [5] and [13].

Procedures and functions. *Procedures and functions* that are together known and referred to as *subprograms* or *stored procedures* are a special kind of block. They can be stored in the database in compiled form and are callable from other programs or stored procedures.

Packages. Stored procedures, together with variables and types can be grouped and stored into a *package.* A package is indeed composed of two parts; one part is the package specification (a.k.a. package header) and the other part is the package body. Packages allow storing related objects together in the database.

Dynamic SQL. Traditionally a programmer needs to know in advance exactly the type, name, number and order of the columns from the tables needed to access in the program. Through *dynamic SQL,* it is now possible to build and execute a SQL statement at runtime. The two dynamic SQL methods existing are the *DBMS_SQL package* and *native dynamic SQL.*

Object types. Like in object-oriented languages, an *object type* has attributes and methods, and can be stored in the database. Object types in PL/SQL allow the access and manipulation of not only relational data but object data as well.

Collections. Are the answer or correspondent construct to arrays in other 3GL-languages. There are three kinds of collections: *index-by tables*, *nested tables* and so called *varrays*.

Built-in packages. Additional functionality is provided by Oracle by a number of *built-in packages.*

One of these built-in packages is of interest for this project, namely DBMS_DEBUG. A more detailed presentation of this package is given in the next Chapter.

2.1.2.5 DBMS_DEBUG: Oracles debug API package.

As a definition of what the DBMS_DEBUG package is let us just use the one given by Oracle in the official documentation for this package[2]: *"DBMS_DEBUG is a PL/SQL API to the PL/SQL debugger layer, 'Probe', in the Oracle server"*.

As mentioned before a package can bundle together variable, types, procedures and functions and DBMS_DEBUG has all of these components.

NOTE: Below are only the most relevant features of the DBMS_DEBUG package presented. For a closer or more extensive study see [2] or [29]. The names given in capitalized style in this Chapter denotes the name of procedures and functions in DBMS_DEBUG. Ex: PRINT_BACKTRACE.

The term of *program unit* comprises: procedures, functions, triggers, packages, package bodies and anonymous blocks.

Variables

The public variables are for timeout behavior and for tracing. These variables permit to set and read the size of the timeout and the possible behavior which are: *retry, continue, nodebug* and *abort* on timeout

Types

There are seven user-defined types in the package and their functions are on one hand to read the program context information and on the other hand to permit giving parameters to the programs. These types are described in Table 1.

Table 1. User defined types for DBMS_DEBUG.

Type	Description
backtrace_table	Used by PRINT_BACKTRACE to store the program stack.
breakpoint_info	Gives information about a breakpoint, its current status and where it is placed.
breakpoint_table	Used by SHOW_BREAKPOINTS to store information about the existing breakpoints in the current session.
index_table	PL/SQL table used by GET_INDEXES to return available indexes for an indexed table.
program_info	Specifies a program location by giving the information for a line in a program unit.
runtime_info	Gives the current context information about the running program.
vc2_table	Type meant to hold lines of code of the procedures to debug and used by SHOW_SOURCE.

Constants
A large number of constants are necessary in a package like DBMS_DEBUG the package the constants can be grouped by their function. The complete list of variables can be found in [2] or [29]. In Table 2 below, the function of the constants in DD are listed grouped by their function.

Table 2. Types of constants in DBMS_DEBUG grouped by their function.

Constant type	Description
Breakpoint	Describe the states of a breakpoint which are: *active, unused, disabled* or *remote.*
Namespaces	Constants for giving the type of the program units that on the server reside in different namespaces. When setting a breakpoint it is necessary to specify the desired namespace.
Libunit types	Useful when presenting information about stack backtrace to the user. Will replace namespaces in the future by overloading. They are important in PROGRAM_INFO to disambiguate among objects that share the same namespace.
Breakflags	Very important constants that are passed to CONTINUE to tell Probe what events are of interest. The Probe will then stop the execution when these events are raised. They break execution on: *next line, any call, any return, return, exception, break handler* and *execution*.
Information flags	These are flags that can be passed as parameter to CONTINUE, SYNCHRONIZE and GET_RUNTIME_INFO to tell Probe what information is of interest.
Reasons for suspension	After CONTINUE is executed, the program will either run to completion or break on some line or event, for example when it reaches: *a new line, a breakpoint, finish, kernel exit, etc.*
Error codes	These values are returned by the various functions that are called in the debug session.
Exceptions	Some exceptions are raised by SELF_CHECK and one if INITIALIZE is called before DEBUG_ON.

Procedures and functions composing the DBMS_DEBUG package
The DBMS_DEBUG package is finally composed of a number of procedures and functions. These procedures are to be executed either by the target or debug session or both of them. Table 3 presents the public procedures and functions in DD and (C), (T)

and (D) shows whether they are common to both sessions, for the target session and for the debug session respectively.

Table 3. Procedures and functions in DBMS_DEBUG. 'C'-Common, 'T'-Target and 'D'-Debug denote in which session they can be executed.

Procedure/function	**Description**
`procedure` **`PROBE_VERSION`**`(C)`	Returns the version number of DBMS_DEBUG on the server.
`procedure` **`SELF_CHECK`** `(C)`	Performs an internal consistency check.
`function` **`SET_TIMEOUT`**`(C)`	Sets the timeout value.
`function` **`INITIALIZE`**`(T)`	Sets debugID in target session.
`procedure` **`DEBUG_ON`** `(T)`	Turns debug-mode on.
`procedure` **`DEBUG_OFF`** `(T)`	Turns debug-mode off.
`procedure` **`ATTACH_SESSION`** `(D)`	Notifies the debug session about the target debugID.
`function` **`SYNCHRONIZE`** `(D)`	Waits for program to start running.
`procedure` **`SHOW_SOURCE`** `(D)`	Fetches program source.
`procedure` **`PRINT_BACKTRACE`** `(D)`	Prints a stack backtrace.
`function` **`CONTINUE`** `(D)`	Continues execution of the target program.
`function` **`SET_BREAKPOINT`** `(D)`	Sets a breakpoint in a program unit.
`function` **`DELETE_BREAKPOINT`** `(D)`	Deletes a breakpoint.
`function` **`DISABLE_BREAKPOINT`**`(D)`	Disables a breakpoint.
`function` **`ENABLE_BREAKPOINT`** `(D)`	Activates an existing breakpoint.
`procedure` **`SHOW_BREAKPOINTS`** `(D)`	Returns a listing of the current breakpoints.
`function` **`GET_VALUE`**`(D)`	Gets a value from the currently-running program.
`function` **`SET_VALUE`**`(D)`	Sets a value in the currently-running program.
`procedure` **`DETACH_SESSION`** `(D)`	Stops debugging the target program.
`function` **`GET_RUNTIME_INFO`** `(D)`	Returns information about the current program.
`function` **`GET_INDEXES`**`(D)`	Returns the set of indexes for an indexed table.
`procedure` **`EXECUTE`**`(D)`	Executes SQL or PL/SQL in the target session.

Program flow for the target session (*debuggee*)

In the market survey there were some debuggers that hung up while debugging, either already at synchronization or after start running the target program. It seems to be because DD follows a strict program flow that must be followed.

Not following the program flow of DD or doing some calls in wrong order (or without fulfilling some conditions) may cause the debugger or the debuggee to hang. Ex. calling **`DEBUG_ON`** before calling **`INITIALIZE`** makes the debuggee session to hang.

The Figure 2.2 describes the order in which the debuggee must make the procedure calls. The debuggee starts its execution by calling **`INITIALIZE`** that returns a unique *debugID*, not necessarily the same as *session ID*. After that, to start debugging **`DEBUG_ON`** is executed followed by the PL/SQL programs that need to be debugged. If the session is to continue after the debugging has been done, **`DEBUG_OFF`** can be called to switch off the debugging mode in the target session.

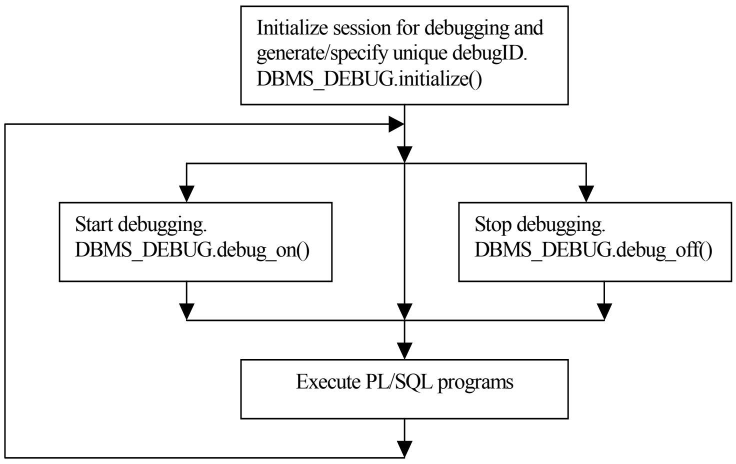

Initialize session for debugging and generate/specify unique debugID. DBMS_DEBUG.initialize()
Start debugging. DBMS_DEBUG.debug_on()
Stop debugging. DBMS_DEBUG.debug_off()
Execute PL/SQL programs

Figure 2.2. Target control flow.

There is one though one step that is skipped in this program flow which should be done previous to **`INITIALIZE:`** the execution of the ***alter statement***. There are two manners for the alter statement, to alter the *session* or to alter *each program unit* to be debugged.

```
alter session set plsql_debug = true;
```

This version instructs the compiler to generate debug information for the rest of the session.

```
     alter [OBJECT_TYPE] [OBJECT_NAME] compile debug;
or   alter [OBJECT_TYPE] [OBJECT_NAME] compile debug body;
```

This second version generates debug information for procedures, functions, packages, triggers and types and the last version for package and type bodies.

Program flow for the target session (*debuggee*)
Figure 2.3 describes the program flow for the debug session. Calling **`ATTACH_SESSION`** is the first call the debugger session does to connect to the session that is to be debugged. After attaching breakpoints can be set and manipulated. The next step is to synchronize with the debuggee and tell the Probe which events are of interest. Next, if the execution cursor is inside a running program: some program context information can be read as the variable values, the program stack, the source code, and breakpoint manipulation in any order. If the cursor is out of a running program, then continue should be called to execute the target program. After executing **`CONTINUE`** the reason for why the debugger stopped can be read in a variable of *runtime_info*.

If the target program has terminated one alternative is to debug another program unit. Another alternative is to execute the target again in order to debug it again. In neither of these cases is necessary for the debugger session to do **`attach_session`** again. A third alternative is to detach the session. In this case the session can continue being used, for instance to attach to a new debuggee session.

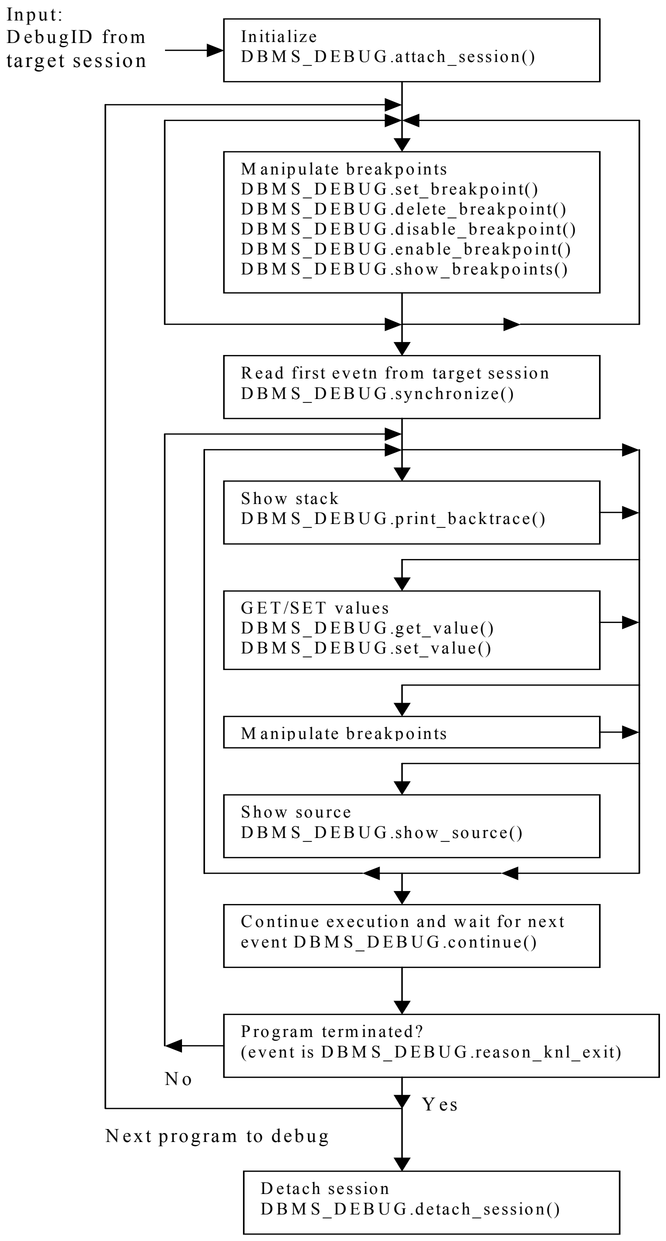

Input:
DebugID from target session
Initialize
DBMS_DEBUG.attach_session()
Manipulate breakpoints
DBMS_DEBUG.set_breakpoint()
DBMS_DEBUG.delete_breakpoint()
DBMS_DEBUG.disable_breakpoint()
DBMS_DEBUG.enable_breakpoint()
DBMS_DEBUG.show_breakpoints()
Read first evetn from target session
DBMS_DEBUG.synchronize()
Show stack
DBMS_DEBUG.print_backtrace()
GET/SET values
DBMS_DEBUG.get_value()
DBMS_DEBUG.set_value()
Manipulate breakpoints
Show source
DBMS_DEBUG.show_source()
Continue execution and wait for next event DBMS_DEBUG.continue()
Program terminated?
(event is DBMS_DEBUG.reason_knl_exit)
No
Yes
Next program to debug
Detach session
DBMS_DEBUG.detach_session()

Figure 2.3. Debug session control flow.

2.1.3 Java Technologies used in this project

This Chapter presents some of the Java technologies for database connectivity and network programming, and some of their properties.

2.1.3.1 Preface

The main objective of this project and the framework in which it will be done, make it necessary to know which platform or platforms it is intended to be run on, what technologies should or will be used and why. The technology used in the base of Corus QuickLink is an Oracle Database Server, and on top there are a number of smaller programs that interact with the core of the system and handle the communication and security within the system. These programs are implemented in Java, using what in general we can call Java Technology.

An intermediate step in the project is to understand how the connectivity between a Java application and the database is done. For this purpose JDBC and SQLJ, that are Java technologies, will be shortly presented and compared.

The remaining object of study regarding possible technologies to use in the implementation of the graphical web-application is to understand what servlets are, how they work, what the advantages of using servlets are and to see some other adequate alternative technologies.

2.1.3.2 Why does Corus use Java Technology and what is it actually?

The frame of this project allows us to naturally narrow the possible choices of technologies to use or combine in the implementation phase. For instance, technologies that expand the functionality of web servers such as Common Gateway Interface scripts (CGI scripts) and FastCGI scripts can be directly discarded. The framework in which Corus QL works is built upon an Oracle Database Server using Java as programming language due to its portability and platform independency. Furthermore, the communication between the clients and the core of the system is handled by servlets.

Talking about "Java" has become a wider and more abstract concept. In the early days of Java it addressed the programming language itself, but nowadays it refers more and more to the platform it constitutes. So, on one hand, whenever talking about "Java" nowadays, it refers to the programming language, the platform, or both. On the other hand, talking about specific "Java technologies" is more concrete and refers to a more narrowed area of programming and techniques.

A question that is necessary to answer before digging into the details of different technologies and alternative architecture models is to know why Corus uses Java. The answer is pretty obvious: in short, Java is platform independent and the most important feature is the network ability offered by Java providing security, modularity and portability features.

Much, much more can be written about *what Java is* and the *network related features*, and it has already been done in many books. Therefore and because to present and explain the Java's features is not the objective of this report the reader who is interested in reading more about this subject can now see: http://java.sun.com/java2/whatis/.

2.1.3.3 JDBC vs. SQLJ

There is an ocean of written material addressing JDBC both in electronic and paper form. Despite this, it is also important to have at least a brief knowledge of what JDBC and SQLJ are. In this Chapter only the main ideas and features and the concept of what JDBC and SQLJ are presented.

The JDBC, which should be addressed more properly as *JDBC application program interface (API)* is a Java application program interface for executing SQL statements. On the contrary to what most of the books say, JDBC is not an acronym for Java DataBase Connectivity, but a trademarked name [6]. JDBC is the standard API used to access SQL databases, and to interact with them via Java classes and methods contained in the Java package *java.sql*. JDBC communicates with the databases using drivers. There are a lot of standard drivers and the user is even able to write its own driver, though in almost all of the cases one of the standard drivers will do just fine to satisfy most of the needs of the user. The driver used in this project is *Oracle's JDBC Thin-driver*.

What JDBC API offers is divided mainly in three features:

- **Access**: permits to login to a database in a secure manner.
- **Queries**: ability to send queries and procedure calls to the database.
- **Results**: the results can easily be handled in the Java application.

For more details see the official web page at http://java.sun.com/products/jdbc/.

SQLJ is a *standard way to* embed *static SQL statements* in Java programs. It can be seen as a complementary role to JDBC. Although static SQL statements can be done in JDBC, SQLJ offers several advantages over JDBC. For example: shorter syntax, type-checking the static code, direct embedding of Java bind expressions within SQL statements and compile-time type-checking. SQLJ applications are portable and able to communicate with databases from multiple vendors using standard JDBC drivers.

Further details about the standard SQLJ can be found at http://technet.oracle.com/docs/products/oracle8i/doc_library/817_doc/Java.817/a83723/toc.htm. Other good sources that were used in this project are found at [1], [6], [27], [30], [32] and [38].

2.1.3.4 Java servlets

A Java technology that still is very popular and expanded is the servlet technology. Despite this fact, nowadays other technologies as JSP [34], Java Beans and EJB are also introduced, expanded and popular. Servlet technology is based, as its name sugges,t on the use and implementation of servlets. Many books have covered how and in what type of applications they are appropriate to be used and in this report only the most important general issues are presented in short. Anyhow, some details are described and discussed wherever it is appropriate in the Chapter 3 that presents the implementation part of this project.

As *applet* is the name given by Sun to Java applications that can be run on or from a web browser, a *servlet* is a service that can be run or called and resides in a web server.
A servlet is a small, platform independent web component; composed of a Java class and that can be dynamically loaded into and run by a web server. A container that implements a request response paradigm manages this Java class that the servlet communicates with the web clients. Therefore, a servlet extends or enhances the functionality of a web server.

A very important feature to know about is the life cycle of a servlet and how they work. Figure 2.4 illustrates a servlet's basic control flow:

- A client access and makes a request to the web server via HTTP.
- The Web server receives the request and forwards it to the servlet container.
- The servlet container determines which servlet to invoke
- The servlet is called with objects representing the request and the response.
- If is the servlet is not loaded, then it will be loaded into the Java virtual machine.
- The servlet get data about the user request from the request object, and some other relevant or necessary data.
- The servlet performs the process it is being programmed to.
- The servlet returns a response back to the Web server via the response object.
- The Web server receives the response from the servlet and forwards it to the client.

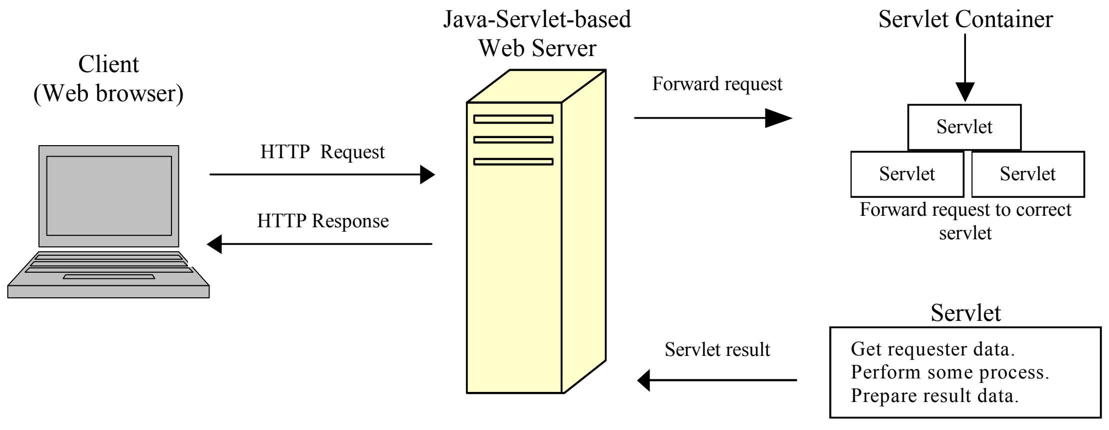


Figure 2.4. Servlet's basic control flow.

After the basic servlet flow, how a servlet works and having and idea of the life cycle of a servlet have been understood, it remains to know why to use them.

Without a deep study or an extensive presentation of the properties and advantages of using servlets (also presented in [7] and [10]), here are some of them presented:

- Persistent. Loaded by the Web server only once and able to maintain services (i.e. database connections).
- Faster and efficient. Because they just need to be loaded once remaining usually in the Web server as a single object instance.
- Highly scalable. Because multiple concurrent requests are handled each in separate threads.
- Platform independent. To work on any client, a servlet only has to work in the machine where it is deployed and tested, unless complete portability is desired.
- Powerful and extensible. Has the ability to use and exploit the full power given by the core Java APIs, such as networking, database access, data compression, internationalization, etc.
- Secure.
- Easy to use with different clients.

In the basic example above in Figure 2.4 the basic servlet flow is presented with a simple request, but there are more examples or use areas in which other examples are applicable. Typical examples of these types of uses are servlet chaining, background processes, communication with applets. For example, in short, servlet chaining refers to the scenario when after a user request is received on the Web server more than one servlet interact sequentially or concurrently before sending back the response. In addition these cooperating servlets doesn't need to reside on the same host. This is of course a very rough description of how servlets can be used and how they do work, but fortunately there are many other sources for using and understanding the use of servlets. A good point of start is at: http://java.sun.com/products/servlet/ and other sources used in this project are found at [1], [7], [10], [12] and [19] are also recommended.

2.1.3.5 Applet – servlet communication and serialization in Java.

How servlets and applets interact and communicate is a very important feature to understand in order to design and choose possible models for the final application in this project. For that reason a somewhat more detailed presentation of this feature is given here, though without going into unnecessary details.

In [10] we found that *tunneling* and *serialization* are two mechanisms used to achieve applet-servlet communication and that the latter one is the one to use with preference. The use of serialization implies that the internal state of an object is stored (*serialized*) on one end of a client-server communication and on the other end this object is retrieved (*deserialized*). This is an easy way for marshaling (packing and unpacking) data between a server and a client.

2.1.4 Applications models

Reminding that the main objective for this project is to verify that the entire chain of the parts or tiers or layers of a graphical web based debugger is realizable; the alternative architecture models commonly used in this type of applications must be studied. The final application is in fact a Java application that uses or communicates with a PL/SQL package that can be seen as a database application, and that according to the requirements for the project could also be used alone as a command-line debugger.

2.1.4.1 Database applications, PL/SQL and Java

Usually, database applications are divided into three parts:

- The user interface that handles the look and feel of the application and the interaction with the user.
- The application logic, controlling the work executed by the user and that in some cases stores the state of the program.
- The database, where the application data is persistently and reliably stored.

The Two-Tier Model

Also known as, the Client-Server model, is composed of a client side and a server side, and is characterized for splitting the application logic between the server and the client. The client side of the application handles in that manner the interface and a part of the application logic. The server side principally stores the application data but it also can or does handle some of the application logic. Figure 2.5 illustrates this type of application model.

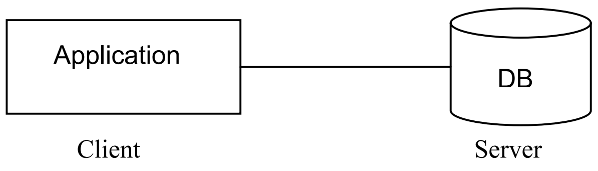


Figure 2.5. Two-tier or Client-server model

Three-tier model

The *three-tier model* is a specialization of the more general *n-tier model* and that is common in many real live examples because many systems have in fact more than three well defined tiers.

In this architecture model the user interface, the application logic and the database storage are split into (whenever possible) well-defined different parts. Usually the first tier is referred as the *presentation layer,* consisting of a graphical user interface. The middle tier is referred as *business layer* and usually contains the application logic. *Data layer* is the third layer where the data for the application is stored.

For some type of applications the middle tier, the business layer, is split into two layers one that takes care of the *presentation logic* and the other handles *business logic.* Figure 2.6 illustrates an adapted model for Java applications that apply to this project.

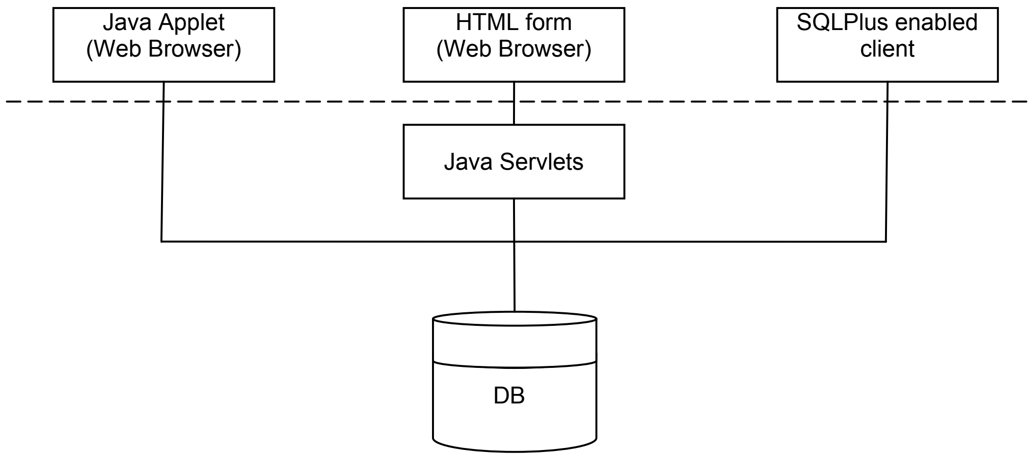


Figure 2.6. Application models applied to this project. 2-tier and n-tier applications.

There are two important elements to know about when addressing model architecture: the PL/SQL Engine and the SQL Statement Executor. PL/SQL blocks are executed in a PL/SQL engine; if a PL/SQL block has a call to a stored procedure in the server then the call to the procedure is sent to the server. Standalone SQL statements and SQL statements within PL/SQL blocks are sent to and executed in the database server. In addition to the PL/SQL engine in the database server, some applications have their own PL/SQL engines. Communication between different PL/SQL engines and the server are done through database links.

2.2 Market survey of third party Oracle PL/SQL debugger tools.

The objective of this market survey was to see what kind of functionality was implemented in third-party vendors' debuggers. The market survey should also help to decide what functionality is required and wished to be implemented in CORUS_DEBUG. It was also done to get some inspiration, to get some knowledge about what is possible to do and what not, and possibly to see some difficulties that might appear in the implementation phase of this project. The objective was not to compare these debuggers with each other and make a rank order of any kind. Therefore the results and insights obtained in this market survey are just summarized in this Chapter.

The tested debuggers are listed below and trial versions of them could be downloaded from the Internet.

- *TOAD* and *SQL Navigator* from Quest Software[2]
- *Xpediter/SQL* from Compuware[3]

[2] A trial version can be downloaded at: http://www.quest.com
[3] A trial version can be downloaded at: http://www.compuware.com/

- *SQL Programmer* from SFI-Software[4]
- *Rapid* SQL from Embarcadero[5]

The results and conclusions obtained from this market survey are in no particular order:

- Initializing and starting a debugging session is not always easy and might require a schema or object browser in order to do this smoothly.

- Synchronization between the target and debug session can be done transparently but it might be difficult to do so. Some of the tested programs just hanged when debugging a program. This hanging-problem was so hard that the process of the application had to be killed from the operating system to continue.

- It should not be so difficult to debug depending programs automatically. A depending program is for example a procedure or package called from another procedure or package.

- Not all the debuggers could read package variables and less set them; only some of them and not in all cases.

- Reading and setting procedure variables is possible.

- Implementing watchpoints is possible but it might be hard to implement as not all the applications have this function.

- Handling breakpoints is easy, but none of the debuggers offers the functionality of setting a breakpoint in a procedure just by pointing at it either graphically or in some input text area. The source code must be displayed and the breakpoint set at a specific line.

- Switching between debug mode and execution mode is difficult to do. Some programs tend to hang or interrupt some windows when switching between debug and executing mode.

- Finally, though it was not the objective of this market survey it must be said that TOAD was the best, most user-friendly and powerful among all the debuggers tested. Unfortunately, not even TOAD is adapted to work in the framework that Corus QL has.

[4] A trial version can be downloaded at: http://www.sfi-software.com
[5] A trial version can be downloaded at: http://www.embarcadero.com/

3 Design and implementation of the web based PL/SQL debugger

Having studied or reviewed the components necessary to develop a debugger and a web application, able to be run over a network and with the conclusions obtained from the market survey, the next step is to put all these pieces together. This must be done in a convenient order, permitting iteration and this is done through a developing plan.

This developing plan, in this case, must first include, a deep understanding of the functionality and capability of Oracle's DBMS_DEBUG package; second, design, implementation and testing of the own debug API in PL/SQL; finally the design of the highest layer of the implementation that consists of the interaction with the database and the network-related features for this application.

Recall that the objective of this project is to design and implement a web based debugger and that a key part goal is the implementation of a PL/SQL package on top of Oracle's DBMS_DEBUG (DD). This package should hide, abstract and facilitate the use of DD for the user who should be able to use the package as a command-line debugger. Furthermore, the details concerning the declaration, initialization and use of the complex variables of the defined data types in DD should need to be hidden and simplified for the user. The web application should implement a user interface for a web browser using this package and probably handle some of the application logic. The design of the web application will permit easy or smooth extensibility of functionality. Finally, both the package and the web application must be designed and implemented in such manner so that they can be integrated into Corus QL's framework with as few changes as possible.

The integration of the PL/SQL package and the web based debugger into Corus QL will add an important, necessary and requested feature: to debug the generated code in Corus QL within the same environment it was created in. They will also reduce the cost for the end-users of Corus QL since licensing for external debuggers will be eliminated.

3.1 Developing plan.

The product of the design and implementation part of this project is a web application called iCODE[6]. iCODE is based on the tailor-made PL/SQL package CORUS_DEBUG (CD) which in turn is based on Oracle's DBMS_DEBUG API (DD). Here follows the intended developing plan to carry out the implementation part of the project.

The milestones to follow in the developing process of this project are:

- Test and deeply study the DBMS_DEBUG package.

[6] iCODE is an acronym for *i Code Online Debugger,* in accordance to the name giving for applications in Corus QuickLink.

- Creation of a plan of what functionality to implement and in which order of preference or convenience, probably in form of a list of requirements.
- Design and implementation of an own package/API on top of DBMS_DEBUG.
- Design and implementation of graphical application using own debug API/package.
- Tests for verification and evaluation.

The results and conclusions of the tests and study of DBMS_DEBUG are presented in Chapter 3.2. Implementation and design decisions taken regarding the CORUS_DEBUG API, are presented in detail in a sort of chronological order in Chapter 3.4. Finally the design and implementations details and technical and strategic decisions related to the graphical application are discussed in Chapters 3.5.2-3.5.4.

In a developing plan it is appropriate to have an application model that give guidance for the design and implementations phases. The next chapters present two guideline models used in the implementation of CORUS_DEBUG and iCODE.

3.1.1 Application model for CORUS_DEBUG

CORUS_DEBUG was intended to be implemented in such a way that it could be used straight away as a console debugger from a SQL client, like as SQLPlus.

In Figure 3.1, there is a client that executes calls to DBMS_DEBUG which is the API to the debugger layer in the Oracle server. This is done transparently through CORUS_DEBUG. The client doesn't need to know anything about the presence of DBMS_DEBUG which is the base component of the debugger.

How the client controls the console debugger was implemented according to the control model that Sommerville describes in [15] as the *call-return model.* The control starts at the top of a subroutine hierarchy and continues down in the hierarchy all the way until DBMS_DEBUG.

Figure 3.1 shows two hosts and two databases. One host can hold two database connections: one for the debugger session and one for the debuggee's session. However, the debuggee and the debugger don't need to reside on the same host, or even in the same database. In the case when the debuggee resides in other database, the target programs should be executed via db-links.

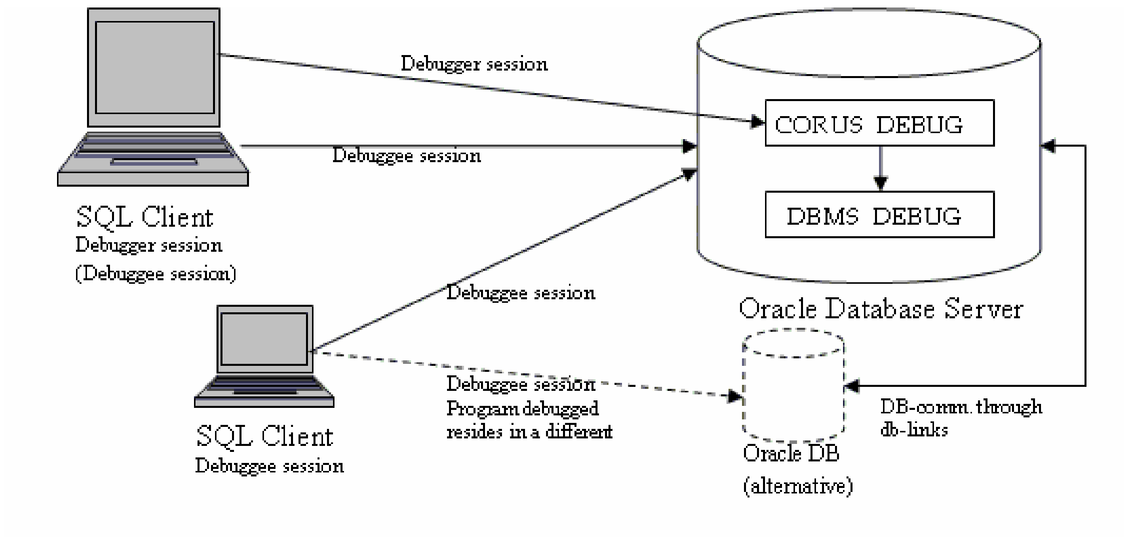


Figure 3.1. General application model for CORUS_DEBUG.

3.1.2 Application model for iCODE, the web application

The graphical web based debugger could be run either from an applet or from dynamic HTML pages that are generated by servlets. The figure below depicts a general model for how this could be implemented.

Figure 3.2 presents two kinds of clients, one is running an applet directly from a web browser, and the other is running the debugger via HTML pages that make requests to servlets stored in a Java enabled web server.

In the first case the applet is loaded to the user's web browser and then it can be run from the browser, keeping the database connection objects within the applet context. In the second case, the database connection objects for the servlets are stored in the web server during the entire debugging process.

In both application models depicted in Figure 3.2 DBMS_DEBUG is at the bottom of the chain of the application. Therefore the next step, as formulated in the developing plan, is the study of this package.

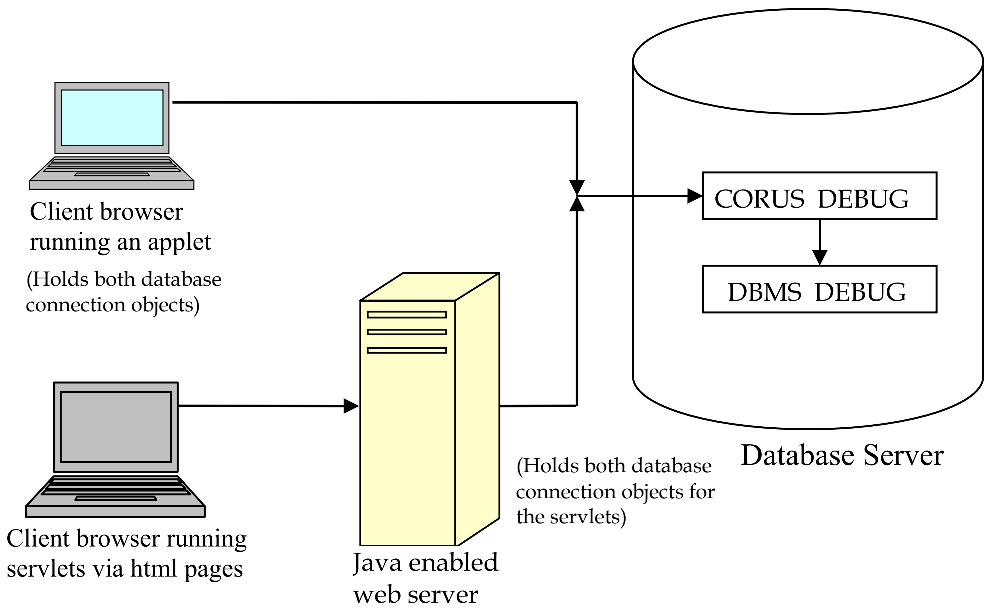

Client browser running an applet
(Holds both database connection objects)
CORUS DEBUG
DBMS DEBUG
Database Server
(Holds both database connection objects for the servlets)
Client browser running servlets via html pages
Java enabled web server

Figure 3.2. General application model for iCODE.

Having these two models presented the next step was to dig into the details, problems, strengths and functionality of Oracle's DBMS_DEBUG.

3.2 Study and tests of the DBMS_DEBUG package

Before going into details of how the DBMS_DEBUG package was tested and what conclusions were obtained out of this study, it is very important to stress the fact that this API was done for third-part vendors. Hence, there was no information available anywhere about this API and how to use it, neither in the open Internet nor in Oracle's web sites: Technet[7] and Metalink[8]. All that was found regarding DD was questions with no answers in different forums on the Internet. Hence, all the conclusions obtained are based in the official documentation for DBMS_DEBUG[9] package and the tests done. This document can be found on [2] or online at [29]:
http://download-west.oracle.com/docs/cd/B10501_01/appdev.920/a96612/ d_debug.htm

To start with, the first thing to do was to study the data types defined inside DBMS_DEBUG and some of the procedures and their parameters. Being familiar with the data types defined in DD was important in order to decide whether these data types were enough to create an own API, or if additional data types were necessary.

The method used for testing DD was not very advanced; each procedure was tested, one at a time. First, they were tested with their default values, if any, and then with alternative valid and invalid values to study and note the behavior of both sessions. The program

[7] *Oracle Technology Network,* Technet can be found at: *http://technet.oracle.com/*
[8] *Oracles's Metalink,* can be found at: *http://metalink.oracle.com/*
[9] Documentation for *DBMS_DEBUG API* is found in ch.7 of Oracle8i Supplied Packages Reference, Release 8.1.5

context information and how it changes was carefully observed. It permitted to find out among other things another bug in DD (the listed values for break_any_return and break_return are backwards), which later on showed to be already reported and well-known but not noted in [2] or [29]. This procedural testing was necessary to make qualified guesses or deductions of what Oracle's engineers and designers intended to do and had in mind when defining certain data types and procedures. Even if it might appear to be inappropriate in a report of this kind it has to be said that it was difficult, very time- and patience-demanding.

Fortunately, after a number of attempts, one could notice a sort of silver thread when studying some procedures. One could see that some procedures absolutely needed some other procedure to be executed first in order to create some automatic and transparency between procedure calls. For instance, it is sometimes necessary to execute get_program_info before setting a breakpoint, and so forth.

The only way to test DD efficiently was by creating an own package with calls to the different procedures in DD and test them for the different cases such are single procedures, packages and triggers. This package became the base for CORUS_DEBUG.

After doing these procedures calls and tests, almost all hard-coded, the following could be noted about DBMS_DEBUG:

Initialization and termination

- The first important conclusion is that, for the nature of a debugging session and the way this API is done, there is no way to make undo procedures as the execution is run and stopped in the Probe (the debugger layer in Oracle Server).

- Initializing and synchronizing are very sensitive parts in any debugging session and probably will be in CORUS_DEBUG as well

- It is not possible to create a database session that creates and controls two sessions at the same time. Two different sessions must always be created.

- Detecting the termination or finalization of a program that is being debugged is also complicated if this is to be done transparently and smoothly.

- It is not possible to make initialization of both the target and the debug session simple, as we need the debugID returned by *dbms_debug.initalize* in the target session in order to attach the debug session to the target session.

Synchronization and execution

- Synchronization showed to be a very sensitive function necessary to do every time a program is going to be debugged.

- There is a warning in the documentation, but it covers only one case, namely when the target session has terminated. In reality, the debug session hangs every time it calls *dbms_debug.synchronize* unless there is a program waiting to be run in the target session or when a program has just finished its execution and the execution cursor is inside the interpreter.

- Stepping through the program execution and stopping at some events are controlled by *dbms_debug.continue*. The *break flag* given to this procedure is a mask that with preference would be created dynamically in an intelligent manner or hard-coded for each debugging function.

Breakpoints

- Deferred breakpoints can't be easily done. It is required to scan the source code to debug to get the line number at which the breakpoint will be set. Thus, some strategy to support deferred breakpoints might be needed to implement this functionality.

- There is a bug, because the debug session hangs if *dbms_debug.show_breakpoints* is called before synchronizing and if the target session releases the debug session. The pipe connection between the two sessions then breaks.

Source Browsing

- Procedures to handle source-related functionality must be implemented taking in consideration these three cases: procedures, packages and triggers. The procedures in DD are note enough and produce faulty errors when called from a Java program.

- One difficulty writing a source code parser to facilitate the setting of deferred breakpoints is that the source code stored in the database doesn't match the source code in the file system. The comment lines in the beginning of a file are removed in the database.

Program context information

- Reading and setting variable values must be taken care of in different ways depending of if the execution cursor is in the scope of a procedure, a trigger or a package. Holding track of namespaces and lib unit types is important for these functions.

- A proper manner to read the program stack and runtime information must be done as the information returned by *get_runtime_info* and *print_backtrace* is good and compounds of many parameters, but this information must be user-friendly formatted.

Other observations

- There are some bugs in DD. If some of the difficulties met with synchronization, program execution and termination are caused by bugs in DD, or are caused by misuse of the *continue* and *synchronize* procedures was at this point unclear.

After summarizing the facts obtained studying and testing the DD API, what can be said is that initializing, synchronization with and termination of a program that is or will be debugged are very sensitive; there are fundamental differences in the namespace, lib unit types, and how the source code is stored in the database between procedures, triggers and packages; there is no additional information besides API's official document, so deductions and assumptions and even guesses must be noted and have a very important role; pending or deferred breakpoints must be handled separately and to do this a strategy must be made up; finally, some own defined data types probably might be needed for setting deferred breakpoints and for automated source browsing.

3.3 Requirements for the package and the final application

With the results obtained by the market survey two lists of requirements have been established. One list is for the PL/SQL package and the other is for the web application and its front-end graphical user interface. The requirements will be gradually fulfilled, probably through a number of iterations implicating changes in design, strategies and implementation.

Requirements for CORUS_DEBUG

The following points will be implemented in the PL/SQL package CORUS_DEBUG, which is based on DBMS_DEBUG. With help of the package the user must, directly or through an application, be able to:

- Establish a connection between the target and the debug session.
- Run a program in debug mode.
- Control the execution of the program by stepping the execution or the program, line by line.
- Read the code that is being debugged. If not wrapped.
- Set breakpoints at a specific line and jump in the execution to that line.
- Step and stop the execution at different events as going out of a procedure, out of a program.
- Set breakpoints by giving the program name as parameter.
- See the program stack and runtime information.
- Read and set variables in single or standalone procedures and functions and in triggers.
- Read and set package variables.
- Debugging of single procedures and functions as well as packages and triggers. By debugging triggers means the PL/SQL programs called from a trigger and the logic that might exist within a trigger.
- Smooth finalization of the debugging session.
- Traditional help functionality.

Requirements for the web based PL/SQL debugger

This list is for the features to be implemented in the graphical web application.

- The call ability properties of CORUS_DEBUG must be demonstrated using JDBC. A set of methods showing that a debugging session can be done via JDBC using the CORUS_DEBUG API.

- As a middle step an applet or a set of applets can be implemented. This applet or applets will be able – if necessary – to collaborate with an object that provides persistent database connections during the debugging session.
- The solution on the preceding point will be lifted up and implemented with servlets.
- The servlet implementation must permit modularity and extensibility in functionality, meaning that if not all the typical functions or all functions that a typical debugger gives, this design will permit to add more functionality in future development.
- A dynamic call to procedures might be done if there is enough time, or at least a study of how this can be implemented resulting in some proposals.
- Finally, the servlet implementation might be adapted for integration to Corus QL's framework. As this point might require a lot of work to have a good design, an implementation of it will be done only if there is enough time, otherwise some outlines of some design alternatives will be enough.

There are no specific requirements regarding security for the entire project. The CORUS_DEBUG API will allow debugging and accessing the files that the user is granted to access. Another security limitation is when the source code of the program to debug has been wrapped, which means the user won't be able to neither read the code nor debug the program. Security in the graphical application will have two layers when introduced in Corus QL. One layer is the same as for the package, which means the security grant properties in the database, and the other is handled by a logging system above some database applications, this logging system is called NAVAJO as is part of Corus QL. Navajo handles a connection pool of database connections, the security features for Corus QL regarding access grant properties, timeouts, etc.

3.4 Design and implementation of the CORUS_DEBUG API.

In the designing phase of CORUS_DEBUG API there were some points kept in mind:

- CD is sort of a wrapper package for DBMS_DEBUG. CD has to be able to be used by different user interfaces.

- CORUS_DEBUG has to provide an interface that implements a standalone command-line debugger.

- A Java-adapted interface is to be implemented as well at the same time.

- Accessing package variable is very important in the Corus environment.

- The system structure for the debugger will be in a layered fashion.

In order to achieve the property of permitting different kinds of user interfaces to access a common API a name giving convention had to be taken.

3.4.1 Procedure overloading and name giving: *sync, syncF and syncJ*

The name of the procedures in CD follows two strategies. One of the main requirements with this package was that it could be used directly for debugging. Therefore it was necessary to make procedures of all functions in DD to facilitate the procedure calling for debugging functions. It is tedious to create bind variables or even more complicated user-defined variables mapping DD's defined types and then use these variables as parameters. There are function-procedure pairs composed of a *procedure* that the user easily can call and of a *function* that returns either a return value or some information text as return value. For example: the procedure *sync* maps *syncF, syncF* can be called either by the procedure *sync* and displayed to the user in a console screen or by a Java class and displayed to the user in an applet, a HTML page, probably after reformatting the result.

Another strategy that has been taken is to have a function that returns to Java objects VARRAYs, because the results are sometimes composed of <key, value> tuples. It is much easier to return results in this way than implementing parsers that should parse a complicated result. Ex. *print_stack* is mapped with *print_stackJ.*

A number of the procedures in CD are help functions done to facilitate the implementation and debugging processes and to enhance the display formatting.

3.4.2 Debugger features implementation

The order in which the procedures in CORUS_DEBUG were implemented was quite natural because it mainly followed the requirements for this API listed in Chapter 3.3. The final version of each procedure and function is the result of several iterations and changes due to certain dependencies between procedures, and due to the differences when debugging the different kinds of program units (procedures, packages and triggers).

Note: The final version of CORUS_DEBUG is the result of a revision done in 2005. The description below describes the problems and decisions taken in both the first and the reviewed version of CORUS_DEBUG. In the final version the BPT_HANDLER package, which was implemented to handle deferred breakpoints, has been removed. Therefore CD is now composed of only one package.

Before discussing what, how and why certain functionality has been implemented in CORUS_DEBUG a couple of points must be noted.

- It is always necessary to open two database sessions: a *debug* session and a *target* session in order to debug a program unit. These two sessions working together to debug a program unit is in this report called a *debugging session*.

- There is no manner to handle and control two database sessions (for the debugger and the debuggee) from only one database session. Pipe functionality to communicate between two sessions exists, but it does not have the functionality necessary to have a debugging session. Another package provided by Oracle with the name DBMS_PIPE was studied and tested.

- Since Corus QL's integration engine and repository have own-defined data types, the functionality to read and set variables of these types was desired that. However, focus should be put on standard PL/SQL and SQL data types.

- The main requirement was that the user could easily and intuitively use the package or set of packages that CORUS_DEBUG would be composed of. This package would also be adapted to be called from Java methods.

With all this requirements in mind, the conclusions obtained by studying the DBMS_DEBUG API, and having learned the programming features available in Oracle's PL/SQL and SQL, finally the implementation of the package started. To begin with *initialize*, *attach_session*, *synchronize* and *debug_on* were implemented, almost without changes when using the mapping procedures and functions in DBMS_DEBUG. The procedures and functions in CORUS_DEBUG here set the value of some global variables. In addition, certain default parameters, that have been strategically chosen, were set.

Source browsing

The next step was to implement procedures to display the source code. These procedures display both the entire source code and just part of it. Here, it must be mentioned that the final version of these procedures are the result of many iterations of CORUS_DEBUG. The source codes for packages, procedures are stored in *all_source* and/or *user_source* tables in the database and they are in fact views, and the source code for a trigger is stored in a field of type long in the *user_triggers* table. *show_source_trigger* parses this field, recreating the source code in the same fashion as procedures or functions. In addition to the difference between procedures, packages and triggers, there is the fact that a package is composed of its specification and of its body. This fact makes it necessary to have a further separation when reading the source code and when reading and setting variables, which is discussed further in the document.

At the beginning a copy of the last read source code was stored in global PL/SQL tables of type *dbms_debug.vc2_table*, which holds strings of type *varchar2(90)*. The type of these PL/SQL tables was changed and a new user-defined data type defined as below:

```
TYPE source_vc2_table IS TABLE OF VARCHAR2(500)
     INDEX BY BINARY_INTEGER;
```

This type was introduced, and all the changes this new type caused took place when the debugger in Java was being done. The reason is, namely, that the field in which procedures and packages source code is stored is of type varchar2(4000) and the corresponding type in DBMS_DEBUG is varchar2(90) causing an exception when called from a Java program. The length of 500 was chosen assuming that no typical line of code is larger than 300 or 400 characters. But if it is shown that there are some code lines longer than 500 characters it is only necessary to change, in principle, the length of the field in the type definition. Handling triggers source code was reduced to access and parse the code for each trigger, which was stored in a variable of type long in *user_triggers* table. The use of this code was removed in the revision version.

The procedures *show_source_procfcn, show_source_pkgbody* and *show_source_trigger* do all the work now. These procedures are called internally by *show_curr_source* and the source browsing functionality was improved. The source code is shown in what is referred to as a window, letting the user to set width and height for the source to be returned or shown. It is done with *set_win_width* and *set_win_height*. The height parameter determines the number of lines to show before and after the current line. In addition, if the user wants to see all the source code or just the 'window', it can be changed with *set_win_mode*.

Breakpoints

Enabling, disabling and deleting breakpoints that have been set were not difficult to implement, only the breakpoint number is needed and it can be obtained by listing the breakpoints that are set. The hard task was to set breakpoints. And setting what was called *pending-breakpoints* was the hardest part of it, which will be discussed later in this Chapter.

When the line number in the code is not an executable line, usually, debuggers stop at the preceding or posterior executable line. Even though there is a parameter to use and the purpose determined in *dbms_debug.set_breakpoint* to accomplish this functionality. This functionality of automatically setting a breakpoint to a previous or posterior executable line has not been implemented yet. This is a common feature in modern debuggers and it was assumed that the future users of CORUS_DEBUG would expect the same feature. Parsing and recognition within the source code that enables finding an executable line can solve this problem. Though, parsing usually tends to be a time-demanding task and after a number of attempts to implement such a parser it has been postponed for future work.

In order to set a breakpoint in CORUS_DEBUG, the line number must be given, but there is a procedure in where you can set a breakpoint in a program that has not yet been accessed by the interpreter, *set_breakpoint_fullinfo**[10]. This procedure permits to set a deferred or pending breakpoint that is hold by the interpreter and set at synchronization.

In the final version a breakpoint can be easiest be set with, for example:

```
          set_break(11)  or set_break(11, 'PROC_NAME');
```

Pending breakpoints

The experienced programmer will find it unnecessary and time demanding looking for the line in a source code in order to set a breakpoint. He or she is probably used to set breakpoints by just giving the name of the procedure, function, method, package or module where he or she wants to stop the execution. Thus, when debugging in PL/SQL setting a breakpoint at a procedure or nested procedure is necessary. Thinking of that fact, another solution was implemented. This functionality is accomplished by several

[10] The procedure and function names marked with a * have been removed from the final version of CORUS_DEBUG.

cooperating procedures and functions. These cooperating procedures store the necessary information for breakpoints in an index-table, 'listen' if the execution cursor is inside of a program corresponding to any pending breakpoint, and finally update the rows of this table after have set a real breakpoint. This scanning for pending breakpoints is done each time a 'step-function' is executed. This function still needs parsing improvement. This functionality was initially implemented in an own package called `BPT_HANDLER*`.

Two types have been defined in order to achieve this functionality:

```
      TYPE bpt_rec is RECORD(line#          number(4),
                             pkg_name       varchar2(30),
                             proc_name      varchar2(30),
                             proc_type      varchar2(12),
                             namespace binary_integer,
                             owner          varchar2(30),
                             dblink         varchar2(30));
      TYPE bpt_table_type is TABLE OF bpt_rec INDEX BY BINARY_INTEGER;
```

Until the next version of DBMS_DEBUG API is released the best manner of setting a breakpoint if the execution cursor is not in the program where the breakpoint is, still is to scan the code. It is necessary to find an executable line appropriated to be set as a breakpoint, and use this line's number in *set_breakpoint_fullinfo** or *set_breakpoint_at**. Yet, the parsing done in this procedure or more exactly in *get_first_exec_line** (called by set_breakpoint_at*), has to be reviewed and improved. In the meanwhile it has been put aside because there are too many cases to handle, and the size of this project doesn't leave space to solve this parsing problem.

Program context

When debugging a program, a very useful function is the ability to get information about the runtime, the current program and especially information about the program stack. In the API for CORUS_DEBUG all of these functions are implemented, and even the ability to set the value for the global variable `l_program_info` (removed in the reviewed version). This was shown to be useful, for instance to call some procedures or functions that depend on this information, e.g. to call *get_pvalue, set_breakpoint, etc.*

dbms_debug.print_stack is a procedure that let the user see the program stack, referred to in this project also as back trace. The program stack is stored in an index-table in CD, namely the `g_backtrace_table*`. From this table the frame number can also be read became important when reading and setting the values of variables.

Local variables

What is important in order to access and read variables, is to keep track of the *frame number* and of the namespace when the variable is a package variable. The default frame number for where the execution cursor is located is 0, which is a copy of the frame number that can be read from the back trace table. This did not turn out to be as simple as one could initially think, in order to make this functionality transparent to the user. Therefore the user has to read the frame number from the back trace table.

```
-- The default case for current program
cd.get_value('var_nr');
-- Returns
var_nr is 4711
4711
```

A variable that is in a program unit that is not the current one

```
cd.get_value('var_nr');
-- Returns
no such var/parm

-- The frame number is needed.
-- The frame number is the same as the entry number
cd.stack;
-- Read the frame number out of the stack table
cd.get_value('var_nr', 2);
```

Setting the values of variables has the same problem. It requires that the user must be careful if he or she wants to set the value of a variable and to see whether the name of this variable exists somewhere else in the scope of the programs in the back trace tables. Thanks to the implementation of *dbms_debug.set_value*, the assignment of a value for a variable is just like an ordinary PL/SQL statement, for example for the variables `var` and `txt`:

```
var := 10; or txt := 'Hello';
```

The assignment statement is a parameter that has to be set as a string, which means between apostrophes, as in this other example to set a field in a record type:

```
'my_rec.some_fld := "new value"; or
'my_rec.some_fld := 'new value';
```

Package variables

Handling package variables was a challenge since some of the tested debuggers couldn't show or set the value for this kind of variables. After studying and understanding how DBMS_DEBUG is built and how it works, it is near at hand to guess that the cause of why these products lack this functionality is that they have missed something in doing this study, either for having to little resources or the patience and perseverance this study requires. This functionality was of special importance because Corus QL uses package variables, also called global variables, in the integration engine. This is a very important and fundamental feature for this project. Fortunately in the CORUS_DEBUG API accessing, reading and setting package variables is possible.

The crucial part to achieve the capability of accessing, reading and setting package variables is the *namespace*. This parameter is silently set by CORUS_DEBUG, so the user doesn't need to hold track of it. However, there is a problem that might occur. It is caused by the fact that not all the developers declare package variables only in the package specification file, but also in the package body. Then it is necessary to set the namespace value manually, like in the following examples:

```
cd.get_pvalue(x_in_body);    -- declare in the package body
```
or
```
cd.get_pvalue(x_in_body, 1); -- declare in the package spec.
```

Execution control

Until this point, what has been discussed is the implementation of the procedures and functions for: preparing and starting a debugging session, reading and scanning the source code, setting breakpoints, reading and setting variable values, reading runtime and program information. The remaining parts are very important for any debugger, namely, the procedures and functions that actually control the execution of the program being debugged. In addition, it remains to discuss about watchpoints and some help procedures that make CORUS_DEBUG capable to be used by a Java application.

The central parts of the 'execution control' procedures (also called as step-procedures): *step, step_in, step_over step_out, trace_out* and *finish* are the break and information flags that are forwarded as parameters to the *continue* function in dbms_debug. The information flag has been set to 14 as default in order to request all the information that can be retrieved when calling *dbms_debug.continue* function (gives stack depth, breakpoint and line data). There is no need founded to request just part of this information, so it has been assumed that these three different flags must exist for consistency reasons. These 'step-procedures' are consequently a sort of alias for different calls to the *corus_debug.continue* procedure. *corus_debug.continue* forwards the procedure calls to dbms_debug.continue with the right parameters, and makes some control of changes in the runtime information. It then takes some measures depending of these changes, e.g. returning a message telling the user that the program execution has ended and the *finish* procedure can be called to get out of the Probe.

When a user attempts to execute *step,* for instance, the procedures described above set the proper *breakflags* and pass the call forward to *continue,* which in its turn forwards the call to the function *continueF.* The purpose of having a function doing this is that it also can be directly called by a Java application. Figure 3.3 below shows the source code for the reviewed version of this function.

```
---------------------------------------------------------------------
function continueF(breakflags in binary_integer,
                   ret_msg out varchar2)--,
                   --info_req_in in binary_integer default 14)
             return binary_integer as
---------------------------------------------------------------------
--! To be called by java methods.
--! See also : 'procedure continue'
---------------------------------------------------------------------
  l_ret_code    binary_integer;
  v_reason      binary_integer;
  v_reason_str  varchar2(40);
begin
  if g_is_attached = true then
    -- info_req_in replaced by mask of breakflags
    l_ret_code := dbms_debug.continue(l_runtime_info, breakflags,
                                 0 + LineInfo +  Breakpoint +
                                 StackDepth + OerInfo);
    v_reason_str := get_reason;
    if(l_ret_code = Success) then
       print_msg('Reason of stop: ' || v_reason_str);
    end if;
    return l_ret_code;
  else -- when not attached
    ret_msg := 'Debug session is not attached to any target
session.';
    return -1;
  end if;
exception
     when others then
  declare
    error_code NUMBER := SQLCODE;
    error_msg VARCHAR2(250) := SQLERRM;
  begin
     print_msg('Exception raised in continue');
     show_error(error_code, error_msg);
     ret_msg := ('Exception raised while executing:continueF. '
                || SQLCODE || ':' || SQLERRM);
     return -2;
  end;  -- end of exception handling
end continueF;
```

Figure 3.3. Source code for *function continueF* in CORUS_DEBUG.

As expected, there are a number of help procedures in CD. Most of these procedures and functions are aimed to make data accessible from a Java application, for instance to retrieve the entries in the index-tables. There are also several procedure-function pairs, which consist of a procedure and a function, called for instance *continue* and *continueF*, that mostly do the same thing but differs in how they return or show the result. A function of this type usually returns some message telling whether the execution succeeded or not. The corresponding procedure displays the result information to the user that is using the package. Having this set of procedures and functions gives the flexibility and callability required to use CD straight away and to build some applications on top of it.

User support

User support is given by the procedure *help*. It gets a procedure name as parameter and displays or returns a description of the function provided, a usage-style description and an example.

Watchpoints

The last feature to discuss is watchpoints which is a much desired feature in any debugger, it is also one of the most difficult to implement because of all the background work necessary. It has been skipped in CORUS_DEBUG for two reasons, the first one is the short time for the project, and the second one is that the coming DD version will possibly have some functionality for this. A possible approach is to implement a pooling procedure, that executes pooling over the programs in the stack looking for changes in the variables listed or marked as watchpoints.

3.4.3 Features and procedures provided by CORUS_DEBUG

As mentioned earlier in 3.4 CD is a wrapper package hiding the implementation and functionality of DD. The features provided in the command-line debugger are listed in Table 4. The public user-friendly procedures are grouped by the features they implement.

Table 4. Features implemented in the standalone debugger and mapping procedures.

Feature	**Mapping procedures**
Initialization and synchronization	initialize (initialize the target session) attach_session (attach the debugger session to the target session) detach (detachs debugger session from the target session) sync (synchronizes the debugger and target sessions)
Breakpoint	set_bp or set_breakpoint (sets a breakpoint) list_breaks or bpts (lists available breakpoints) del_break (delete a given breakpoint) disable_break (disable a given breakpoint) enable_break (enables a given breakpoint)
Execution control	step (executes to next line) step_in (steps into the next called program unit) step_over (executes to next breakpoint or to next return) step_out (executes to next return) trace_out (executes to next return, to leave the interpreter) abort (aborts target execution)
Variable access and manipulation	get_value (gets the value of a local variable) get_pvalue (gets the value of a package variable) set_value (sets the value of a local variable) set_pvalue (sets the value of a package variable)
Source browsing	curr_source_line (prints source code of the current line) show_curr_source (prints a portion of source code around the current line) show_source_procfcn (prints source code of a given procedure) show_source_pkgbody (prints source code of a given function) show_source_trigger (prints source code of a trigger)
Program context	print_stack (prints the program stack) print_runtime (prints the current runtime info)
Support	help (prints usage for a given command, or a list of commands)

The complete CORUS_DEBUG API with the parameters lists for each procedure in the standalone debugger and the web application is found in Appendix A**.** For further details about CORUS_DEBUG contact Corus Technologies[11].

3.4.4 Facts about CORUS_DEBUG API.

The CORUS_DEBUG API was first composed of two packages, one that maps most of the procedures in DD and one that handles deferred breakpoints called BPT_HANDLER (BPT). The latter package was discarded during the reviewing of the project. The user-defined type *source_vc2_table* created and used in the first version of CD (the package that resulted of the study of DD) has been replaced with the procedures *show_source_procfcn*, *show_source_pkgbody* and *show_source_trigger*.

The final version of CORUS_DEBUG contains 72 procedures and functions out of which 35 are in the public API. The public procedures are intended to be used directly as a command-line debugger.

The features implemented in CORUS_DEBUG are:

- Initialization
- Session attaching
- Turning on and off the debug mode of a session
- Setting, deleting, enabling, disabling and listing of breakpoints
- Step, step in, step over, step out, trace out and abort execution
- Access and ability to set local variables
- Access and ability to set package variables
- Source code browsing of current executing program, procedures, functions, package specifications and package bodies and triggers
- Stack information
- Runtime and program information
- Usage help with examples

3.5 Design and Implementation of the web based PL/SQL debugger

The final application, the graphical debugger, have grown up in mainly three steps or three versions. In each of these versions improvements has been done in comparison to the prior version. The denominations for these versions are *test version, applet version* and *servlet version.*

3.5.1 CORUS_DEBUG API - the base component

The basic component of the final and the intermediate versions of the graphical debugger is the CORUS_DEBUG API. The complete package specification is found in Appendix A and the public procedures in Appendix B.

[11] Corus Technologies web site: www.corustechnologies.com

3.5.2 Technologies (SQL, PL/SQL, JDBC, Java servlets)

For consistency reasons, the technologies and programming languages utilized in the development and implementation of the web based debugger must be listed and their selection motivated.

SQL was used among other motives to formulate and execute queries on the database. For instance, to read the source code of the program units, to create the command help table, to give the right grants to users, etc.

PL/SQL is an obvious element used to create CORUS_DEBUG and test program units.

JDBC was chosen as the only technology to connect to the database because the other alternative, SQLJ, is not appropriate since the latter technology is for static calls.

The choice of using **Java applets** has probably caused some readers of this report to ask themselves: why use applets if the goal is to make a graphical implementation with servlets? The answer is that the choice to implement an applet debugger was first based on the fact that once the user's browser has loaded the applet the user will remain in the same page during the remaining time of the debugging session. On the other hand, applets offer a wider gamma of elements to use than HTML, they are easier to implement, and the look and feel of the GUI can easier be changed and improved later on.

The requirement of using **Java servlets** is mainly for the possibility of integrating the debugger into Corus QL's framework. Besides, the users browsers only need to understand HTML and don't need to have a JRE (Java Runtime Environment) installed on his or her machine, which is a requisite in order to run applets. Furthermore, the probable need to make some static adaptations (for example of global variables, etc) in the servlets codes may be done at installation time of Corus QL.[12]

Javascript was supposed to be used, among other functions, to generate a linkable dynamic tree of objects. This object tree was supposed to improve source browsing.

3.5.3 User interface, control and interaction models

As an ambition, a state-of-the-art final graphical user interface using most of the adequate graphical elements available was the goal. Though, due to constraints of time for the project more modest GUIs were implemented.

The principles for *user interface design* recalled in [15] were taken in consideration when decisions were taken about the layout and what graphical components to use. For example in the servlet version, *user familiarity* was followed by trying to imitate the toolbar that some GUIs in most of IDEs provide. In the applet version, the use of pop-up windows (modal windows) was used for parameter reading, etc. The other principles that

[12] Corus QL which is used for system integration generates PL/SQL and Java code base on a general data model of the participating systems of the integration.

have been followed with different but acceptable grades of success were: *consistency* (operations are activated in same alternative ways), *user guidance* (help function available) *and user diversity* (support command line and buttons). Though, the principle of *minimal surprise* couldn't be followed due to the problems with synchronization between debugger and debuggee. The user has to know previous to the use of the debugger, when, why a session should hang, and how to make them running again.

Regarding the *user interaction model* used, two styles were applied: *form fill-in* (with the forms in the servlet, and in the popup windows in the applet) and *command language*. Both versions have the ability to parse and execute code according to a SQL client, in a similar way that it is done from a console client.

3.5.4 Gradually incremented GUIs.

The strategy chosen for implementing the graphical application was to use a gradual incrimination approach. Starting with a very simple Java application with just some commands, followed by an applet holding both connections (debugger and debuggee connections) was the next step, which if successful was accorded to probably be the last version. The final version should be a single servlet, if possible, holding both connections and at the same time offering an acceptable user interface as similar as possible to a state-of-the-art debugger. If necessary, which was understood at an early stage but accepted and adopted first when reviewing the project, a set of two or more servlets are acceptable.

A database connection obtained with JDBC corresponds or is equivalent to a database session. This equivalency implies that in the descriptions below these concepts are used as synonyms.

3.5.4.1 The test version

The first deployed program's purposes were:

- To demonstrate and test the "call-ability" of the CORUS_DEBUG API from a Java application
- To test if debugging of PL/SQL programs by using CORUS_DEBUG was possible to do from a Java program.

To demonstrate these two assertions it was decided to be enough to do a program that could create a database connection (having this way a database session) and that could call some procedures from the CORUS_DEBUG. With a set of two instances of this program opened, one corresponding to the *target session* and one to the *debug session,* a debugging session could then take place. One of these database connections or sessions (target session) should be able to make an initialization to be debugged, and the other (debug session) should be able to attach itself to the first one (target session). Furthermore, at least one program might be executable in the target session. And in the debug session reading the source code, doing single-steps should be possible to do and maybe even setting breakpoints. A command parser was not required for this test version.

This 'alpha' version confirmed the assertions above

3.5.4.2 The applet version

The purpose of this implementation was principally to test some different graphical interfaces and to implement methods to make the procedure calls necessary to debug a PL/SQL program using CORUS_DEBUG in the base. If possible, these methods should be transformed into classes, making it possible to achieve extensibility and modularity by writing new classes for new commands or functions for debugging. Even other types of non-debugging functionality could be added to the GUI.

The Swing API[13], was chosen for implementing the applet version, due to its properties over the almost deprecated AWT[14] API. Both are part of the Java Foundation Classes (JFC) -- the standard API for providing graphical user interfaces.

First, an applet should be done to have the same behavior as when debugging from SQLPlus[15] and – as contrary to the test version – should permit the usage of all the debugging functions implemented in CORUS_DEBUG. An applet following this requirement was deployed. This applet can run either a debugger or a debuggee session and the counter-session is run from a console using SQLPlus. This applet shows that the use of CORUS_DEBUG can be accessed and combined from different type of SQL clients, i.e. having the debuggee in the applet being debugged from a database session in SQLPlus.

Second, an applet with the capability of holding and handling both sessions from within the same graphical interface was implemented. This second implementation of a debugger-applet was supposed to hide the initialization, attaching, and synchronization steps between the debug and the target session. After this initialization procedure the user should be able to continue the debugging session as he prefers.

A command parser could also be done in this part of the implementation phase, but testing the interactivity with help of buttons, menus, lists, dialog windows, etc should be the main target when designing and implementing this type of debugger interface.

A great part of the time for this project was dedicated to do this second applet. The problem that emerged was that once the debuggee started to hang (which, of course was expected), there was no way to get to the debugger connection or, in other words, to get to the graphical elements that affect the debugger session (connection). Swing was used as the graphical package to implement the GUI. Different approaches were taken to tackle this problem:

- One single GUI that contained all the graphical elements (TextAreas, Labels and Buttons). This GUI didn't have any structure of subclasses or components containing other components. It was a simple and straight ahead approach.

[13] "Swing" was the code name of the project that developed new components for the Java Foundations Classes (JFC)
[14] AWT stands for *Abstract Window Toolkit.* The AWT is part of the Java Foundations Classes (JFC).
[15] SQLPLUS is an interface program to execute sql statements and pl/sql procedures.

- Another approach was to have two threaded Panels, each of them holding a connection. The main class started both panels' threads and they were expected to be run concurrently and independently of each other. That was not the case. The entire applet already hung when the debuggee was trying to execute debug_on. These panels contained a TextArea for output to the user, and a TextField and Buttons combined with popup or ModalWindows for user input.

- The next approach was implemented with four threaded objects. Two threads for the panels, and two for the database connections. The same behavior occurred.

- The last approach was to have each query execution in JDBC executed within a thread. Even this approach ended without success.

After several weeks deploying and testing the approaches listed above a set of three articles regarding Swing and Threads were discovered. This article series is available online in [20], [24] and [25]. Then, a few numbers of attempts were done.

- This time, the SwingWorker class described and exemplified in the articles just mentioned was used. Unfortunately even these versions without success.
  Because the project was running out of time it was decided not to make more attempts to solve this concurrency problem. Nevertheless, the idea that this problem can be solved still remains.

Two different applets that can be run from within the same HTML page or from two different HTML pages was the next step. This step was though skipped in order to continue with the remaining parts of the project. It was decided that if wished or necessary these new approaches could be done in a posterior project based on this one.

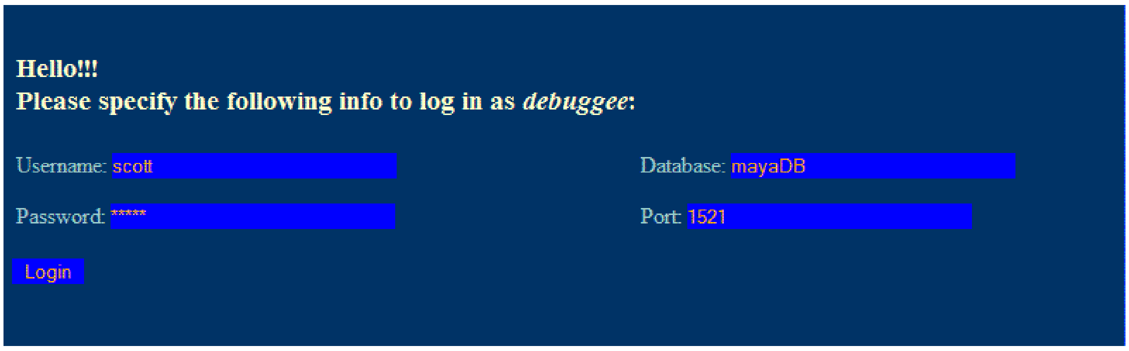
3.5.4.3 Servlet version

Recall that the main goal is to confirm and demonstrate whether the entire chain from DD through CD and with a front-end GUI implemented with servlets is possible or not. Most of the implementation of the servlet version for the GUI was done during the revision of the project. The technique called refactoring was applied as well as modularity by reusing and rewriting parts of the code used in the applet versions.

The same model and the same principles for designing user interfaces that were applied in the applet were applied here. The difference is the difficulty of having popup windows for getting the parameters for the functions that are to be executed. In addition it was decided that using a toolbar with descriptive icons for the different debugging functions was much better than having a lot of labeled buttons as it was done in the applets.

A major difference is that at this point of the project it was accepted to have two browser windows open for the debugging process; one for the debugger and one for the debuggee. It is accepted because running an application from a web browser has a lot of limitations and since CORUS_DEBUG is also used as a standalone console debugger, the target users are or will be used to have two different windows when debugging. In addition this measure is consequent with the principle of *user familiarity*.

Figure 3.4 shows TgtLoginServlet which permits giving as parameters *username, database name, port nummer used by the database for connections and password.*


Hello!!!
Please specify the following info to log in as *debuggee*:

Username: scott
Database: mayaDB
Password: *****
Port: 1521

Figure 3.4. Interface for the debuggee logging servlet TgtLogginServlet.

Figure 3.5 below shows the part of code that generates the output illustrated in Figure 3.6 and illustrates the easy accomplished manner for setting together commands**.** *commandLine* which is central in DebuggeeServlet is manipulated to let three other objects to actually execute the command *version, selfcheck* and *initialize.* The first of those command objects display directly their outputs via a *DbmsOutputToServlet* which is a modification of the example found in [9]. Finding a way in which the output of stored procedures could be accessed in a Java object took long time without success; fortunately, during the revision of the project the mentioned example given by Thomas Kyte, who is a guru in Oracle databases, was found. It permitted to directly access and retrieve the output results of the procedures in CD whenever wanted or necessary. The last command's result is displayed by *DebuggeeServlet.*

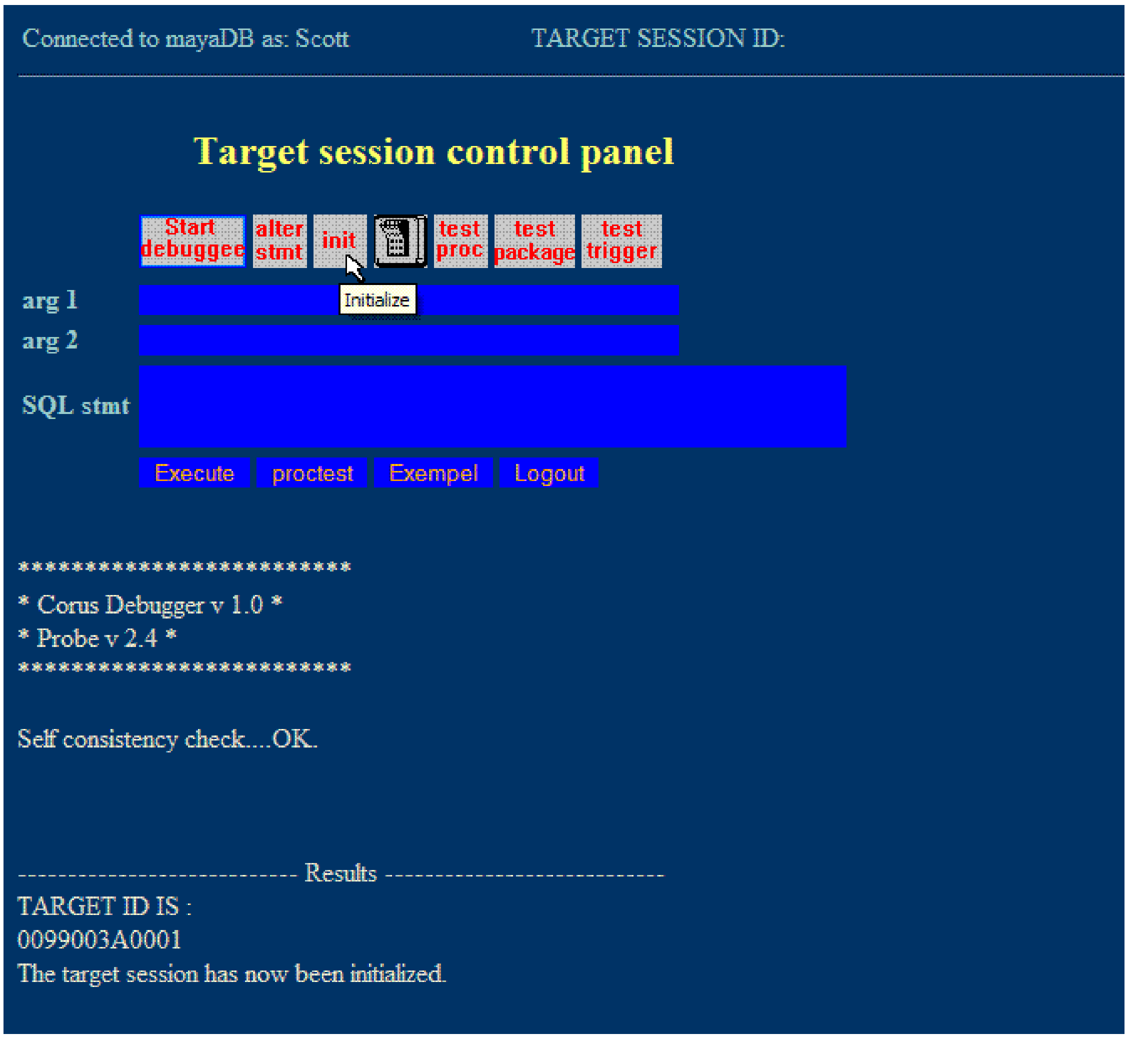
```
}else if(command.equalsIgnoreCase(CMD INIT)){
   commandLine = new String[1];
   commandLine[0] = "version";
   executeCommandAndDisplay(commandLine, out, connection);
   commandLine[0] = "selfcheck";
   executeCommandAndDisplay(commandLine, out, connection);
   commandLine[0] = command;
   executeCommand(commandLine, out, connection);

   // Save target session id in session object
   if(queryResultVector.elementAt(1)!= null){
      this.targetID = (String)queryResultVector.elementAt(1);
      session.setAttribute(ATTR_TGTSESSIONID, this.targetID);
   }
```

Figure 3.5. Code example for execution of *initialize*.

Figure 3.6 shows DebuggeeServlet after the user has logged into the system and has pressed the *init* button. The command *init* can be given by pressing the *init* button or by typing init in *SQL stmt* text area and then pressing the *Execute* button.

Figure 3.6 also depicts the interface provided by the servlet for the debuggee, called DebuggeeServlet. On the top is the output of TgtBannerServlet showing the name of the user and the database to which the user is connected. Below there are some icons for executing *alter session, alter [object], initialize, debug_on* and for three test programs. There is also a button for executing *alter session, initialize* and *debug_on,* all at once. The icons are kept to demonstrate CORUS_DEBUG; in a release version *alter session* and *initialize* will be done automatically and transparently when logging in.

Two strategies have been implemented for parameter reading. One strategy is to have a number of textfields. Each textfield maps to one argument of the debugging functions.
The second strategy is to have a textarea in which commands can be written just in the same way as in a console. A parser takes care of these parameters and creates a *commandLine array*. This command line array is a String array which is passed to command objects, each of them handling one debugging function. Each new function to be added to this interface has to have the ability of receiving its parameters in a String array. This model for passing parameters to objects that will execute some query on the database was chosen because handling an array is a pretty straight forward way to pass indexed parameters. In addition if some additional parsing is to be done, it is easier to have an array with String objects to do that. Other alternatives as passing tuples of type=value has been considerated and are a good option. This alternative is noted for future development where knowing the types of the parameters is crucial.



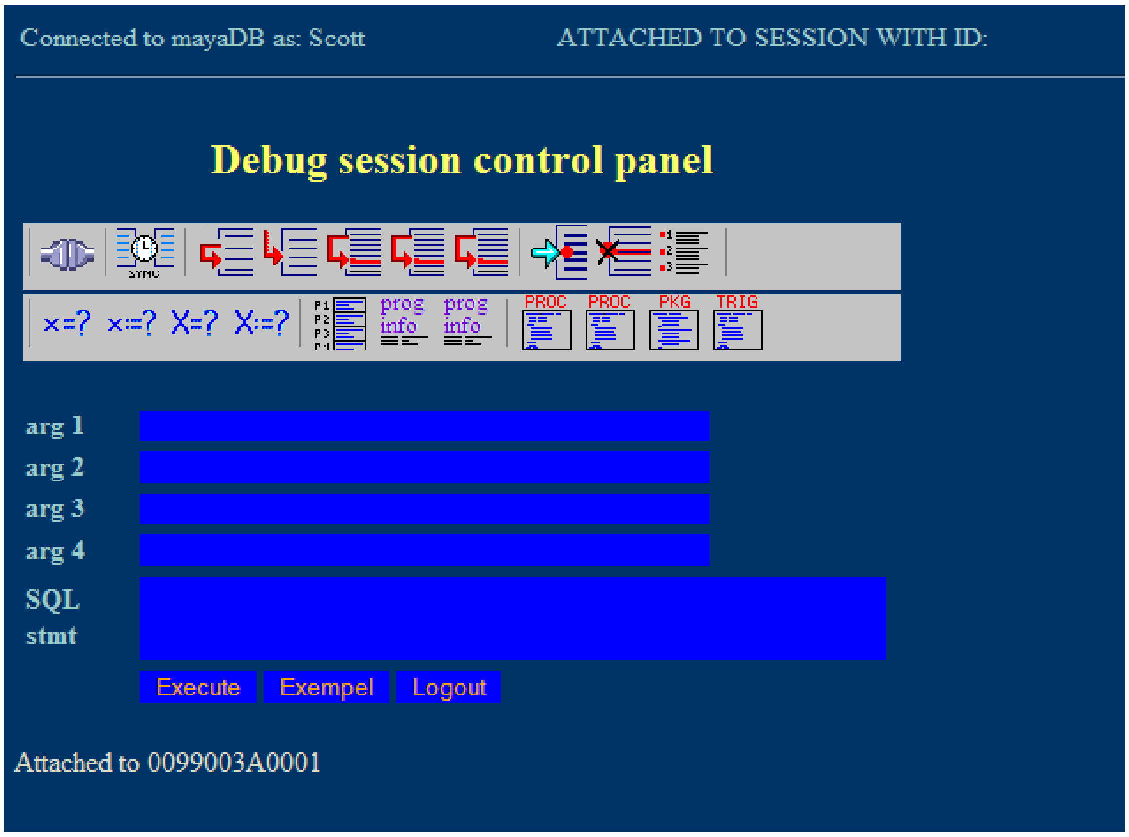
Connected to mayaDB as: Scott TARGET SESSION ID:

Target session control panel

```
*************************
* Corus Debugger v 1.0 *
* Probe v 2.4 *
*************************

Self consistency check....OK.

---------------------------- Results ----------------------------
TARGET ID IS :
0099003A0001
The target session has now been initialized.
```

Figure 3.6. Output of *DebuggeeServlet* after executing *initialize.*

The result of *initialize* is the *target session id* which in the example illustrated in Figure 3.6 has the value of `0099003A0001` and it will be used in the *DebuggerServlet (debugger)* in order to attach it to the *DebuggeeServlet (debuggee).* Figure 3.7 shows the result of *DebuggerServlet* after is has been attached to the debuggee. This servlet has two rows with icons for each command and four textfields for the parameters. It also has a text area for typing the commands just like in a console. The way in which DebuggerServlet and DebuggeeServlet are designed is similar. The difference is the composed commands inside them (without knowledge for the user) and the number of commands that can be executed in each of them.


Connected to mayaDB as: Scott ATTACHED TO SESSION WITH ID:

Debug session control panel

arg 1
arg 2
arg 3
arg 4
SQL stmt

Attached to 0099003A0001

Figure 3.7. Output of *DebuggerServlet* after attaching to debuggee in Figure 3.6.

In a similar way the results of the other commands are either directly displayed by the command objects or by the servlet.

3.5.5 Environment

For the tests, deployment and evaluation in this project the following software has been used: an Oracle Database Server v.8.1.6, some other database administrating and programming tools from Oracle as SQLPlus and DBA Assistant, an Apache Web Server, the Java Development Kit (jdk) v.1.2.2 with Java Servlet Development Kit (jsdk) v 2.0 in the first version. The test and trial versions of the different debugging tools that were used in the market survey are the ones in the accompanying CD to [13].

During the project's revision the main used tools and programs are an Oracle Server 10*g*, the Java Development Kit Standard Edition (j2se) v.1.4.2_04 and the web server used is the built-in server in the Oracle Server. A Mozilla web browser v 1.7 was used for testing the functionality and layout during the revision.

3.6 System Architecture

After presenting the details of the web based debugger at a lower level, it remains to present a general description of the system, which models that have been adapted for designing and implementing system control, user interaction, user interfaces, user support, model pattern for achieving modularity and extensibility and others.

It is suitable to start by listing some use cases which will be a complement to the requirements listed in Chapter 3.3**.**

3.6.1 Use cases

The use of use cases is adequate to develop an understanding of the dynamic relationship between a system and its external environment as exposed in [15]. Use cases help to achieve understanding of the system usage and this is also the case in this project.

Table 5 lists five use cases. The use cases A-C are fundamental and sufficient for this project, they permit to determine whether the project's main goal is attainable or not. Use case E is used to demonstrate the availability of CD and the web based debugger.

Table 5. Table of use cases.

Case	**Program unit**	**Description**
A	Procedure	A standalone procedure or function.
B	Package	A package with global and local variables.
C	Trigger	A trigger.
D	Composed	Several program units that are called from each other.
E	Database linked	Debugging of a program unit through a database link.

These use cases was also used for the verification phase of the system.

3.6.2 System structure

The web based debugger is a client-server application having DD at the bottom of the server side. On the client side there are three alternatives provided: dynamic HTML pages (generated with servlets, or another technologies such as JSP and Java Beans); a Java applet and a console client that is enable to execute stored procedures and SQL statements. In between there is CD that is a wrapping component around DD. This package comprises the interface towards the different clients. The clients are composed of a set of Java servlets, or a set of Java objects for the applet, or a SQL enabled client that execute a subset of CD which builds a standalone console client. This system structure, of which the OSI model[16] is a good example, is better described with a model that Sommervile call *the abstract machine model* but could also be called *layered model* in [15].

Figure 3.8 illustrates the structure that now has been described. In the middle of the model is the database that has the debug layer, *Probe*.

[16] The *OSI model* (Open Systems Interconnect) model describes seven defined layers in a network operating system. It was created by the International Standards Organization (ISO)

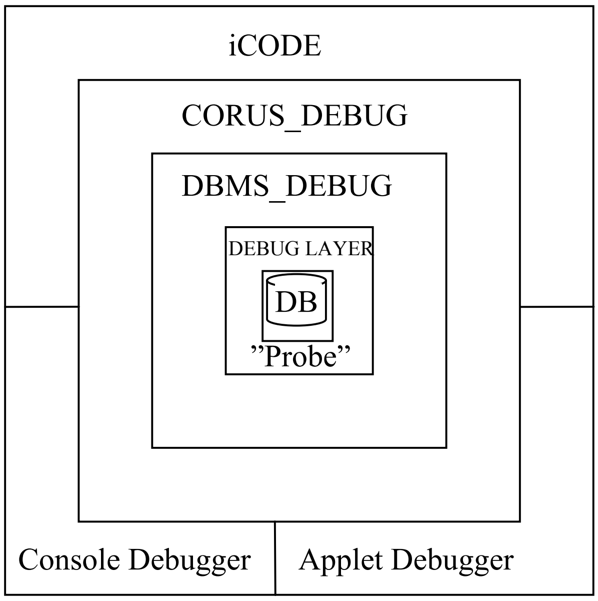


Figure 3.8 System structure, applying the *layered model*

How the system is controlled is given by the *call-return model* for many reasons. The whole system is sequential; the control starts by a subroutine call at the top of a subroutine hierarchy (by a command line, or a button press), through subroutine calls in the lower layers, all the way to the lowest layer where data and Probe reside.

The entire system is composed of, or can be seen as divided into modules. Each of these modules represents a sort of a "white-box" in which we expect a certain output as response to a determinate input. An example illustrating the control of the system is given in Figure 3.9 below.

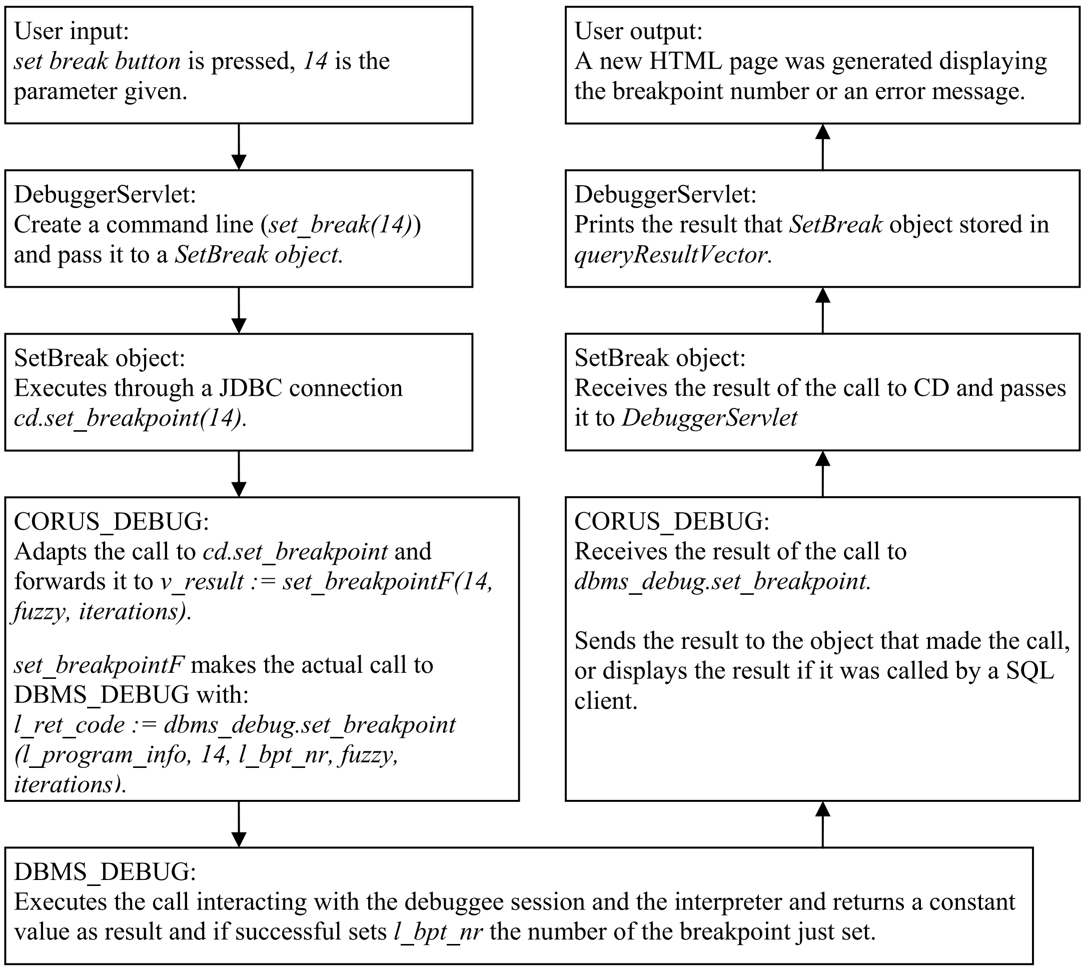

User input:
set break button is pressed, *14* is the parameter given.
DebuggerServlet:
Create a command line (*set_break(14)*) and pass it to a *SetBreak object.*
SetBreak object:
Executes through a JDBC connection *cd.set_breakpoint(14).*
CORUS_DEBUG:
Adapts the call to *cd.set_breakpoint* and forwards it to *v_result := set_breakpointF(14, fuzzy, iterations).*
set_breakpointF makes the actual call to DBMS_DEBUG with:
l_ret_code := dbms_debug.set_breakpoint (l_program_info, 14, l_bpt_nr, fuzzy, iterations).
DBMS_DEBUG:
Executes the call interacting with the debuggee session and the interpreter and returns a constant value as result and if successful sets *l_bpt_nr* the number of the breakpoint just set.
CORUS_DEBUG:
Receives the result of the call to *dbms_debug.set_breakpoint.*
Sends the result to the object that made the call, or displays the result if it was called by a SQL client.
SetBreak object:
Receives the result of the call to CD and passes it to *DebuggerServlet*
DebuggerServlet:
Prints the result that *SetBreak* object stored in *queryResultVector.*
User output:
A new HTML page was generated displaying the breakpoint number or an error message.

Figure 3.9. Example of system control and program execution at the different layers by a servlet client.

Figure 3.9 also illustrates the data-flow of the system when a command is executed from a HTML page via a Java servlet and other Java objects all the way to DBMS_DEBUG. The input in this example can also be created in an applet and the data-flow is almost the same.

Figure 3.10 shows the complete system with the three types of clients. Type 1 is a console client that utilizes CD as a command-line debugger. Type 2 is an applet. The latter client downloads the applet from a web browser at start, and then it opens two connections to the database and proceeds with debugging from the web browser directly to the database. Type 3 is conformed by servlet generated HTML pages. The user sends a request to a web server and the URL that is sent corresponds to a servlet. The servlet then makes the connection to the database and sends back an HTML page to the user. The debugging process follows the same request-response paradigm. The difference against type 2 is that the debugging process is actually done from within the web server, where the database connection objects reside.

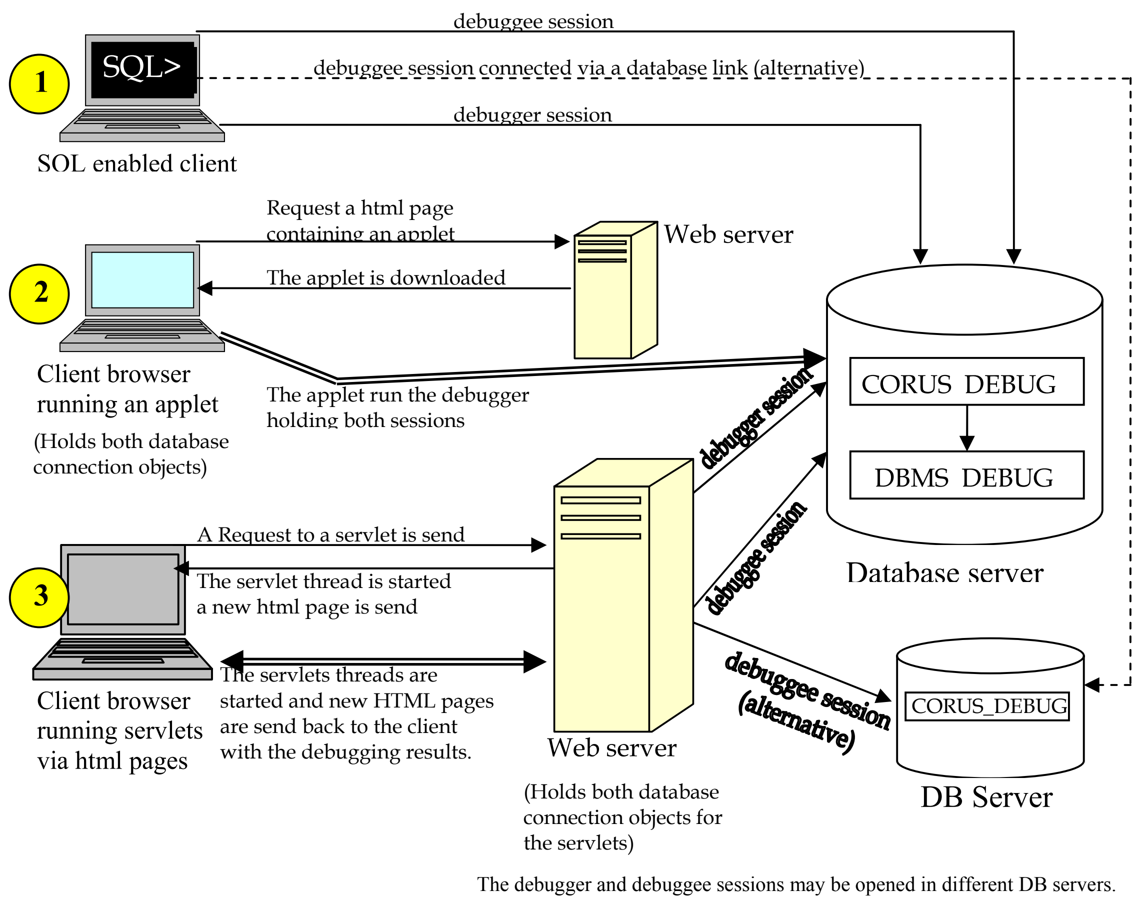

debuggee session
SQL>
1
debuggee session connected via a database link (alternative)
debugger session
SQL enabled client
Request a html page containing an applet
Web server
The applet is downloaded
2
Client browser running an applet
(Holds both database connection objects)
The applet run the debugger holding both sessions
CORUS DEBUG
DBMS DEBUG
debugger session
debuggee session
Database server
A Request to a servlet is send
3
The servlet thread is started a new html page is send
The servlets threads are started and new HTML pages are send back to the client with the debugging results.
Client browser running servlets via html pages
Web server
(Holds both database connection objects for the servlets)
debuggee session (alternative)
CORUS_DEBUG
DB Server
The debugger and debuggee sessions may be opened in different DB servers.

Figure 3.10. Entire system with the three types of clients accessing CORUS_DEBUG.

3.6.3 Class diagrams

Figure 3.11 shows a general class diagram for iCODE. The `Command` class is an abstract class that has two abstract methods to be implemented: *performCommand* and *performCommandAndDisplay*. By implementing these methods new subclasses can easily add functionality to iCODE either by using CORUS_DEBUG for calls to stored procedures or executing independent SQL queries without using CORUS_DEBUG. *TgtBannerServlet* and *DbgBannerServlet* classes include information about the user and the database that he or she is connected to. These objects can also be used to implement filtering if necessary. Instances of the *DbmsOutputToServlet* class are instantiated by all subclasses to `Command` class and by `DebuggerServlet` and `DebuggeeServlet`. It is necessary to have access to the database output generated by stored procedures that use the `DBMS_OUTPUT`. This is the only way to display output from the database server. Most of the presentation logic of the application is located and implemented in `DebuggerServlet` as well as some of the business logic.

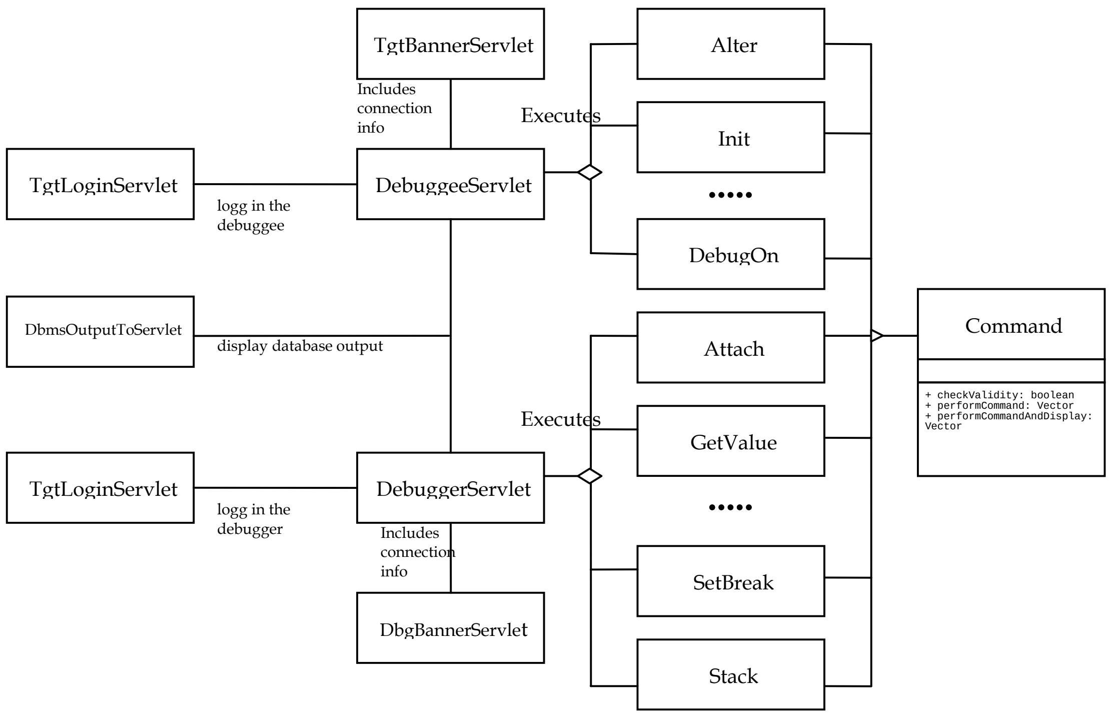


Figure 3.11 Class diagram for the web based debugger using Java servlets.

Documentation about the classes in the web application implemented with servlets is found in Appendix C in the well-known javadoc style.

3.6.4 Modularity and functional extensibility

One of the requirements was that the servlet should permit modularity and extensibility in functionality. As seen in Figure 3.11, the strategy or implementing a Command model where each sub class to the *Command* class takes independently care of a function, permits to easily extend or change the functionality of the web application. For example, to add the functionality of browsing the source code of procedures or functions *cd.show_source_procfcn* can be executed. The necessary steps to introduce new functionality into the web application are described below.

A new class that extends the *Command* class must be implemented. This class should completely implement at least one but preferably both of the methods *performCommandAndDisplay* and *performCommand* depending on the function of the command. To introduce a new command at least steps 1-3 in Figure 3.12 must be followed. Step 4 is optional for inserting an icon that executes the command. It is also optional but advisable to add a constant variable in *iCODEdefinitions interface* to be directly used in the servlet and other classes, and to avoid mistyping. It also may facilitate a way to hold track of all new commands. In the example below, `CMD_SOURCE_PROC` has been inserted in *iCODEdefinitions*.

Part of the source code for method *performCommandAndDisplay* in *SourceProc* class:

```
performCommandAndDisplay(String[] cmdArrayin, java.sql.Connection
connection, java.io.PrintWriter out) {
   String query = "{call dbg.show_source_procfcn(?)}";
   String procName = cmdArray[1].toUpperCase();
   ………………
   if(checkValidity(cmdArray)){
      try{
          deb.DbmsOuputToServlet dout = new
deb.DbmsOuputToServlet(connection);
          dout.enable(1000);
          cstmt = connection.prepareCall(query);
          cstmt.setString(1, procName);
          cstmt.execute ();
          cstmt.execute();
          dout.show(out);
          dout.disable();
          dout.close();
      }catch (SQLException e) {
          vector.addElement("SQLException in " + className");
          vector.addElement(e.getMessage());
      }finally {
          if (cstmt != null) try{ cstmt.close();}
             catch(SQLException ignore) {}
          }
      }
```

1

The method has `performCommand` also to be implemented

In DebuggerServlet three methods are modified (total: 7 lines).

In method *initcommandTable*

```
      commandTable.put(CMD_SOURCE_PROC,   new SourceProc());
```

2

part of the code inserted in *doRealProcessRequest*

```
     if(command.equalsIgnoreCase(CMD_SOURCE_PROC)){
        executeCommandAndDisplay(commandLine, out, connection);
        displayResultsOnScreen = false;
     }
```

3

Edited code inserted in *printControlForm*. This part is optional if an icon is wanted in the toolbar.

4

```
 out.println("<input name=\"command\" type=\"image\"
value=\""+CMD_SOURCE_PROC+"\"  title=\"Show Current Source code\"");
 out.println("src=\"../icode/images/sourceproc.gif\" alt=\"[Show
PROCEDUREs Source code]\" width=\"32\" height=\"32\" border=\"0\" >");
```

Figure 3.12 Piece of code necessary to add a new function to the servlet debugger.

3.6.5 GUI design

Regarding the technologies chosen to implement the GUIs for the clients in this project, the reasons for having chosen Java applets and Java servlets are explained in Chapter 3.5.2. The parameters taken into consideration when deciding whether a GUI in command line fashion, or a GUI with buttons, menus and forms should be chosen, are already described in Chapter 3.5.4. An incremental implementation and evolution of the GUI is also described in Chapter 3.5.4.

3.6.6 Security and performance

Neither security nor performance was an issue in this project. One of Corus QL's components is a web application called NAVAJO. This application is designed and designated to handle data access and granted roles and properties. Therefore the aspect of security in the scope of this project has been limited to the user granted roles.

3.7 Industry version of CORUS_DEBUG

The execution of this Master Thesis project provided Corus Technologies with a lot of knowledge about how DBMS_DEBUG can be used and how a web based debugger can be built. This knowledge, the difficulties encountered and alternative solutions, the application and control models, the first version of CORUS_DEBUG, the different approaches of the applet debugger and mode, were noted and documented in the first draft of this thesis report.

Based on these notes, the report's first draft and a study of how a PL/SQL debugger can be built in the core of Corus QL, a completely new version of CORUS_DEBUG was implemented in 2001. This version, which properly can be called *industry version*, was integrated into Corus QL. This integration, as expected, implied the creation of new database objects in order to interact and cooperate with the rest of a Corus generated environment. A complete user manual with details of how to use the debugger within a Corus generated system was written, as well as an online version.

There are differences between the first version and the industry version. Most of the dependencies between procedures were removed. Source parsing, abstraction to the user and error report were improved. The help package for handling pending breakpoints was removed, because parsing was improved. Alias procedures for most of the procedures were added to simplify the use of the package. Database objects were created. Finally, a new function was added to one of Corus QL's tools in order to use the debugger, since the *alter statement* changes the mode of the current session, which is an important detail.

4 Analysis

Once the design, implementation and system architecture has been exposed an evaluation of CORUS_DEBUG and especially of the web based debugger must be presented. Even though testing has been continually done during the different phases of the project, a final evaluation has to be carried out.

4.1 Use Cases

The use cases presented in Chapter 3.6.1 have been shortened to just four cases since database links have not been covered in the implementation in the final version of CORUS_DEBUG.

Table 6. Test use cases focused in the evaluation phase

Case	**Program unit**	**Description**
A	Procedure	A standalone procedure or function.
B	Package	A package with global and local variables.
C	Trigger	A trigger.
D	Composed	Collaborating program units.

4.2 Evaluation model

The technique used in the verification and validation (V&V) phase was *software testing. Use case based testing* is the appropriate testing model for this project because the target users of the applications of this project are very specific, namely developers using Corus QL, and that the known target program units are procedures, packages and triggers written in PL/SQL.

Defect testing was applied in order to carry out a *component evaluation* and *integration evaluation*. The objective was to see whether the goals of this project were achieved or not and to get a qualitative measuring of how well the requirements listed in Chapter 3.3 were fulfilled. The deployed web based debugger is not fully complete and has not been integrated into Corus QL's framework, therefore necessary *acceptance tests* has been left until a later phase in the project.

4.3 Component evaluation

The final application is the result of collaborating components of which the basic and most important is CORUS_DEBUG. The other results are the AppletDebugger and DebuggerServlet.

The first type of test to carry out is *structural defect testing*. Therefore, each procedure in CORUS_DEBUG has been tested with satisfactory results in the use cases. The test programs used were simple programs that easily permit the demonstration of control execution and the access and manipulation of program context information. The obtained results have resulted in changes in CORUS_DEBUG and new iterations of the testing phase. The final test results have been noted for further development and improvement.

Unfortunately, the final tests for the AppletDebugger could not be run since the applet use the first version of CORUS_DEBUG done in spring 2000. Nevertheless the results noted from that date show that the applet hung with no manner for the user to interrupt this state or retake program control when both the debugger and debuggee sessions were handled by the applet. On the other hand if the debuggee was run in a console client, the applet was able to control the target session and to show the results for the user. The reason for why the applet hangs and the user loose control of the debugging is apparently some issues that Swing has handling threads. Despite different approaches to solve the concurrency problem handling the debugger and debuggee sessions, and the information found in [20], [24] and [25], this problem could not be solved within the stipulated timeframe.

iCODE, which is the name for the application composed of a set of servlets of which DebuggerServlet and DebuggeeServlet are the central ones, has also been tested with the same use cases listed in Chapter 4.1. The results were also satisfactory with this type of client. The behavior of the iCODE differs from the way CORUS_DEBUG behaves when used directly from a console. The main differences are that the output is not as clear as in an SQL environment and that the feeling of having a web window that hangs is not as natural as in a console window. When using a graphical interface with a window there is always a feeling that something is wrong if a window just hangs, but it is assumed that the users will get used to this behavior.

4.4 Integration test

How the AppletDebugger and iCODE work together with CORUS_DEBUG has also been tested. The *integration testing* was down according to the *top-down model.*

Since the final GUI has been gradually incremented it wasn't necessary to run special tests for the integration testing phase. The first GUI version from a simple Java application already showed the connectivity between a Java application and CORUS_DEBUG.

The results were negative with the AppletDebugger due to problems handling threads that are caused by the implementation in the javax.swing package. With the servlets compounding iCODE the results were, on the other hand, satisfactory and positive. What have been tested later on were the different ways of interaction between iCODE and CORUS_DEBUG. Accessing directly the output generated by DBMS_OUTPUT package, retrieving single results in *ResultSets* and retrieving complex results from *VARRAYs* were also tested.

4.5 Results of evaluation

A good overview of the results for the final web based debugger is provided in Table 7. The results are organized according to the requirements of the application and the features/functionality desired.

Table 7. Results of the evaluation

Feature/function	Result description
Initialization	Achieved.
Smooth termination	Ambiguous. Not completely clear but realizable.
Synchronization	Achieved. Necessary even to get out of the interpreter.
Control execution	Achieved, may introduce some ambiguity due to different behavior compared to other debuggers.
Breakpoints	Achieved. Setting and managing of breakpoints at specified valid line achieved. Breakpoint setting with program unit name as parameter left for future work; parsing improvement necessary.
Source browsing	Achieved.
Local variable accessing	Achieved. Local variables can be accessed and set. Though :new and :old cannot be accessed.
Package variable accessing	Achieved. Package variables can be accessed and set.
Program context	Achieved. Runtime, stack info accessible.
Easy of use	Achieved. Applied user familiarity, fill-in form and command line control.
User support	Built in help functionality was implemented. Error messages for procedure/function failure without further instructions.
Components connectivity	Achieved. CORUS_DEBUG can easily be called and used by Java applications.
Extensibility	Achieved. Easy extension of functionality.

4.5.1 Discussion

In the tests no response-time or resource allocation measuring has been done. The type of performance tests realized in [16] - and that might be expected when testing a debugger - were not done in this project. Performance tests have not been run because performance is neither an issue nor are needed or applicable in the scope of this project. Response-time and memory allocation are not important when debugging the type of programs that will be debugged with iCODE or CORUS_DEBUG. Acceptance tests have been left to a later phase in the development of iCODE.

The results of the evaluation tests show that most of the goals have been achieved. They also show that improvements might still be necessary and that iCODE even though it can be used for debugging at the current stage, should more appropriately be seen as a prototype.

5 Discussion

The decisions made during the execution of this project and the development resulting web application must be discussed and exposed as well as the choices done in the design and implementation.

Overloading most of the procedures in CORUS_DEBUG was necessary in order to implement call-ability and usability of the package from both types of target clients: a console client like SQLPlus and Java applications. This property could be implemented in separate packages, one for each type of client, or having a standard way to handle input and output to and from the procedures. For instance, using the `IN` and `IN OUT` properties of the parameters, but it could also cause the implementation to be more complex and tedious, it would also affect the usage of the package from a console client.

A number of similarities and a major difference were noted between the functionality provided by iCODE and CORUS_DEBUG, and the commercial debuggers used in the market survey. The commercial debuggers are integrated in state-of-the-art IDEs providing a lot more of functionality via their GUIs than iCODE does. But, on the other hand, iCODE provides easy extensibility of functionality. Starting the debugging process is easier and safer with iCODE than with some of the other debuggers; iCODE doesn't hang in a way that the user is forced to close the web browser or kill some process at OS level. Killing processes at OS level was shown to be necessary with some other debuggers. The major difference, which was one of the requirements, is the ability to access and manipulate package variables which was not implemented by all the other debuggers.

The conviction that all the advantages of using a graphical interface package like Swing's (javax.swing) could compensate all the time and energy invested will pay off at the end, caused that the servlet application didn't have enough resting time for its implementation. Reusing the strategy of having separate classes for each command call, saved though some time, demonstrating also that easy extensibility was achieved.

The final web based debugger run by cooperating servlets and command classes, has a graphical interface with limited functionality and poor layout. It is due to limitations in HTML. But the layout can be enhanced greatly with Javascripts or other techniques for dynamic HTML pages. There was no time for making these enhancements in the layout.

A GUI design providing a layout as similar as possible to state-of-the-art graphical interfaces, and at the same time providing a command-line control may not appear to be necessary at first sight. It was implemented anyway for holding the user familiarity property and to be consequent with the use of CORUS_DEBUG from a console client.

User support, which is a fundamental feature for any type of application, according to the author of this project, was achieved and implemented in a similar manner as Java's command-line debugger `jdb`. Help functionality was implemented even though it wasn't really required from the beginning.

6 Summary and Conclusion

In this Chapter the difficulties, results and conclusions obtained during the development of this Master Thesis are presented in short.

The verification of the thesis - as stated as the main goal of this Master Thesis project- "the entire chain is realizable, from the database using a tailor-made debug API on top of DBMS_DEBUG, via servlets and eventually JavaScript scripts, to the graphical representation (in Chapter 1.3)" makes the main objective of this Master Thesis achieved.

The market survey which was the first phase of the project resulted in ideas and insights that later on, during the development phase, were proved and experienced. They showed that the implementation of a debugger for PL/SQL would be a challenge.

During the study of DBMS_DEBUG, which was very time and patience demanding, an insight was gained: the main goal and part goals of this project needed much more time and even more human resources than the available for this Master Thesis. Therefore, due to the actual size and complexity of the project some limitations in the required debugger features were done. Based on the studied material, the market survey and the study of DBMS_DEBUG a list of requirements was produced for the PL/SQL package and for the web application.

A PL/SQL package was implemented on top of DBMS_DEBUG, called CORUS_DEBUG. This package hides the details concerning the variable and data types necessary for a debugging session, holding the state and program context information and controlling the debuggee program. The use of DBMS_DEBUG, which is at the very bottom layer of this application is facilitated by CORUS_DEBUG; resulting in a command-line debugger. It also provides functions and procedures to be called by a Java application.

A web based application was finally implemented using Java servlets. This web application provides a user interface for using CORUS_DEBUG. It's has limited functionality at its current state, but is designed in such manner that new functionality can easily be added or modified.

A GUI can be provided with Java servlets, but in that case it needs enhancements, for example with Javascript scripts to make a more state-of-the-art user interface. Despite, the failed attempts of implementing a GUI with applets the idea that it is a better alternative than servlets from the graphic point of view still remains.

API documentation of selected classes of the debugger is provided in Appendix C.

The complete documentation and the source code of the debugger can be requested from the Corus Technologies or from the author.

Positive results were obtained in the evaluation phase of the project in which simple programs were used for component and integration testing and to illustrate the different features provided by the web based application and by CORUS_DEBUG. Defect testing was performed in the evaluation tests.

Based on the difficulties, results and conclusions summarized above, the following final conclusion is presented:

A web based debugger for PL/SQL programs based on Oracle's debug API (DBMS_DEBUG) is realizable with help of a wrapping package (CORUS_DEBUG) which on one hand provides transparency from DBMS_DEBUG and on the other hand provide full functionality to be used as a command-line debugger and that can also be used as a debug API for java applications.

6.1 Limitations

The major limitation for this thesis project is the relative short amount of available time. A project of this magnitude requires much more time and with preference more human resources in order to produce a fancy professional state-of-the-art web based debugger.

Some limitations were done during the implementation phase of this project due to the limited time for the project when it was carried out at Corus. Prioritizations in favor of the most common features in debugger were done. Other limitations were done due to lack of implementation in DBMS_DEBUG, as for example: access to`:OLD`, `:NEW` in triggers, watchpoints functionality and access to bind variables. It has to be mentioned and stressed one more time that the major limitation is the poor amount of available information about DBMS_DEBUG.

Among the limitations due lack of time assigned for this thesis project, the following can be mentioned: implementation of watchpoints, implementation of breakpoints set just by program unit names, a better error messaging, and the most important was the integration of iCODE into Corus QL's framework. The latter limitation requires a deep study of Corus QL and especially of some of its tools, in order to integrate iCODE in the core of it. This study may also take some considerable time.

The size and complexity in designing and building an application as the one in this project; the relative short time for this thesis project; and finally, the poor information available about DBMS_DEBUG, are the cause for some of the limitations introduced in this project.

Another limitation that has to be mentioned here is the difficulty that the Swing package has with threads. There is, though, the possibility that the problem in this case is not Swing but JWM working together with JDBC and hanging database sessions, as in the case of the debuggee.

7 Further development

Design and implementation of a debugger requires a lot of resources. The complexity of this task was experienced in this project. Porting over the functionality of a debugger to a web application requires at an early stage decisions made regarding the architecture model and technologies to utilize.

The web based debugger implemented in this Master Thesis can be used in different ways in its current state. It can be used as it is, because despite its limited functionality and simple GUI, it provides the necessary and enough functionality to debug PL/SQL programs. It can be also used as a prototype together with the conclusions obtained and the difficulties noted in order to build a new web based debugger in, with preference, a development team with an application architect, and a couple of implementers and a tester.

It may also be improved. Here follow some ideas for the next steps for improvement of iCODE:

A source parser that returns the first execution line in the *executable block* of a PL/SQL program unit is necessary. It should handle the special cases of triggers and nested procedure inside PL/SQL packages. This parser makes possibly to set breakpoint by giving the name of a program unit as parameter. For instance for a procedure called *break_at.*

CORUS_DEBUG should be reviewed and tested to implement the access of program units via database links. Working with different user schemas in the database may also be improved. Handling user-defined exceptions and common exceptions for setting breakpoints and for execution control are in the list of desired features.

In order to implement watchpoints, pooling of the context program information and of the program units listed in the program stack can be used.

A new model or pattern for concurrent execution that especially handles asynchronous messaging when using the Swing package could enhance the GUI by having an applet instead of generated HTML pages. In addition this applet can be adapted to be also used as a Java application, implementing the debugger interface in this way gives two applications instead of only one. The look and feel would also be easier to improve and change.

Finally the ability to browse the code with clickable lines that permit to set and delete breakpoints will be a great improvement for the GUI.

8 Acronyms and Abbreviations

ALS	Application Linking System
AWT	Abstract Window Toolkit
BPT	Short name for BPT_HANDLER, package for deferred breakpoints.
CD	CORUS_DEBUG
CGI	Common Gateway Interface
CPU	Central Process Unit
DBMS	Database Management System
DBA	Database Administration
DCL	Data Control Language
DD	DBMS_DEBUG
DDL	Data Description Language
DML	Data Manipulation Language
EAI	Enterprise Application Integration
EJB	Enterprise Java Beans
IDE	Integrated Development Environment
IEI	Inter Enterprise Integration
JDBC	Unofficial acronym for Java Database Connectivity. It is a trademarked name.
JFC	Java Foundation Classes
JSP	Java Server Pages
GUI	Graphical User Interface
HTTP	Hypertext Transfer Protocol
HTTPS	Secure Hypertext Transfer Protocol
iCODE	iCorus Online Debugger
ISO	International Organization for Standardization
QL	QuickLink, as in Corus QuickLink.
OS	Operating System
OSI	Open Standards Interconnect
RDBMS	Relational Database Management System
SQL	Structured Query Language
TCP	Transport Control Protocol
UML	Unified Modeling Language
V&V	Validation and verification

9 References

[1] Bales, Donald. *Java Programming with Oracle JDBC.* ISBN 0-596-00088-X, O'Reilly & Associates, Inc, 2002.

[2] Cyran, Michele. *Supplied Packages Reference. Release 8.1.5,Part No. A68001-01 Chapter 7.DBMS_DEBUG..* Oracle Corporation, February 1999.

[3] Date, C. J. *An Introduction to Database Systems, 6th Edition.* ISBN 0-201-54329-X, Addison-Wesley Publishing Company, Inc, 1995.

[4] Feuerstein, Steven & Pribyl, Bill. *Oracle PL/SQL Programming, Second Edition*. ISBN 1-56592-335-9E, O'Reilly & Associates, Inc.,1997.

[5] Feuerstein, Steven. *Advanced Oracle PL/SQL Programming with Packages.* ISBN 1-56592-238-7E, O'Reilly & Associates, Inc., 2000.

[6] Graham, Hamilton et al. *JDBC Database Access with Java: A Tutorial and Annotated Reference.* ISBN 0201309955, Addison Wesley Publishing Company, 1997.

[7] Harold, Elliotte Rusty. *Java Network Programming, 2nd Edition.* ISBN 1565928709, O'Reilly & Associates, Inc, 2000.

[8] Kamkar M. AADEBUG'97. Third International Workshop on Automatic Debugging. Linköping, Sweden, UniTryck, 1997, pp.103-122.

[9] Kyte, Thomas, *Expert One-On-One Oracle*, USA, Apress, 2003

[10] Moss, Karl. *Java Servlets, Second Edition.* ISBN 0-07-135188-4, The McGraw-Hill Companies, Inc., 1999.

[11] Rosenberg, J. B. *How debuggers work?. Algorithms, Data Structures and Architecture.* ISBN 0-471-14966-7, John Wiley & Sons, Inc. 1996.

[12] Subrahmanyam, Allamaraju et al. *Professional Java Server Programming, 1.3 Edition.* ISBN 1-861005-37-7, Wrox Press Ltd, 2001.

[13] Urman, Scott. *Oracle 8i Advanced PL/SQL Programming.* ISBN 0-07-212146-7, The McGraw-Hill Companies, Inc., 2000.

[14] Telles M. & Hsieh Y. *The Science of debugging.* ISBN 1-57610-917-8, The Coriolis Group, 2001.

[15] Sommerville I. *Software Engineering, 6th edition.* ISBN 0-201-39815-X, Addison-Wesley Publishing Company, Inc, Imprint of Pearson Education Limited, 2001.

[16] Säteri, Mattias. Design and performance analysis of a system level debugger for a real time operating system. ISSN 1401-5757, Uppsala University School of Engineering, 1999.

[17] Zellweger, Polle T. *Interactive source-level debugging.* Ph.D. Diss, Xevox Parc Palo Alto Research Center, Technical Report CSL-84-5, 1984

Web Resources

[18] American Institute of Physics & Cassidy, David. *Heisenberg - Quantum Mechanics, 1925-1927: The Uncertainty Principle*, http://www.aip.org/history/heisenberg/p08.htm, last visited on 2005-03-29.

[19] Bodoff, Stephanie. *Java Servlet Technology,* http://java.sun.com/webservices/docs/1.0/tutorial/doc/Servlets.html

[20] Bowbeer, Joseph. *The Last Word in Swing Threads*, article, http://java.sun.com/products/jfc/tsc/articles/threads/threads3.html. Retrieved, april 2005.

[21] Brevard's users group, BUG Club. *What is Debug? Debuggers MS-DOS Assembly Source Break Points what is a debug?*, http://bugclub.org/beginners/dos/debug.html

[22] Digital Equipment Corporation
Maynard, Massachusetts. *Digital UNIX, Ladebug Debugger Manual*, http://discovery.cc.vt.edu/dec/APZ7EETE/TITLE.HTM

[23] iSYSTEM AG, Munich, Germany. *In-circuit emulator and debugging tools for microprocessors and microcontrollers,* http://www.isystem.com/in-circuit-emulator, last visited 2005.

[24] Muller, Hans & Walrath, Kathy. *Threads and Swing*, article, http://java.sun.com/products/jfc/tsc/articles/threads/threads1.html. Last retrieved, april 2005.

[25] Muller, Hans & Walrath, Kathy. *Using a Swing Worker Thread*, article, http://java.sun.com/products/jfc/tsc/articles/threads/threads2.html. Retrieved, april 2005.

[26] Network Security Resource (NSR). *NSRP: Glossary*, http://www.gslis.utexas.edu/~netsec/gloss.html

[27] Oracle Corporation. *JDBC,* http://www.oracle.com/technology/tech/java/sqlj_jdbc/index.html

[28] Oracle Corporation. *Oracle Corporation*. http://www.oracle.com/

[29] Oracle Corporation. *Oracle9i Supplied PL/SQL Packages and Types Reference Release 2 (9.2) Part Number A96612-01 Chapter 10, DBMS_DEBUG* http://download-west.oracle.com/docs/cd/B10501_01/appdev.920/a96612/ d_debug.htm

[30] Pfaeffle, Thomas. *Oracle8i JDBC Developer's Guide and Reference. Release 8.1.5 Part Number A64685-01* http://www.csee.umbc.edu/help/oracle8/Java.815/a64685/title.htm (new)

[31] Randy, Kath. "*The Win32 Debugging Application Programming Interface*" http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dndebug/html/msdn_debugEH.asp, 2004

[32] Sun Microsystems, Inc. *JDBC API Documentation*, http://java.sun.com/j2se/1.3/docs/guide/jdbc/index.html

[33] Sun Microsystems, Inc. *J2EE Java Servlet Technology*, http://java.sun.com/products/servlet/

[34] Sun Microsystems, Inc. *J2EE JavaServer Pages Technology*, http://java.sun.com/products/jsp/

[35] Sun Microsystems, Inc. *Sun global glossary collection,* http://docs.sun.com/app/docs/coll/417.1

[36] Tics Realtime. *Tics Realtime Kernel Definitions,* http://www.cris.com/~Tics/ticsdef.htm

[37] Winterbottom, Phil. "*Acid: A Debugger Built From A Language*", http://plan9.bell-labs.com/sys/doc/acidpaper.html, Winter 1994.

[38] Wright, Brian et al. *Oracle8i SQLJ Developer's Guide and Reference. Release 8.1.5 Part Number A64685-01,* http://www.csee.umbc.edu/help/oracle8/java.815/a64684/title.htm

Appendices

A. corus_debug_h.sql
   Outprint of package specification for CORUS_DEBUG PL/SQL package

B. CORUS_DEBUG API
   The public procedures of CORUS_DEBUG.

C. Java documentation
   Java documentation for the web application's classes, generated with javadoc.

Appendix A. corus_debug_h.sql

Outprint of package specification for CORUS_DEBUG PL/SQL package

```
/* CORUS_DEBUG is a wrapping PL/SQL package on top of DBMS_DEBUG. It is
 * aimed to be used as a command-line debugger.
 * It was implemented as part of a Master Thesis project at IMIT,
 * Department of Microelectronics and Information Technology, KTH, Sweden.
 *
 * The project was realized at Corus Technologies AB.
 * Author: Alvaro Mayorga
 */
CREATE OR REPLACE package corus_debug as
  -- Can be deleted. just for get_source_between. Deleter afterwards.
  --  TYPE source_vc2_table IS TABLE OF VARCHAR2(500) INDEX BY BINARY_INTEGER;

  ------------------------------------
  --           VARIABLES            --
  ------------------------------------
  g_tgt_target_session_id varchar2(12);-- default null; -- For target session
  g_attached_session_id   varchar2(12);-- default null; -- For debug session
  g_is_attached     boolean;
  g_trace_out_flag  boolean := true;-- Used in step_to and trace_out.When a
                                    -- bpt is set in order to jump to this bpt.
  g_breakpoint binary_integer := 0;
  g_is_javaapp     boolean := false;


  -- record variables
  l_program_info      dbms_debug.program_info;
  l_runtime_info      dbms_debug.runtime_info;
  l_breakpoint_info   dbms_debug.breakpoint_info;

  g_breakpoint_table  dbms_debug.breakpoint_table;

  ------------------------------------
  --           CONSTANTS            --
  ------------------------------------
  StackDepth CONSTANT PLS_INTEGER := dbms_debug.info_getStackDepth;-- 2;
  Breakpoint CONSTANT PLS_INTEGER := dbms_debug.info_getBreakpoint;-- 4;
  LineInfo   CONSTANT PLS_INTEGER := dbms_debug.info_getLineinfo;  -- 8;
  OerInfo    CONSTANT PLS_INTEGER := dbms_debug.info_getOerInfo;   -- 32;

  BreakException CONSTANT PLS_INTEGER := dbms_debug.break_exception ;--     2;
(step over calls).
  BreakAnyCall   CONSTANT PLS_INTEGER := dbms_debug.break_any_call  ;--    12;
-- 4 | 8   (step into calls).
  BreakReturn    CONSTANT PLS_INTEGER := dbms_debug.break_return    ;--    16;
-- Indeed 512.
  BreakNextLine  CONSTANT PLS_INTEGER := dbms_debug.break_next_line ;--    32;
  BreakAnyReturn CONSTANT PLS_INTEGER := dbms_debug.break_any_return;--   512;
-- Indeed 16.
  BreakHandler   CONSTANT PLS_INTEGER := dbms_debug.break_handler   ;--  2048;
  BreakExecution CONSTANT PLS_INTEGER := dbms_debug.abort_execution ;--  8192;

  BreakStep      CONSTANT PLS_INTEGER := dbms_debug.break_next_line ;--    32;
  BreakStepIn    CONSTANT PLS_INTEGER := dbms_debug.break_any_call  ;--    12;
-- 4 | 8   (step into calls)
  BreakStepOut   CONSTANT PLS_INTEGER := dbms_debug.break_return    ;--    16;
-- Indeed 512.
```

```
  BreakStepOver  CONSTANT PLS_INTEGER := dbms_debug.break_exception ;--     2;
(step over calls).
  BreakTraceOut  CONSTANT PLS_INTEGER := dbms_debug.break_any_return;--   512;
-- Indeed 16.

  VERSION_MAJOR CONSTANT BINARY_INTEGER := 1;
  VERSION_MINOR CONSTANT BINARY_INTEGER := 0;
  WIN_SIZE_MIN CONSTANT  BINARY_INTEGER := 1;
  WIN_SIZE               BINARY_INTEGER := 3;
  WIN_MODE               BINARY_INTEGER := 1; -- 1:to see part of the code
                                              -- 0:to see the entire code
  WIN_WIDTH              BINARY_INTEGER := 70;
  WIN_WIDTH_MIN CONSTANT BINARY_INTEGER := 50;
  Success    CONSTANT BINARY_INTEGER := dbms_debug.success; -- 0.

  ---------------------------------------
  ---         COMMOM SESSION         ---
  ---------------------------------------
  procedure version;
  procedure self_check(timeout IN binary_integer := 60);
  function set_timeout(timeout BINARY_INTEGER)
           return varchar2;-- BINARY_INTEGER;
  procedure set_diagnostic_level(dlevel IN BINARY_INTEGER);

  ---------------------------------------
  ---         TARGET SESSION         ---
  ---------------------------------------
  procedure initialize;--(session_id_in in varchar2 default null,
                     -- diagnostics   in binary_integer default 0);
  function initializeF(session_id_in in varchar2 default null,
                     diagnostics   in binary_integer default 0)
                return varchar2;
  procedure debug_on(no_client_side_plsql_engine in boolean default true,
  immediate in boolean default false);
  function debug_onF(no_client_side_plsql_engine in boolean default true,
                   immediate in boolean default false)
              return varchar2;
  procedure debug_off;
  function  debug_offF return varchar2;

  ---------------------------------------
  ---          DEBUG SESSION         ---
  ---------------------------------------
  procedure is_javaapp;
  procedure attach_session(debug_session_id in varchar2,
                         diagnostics in binary_integer := 0);
  function attach_sessionF(debug_session_id in varchar2,
                         diagnostics in binary_integer default 0)
           return varchar2;
  procedure attach_sessionJ(debug_session_id in varchar2);
  procedure sync(info_requested_in in binary_integer default null);
  function  syncF(info_requested in binary_integer default null)
     return varchar2;

  --- STEP PROCEDURES AND FUNCTIONS
  procedure continue(breakflags  in binary_integer := 12);--,
  --info_req_in in binary_integer default 14);
  function continueF(breakflags  in binary_integer,
                   ret_msg out varchar2)
        return binary_integer;

  procedure step;
  procedure step_in;
  procedure step_over;-- (To step until a breakpoint)
  procedure step_out;
  procedure step_to;
  procedure trace_out;
  procedure abort;

  -- SOURCE PROCEDURES AND FUNCTIONS
  procedure show_source_line;
  procedure show_curr_source;
  procedure source(first_line  in binary_integer,
                   last_line   in binary_integer,
                   win_size in binary_integer);
  procedure show_source_procfcn(name_in in varchar2,
                                line_in in binary_integer default 0);
  procedure show_source_pkgbody(name_in in varchar2,
                                line_in in binary_integer default 0);
  procedure show_source_trigger(name_in in varchar2,
                                line_in in binary_integer default 0);
  procedure set_win_size(size_in in binary_integer);
  procedure set_win_width(width_in in binary_integer);
  procedure set_win_mode(mode_in in binary_integer);

  -- VARIABLE ACCESSING AND SETTING.
  procedure get_value(variable_name in varchar2,
                frame#        in binary_integer default 0,
                format        in varchar2 := null);
  function get_valueF(variable_name in varchar2,
                frame#        in binary_integer default 0,
                format        in varchar2 := null)
                return varchar2;
  procedure set_value(assignment_statement in varchar2,
                      frame#               in binary_integer := 0);

  function set_valueF(assignment_statement in varchar2,
                      frame#               in binary_integer := 0,
                      res_msg           out varchar2)
               return binary_integer;
  procedure get_pvalue(variable_name    in varchar2,
                       name_space_in    in BINARY_INTEGER default 2,
                       progname_in      in varchar2 default null,
                       format           in varchar2 default null);

                                 --frame#        in binary_integer default 0,
                       --format        in varchar2 default null);
  function get_pvalueF(variable_name in varchar2,
                       name_space_in IN  BINARY_INTEGER default 2,
                       progname_in       IN varchar2 default null,
                       format in varchar2 default null)
                return varchar2;
  procedure set_pvalue(assignment_statement in varchar2);
  function set_pvalueF(assignment_statement in varchar2)
                return binary_integer;
  --------------------------------------
  ---           BREAKPOINTS          ---
  --------------------------------------
  procedure set_break(line#      in binary_integer,
                    progname   in varchar2 default NULL,
                    username   in varchar2 default USER,
                    fuzzy      in binary_integer := 1,
                    iterations in binary_integer := 0);
  procedure set_breakpoint(line#     in binary_integer,
                          fuzzy      in binary_integer := 1,
                          iterations in binary_integer := 0);
  function set_breakpointF(line#     in binary_integer,
                          fuzzy      in binary_integer := 1,
                         iterations in binary_integer := 0)
       return varchar2;
  procedure del_break(breakpoint_nr_in in binary_integer);
  procedure delete_breakpoint(breakpoint_nr_in in binary_integer);
  procedure breaks;
  procedure show_breakpoints;
  function get_type_of(name_in  in varchar2,
                owner_in in varchar2 default USER) return varchar2;
  procedure break_info(line#_in       in binary_integer,
                       prog_name_in   in varchar2,
                       prog_owner_in  in varchar2 default USER,
                       prog_dblink_in in varchar2 default null,
                       fuzzy          in binary_integer default 1,
                       iterations     in binary_integer default 0);
  function break_infoF(line#_in       in binary_integer,
                       prog_name_in   in varchar2,
                       prog_owner_in  in varchar2 default USER,
                       prog_dblink_in in varchar2 default null,
                       fuzzy          in binary_integer default 1,
                       iterations     in binary_integer default 0)
               return varchar2 ;

  -- STACK, RUNTIME AND PROGRAM INFO.
  procedure print_stack; -- print_backtrace;
  function stackJ return varray; -- print_backtrace;
  procedure print_runtime_info(runinfo_in IN dbms_debug.runtime_info default
null);
  procedure print_program_info;
  procedure update_program_info;

  --------------------------------------
  --- COMMON PART AND HELP FUNCTIONS ---
  --------------------------------------
  function get_reason return varchar2;
  function predefined_reasons(reason_nr in binary_integer) return varchar2;
  function predefined_errors(error_nr in binary_integer)   return varchar2;
  function predef_libs(type_in in binary_integer)          return varchar2;
  function predef_names(name_in in binary_integer)         return varchar2;
  function predefined_namespaces(name_in in varchar2)    return binary_integer;
  function predefined_libunittypes(name_in in varchar2)  return binary_integer;
  procedure help(cmd in varchar2 default null);
  procedure print_msg(msg_in in varchar2);
  procedure show_error(error_code in number, error_msg in varchar2);
  function newline return varchar2;

  --------------------------------------
  ---    ALIAS TO SOME PROCEDURES    ---
  --------------------------------------
  -- shortcut for attach_session
  procedure attach(debug_session_id in varchar2,
                   diagnostics in binary_integer := 0);
  -- shortcut for print_runtime_info.
  procedure runtime(r_in IN dbms_debug.runtime_info default null);
  procedure stack; -- shortcut for print_stack.
  procedure program; -- shortcut for print_program_info.
  procedure s;     -- shortcut for step
  procedure to;    -- shortcut for trace_out
end corus_debug;
/
```

Appendix B. CORUS_DEBUG API

The public procedures of CORUS_DEBUG.

```
/* CORUS_DEBUG is a wrapping PL/SQL package on top of DBMS_DEBUG. It is
 * aimed to be used as a command-line debugger.
 * It was implemented as part of a Master Thesis project at IMIT,
 * Department of Microelectronics and Information Technology, KTH, Sweden.
 *
 * The project was realized at Corus Technologies AB.
 * Author: Alvaro Mayorga
 */
CREATE OR REPLACE package corus_debug as
  ------------------------------------
  --           VARIABLES            --
  ------------------------------------
  g_tgt_target_session_id varchar2(12);-- default null; -- For target session
  g_attached_session_id   varchar2(12);-- default null; -- For debug session
  g_is_attached      boolean;
  g_breakpoint binary_integer := 0;

  -- record variables
  l_program_info      dbms_debug.program_info;
  l_runtime_info      dbms_debug.runtime_info;
  l_breakpoint_info   dbms_debug.breakpoint_info;

  g_breakpoint_table  dbms_debug.breakpoint_table;

  ------------------------------------
  --           CONSTANTS            --
  ------------------------------------
  StackDepth CONSTANT PLS_INTEGER := dbms_debug.info_getStackDepth;-- 2;
  Breakpoint CONSTANT PLS_INTEGER := dbms_debug.info_getBreakpoint;-- 4;
  LineInfo   CONSTANT PLS_INTEGER := dbms_debug.info_getLineinfo;  -- 8;
  OerInfo    CONSTANT PLS_INTEGER := dbms_debug.info_getOerInfo;   -- 32;

  BreakException CONSTANT PLS_INTEGER := dbms_debug.break_exception ;
  BreakAnyCall   CONSTANT PLS_INTEGER := dbms_debug.break_any_call  ;
  BreakReturn    CONSTANT PLS_INTEGER := dbms_debug.break_return    ;
  BreakNextLine  CONSTANT PLS_INTEGER := dbms_debug.break_next_line ;
  BreakAnyReturn CONSTANT PLS_INTEGER := dbms_debug.break_any_return;
  BreakHandler   CONSTANT PLS_INTEGER := dbms_debug.break_handler   ;
  BreakExecution CONSTANT PLS_INTEGER := dbms_debug.abort_execution ;

  WIN_SIZE_MIN CONSTANT  BINARY_INTEGER := 1;
  WIN_SIZE               BINARY_INTEGER := 3;
  WIN_MODE               BINARY_INTEGER := 1; -- 1:to see part of the code
                                              -- 0:to see the entire code
  WIN_WIDTH              BINARY_INTEGER := 70;
  WIN_WIDTH_MIN CONSTANT BINARY_INTEGER := 50;
  Success    CONSTANT BINARY_INTEGER := dbms_debug.success; -- 0.
  --------------------------------------
  ---         COMMOM SESSION         ---
  --------------------------------------
  procedure version;
  procedure self_check(timeout IN binary_integer := 60);
  procedure set_diagnostic_level(dlevel IN BINARY_INTEGER);

  --------------------------------------
  ---         TARGET SESSION         ---
  --------------------------------------
  procedure initialize;--(session_id_in in varchar2 default null,
                       -- diagnostics   in binary_integer default 0);
  procedure debug_on(no_client_side_plsql_engine in boolean default true,
                     immediate in boolean default false);
  procedure debug_off;

  --------------------------------------
  ---          DEBUG SESSION         ---
  --------------------------------------
  procedure attach_session(debug_session_id in varchar2,
                           diagnostics in binary_integer := 0);
  procedure sync(info_requested_in in binary_integer default null);

  --- STEP PROCEDURES AND FUNCTIONS
  procedure continue(breakflags  in binary_integer := 12);
  procedure step;
  procedure step_in;
  procedure step_over;-- (To step until a breakpoint)
  procedure step_out;
  procedure step_to;
  procedure trace_out;
  procedure abort;

  -- SOURCE PROCEDURES AND FUNCTIONS
  procedure show_source_line;
  procedure show_curr_source;
  procedure source(first_line  in binary_integer,
                   last_line   in binary_integer,
                   win_size in binary_integer);
  procedure show_source_procfcn(name_in in varchar2,
                                line_in in binary_integer default 0);
  procedure show_source_pkgbody(name_in in varchar2,
                                line_in in binary_integer default 0);
  procedure show_source_trigger(name_in in varchar2,
                                line_in in binary_integer default 0);
  procedure set_win_size(size_in in binary_integer);
  procedure set_win_width(width_in in binary_integer);
  procedure set_win_mode(mode_in in binary_integer);

  -- VARIABLE ACCESSING AND SETTING.
  procedure get_value(variable_name in varchar2,
                      frame#        in binary_integer default 0,
                format        in varchar2 := null);
  procedure set_value(assignment_statement in varchar2,
                      frame#               in binary_integer := 0);
  procedure get_pvalue(variable_name    in varchar2,
                       name_space_in    in BINARY_INTEGER default 2,
                       progname_in      in varchar2 default null,
                       format           in varchar2 default null);
  procedure set_pvalue(assignment_statement in varchar2);

  --------------------------------------
  ---           BREAKPOINTS          ---
  --------------------------------------
  procedure set_break(line#      in binary_integer,
                    progname   in varchar2 default NULL,
                    username   in varchar2 default USER,
                    fuzzy      in binary_integer := 1,
                    iterations in binary_integer := 0);
  procedure set_breakpoint(line#     in binary_integer,
                          fuzzy      in binary_integer := 1,
                          iterations in binary_integer := 0);
  procedure del_break(breakpoint_nr_in in binary_integer);
  procedure delete_breakpoint(breakpoint_nr_in in binary_integer);
  procedure breaks;
```

```
  procedure show_breakpoints;
  procedure break_info(line#_in       in binary_integer,
                       prog_name_in   in varchar2,
                       prog_owner_in  in varchar2 default USER,
                       prog_dblink_in in varchar2 default null,
                       fuzzy          in binary_integer default 1,
                       iterations     in binary_integer default 0);

  -- STACK, RUNTIME AND PROGRAM INFO.
  procedure print_stack; -- print_backtrace;
  procedure print_runtime_info(runinfo_in IN dbms_debug.runtime_info := null);
  procedure print_program_info;
  procedure update_program_info;

  --------------------------------------
  --- COMMON PART AND HELP FUNCTIONS ---
  --------------------------------------
  procedure help(cmd in varchar2 default null);
  procedure print_msg(msg_in in varchar2);
  procedure show_error(error_code in number, error_msg in varchar2);

  --------------------------------------
  ---    ALIAS TO SOME PROCEDURES    ---
  --------------------------------------
  procedure attach(debug_session_id in varchar2,
                   diagnostics in binary_integer := 0);
  procedure runtime(r_in IN dbms_debug.runtime_info default null);
  procedure stack; -- shortcut for print_stack.
  procedure program; -- shortcut for print_program_info.
  procedure s;     -- shortcut for step
  procedure to;    -- shortcut for trace_out
end corus_debug;
```

Appendix C. Java documentation for the web application with servlets.

Debug and implementation ofa web based PL/SQL debugger using Oracle's debug API

Packages	
deb	
deb.command	

Package deb

Interface Summary	
ICODEdefinitions	

Class Summary	
DbgLogin	Class that handles the loggin of the user as a *debugger*.
DbgUserBanner	Class to display user and connection information for the debugger session.
DbmsOuputToServlet	Class that retrieves and displays the output of the DBMS_OUTPUT package in an Oracle Database.
DebuggeeServlet	Handles the start, control and finalization of debuggee session.
DebuggerServlet	Handles the start, control and finalization of debugger session as well that controlls the debuggee session.
TgtLogin	Class that handles the loggin of the user as a *debuggee*.
TgtUserBanner	Class to display user and connection information for the debugger session.

deb

Interface ICODEdefinitions

All Known Implementing Classes:

DbgLogin, DbgUserBanner, DebuggeeServlet, DebuggerServlet, TgtLogin, TgtUserBanner public interface **ICODEdefinitions**

Author:
Alvaro Mayorga

Field Summary

static java.lang.String	**ATTR_DBG_DBMS_OUTPUT**
static java.lang.String	**ATTR_DBGATTACHED_TO**
static java.lang.String	**ATTR_DBGCOMMAND**
static java.lang.String	**ATTR_DBGCONNECTION**
static java.lang.String	**ATTR_DBGDATABASE**
static java.lang.String	**ATTR_DBGUSER** SESSION ATTRIBUTES
static java.lang.String	**ATTR_TGTCOMMAND**
static java.lang.String	**ATTR_TGTCONNECTION**
static java.lang.String	**ATTR_TGTDATABASE**
static java.lang.String	**ATTR_TGTSESSIONID**
static java.lang.String	**ATTR_TGTUSER**
static int	**BYTEBUFF_LEN**
static java.lang.String	**CMD_ABORT**
static java.lang.String	**CMD_ABOUT**
static java.lang.String	**CMD_ALTER_PROC**
static java.lang.String	**CMD_ATTACH** COMMANDS DEBUG
static java.lang.String	**CMD_DEBUG_ON**
static java.lang.String	**CMD_DEL_BREAK**
static java.lang.String	**CMD_DETACH**
static java.lang.String	**CMD_EXECUTE**
static java.lang.String	**CMD_EXECUTE_QUERY** COMMANDS COMMON
static java.lang.String	**CMD_EXEMPEL**
static java.lang.String	**CMD_GET_PKG_VALUE**

static java.lang.String	CMD_GET_VALUE
static java.lang.String	CMD_HELP
static java.lang.String	CMD_INIT
static java.lang.String	CMD_LIST_BREAK
static java.lang.String	CMD_PKG_TEST
static java.lang.String	CMD_PROC_TEST
static java.lang.String	CMD_PROGRAM
static java.lang.String	CMD_SELF_CHECK
static java.lang.String	CMD_SET_BREAK
static java.lang.String	CMD_SET_PKG_VALUE
static java.lang.String	CMD_SET_VALUE
static java.lang.String	CMD_SOURCE_CURRENT
static java.lang.String	CMD_SOURCE_PKG
static java.lang.String	CMD_SOURCE_PROC
static java.lang.String	CMD_SOURCE_TRIG
static java.lang.String	CMD_STACK
static java.lang.String	CMD_START_DEBUGGEE COMMANDS TARGET
static java.lang.String	CMD_STEP
static java.lang.String	CMD_STEP_IN
static java.lang.String	CMD_STEP_OUT
static java.lang.String	CMD_STEP_OVER
static java.lang.String	CMD_SYNC
static java.lang.String	CMD_TRACE_OUT
static java.lang.String	CMD_TRIG_TEST
static java.lang.String	CMD_VERSION
static java.lang.String	DEFAULT_BACKGROUND
static java.lang.String	DEFAULT_BGCOLOR

static java.lang.String	DEFAULT_CONTENT CONFIGURATION PARAMETERS
static java.lang.String	DEFAULT_JDBC_DRIVER
static java.lang.String	DEFAULT_PROPERTIES
static java.lang.String	DEFAULT_PROPERTIES_FILE
static java.lang.String	DEFAULT_TCOLOR
static java.lang.String	FRAME_BROWSE
static java.lang.String	FRAME_HELP
static java.lang.String	FRAME_MAIN
static java.lang.String	FRAME_MENU
static java.lang.String	GIF_ABORT
static java.lang.String	GIF_ALTER
static java.lang.String	GIF_ATTACH
static java.lang.String	GIF_CURR_SOURCE
static java.lang.String	GIF_DEBUG_OFF
static java.lang.String	GIF_DEBUG_ON
static java.lang.String	GIF_DEL_BREAK
static java.lang.String	GIF_GET_PVALUE
static java.lang.String	GIF_GET_VALUE
static java.lang.String	GIF_HELP
static java.lang.String	GIF_INIT
static java.lang.String	GIF_LINK_CORUS_ICON
static java.lang.String	GIF_LINK_ICODE_ICON
static java.lang.String	GIF_LIST_BREAK
static java.lang.String	GIF_MENU
static java.lang.String	GIF_PROG_INFO
static java.lang.String	GIF_ROOT IMAGE FILE NAMES
static java.lang.String	GIF_SET_BREAK
static java.lang.String	GIF_SET_PVALUE

static java.lang.String	**GIF_SET_VALUE**
static java.lang.String	**GIF_SOURCE_PKG_BODY**
static java.lang.String	**GIF_SOURCE_PROC**
static java.lang.String	**GIF_SOURCE_TRIGGER**
static java.lang.String	**GIF_STACK**
static java.lang.String	**GIF_START_DEBUGGEE**
static java.lang.String	**GIF_STEP**
static java.lang.String	**GIF_STEP_IN**
static java.lang.String	**GIF_STEP_OUT**
static java.lang.String	**GIF_STEP_OVER**
static java.lang.String	**GIF_SYNC**
static java.lang.String	**GIF_TEST_PKG**
static java.lang.String	**GIF_TEST_PROC**
static java.lang.String	**GIF_TEST_TRIG**
static java.lang.String	**GIF_TRACE_OUT**
static java.lang.String	**MENU_BACKGROUND** COLORS FOR BACK-, FOREGROUND and OTHERS
static java.lang.String	**MENU_BGCOLOR**
static java.lang.String	**MENU_TCOLOR**
static java.lang.String	**MULTI_BACKGROUND**
static java.lang.String	**MULTI_BGCOLOR**
static java.lang.String	**MULTI_TCOLOR**
static java.lang.String	**NAV_BACKGROUND**
static java.lang.String	**NAV_BGCOLOR**
static java.lang.String	**NAV_TCOLOR**
static java.lang.String	**NOTHING_TO_DO**
static int	**NUMBER_OF_DBG_ARGUMENTS**
static int	**NUMBER_OF_TGT_ARGUMENTS**

static java.lang.String	**PAR_DBGCOMMAND**
static java.lang.String	**PAR_DBGDATABASE**
static java.lang.String	**PAR_DBGPASSWORD**
static java.lang.String	**PAR_DBGPORT**
static java.lang.String	**PAR_DBGSQL_STMT**
static java.lang.String	**PAR_DBGUSER** FORM PARAMETERS
static java.lang.String	**PAR_DEF_DBGDATABASE**
static java.lang.String	**PAR_DEF_DBGPASSWORD**
static java.lang.String	**PAR_DEF_DBGPORT**
static java.lang.String	**PAR_DEF_DBGUSER** DEFAULT VALUES FOR FORM PARAMETERS
static java.lang.String	**PAR_DEF_TGTDATABASE**
static java.lang.String	**PAR_DEF_TGTPASSWORD**
static java.lang.String	**PAR_DEF_TGTPORT**
static java.lang.String	**PAR_DEF_TGTUSER**
static java.lang.String	**PAR_LOGOUT**
static java.lang.String	**PAR_TGTCOMMAND**
static java.lang.String	**PAR_TGTDATABASE**
static java.lang.String	**PAR_TGTPASSWORD**
static java.lang.String	**PAR_TGTPORT**
static java.lang.String	**PAR_TGTSQL_STMT**
static java.lang.String	**PAR_TGTUSER**
static int	**STRBUFF_LEN**
static java.lang.String	**URL_BROWSE_DIR_SERVLET**
static java.lang.String	**URL_BROWSE_OBJ_SERVLET**
static java.lang.String	**URL_DEBUG_BANNER_SERVLET**
static java.lang.String	**URL_DEBUG_LOGIN_SERVLET** SERVLETS, URLS AND FRAME NAMES
static java.lang.String	**URL_DEBUG_SERVLET**
static java.lang.String	**URL_HELP_DIR_SERVLET**

static java.lang.String	**URL_HELP_OBJ_SERVLET**
static java.lang.String	**URL_TARGET_BANNER_SERVLET**
static java.lang.String	**URL_TARGET_LOGIN_SERVLET**
static java.lang.String	**URL_TARGET_SERVLET**

deb

Class DbgLogin

```
java.lang.Object
  └javax.servlet.GenericServlet
      └javax.servlet.http.HttpServlet
          └deb.DbgLogin
```

All Implemented Interfaces:
ICODEdefinitions, java.io.Serializable, javax.servlet.Servlet, javax.servlet.ServletConfig

public class **DbgLogin**
extends javax.servlet.http.HttpServlet
implements ICODEdefinitions

See Also:
Serialized Form

Field Summary

Fields inherited from interface deb.ICODEdefinitions
ATTR_DBG_DBMS_OUTPUT, ATTR_DBGATTACHED_TO, ATTR_DBGCOMMAND, ATTR_DBGCONNECTION, ATTR_DBGDATABASE, ATTR_DBGUSER, ATTR_TGTCOMMAND, ATTR_TGTCONNECTION, ATTR_TGTDATABASE, ATTR_TGTSESSIONID, ATTR_TGTUSER, BYTEBUFF_LEN, CMD_ABORT, CMD_ABOUT, CMD_ALTER_PROC, CMD_ATTACH, CMD_DEBUG_ON, CMD_DEL_BREAK, CMD_DETACH, CMD_EXECUTE, CMD_EXECUTE_QUERY, CMD_EXEMPEL, CMD_GET_PKG_VALUE, CMD_GET_VALUE, CMD_HELP, CMD_INIT, CMD_LIST_BREAK, CMD_PKG_TEST, CMD_PROC_TEST, CMD_PROGRAM, CMD_SELF_CHECK, CMD_SET_BREAK, CMD_SET_PKG_VALUE, CMD_SET_VALUE, CMD_SOURCE_CURRENT, CMD_SOURCE_PKG, CMD_SOURCE_PROC, CMD_SOURCE_TRIG, CMD_STACK, CMD_START_DEBUGGEE, CMD_STEP, CMD_STEP_IN, CMD_STEP_OUT, CMD_STEP_OVER, CMD_SYNC, CMD_TRACE_OUT, CMD_TRIG_TEST, CMD_VERSION, DEFAULT_BACKGROUND, DEFAULT_BGCOLOR, DEFAULT_CONTENT, DEFAULT_JDBC_DRIVER, DEFAULT_PROPERTIES, DEFAULT_PROPERTIES_FILE, DEFAULT_TCOLOR, FRAME_BROWSE, FRAME_HELP, FRAME_MAIN, FRAME_MENU, GIF_ABORT, GIF_ALTER, GIF_ATTACH, GIF_CURR_SOURCE, GIF_DEBUG_OFF, GIF_DEBUG_ON, GIF_DEL_BREAK, GIF_GET_PVALUE, GIF_GET_VALUE, GIF_HELP, GIF_INIT, GIF_LINK_CORUS_ICON, GIF_LINK_ICODE_ICON, GIF_LIST_BREAK, GIF_MENU, GIF_PROG_INFO, GIF_ROOT, GIF_SET_BREAK, GIF_SET_PVALUE, GIF_SET_VALUE, GIF_SOURCE_PKG_BODY, GIF_SOURCE_PROC, GIF_SOURCE_TRIGGER, GIF_STACK, GIF_START_DEBUGGEE, GIF_STEP, GIF_STEP_IN, GIF_STEP_OUT, GIF_STEP_OVER, GIF_SYNC, GIF_TEST_PKG, GIF_TEST_PROC, GIF_TEST_TRIG, GIF_TRACE_OUT, MENU_BACKGROUND, MENU_BGCOLOR, MENU_TCOLOR, MULTI_BACKGROUND, MULTI_BGCOLOR, MULTI_TCOLOR, NAV_BACKGROUND, NAV_BGCOLOR, NAV_TCOLOR, NOTHING_TO_DO, NUMBER_OF_DBG_ARGUMENTS, NUMBER_OF_TGT_ARGUMENTS, PAR_DBGCOMMAND, PAR_DBGDATABASE, PAR_DBGPASSWORD, PAR_DBGPORT,

PAR_DBGSQL_STMT, PAR_DBGUSER, PAR_DEF_DBGDATABASE, PAR_DEF_DBGPASSWORD, PAR_DEF_DBGPORT, PAR_DEF_DBGUSER, PAR_DEF_TGTDATABASE, PAR_DEF_TGTPASSWORD, PAR_DEF_TGTPORT, PAR_DEF_TGTUSER, PAR_LOGOUT, PAR_TGTCOMMAND, PAR_TGTDATABASE, PAR_TGTPASSWORD, PAR_TGTPORT, PAR_TGTSQL_STMT, PAR_TGTUSER, STRBUFF_LEN, URL_BROWSE_DIR_SERVLET, URL_BROWSE_OBJ_SERVLET, URL_DEBUG_BANNER_SERVLET, URL_DEBUG_LOGIN_SERVLET, URL_DEBUG_SERVLET, URL_HELP_DIR_SERVLET, URL_HELP_OBJ_SERVLET, URL_TARGET_BANNER_SERVLET, URL_TARGET_LOGIN_SERVLET, URL_TARGET_SERVLET

Constructor Summary

DbgLogin()

Method Summary

void	**doGet**(javax.servlet.http.HttpServletRequest request, javax.servlet.http.HttpServletResponse response)
void	**doPost**(javax.servlet.http.HttpServletRequest request, javax.servlet.http.HttpServletResponse response)
void	**init**(javax.servlet.ServletConfig config)
protected void	**mylog**(java.lang.String str)
void	**processRequest**(javax.servlet.http.HttpServletRequest request, javax.servlet.http.HttpServletResponse response)

Methods inherited from class javax.servlet.http.HttpServlet

doDelete, doHead, doOptions, doPut, doTrace, getLastModified, service, service

Methods inherited from class javax.servlet.GenericServlet

destroy, getInitParameter, getInitParameterNames, getServletConfig, getServletContext, getServletInfo, getServletName, init, log, log

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

DbgLogin

```
public DbgLogin()
```

Method Detail

init

```
public void init(javax.servlet.ServletConfig config)
          throws javax.servlet.ServletException
```

Specified by:
init in interface javax.servlet.Servlet

Throws:
javax.servlet.ServletException

doGet

```
public void doGet(javax.servlet.http.HttpServletRequest request,
                  javax.servlet.http.HttpServletResponse response)
           throws java.io.IOException,
                  javax.servlet.ServletException
```

Throws:

`java.io.IOException`

`javax.servlet.ServletException`

---

doPost

```
public void doPost(javax.servlet.http.HttpServletRequest request,
                   javax.servlet.http.HttpServletResponse response)
            throws java.io.IOException,
                   javax.servlet.ServletException
```

Throws:

`java.io.IOException`

`javax.servlet.ServletException`

---

processRequest

```
public void processRequest(javax.servlet.http.HttpServletRequest request,
                           javax.servlet.http.HttpServletResponse response)
                    throws java.io.IOException,
                           javax.servlet.ServletException
```

Throws:

`java.io.IOException`

`javax.servlet.ServletException`

---

mylog

```
protected void mylog(java.lang.String str)
```

---

deb

Class DbgUserBanner

```
java.lang.Object
  └javax.servlet.GenericServlet
      └javax.servlet.http.HttpServlet
          └deb.DbgUserBanner
```

All Implemented Interfaces:

ICODEdefinitions, java.io.Serializable, javax.servlet.Servlet, javax.servlet.ServletConfig

---

public class **DbgUserBanner**
extends javax.servlet.http.HttpServlet
implements ICODEdefinitions

Author:
Alvaro Mayorga
See Also:
Serialized Form

Field Summary

Fields inherited from interface deb.ICODEdefinitions

`ATTR_DBG_DBMS_OUTPUT, ATTR_DBGATTACHED_TO, ATTR_DBGCOMMAND, ATTR_DBGCONNECTION, ATTR_DBGDATABASE, ATTR_DBGUSER, ATTR_TGTCOMMAND, ATTR_TGTCONNECTION, ATTR_TGTDATABASE, ATTR_TGTSESSIONID, ATTR_TGTUSER, BYTEBUFF_LEN, CMD_ABORT, CMD_ABOUT, CMD_ALTER_PROC, CMD_ATTACH, CMD_DEBUG_ON, CMD_DEL_BREAK, CMD_DETACH, CMD_EXECUTE, CMD_EXECUTE_QUERY, CMD_EXEMPEL, CMD_GET_PKG_VALUE, CMD_GET_VALUE, CMD_HELP, CMD_INIT, CMD_LIST_BREAK, CMD_PKG_TEST, CMD_PROC_TEST, CMD_PROGRAM, CMD_SELF_CHECK, CMD_SET_BREAK, CMD_SET_PKG_VALUE, CMD_SET_VALUE, CMD_SOURCE_CURRENT, CMD_SOURCE_PKG, CMD_SOURCE_PROC, CMD_SOURCE_TRIG, CMD_STACK, CMD_START_DEBUGGEE, CMD_STEP, CMD_STEP_IN, CMD_STEP_OUT, CMD_STEP_OVER, CMD_SYNC, CMD_TRACE_OUT, CMD_TRIG_TEST, CMD_VERSION, DEFAULT_BACKGROUND, DEFAULT_BGCOLOR, DEFAULT_CONTENT, DEFAULT_JDBC_DRIVER, DEFAULT_PROPERTIES, DEFAULT_PROPERTIES_FILE, DEFAULT_TCOLOR, FRAME_BROWSE, FRAME_HELP, FRAME_MAIN, FRAME_MENU, GIF_ABORT, GIF_ALTER, GIF_ATTACH, GIF_CURR_SOURCE, GIF_DEBUG_OFF, GIF_DEBUG_ON, GIF_DEL_BREAK, GIF_GET_PVALUE, GIF_GET_VALUE, GIF_HELP, GIF_INIT, GIF_LINK_CORUS_ICON, GIF_LINK_ICODE_ICON, GIF_LIST_BREAK, GIF_MENU, GIF_PROG_INFO, GIF_ROOT, GIF_SET_BREAK, GIF_SET_PVALUE, GIF_SET_VALUE, GIF_SOURCE_PKG_BODY, GIF_SOURCE_PROC, GIF_SOURCE_TRIGGER, GIF_STACK, GIF_START_DEBUGGEE, GIF_STEP, GIF_STEP_IN, GIF_STEP_OUT, GIF_STEP_OVER, GIF_SYNC, GIF_TEST_PKG, GIF_TEST_PROC, GIF_TEST_TRIG, GIF_TRACE_OUT, MENU_BACKGROUND, MENU_BGCOLOR, MENU_TCOLOR, MULTI_BACKGROUND, MULTI_BGCOLOR, MULTI_TCOLOR, NAV_BACKGROUND, NAV_BGCOLOR, NAV_TCOLOR, NOTHING_TO_DO, NUMBER_OF_DBG_ARGUMENTS, NUMBER_OF_TGT_ARGUMENTS, PAR_DBGCOMMAND, PAR_DBGDATABASE, PAR_DBGPASSWORD, PAR_DBGPORT, PAR_DBGSQL_STMT, PAR_DBGUSER, PAR_DEF_DBGDATABASE, PAR_DEF_DBGPASSWORD, PAR_DEF_DBGPORT, PAR_DEF_DBGUSER, PAR_DEF_TGTDATABASE, PAR_DEF_TGTPASSWORD, PAR_DEF_TGTPORT, PAR_DEF_TGTUSER, PAR_LOGOUT, PAR_TGTCOMMAND, PAR_TGTDATABASE, PAR_TGTPASSWORD, PAR_TGTPORT, PAR_TGTSQL_STMT, PAR_TGTUSER, STRBUFF_LEN, URL_BROWSE_DIR_SERVLET, URL_BROWSE_OBJ_SERVLET, URL_DEBUG_BANNER_SERVLET, URL_DEBUG_LOGIN_SERVLET, URL_DEBUG_SERVLET, URL_HELP_DIR_SERVLET, URL_HELP_OBJ_SERVLET, URL_TARGET_BANNER_SERVLET, URL_TARGET_LOGIN_SERVLET, URL_TARGET_SERVLET`

Constructor Summary

`DbgUserBanner``()`

Method Summary

`protected void`	**`doGet`**`(javax.servlet.http.HttpServletRequest request, javax.servlet.http.HttpServletResponse response)` Handles the HTTP `GET` method.
`protected void`	**`doPost`**`(javax.servlet.http.HttpServletRequest request, javax.servlet.http.HttpServletResponse response)` Handles the HTTP `POST` method.
`protected void`	**`processRequest`**`(javax.servlet.http.HttpServletRequest request, javax.servlet.http.HttpServletResponse response)` Processes requests for both HTTP `GET` and `POST` methods.

Methods inherited from class javax.servlet.http.HttpServlet

`doDelete, doHead, doOptions, doPut, doTrace, getLastModified, service, service`

Methods inherited from class javax.servlet.GenericServlet

`destroy, getInitParameter, getInitParameterNames, getServletConfig, getServletContext,`

`getServletInfo, getServletName, init, init, log, log`

Methods inherited from class java.lang.Object
`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Constructor Detail

DbgUserBanner

```
public DbgUserBanner()
```

Method Detail

processRequest

```
protected void processRequest(javax.servlet.http.HttpServletRequest request,
                              javax.servlet.http.HttpServletResponse response)
                       throws javax.servlet.ServletException,
                              java.io.IOException
```

Processes requests for both HTTP `GET` and `POST` methods.
Parameters:
`request` - servlet request
`response` - servlet response
Throws:
`javax.servlet.ServletException`
`java.io.IOException`

---

doGet

```
protected void doGet(javax.servlet.http.HttpServletRequest request,
                     javax.servlet.http.HttpServletResponse response)
              throws javax.servlet.ServletException,
                     java.io.IOException
```

Handles the HTTP `GET` method.
Parameters:
`request` - servlet request
`response` - servlet response
Throws:
`javax.servlet.ServletException`
`java.io.IOException`

---

doPost

```
protected void doPost(javax.servlet.http.HttpServletRequest request,
                      javax.servlet.http.HttpServletResponse response)
               throws javax.servlet.ServletException,
                      java.io.IOException
```

Handles the HTTP `POST` method.
Parameters:
`request` - servlet request
`response` - servlet response
Throws:
`javax.servlet.ServletException`
`java.io.IOException`

deb

Class DbmsOuputToServlet

```
java.lang.Object
  └ deb.DbmsOuputToServlet
```

public class **DbmsOuputToServlet**
extends java.lang.Object

Author:
Alvaro Mayorga

Constructor Summary
`DbmsOuputToServlet(java.sql.Connection conn)`
`DbmsOuputToServlet(java.sql.Connection conn, java.io.PrintWriter out_in)` Creates a new instance of DbmsOuput

Method Summary	
`void`	`close()`
`void`	`disable()`
`void`	`enable(int size)`
`void`	`show(java.io.PrintWriter out)`

Methods inherited from class java.lang.Object
`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Constructor Detail

DbmsOuputToServlet

```
public DbmsOuputToServlet(java.sql.Connection conn,
                          java.io.PrintWriter out_in)
                   throws java.sql.SQLException
```

Creates a new instance of DbmsOuput

DbmsOuputToServlet

```
public DbmsOuputToServlet(java.sql.Connection conn)
                   throws java.sql.SQLException
```

Method Detail

enable

```
public void enable(int size)
            throws java.sql.SQLException
```

Throws:
`java.sql.SQLException`

disable

```
public void disable()
             throws java.sql.SQLException
```

Throws:
`java.sql.SQLException`

show

```
public void show(java.io.PrintWriter out)
          throws java.sql.SQLException
```

Throws:
`java.sql.SQLException`

close

```
public void close()
           throws java.sql.SQLException
```

Throws:
`java.sql.SQLException`

deb
Class DebuggeeServlet

```
java.lang.Object
  └ javax.servlet.GenericServlet
      └ javax.servlet.http.HttpServlet
          └ deb.DebuggeeServlet
```

All Implemented Interfaces:
ICODEdefinitions, java.io.Serializable, javax.servlet.Servlet, javax.servlet.ServletConfig

---

public class **DebuggeeServlet**
extends javax.servlet.http.HttpServlet
implements ICODEdefinitions

See Also:
Serialized Form

Field Summary

protected java.util.Hashtable	**commandTable**
DbmsOuputToServlet	**dout**
protected java.lang.String	**dummyResult**
protected Alter	**myAlter**
protected AttachSession	**myAttach**
protected Help	**myHelp**
protected Initialize	**myInitialize**
protected RunProc1	**myRunProc1**
protected Sync	**mySync**
protected java.lang.StringBuffer	**queryResultText**
protected java.util.Vector	**queryResultVector**
protected java.lang.String	**stmtString**

Fields inherited from interface deb.ICODEdefinitions

ATTR_DBG_DBMS_OUTPUT, ATTR_DBGATTACHED_TO, ATTR_DBGCOMMAND, ATTR_DBGCONNECTION, ATTR_DBGDATABASE, ATTR_DBGUSER, ATTR_TGTCOMMAND, ATTR_TGTCONNECTION, ATTR_TGTDATABASE, ATTR_TGTSESSIONID, ATTR_TGTUSER, BYTEBUFF_LEN, CMD_ABORT, CMD_ABOUT, CMD_ALTER_PROC, CMD_ATTACH, CMD_DEBUG_ON, CMD_DEL_BREAK, CMD_DETACH, CMD_EXECUTE, CMD_EXECUTE_QUERY, CMD_EXEMPEL, CMD_GET_PKG_VALUE, CMD_GET_VALUE, CMD_HELP, CMD_INIT, CMD_LIST_BREAK, CMD_PKG_TEST, CMD_PROC_TEST, CMD_PROGRAM, CMD_SELF_CHECK, CMD_SET_BREAK, CMD_SET_PKG_VALUE, CMD_SET_VALUE, CMD_SOURCE_CURRENT, CMD_SOURCE_PKG, CMD_SOURCE_PROC, CMD_SOURCE_TRIG, CMD_STACK, CMD_START_DEBUGGEE, CMD_STEP, CMD_STEP_IN, CMD_STEP_OUT, CMD_STEP_OVER, CMD_SYNC, CMD_TRACE_OUT, CMD_TRIG_TEST, CMD_VERSION, DEFAULT_BACKGROUND, DEFAULT_BGCOLOR, DEFAULT_CONTENT, DEFAULT_JDBC_DRIVER, DEFAULT_PROPERTIES, DEFAULT_PROPERTIES_FILE, DEFAULT_TCOLOR, FRAME_BROWSE, FRAME_HELP, FRAME_MAIN, FRAME_MENU, GIF_ABORT, GIF_ALTER, GIF_ATTACH, GIF_CURR_SOURCE, GIF_DEBUG_OFF, GIF_DEBUG_ON, GIF_DEL_BREAK, GIF_GET_PVALUE, GIF_GET_VALUE, GIF_HELP, GIF_INIT, GIF_LINK_CORUS_ICON, GIF_LINK_ICODE_ICON, GIF_LIST_BREAK, GIF_MENU, GIF_PROG_INFO, GIF_ROOT, GIF_SET_BREAK, GIF_SET_PVALUE, GIF_SET_VALUE, GIF_SOURCE_PKG_BODY, GIF_SOURCE_PROC, GIF_SOURCE_TRIGGER, GIF_STACK, GIF_START_DEBUGGEE, GIF_STEP, GIF_STEP_IN, GIF_STEP_OUT, GIF_STEP_OVER, GIF_SYNC, GIF_TEST_PKG, GIF_TEST_PROC, GIF_TEST_TRIG, GIF_TRACE_OUT, MENU_BACKGROUND, MENU_BGCOLOR, MENU_TCOLOR, MULTI_BACKGROUND, MULTI_BGCOLOR, MULTI_TCOLOR, NAV_BACKGROUND, NAV_BGCOLOR, NAV_TCOLOR, NOTHING_TO_DO, NUMBER_OF_DBG_ARGUMENTS, NUMBER_OF_TGT_ARGUMENTS, PAR_DBGCOMMAND, PAR_DBGDATABASE, PAR_DBGPASSWORD, PAR_DBGPORT, PAR_DBGSQL_STMT, PAR_DBGUSER, PAR_DEF_DBGDATABASE, PAR_DEF_DBGPASSWORD, PAR_DEF_DBGPORT, PAR_DEF_DBGUSER, PAR_DEF_TGTDATABASE, PAR_DEF_TGTPASSWORD, PAR_DEF_TGTPORT, PAR_DEF_TGTUSER, PAR_LOGOUT, PAR_TGTCOMMAND, PAR_TGTDATABASE, PAR_TGTPASSWORD, PAR_TGTPORT, PAR_TGTSQL_STMT, PAR_TGTUSER, STRBUFF_LEN, URL_BROWSE_DIR_SERVLET, URL_BROWSE_OBJ_SERVLET, URL_DEBUG_BANNER_SERVLET, URL_DEBUG_LOGIN_SERVLET, URL_DEBUG_SERVLET, URL_HELP_DIR_SERVLET, URL_HELP_OBJ_SERVLET, URL_TARGET_BANNER_SERVLET, URL_TARGET_LOGIN_SERVLET, URL_TARGET_SERVLET

Constructor Summary

DebuggeeServlet()

Method Summary

void	**doGet**(javax.servlet.http.HttpServletRequest request, javax.servlet.http.HttpServletResponse response)
protected void	**doLogout**(javax.servlet.http.HttpServletResponse response, javax.servlet.http.HttpSession session)
void	**doPost**(javax.servlet.http.HttpServletRequest request, javax.servlet.http.HttpServletResponse response)
protected void	**doRealProcessRequest**(javax.servlet.http.HttpServletRequest request, javax.servlet.http.HttpServletResponse response, javax.servlet.http.HttpSession session)
protected void	**executeCommand**(java.lang.String[] commandString, java.io.PrintWriter out, java.sql.Connection conn)
protected void	**executeCommandAndDisplay**(java.lang.String[] commandString, java.io.PrintWriter out, java.sql.Connection conn)
void	**init**(javax.servlet.ServletConfig config)
protected void	**loadParameters**(javax.servlet.http.HttpServletRequest request, javax.servlet.http.HttpServletResponse response, javax.servlet.http.HttpSession session)
protected void	**mylog**(java.lang.String str)
protected java.lang.String[]	**parseStmtString**(java.lang.String cmdin)
protected void	**printControlForm**(java.io.PrintWriter out)
protected void	**processRequest**(javax.servlet.http.HttpServletRequest request, javax.servlet.http.HttpServletResponse response)

Methods inherited from class javax.servlet.http.HttpServlet

doDelete, doHead, doOptions, doPut, doTrace, getLastModified, service, service

Methods inherited from class javax.servlet.GenericServlet

destroy, getInitParameter, getInitParameterNames, getServletConfig, getServletContext, getServletInfo, getServletName, init, log, log

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

dout

```
public DbmsOuputToServlet dout
```

---

queryResultVector

```
protected java.util.Vector queryResultVector
```

---

queryResultText

```
protected java.lang.StringBuffer queryResultText
```

---

stmtString

```
protected java.lang.String stmtString
```

---

dummyResult

```
protected java.lang.String dummyResult
```

---

commandTable

```
protected java.util.Hashtable commandTable
```

---

myHelp

```
protected Help myHelp
```

---

myAlter

```
protected Alter myAlter
```

---

myInitialize

```
protected Initialize myInitialize
```

---

myRunProc1

```
protected RunProc1 myRunProc1
```

---

myAttach

```
protected AttachSession myAttach
```

---

mySync

```
protected Sync mySync
```

Constructor Detail

DebuggeeServlet

```
public DebuggeeServlet()
```

Method Detail

init

```
public void init(javax.servlet.ServletConfig config)
          throws javax.servlet.ServletException
```

Specified by:
`init` in interface `javax.servlet.Servlet`
Throws:
`javax.servlet.ServletException`

---

processRequest

```
protected void processRequest(javax.servlet.http.HttpServletRequest request,
                              javax.servlet.http.HttpServletResponse response)
                       throws javax.servlet.ServletException,
                              java.io.IOException
```

Throws:
`javax.servlet.ServletException`
`java.io.IOException`

---

loadParameters

```
protected void loadParameters(javax.servlet.http.HttpServletRequest request,
                              javax.servlet.http.HttpServletResponse response,
                              javax.servlet.http.HttpSession session)
                       throws java.io.IOException,
                              javax.servlet.ServletException
```

Throws:
`java.io.IOException`
`javax.servlet.ServletException`

doGet

```
public void doGet(javax.servlet.http.HttpServletRequest request,
                  javax.servlet.http.HttpServletResponse response)
           throws java.io.IOException,
                  javax.servlet.ServletException
```

Throws:

```
java.io.IOException
javax.servlet.ServletException
```

---

doPost

```
public void doPost(javax.servlet.http.HttpServletRequest request,
                   javax.servlet.http.HttpServletResponse response)
            throws java.io.IOException,
                   javax.servlet.ServletException
```

Throws:

```
java.io.IOException
javax.servlet.ServletException
```

---

doLogout

```
protected void doLogout(javax.servlet.http.HttpServletResponse response,
                        javax.servlet.http.HttpSession session)
                 throws java.io.IOException,
                        javax.servlet.ServletException
```

Throws:

```
java.io.IOException
javax.servlet.ServletException
```

---

doRealProcessRequest

```
protected void doRealProcessRequest(javax.servlet.http.HttpServletRequest request,
                                    javax.servlet.http.HttpServletResponse response,
                                    javax.servlet.http.HttpSession session)
                             throws java.io.IOException,
                                    javax.servlet.ServletException
```

Throws:

```
java.io.IOException
javax.servlet.ServletException
```

---

printControlForm

```
protected void printControlForm(java.io.PrintWriter out)
                         throws java.io.IOException,
                                javax.servlet.ServletException
```

Throws:

```
java.io.IOException
javax.servlet.ServletException
```

parseStmtString

```
protected java.lang.String[] parseStmtString(java.lang.String cmdin)
```

executeCommand

```
protected void executeCommand(java.lang.String[] commandString,
                              java.io.PrintWriter out,
                              java.sql.Connection conn)
```

executeCommandAndDisplay

```
protected void executeCommandAndDisplay(java.lang.String[] commandString,
                                        java.io.PrintWriter out,
                                        java.sql.Connection conn)
```

mylog

```
protected void mylog(java.lang.String str)
              throws java.util.MissingResourceException
```

Throws:
`java.util.MissingResourceException`

deb
Class DebuggerServlet

```
java.lang.Object
  └javax.servlet.GenericServlet
      └javax.servlet.http.HttpServlet
          └deb.DebuggerServlet
```

All Implemented Interfaces:
ICODEdefinitions, java.io.Serializable, javax.servlet.Servlet, javax.servlet.ServletConfig

public class **DebuggerServlet**
extends javax.servlet.http.HttpServlet
implements ICODEdefinitions

See Also:
Serialized Form

Field Summary

protected java.util.Hashtable	**commandTable**
DbmsOuputToServlet	**dout**
protected java.lang.String	**dummyResult**
protected Alter	**myAlter**
protected AttachSession	**myAttach**
protected Help	**myHelp**
protected Initialize	**myInitialize**
protected RunProc1	**myRunProc1**
protected Sync	**mySync**
protected java.lang.StringBuffer	**queryResultText**
protected java.util.Vector	**queryResultVector**
protected java.lang.String	**stmtString**

Fields inherited from interface deb.ICODEdefinitions

ATTR_DBG_DBMS_OUTPUT, ATTR_DBGATTACHED_TO, ATTR_DBGCOMMAND, ATTR_DBGCONNECTION, ATTR_DBGDATABASE, ATTR_DBGUSER, ATTR_TGTCOMMAND, ATTR_TGTCONNECTION, ATTR_TGTDATABASE, ATTR_TGTSESSIONID, ATTR_TGTUSER, BYTEBUF_LEN, CMD_ABORT, CMD_ABOUT, CMD_ALTER_PROC, CMD_ATTACH, CMD_DEBUG_ON, CMD_DEL_BREAK, CMD_DETACH, CMD_EXECUTE, CMD_EXECUTE_QUERY, CMD_EXEMPEL, CMD_GET_PKG_VALUE, CMD_GET_VALUE, CMD_HELP, CMD_INIT, CMD_LIST_BREAK, CMD_PKG_TEST, CMD_PROC_TEST, CMD_PROGRAM, CMD_SELF_CHECK, CMD_SET_BREAK, CMD_SET_PKG_VALUE, CMD_SET_VALUE, CMD_SOURCE_CURRENT, CMD_SOURCE_PKG, CMD_SOURCE_PROC, CMD_SOURCE_TRIG, CMD_STACK, CMD_START_DEBUGGEE, CMD_STEP, CMD_STEP_IN, CMD_STEP_OUT, CMD_STEP_OVER, CMD_SYNC, CMD_TRACE_OUT, CMD_TRIG_TEST, CMD_VERSION, DEFAULT_BACKGROUND, DEFAULT_BGCOLOR, DEFAULT_CONTENT, DEFAULT_JDBC_DRIVER, DEFAULT_PROPERTIES, DEFAULT_PROPERTIES_FILE, DEFAULT_TCOLOR, FRAME_BROWSE, FRAME_HELP, FRAME_MAIN, FRAME_MENU, GIF_ABORT, GIF_ALTER, GIF_ATTACH, GIF_CURR_SOURCE, GIF_DEBUG_OFF, GIF_DEBUG_ON, GIF_DEL_BREAK, GIF_GET_PVALUE, GIF_GET_VALUE, GIF_HELP, GIF_INIT, GIF_LINK_CORUS_ICON, GIF_LINK_ICODE_ICON, GIF_LIST_BREAK, GIF_MENU, GIF_PROG_INFO, GIF_ROOT, GIF_SET_BREAK, GIF_SET_PVALUE, GIF_SET_VALUE, GIF_SOURCE_PKG_BODY, GIF_SOURCE_PROC, GIF_SOURCE_TRIGGER, GIF_STACK, GIF_START_DEBUGGEE, GIF_STEP, GIF_STEP_IN, GIF_STEP_OUT, GIF_STEP_OVER, GIF_SYNC, GIF_TEST_PKG, GIF_TEST_PROC, GIF_TEST_TRIG, GIF_TRACE_OUT, MENU_BACKGROUND, MENU_BGCOLOR, MENU_TCOLOR, MULTI_BACKGROUND, MULTI_BGCOLOR, MULTI_TCOLOR, NAV_BACKGROUND, NAV_BGCOLOR, NAV_TCOLOR, NOTHING_TO_DO, NUMBER_OF_DBG_ARGUMENTS, NUMBER_OF_TGT_ARGUMENTS, PAR_DBGCOMMAND, PAR_DBGDATABASE, PAR_DBGPASSWORD, PAR_DBGPORT, PAR_DBGSQL_STMT, PAR_DBGUSER, PAR_DEF_DBGDATABASE, PAR_DEF_DBGPASSWORD, PAR_DEF_DBGPORT, PAR_DEF_DBGUSER, PAR_DEF_TGTDATABASE, PAR_DEF_TGTPASSWORD, PAR_DEF_TGTPORT, PAR_DEF_TGTUSER, PAR_LOGOUT, PAR_TGTCOMMAND, PAR_TGTDATABASE, PAR_TGTPASSWORD, PAR_TGTPORT, PAR_TGTSQL_STMT, PAR_TGTUSER, STRBUFF_LEN, URL_BROWSE_DIR_SERVLET, URL_BROWSE_OBJ_SERVLET, URL_DEBUG_BANNER_SERVLET, URL_DEBUG_LOGIN_SERVLET, URL_DEBUG_SERVLET, URL_HELP_DIR_SERVLET, URL_HELP_OBJ_SERVLET, URL_TARGET_BANNER_SERVLET, URL_TARGET_LOGIN_SERVLET, URL_TARGET_SERVLET

Constructor Summary

DebuggerServlet()

Method Summary

void	**doGet**(javax.servlet.http.HttpServletRequest request, javax.servlet.http.HttpServletResponse response)

protected void	**doLogout**(javax.servlet.http.HttpServletResponse response, javax.servlet.http.HttpSession session)
void	**doPost**(javax.servlet.http.HttpServletRequest request, javax.servlet.http.HttpServletResponse response)
protected void	**doRealProcessRequest**(javax.servlet.http.HttpServletRequest request, javax.servlet.http.HttpServletResponse response, javax.servlet.http.HttpSession session)
protected void	**executeCommand**(java.lang.String[] commandString, java.io.PrintWriter out, java.sql.Connection conn)
protected void	**executeCommandAndDisplay**(java.lang.String[] commandString, java.io.PrintWriter out, java.sql.Connection conn)
void	**init**(javax.servlet.ServletConfig config)
protected void	**loadParameters**(javax.servlet.http.HttpServletRequest request, javax.servlet.http.HttpServletResponse response, javax.servlet.http.HttpSession session)
protected void	**mylog**(java.lang.String str)
protected java.lang.String[]	**parseStmtString**(java.lang.String cmdin)
protected void	**printControlForm**(java.io.PrintWriter out)
protected void	**processRequest**(javax.servlet.http.HttpServletRequest request, javax.servlet.http.HttpServletResponse response)

Methods inherited from class javax.servlet.http.HttpServlet
doDelete, doHead, doOptions, doPut, doTrace, getLastModified, service, service

Methods inherited from class javax.servlet.GenericServlet
destroy, getInitParameter, getInitParameterNames, getServletConfig, getServletContext, getServletInfo, getServletName, init, log, log

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

dout

`public DbmsOuputToServlet` **`dout`**

---

queryResultVector

`protected java.util.Vector` **`queryResultVector`**

---

queryResultText

`protected java.lang.StringBuffer **queryResultText**`

---

stmtString

`protected java.lang.String stmtString`

---

dummyResult

`protected java.lang.String dummyResult`

---

commandTable

`protected java.util.Hashtable commandTable`

---

myHelp

`protected Help myHelp`

---

myAlter

`protected Alter myAlter`

---

myInitialize

`protected Initialize myInitialize`

---

myRunProc1

`protected RunProc1 myRunProc1`

---

myAttach

`protected AttachSession myAttach`

---

mySync

`protected Sync mySync`

Constructor Detail

DebuggerServlet

```
public DebuggerServlet()
```

Method Detail

init

```
public void init(javax.servlet.ServletConfig config)
          throws javax.servlet.ServletException
```

Specified by:
`init` in interface `javax.servlet.Servlet`

Throws:
`javax.servlet.ServletException`

---

processRequest

```
protected void processRequest(javax.servlet.http.HttpServletRequest request,
                              javax.servlet.http.HttpServletResponse response)
                       throws javax.servlet.ServletException,
                              java.io.IOException
```

Throws:
`javax.servlet.ServletException`
`java.io.IOException`

---

loadParameters

```
protected void loadParameters(javax.servlet.http.HttpServletRequest request,
                              javax.servlet.http.HttpServletResponse response,
                              javax.servlet.http.HttpSession session)
                       throws java.io.IOException,
                              javax.servlet.ServletException
```

Throws:
`java.io.IOException`
`javax.servlet.ServletException`

---

doGet

```
public void doGet(javax.servlet.http.HttpServletRequest request,
                  javax.servlet.http.HttpServletResponse response)
           throws java.io.IOException,
                  javax.servlet.ServletException
```

Throws:
`java.io.IOException`
`javax.servlet.ServletException`

---

doPost

```
public void doPost(javax.servlet.http.HttpServletRequest request,
                   javax.servlet.http.HttpServletResponse response)
            throws java.io.IOException,
                   javax.servlet.ServletException
```

Throws:
    `java.io.IOException`
    `javax.servlet.ServletException`

---

doLogout

```
protected void doLogout(javax.servlet.http.HttpServletResponse response,
                        javax.servlet.http.HttpSession session)
                 throws java.io.IOException,
                        javax.servlet.ServletException
```

Throws:
    `java.io.IOException`
    `javax.servlet.ServletException`

---

doRealProcessRequest

```
protected void doRealProcessRequest(javax.servlet.http.HttpServletRequest request,
                                    javax.servlet.http.HttpServletResponse response,
                                    javax.servlet.http.HttpSession session)
                             throws java.io.IOException,
                                    javax.servlet.ServletException
```

Throws:
    `java.io.IOException`
    `javax.servlet.ServletException`

---

printControlForm

```
protected void printControlForm(java.io.PrintWriter out)
                         throws java.io.IOException,
                                javax.servlet.ServletException
```

Throws:
    `java.io.IOException`
    `javax.servlet.ServletException`

---

parseStmtString

```
protected java.lang.String[] parseStmtString(java.lang.String cmdin)
```

---

executeCommand

```
protected void executeCommand(java.lang.String[] commandString,
                              java.io.PrintWriter out,
                              java.sql.Connection conn)
```

---

executeCommandAndDisplay

```
protected void executeCommandAndDisplay(java.lang.String[] commandString,
                                        java.io.PrintWriter out,
                                        java.sql.Connection conn)
```

---

mylog

```
protected void mylog(java.lang.String str)
              throws java.util.MissingResourceException
```

Throws:
    `java.util.MissingResourceException`

---

deb
Class TgtLogin

```
java.lang.Object
  └ javax.servlet.GenericServlet
      └ javax.servlet.http.HttpServlet
          └ deb.TgtLogin
```

All Implemented Interfaces:
    ICODEdefinitions, java.io.Serializable, javax.servlet.Servlet, javax.servlet.ServletConfig

---

public class **TgtLogin**
extends javax.servlet.http.HttpServlet
implements ICODEdefinitions

See Also:
    Serialized Form

Field Summary

Fields inherited from interface deb.ICODEdefinitions

`ATTR_DBG_DBMS_OUTPUT, ATTR_DBGATTACHED_TO, ATTR_DBGCOMMAND, ATTR_DBGCONNECTION, ATTR_DBGDATABASE, ATTR_DBGUSER, ATTR_TGTCOMMAND, ATTR_TGTCONNECTION, ATTR_TGTDATABASE, ATTR_TGTSESSIONID, ATTR_TGTUSER, BYTEBUFF_LEN, CMD_ABORT, CMD_ABOUT, CMD_ALTER_PROC, CMD_ATTACH, CMD_DEBUG_ON, CMD_DEL_BREAK, CMD_DETACH, CMD_EXECUTE, CMD_EXECUTE_QUERY, CMD_EXEMPEL, CMD_GET_PKG_VALUE, CMD_GET_VALUE, CMD_HELP, CMD_INIT, CMD_LIST_BREAK, CMD_PKG_TEST, CMD_PROC_TEST, CMD_PROGRAM, CMD_SELF_CHECK, CMD_SET_BREAK, CMD_SET_PKG_VALUE, CMD_SET_VALUE, CMD_SOURCE_CURRENT, CMD_SOURCE_PKG, CMD_SOURCE_PROC, CMD_SOURCE_TRIG, CMD_STACK, CMD_START_DEBUGGEE, CMD_STEP, CMD_STEP_IN, CMD_STEP_OUT, CMD_STEP_OVER, CMD_SYNC, CMD_TRACE_OUT, CMD_TRIG_TEST, CMD_VERSION, DEFAULT_BACKGROUND, DEFAULT_BGCOLOR, DEFAULT_CONTENT, DEFAULT_JDBC_DRIVER, DEFAULT_PROPERTIES, DEFAULT_PROPERTIES_FILE, DEFAULT_TCOLOR, FRAME_BROWSE, FRAME_HELP, FRAME_MAIN, FRAME_MENU, GIF_ABORT, GIF_ALTER, GIF_ATTACH, GIF_CURR_SOURCE, GIF_DEBUG_OFF, GIF_DEBUG_ON, GIF_DEL_BREAK,` GIF_GET_PVALUE, GIF_GET_VALUE, GIF_HELP, GIF_INIT, GIF_LINK_CORUS_ICON, GIF_LINK_ICODE_ICON, GIF_LIST_BREAK, GIF_MENU, GIF_PROG_INFO, GIF_ROOT, GIF_SET_BREAK, GIF_SET_PVALUE, GIF_SET_VALUE, GIF_SOURCE_PKG_BODY, GIF_SOURCE_PROC, GIF_SOURCE_TRIGGER, GIF_STACK, GIF_START_DEBUGGEE, GIF_STEP, GIF_STEP_IN, GIF_STEP_OUT, GIF_STEP_OVER, GIF_SYNC, GIF_TEST_PKG, GIF_TEST_PROC, GIF_TEST_TRIG, GIF_TRACE_OUT, MENU_BACKGROUND, MENU_BGCOLOR, MENU_TCOLOR, MULTI_BACKGROUND, MULTI_BGCOLOR, MULTI_TCOLOR, NAV_BACKGROUND, NAV_BGCOLOR, NAV_TCOLOR, NOTHING_TO_DO, NUMBER_OF_DBG_ARGUMENTS, NUMBER_OF_TGT_ARGUMENTS, PAR_DBGCOMMAND, PAR_DBGDATABASE, PAR_DBGPASSWORD, PAR_DBGPORT, PAR_DBGSQL_STMT, PAR_DBGUSER, PAR_DEF_DBGDATABASE, PAR_DEF_DBGPASSWORD, PAR_DEF_DBGPORT, PAR_DEF_DBGUSER, PAR_DEF_TGTDATABASE, PAR_DEF_TGTPASSWORD, PAR_DEF_TGTPORT, PAR_DEF_TGTUSER, PAR_LOGOUT, PAR_TGTCOMMAND, PAR_TGTDATABASE, PAR_TGTPASSWORD, PAR_TGTPORT, PAR_TGTSQL_STMT, PAR_TGTUSER, STRBUFF_LEN, URL_BROWSE_DIR_SERVLET, URL_BROWSE_OBJ_SERVLET, URL_DEBUG_BANNER_SERVLET, URL_DEBUG_LOGIN_SERVLET, URL_DEBUG_SERVLET, URL_HELP_DIR_SERVLET, URL_HELP_OBJ_SERVLET, URL_TARGET_BANNER_SERVLET, URL_TARGET_LOGIN_SERVLET, URL_TARGET_SERVLET

Constructor Summary

TgtLogin()

Method Summary

void	**doGet**(javax.servlet.http.HttpServletRequest request, javax.servlet.http.HttpServletResponse response)
void	**doPost**(javax.servlet.http.HttpServletRequest request, javax.servlet.http.HttpServletResponse response)
void	**init**(javax.servlet.ServletConfig config)
protected void	**mylog**(java.lang.String str)
void	**processRequest**(javax.servlet.http.HttpServletRequest request, javax.servlet.http.HttpServletResponse response)

Methods inherited from class javax.servlet.http.HttpServlet

doDelete, doHead, doOptions, doPut, doTrace, getLastModified, service, service

Methods inherited from class javax.servlet.GenericServlet

destroy, getInitParameter, getInitParameterNames, getServletConfig, getServletContext, getServletInfo, getServletName, init, log, log

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

TgtLogin

```
public TgtLogin()
```

Method Detail

init

```
public void init(javax.servlet.ServletConfig config)
          throws javax.servlet.ServletException
```

Specified by:
init in interface javax.servlet.Servlet

Throws:

```
javax.servlet.ServletException
```

---

doGet

```
public void doGet(javax.servlet.http.HttpServletRequest request,
                  javax.servlet.http.HttpServletResponse response)
           throws java.io.IOException,
                  javax.servlet.ServletException
```

Throws:

```
java.io.IOException
javax.servlet.ServletException
```

---

doPost

```
public void doPost(javax.servlet.http.HttpServletRequest request,
                   javax.servlet.http.HttpServletResponse response)
            throws java.io.IOException,
                   javax.servlet.ServletException
```

Throws:

```
java.io.IOException
javax.servlet.ServletException
```

---

processRequest

```
public void processRequest(javax.servlet.http.HttpServletRequest request,
                           javax.servlet.http.HttpServletResponse response)
                    throws java.io.IOException,
                           javax.servlet.ServletException
```

Throws:

```
java.io.IOException
javax.servlet.ServletException
```

---

mylog

```
protected void mylog(java.lang.String str)
              throws java.util.MissingResourceException
```

Throws:

```
java.util.MissingResourceException
```

Overview Package Class **Tree Deprecated Index Help**

Debug and implementation of a web based PL/SQL debugger using Oracle's debug API

PREV CLASS NEXT CLASS FRAMES NO FRAMES All Classes

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

---

deb

Class TgtUserBanner

```
java.lang.Object
  └─javax.servlet.GenericServlet
      └─javax.servlet.http.HttpServlet
          └─deb.TgtUserBanner
```

All Implemented Interfaces:
ICODEdefinitions, java.io.Serializable, javax.servlet.Servlet, javax.servlet.ServletConfig

---

public class **TgtUserBanner**
extends javax.servlet.http.HttpServlet
implements ICODEdefinitions

Author:
Alvaro Mayorga
See Also:
Serialized Form

Field Summary

Fields inherited from interface deb.ICODEdefinitions

`ATTR_DBG_DBMS_OUTPUT, ATTR_DBGATTACHED_TO, ATTR_DBGCOMMAND, ATTR_DBGCONNECTION, ATTR_DBGDATABASE, ATTR_DBGUSER, ATTR_TGTCOMMAND, ATTR_TGTCONNECTION, ATTR_TGTDATABASE, ATTR_TGTSESSIONID, ATTR_TGTUSER, BYTEBUFF_LEN, CMD_ABORT, CMD_ABOUT, CMD_ALTER_PROC, CMD_ATTACH, CMD_DEBUG_ON, CMD_DEL_BREAK, CMD_DETACH, CMD_EXECUTE, CMD_EXECUTE_QUERY, CMD_EXEMPEL, CMD_GET_PKG_VALUE, CMD_GET_VALUE, CMD_HELP, CMD_INIT, CMD_LIST_BREAK, CMD_PKG_TEST, CMD_PROC_TEST, CMD_PROGRAM, CMD_SELF_CHECK, CMD_SET_BREAK, CMD_SET_PKG_VALUE, CMD_SET_VALUE, CMD_SOURCE_CURRENT, CMD_SOURCE_PKG, CMD_SOURCE_PROC, CMD_SOURCE_TRIG, CMD_STACK, CMD_START_DEBUGGEE, CMD_STEP, CMD_STEP_IN, CMD_STEP_OUT, CMD_STEP_OVER, CMD_SYNC, CMD_TRACE_OUT, CMD_TRIG_TEST, CMD_VERSION, DEFAULT_BACKGROUND, DEFAULT_BGCOLOR, DEFAULT_CONTENT, DEFAULT_JDBC_DRIVER, DEFAULT_PROPERTIES, DEFAULT_PROPERTIES_FILE, DEFAULT_TCOLOR, FRAME_BROWSE, FRAME_HELP, FRAME_MAIN, FRAME_MENU, GIF_ABORT, GIF_ALTER, GIF_ATTACH, GIF_CURR_SOURCE, GIF_DEBUG_OFF, GIF_DEBUG_ON, GIF_DEL_BREAK, GIF_GET_PVALUE, GIF_GET_VALUE, GIF_HELP, GIF_INIT, GIF_LINK_CORUS_ICON, GIF_LINK_ICODE_ICON, GIF_LIST_BREAK, GIF_MENU, GIF_PROG_INFO, GIF_ROOT, GIF_SET_BREAK, GIF_SET_PVALUE, GIF_SET_VALUE, GIF_SOURCE_PKG_BODY, GIF_SOURCE_PROC, GIF_SOURCE_TRIGGER, GIF_STACK, GIF_START_DEBUGGEE, GIF_STEP, GIF_STEP_IN, GIF_STEP_OUT, GIF_STEP_OVER, GIF_SYNC, GIF_TEST_PKG, GIF_TEST_PROC, GIF_TEST_TRIG, GIF_TRACE_OUT, MENU_BACKGROUND, MENU_BGCOLOR, MENU_TCOLOR, MULTI_BACKGROUND, MULTI_BGCOLOR, MULTI_TCOLOR, NAV_BACKGROUND, NAV_BGCOLOR, NAV_TCOLOR, NOTHING_TO_DO, NUMBER_OF_DBG_ARGUMENTS, NUMBER_OF_TGT_ARGUMENTS, PAR_DBGCOMMAND, PAR_DBGDATABASE, PAR_DBGPASSWORD, PAR_DBGPORT, PAR_DBGSQL_STMT, PAR_DBGUSER, PAR_DEF_DBGDATABASE, PAR_DEF_DBGPASSWORD, PAR_DEF_DBGPORT, PAR_DEF_DBGUSER, PAR_DEF_TGTDATABASE, PAR_DEF_TGTPASSWORD, PAR_DEF_TGTPORT, PAR_DEF_TGTUSER, PAR_LOGOUT, PAR_TGTCOMMAND, PAR_TGTDATABASE, PAR_TGTPASSWORD, PAR_TGTPORT, PAR_TGTSQL_STMT, PAR_TGTUSER, STRBUFF_LEN, URL_BROWSE_DIR_SERVLET, URL_BROWSE_OBJ_SERVLET, URL_DEBUG_BANNER_SERVLET, URL_DEBUG_LOGIN_SERVLET, URL_DEBUG_SERVLET, URL_HELP_DIR_SERVLET, URL_HELP_OBJ_SERVLET, URL_TARGET_BANNER_SERVLET, URL_TARGET_LOGIN_SERVLET, URL_TARGET_SERVLET`

Constructor Summary

`TgtUserBanner()`

Method Summary

`protected void`	`doGet(javax.servlet.http.HttpServletRequest request, javax.servlet.http.HttpServletResponse response)` Handles the HTTP `GET` method.
`protected void`	`doPost(javax.servlet.http.HttpServletRequest request, javax.servlet.http.HttpServletResponse response)` Handles the HTTP `POST` method.
`protected void`	`processRequest(javax.servlet.http.HttpServletRequest request, javax.servlet.http.HttpServletResponse response)`

	Processes requests for both HTTP `GET` and `POST` methods.

Methods inherited from class javax.servlet.http.HttpServlet
`doDelete, doHead, doOptions, doPut, doTrace, getLastModified, service, service`

Methods inherited from class javax.servlet.GenericServlet
`destroy, getInitParameter, getInitParameterNames, getServletConfig, getServletContext, getServletInfo, getServletName, init, init, log, log`

Methods inherited from class java.lang.Object
`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Constructor Detail

TgtUserBanner

```
public TgtUserBanner()
```

Method Detail

processRequest

```
protected void processRequest(javax.servlet.http.HttpServletRequest request,
                              javax.servlet.http.HttpServletResponse response)
                       throws javax.servlet.ServletException,
                              java.io.IOException
```

Processes requests for both HTTP `GET` and `POST` methods.
Parameters:
`request` - servlet request
`response` - servlet response
Throws:
`javax.servlet.ServletException`
`java.io.IOException`

---

doGet

```
protected void doGet(javax.servlet.http.HttpServletRequest request,
                     javax.servlet.http.HttpServletResponse response)
              throws javax.servlet.ServletException,
                     java.io.IOException
```

Handles the HTTP `GET` method.
Parameters:
`request` - servlet request
`response` - servlet response
Throws:
`javax.servlet.ServletException`
`java.io.IOException`

---

doPost

```
protected void doPost(javax.servlet.http.HttpServletRequest request,
                      javax.servlet.http.HttpServletResponse response)
               throws javax.servlet.ServletException,
java.io.IOException
```

Handles the HTTP `POST` method.

Parameters:

`request` - servlet request

`response` - servlet response

Throws:

`javax.servlet.ServletException`

`java.io.IOException`

Package deb.command

Class Summary	
Alter	This class is altered alter a session as in `alter session set plsql_debug = true alter {OBJECT_TYPE} {OBJECT_NAME}` Exempel: `alter procedure myProc compile debug`
AttachSession	
Command	Abstract `Command` Class. Base class to extend in order to implement a function in iCODE.
DebugOn	Sets the session's debug mode on. It uses `corus_debug.debug_on`
Exempel	
GetValue	Gets the value of a variable. It uses `corus_debug.get_value`
Help	Displays help information about command in iCODE and the `corus_debug` package. It uses `corus_debug.help.`
Initialize	Initializes the debugee session retrieving and displaying the obtained *debugID* from the database. It uses `corus_debug.initialize.`
Log	
NullCommand	
RunPkg1	
RunProc1	
SelfCheck	
SetBreak	Set a breakpoint at given line.. It uses `corus_debug.set_break.`
SetValue	
ShowSource	
SourceProc	Displays the source code of a given procedure. It uses `corus_debug.show_source_proc.`
Stack	
Step	
StepIn	
StepOut	
StepOver	
Sync	
TraceOut	
Version	

Debug and implementation of a web based PL/SQL debugger using Oracle's debug API

deb.command
Class Alter

```
java.lang.Object
  └ deb.command.Command
      └ deb.command.Alter
```

public class **Alter**
extends Command

This class is altered alter a session as in `alter session set plsql_debug = true` `alter {OBJECT_TYPE} {OBJECT_NAME}`
Exempel: `alter procedure myProc compile debug`

Author:
Alvaro Mayorga

Field Summary

`protected java.sql.CallableStatement`	**cstmt**
`protected java.lang.String`	**help**
`protected java.lang.String`	**info**
`protected java.sql.Statement`	**stmt**
`protected java.util.Vector`	**vector**

Fields inherited from class deb.command.Command
`logFile`

Constructor Summary

Alter()

Method Summary

`protected boolean`	**checkValidity**(java.lang.String[] cmdArray) Should perform data control of parameters to the command.
`protected void`	**mylog**(java.lang.String str)
`java.util.Vector`	**performCommand**(java.lang.String[] cmdArrayin, java.sql.Connection myCon) The actual command execution.
`java.util.Vector`	**performCommandAndDisplay**(java.lang.String[] cmdArrayin,

	`java.sql.Connection connection, java.io.PrintWriter out)` The actual command execution.

Methods inherited from class deb.command.Command
`ifValidThenPerformCommand, info`

Methods inherited from class java.lang.Object
`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Field Detail

info

```
protected java.lang.String info
```

---

help

```
protected java.lang.String help
```

---

vector

```
protected java.util.Vector vector
```

---

stmt

```
protected java.sql.Statement stmt
```

---

cstmt

```
protected java.sql.CallableStatement cstmt
```

Constructor Detail

Alter

```
public Alter()
```

Method Detail

mylog

```
protected void mylog(java.lang.String str)
              throws java.util.MissingResourceException
```

Overrides:
`mylog` in class `Command`

Throws:

```
java.util.MissingResourceException
```

---

checkValidity

```
protected boolean checkValidity(java.lang.String[] cmdArray)
```

Description copied from class: `Command`
Should perform data control of parameters to the command.
Specified by:
`checkValidity` in class `Command`
Parameters:
`cmdArray` - Array containing *command keyword, modifier(s) and , parameters*
Returns:
Returns `true` if valid else `false`.

---

performCommand

```
public java.util.Vector performCommand(java.lang.String[] cmdArrayin,
                                      java.sql.Connection myCon)
```

Description copied from class: `Command`
The actual command execution.
Specified by:
`performCommand` in class `Command`
Parameters:
`cmdArrayin` - Array with *command keyword, modifier(s) and , parameters*
`myCon` - A Connection to execute stored procedures in the database within the *correct* session.
Returns:
Returns a `java.util.Vector` containing the results of executions as objects

---

performCommandAndDisplay

```
public java.util.Vector performCommandAndDisplay(java.lang.String[] cmdArrayin,
                                                java.sql.Connection connection,
                                                java.io.PrintWriter out)
```

Description copied from class: `Command`
The actual command execution. Displays also the output generated by DBMS_OUTPUT on the DB.
Overrides:
`performCommandAndDisplay` in class `Command`
Parameters:
`cmdArrayin` - Array with *command keyword, modifier(s) and , parameters*
`connection` - A Connection to execute stored procedures in the database within the *correct* session.
Returns:
Returns a `java.util.Vector` containing the results of executions as objects

---

---

deb.command
Class AttachSession

```
java.lang.Object
  └ deb.command.Command
      └ deb.command.AttachSession
```

public class **AttachSession**
extends Command

Field Summary	
protected java.sql.CallableStatement	**cstmt**
protected java.lang.String	**help**
protected java.lang.String	**info**
protected java.sql.Statement	**stmt**
protected java.util.Vector	**vector**

Fields inherited from class deb.command.Command
logFile

Constructor Summary
AttachSession()

Method Summary	
protected boolean	**checkValidity**(java.lang.String[] cmdArray) Returns true in order to display an error message
java.util.Vector	**performCommand**(java.lang.String[] cmdArray, java.sql.Connection myCon) The actual command execution.
java.util.Vector	**performCommandAndDisplay**(java.lang.String[] cmdArray, java.sql.Connection connection, java.io.PrintWriter out) The actual command execution.

Methods inherited from class deb.command.Command
ifValidThenPerformCommand, info, mylog

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

info

```
protected java.lang.String info
```

---

help

```
protected java.lang.String help
```

---

stmt

```
protected java.sql.Statement stmt
```

---

cstmt

```
protected java.sql.CallableStatement cstmt
```

---

vector

```
protected java.util.Vector vector
```

Constructor Detail

AttachSession

```
public AttachSession()
```

Method Detail

checkValidity

```
protected boolean checkValidity(java.lang.String[] cmdArray)
```

Returns true in order to display an error message

Specified by:

`checkValidity` in class `Command`

Parameters:

`cmdArray` - Array containing *command keyword, modifier(s) and , parameters*

Returns:

Returns `true` if valid else `false`.

---

performCommand

```
public java.util.Vector performCommand(java.lang.String[] cmdArray,
                                       java.sql.Connection myCon)
```

Description copied from class: `Command`

The actual command execution.

Specified by:

`performCommand` in class `Command`

Parameters:

`cmdArray` - Array with *command keyword, modifier(s) and , parameters*

`myCon` - A Connection to execute stored procedures in the database within the *correct* session.

Returns:

Returns a `java.util.Vector` containing the results of executions as objects

---

performCommandAndDisplay

```
public java.util.Vector performCommandAndDisplay(java.lang.String[] cmdArray,
                                                  java.sql.Connection connection,
                                                  java.io.PrintWriter out)
```

Description copied from class: `Command`
The actual command execution. Displays also the output generated by DBMS_OUTPUT on the DB.
Overrides:
`performCommandAndDisplay` in class `Command`
Parameters:
`cmdArray` - Array with *command keyword, modifier(s) and , parameters*
`connection` - A Connection to execute stored procedures in the database within the *correct* session.
Returns:
Returns a `java.util.Vector` containing the results of executions as objects

deb.command
Class Command

```
java.lang.Object
  └ deb.command.Command
```

Direct Known Subclasses:
Alter, AttachSession, DebugOn, Exempel, GetValue, Help, Initialize, NullCommand, RunPkg1, RunProc1, SelfCheck, SetBreak, SetValue, ShowSource, SourceProc, Stack, Step, StepIn, StepOut, StepOver, Sync, TraceOut, Version

public abstract class **Command**
extends java.lang.Object

Abstract `Command` Class. Basic implementation of a Command superclass.

Field Summary	
`protected java.lang.String`	**`className`**
`protected java.lang.String`	**`help`**
`protected java.lang.String`	**`info`**
`protected java.lang.String`	**`logFile`**

Constructor Summary	
`protected`	**`Command`**`()` Sole constructor.

Method Summary

protected abstract boolean	**checkValidity**(java.lang.String[] cmdArray) Should perform data control of parameters to the command.
protected java.util.Vector	**ifValidThenPerformCommand**(java.lang.String[] cmdArray, java.sql.Connection con) If parameters are valid then perform the command execution.
java.lang.String	**info**() Returns info
protected void	**mylog**(java.lang.String str)
abstract java.util.Vector	**performCommand**(java.lang.String[] cmdArray, java.sql.Connection con) The actual command execution.
java.util.Vector	**performCommandAndDisplay**(java.lang.String[] cmdArray, java.sql.Connection connection, java.io.PrintWriter out) The actual command execution.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

className

`protected java.lang.String className`

---

info

`protected java.lang.String info`

---

help

`protected java.lang.String help`

---

logFile

`protected java.lang.String logFile`

Constructor Detail

Command

`protected Command()`

Sole constructor.

Method Detail

mylog

```
protected void mylog(java.lang.String str)
              throws java.util.MissingResourceException
```

Throws:
`java.util.MissingResourceException`

---

info

```
public java.lang.String info()
```

Returns info
Returns:
Returns `info`

---

checkValidity

```
protected abstract boolean checkValidity(java.lang.String[] cmdArray)
```

Should perform data control of parameters to the command.
Parameters:
`cmdArray` - Array containing *command keyword, modifier(s) and , parameters*
Returns:
Returns `true` if valid else `false`.

---

performCommand

```
public abstract java.util.Vector performCommand(java.lang.String[] cmdArray,
                                                java.sql.Connection con)
```

The actual command execution.
Parameters:
`cmdArray` - Array with *command keyword, modifier(s) and , parameters*
`con` - A Connection to execute stored procedures in the database within the *correct* session.
Returns:
Returns a `java.util.Vector` containing the results of executions as objects

---

performCommandAndDisplay

```
public java.util.Vector performCommandAndDisplay(java.lang.String[] cmdArray,
                                                 java.sql.Connection connection,
                                                 java.io.PrintWriter out)
```

The actual command execution. Displays also the output generated by DBMS_OUTPUT on the DB.
Parameters:
`cmdArray` - Array with *command keyword, modifier(s) and , parameters*
`connection` - A Connection to execute stored procedures in the database within the *correct* session.
Returns:
Returns a `java.util.Vector` containing the results of executions as objects

---

ifValidThenPerformCommand

```
protected java.util.Vector ifValidThenPerformCommand(java.lang.String[] cmdArray,
                                                     java.sql.Connection con)
```

If parameters are valid then perform the command execution.
Parameters:
`cmdArray` - Array containig `command keyword, modifiers` and `parameters` for command execution
`con` - Database connection in which the command should be performed

deb.command
Class DebugOn

```
java.lang.Object
  └ deb.command.Command
      └ deb.command.DebugOn
```

public class **DebugOn**
extends Command

Author:
Alvaro Mayorga

Field Summary	
protected java.lang.String	**help**
protected java.lang.String	**info**

Fields inherited from class deb.command.Command
`logFile`

Constructor Summary
DebugOn()

Method Summary	
protected boolean	**checkValidity**(java.lang.String[] cmdArray) Returns true in order to display an error message
java.util.Vector	**performCommand**(java.lang.String[] cmdArray, java.sql.Connection myCon) The actual command execution.
java.util.Vector	**performCommandAndDisplay**(java.lang.String[] cmdArray, java.sql.Connection connection, java.io.PrintWriter out) The actual command execution.

Methods inherited from class deb.command.Command
`ifValidThenPerformCommand, info, mylog`

Methods inherited from class java.lang.Object
`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Field Detail

info

```
protected java.lang.String info
```

---

help

```
protected java.lang.String help
```

Constructor Detail

DebugOn

```
public DebugOn()
```

Method Detail

checkValidity

```
protected boolean checkValidity(java.lang.String[] cmdArray)
```

Returns true in order to display an error message
Specified by:
`checkValidity` in class `Command`
Parameters:
`cmdArray` - Array containing *command keyword, modifier(s) and , parameters*
Returns:
Returns `true` if valid else `false`.

---

performCommand

```
public java.util.Vector performCommand(java.lang.String[] cmdArray,
                                       java.sql.Connection myCon)
```

Description copied from class: `Command`
The actual command execution.
Specified by:
`performCommand` in class `Command`
Parameters:
`cmdArray` - Array with *command keyword, modifier(s) and , parameters*
`myCon` - A Connection to execute stored procedures in the database within the *correct* session.
Returns:
Returns a `java.util.Vector` containing the results of executions as objects

---

performCommandAndDisplay

```
public java.util.Vector performCommandAndDisplay(java.lang.String[] cmdArray,
                                                 java.sql.Connection connection,
```

`java.io.PrintWriter out)`

Description copied from class: `Command`
The actual command execution. Displays also the output generated by DBMS_OUTPUT on the DB.
Overrides:
`performCommandAndDisplay` in class `Command`
Parameters:
`cmdArray` - Array with *command keyword, modifier(s) and , parameters*
`connection` - A Connection to execute stored procedures in the database within the *correct* session.
Returns:
Returns a `java.util.Vector` containing the results of executions as objects

---

---

deb.command
Class Command

```
java.lang.Object
  └ deb.command.Command
```

Direct Known Subclasses:
Alter, AttachSession, DebugOn, Exempel, GetValue, Help, Initialize, NullCommand, RunPkg1, RunProc1, SelfCheck, SetBreak, SetValue, ShowSource, SourceProc, Stack, Step, StepIn, StepOut, StepOver, Sync, TraceOut, Version

---

public abstract class **Command**
extends java.lang.Object

Abstract `Command` Class. Basic implementation of a Command superclass.

Field Summary

`protected java.lang.String`	**`className`**
`protected java.lang.String`	**`help`**
`protected java.lang.String`	**`info`**
`protected java.lang.String`	**`logFile`**

Constructor Summary

`protected`	**`Command`**`()` Sole constructor.

Method Summary

`protected`	

abstract boolean	checkValidity(java.lang.String[] cmdArray) Should perform data control of parameters to the command.
protected java.util.Vector	ifValidThenPerformCommand(java.lang.String[] cmdArray, java.sql.Connection con) If parameters are valid then perform the command execution.
java.lang.String	info() Returns info
protected void	mylog(java.lang.String str)
abstract java.util.Vector	performCommand(java.lang.String[] cmdArray, java.sql.Connection con) The actual command execution.
java.util.Vector	performCommandAndDisplay(java.lang.String[] cmdArray, java.sql.Connection connection, java.io.PrintWriter out) The actual command execution.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

className

```
protected java.lang.String className
```

---

info

```
protected java.lang.String info
```

---

help

```
protected java.lang.String help
```

---

logFile

```
protected java.lang.String logFile
```

Constructor Detail

Command

```
protected Command()
```

Sole constructor.

Method Detail

mylog

```
protected void mylog(java.lang.String str)
              throws java.util.MissingResourceException
```

Throws:
`java.util.MissingResourceException`

---

info

```
public java.lang.String info()
```

Returns info

Returns:
Returns `info`

---

checkValidity

```
protected abstract boolean checkValidity(java.lang.String[] cmdArray)
```

Should perform data control of parameters to the command.

Parameters:
`cmdArray` - Array containing *command keyword, modifier(s) and , parameters*
Returns:
Returns `true` if valid else `false`.

---

performCommand

```
public abstract java.util.Vector performCommand(java.lang.String[] cmdArray,
                                                java.sql.Connection con)
```

The actual command execution.

Parameters:
`cmdArray` - Array with *command keyword, modifier(s) and , parameters*
`con` - A Connection to execute stored procedures in the database within the *correct* session.
Returns:
Returns a `java.util.Vector` containing the results of executions as objects

---

performCommandAndDisplay

```
public java.util.Vector performCommandAndDisplay(java.lang.String[] cmdArray,
                                                 java.sql.Connection connection,
                                                 java.io.PrintWriter out)
```

The actual command execution. Displays also the output generated by DBMS_OUTPUT on the DB.

Parameters:
`cmdArray` - Array with *command keyword, modifier(s) and , parameters*
`connection` - A Connection to execute stored procedures in the database within the *correct* session.
Returns:
Returns a `java.util.Vector` containing the results of executions as objects

---

ifValidThenPerformCommand

```
protected java.util.Vector ifValidThenPerformCommand(java.lang.String[] cmdArray,
                                                     java.sql.Connection con)
```

If parameters are valid then perform the command execution.

Parameters:

`cmdArray` - Array containig `command keyword, modifiers` and `parameters` for command execution

`con` - Database connection in which the command should be performed

---

Note: The remaining javadoc documents of the `deb.command` package has mainly the same class documentation as the `AttachSession` and the `DebugOn` classes.
All of them extends the `Command` class.
These documents have not been inserted in this appendix because the information they provide is mostly repetitive.

The full documentation can be requested from the author.