# Chapter2 Background

Software debugging and visualization manifest the dynamic behaviors of programs. Debugging is the process to find and correct errors in a program. Software visualization can be used to understand software or to help program debugging Software debugging and visualization may be combined in a single tool (e.g. Lens[14, 15] and Leonardo[4]).

This section describes the background knowledge of debuggers and software visualization. Debuggers and software visualization tools designed for procedure or object-oriented languages such as C, C++, and Java are surveyed. Tools built for other languages (such as Prolog or Lisp) are ignored in this thesis. The language domain can result in a difference in the visualization problems for data structure and program executions. Besides, the general-purpose program animation tools are described.

This section is divided into four parts. Section 2.1 explains what the debugger is, and lists some debuggers related to our works. Section 2.2 describes the definition of software visualization and lists some surveyed tools. Section 2.3 and Section 2.4 describe the implement technology. Section 2.3 describes the related XML background knowledge and the related JAVA technology explained in section 2.4.

## 2.1 Debugger

Jonathan B. Rosenberg defined a debugger as a tool to help track down, isolate, and remove bugs from software programs. In truth, debuggers are tools to illuminate the dynamic nature of a program in order to understand it as well as find and fix defects[18]. Software debugging is a critical part of the software life cycle. It is the process of the finding, analyzing, and fixing bugs. Adam Kolawa writes that it is not uncommon for the debugging phase of software development to take 60-70% of the overall development time. In fact, debugging is responsible for 80% of all software project overruns [9]. In other words, a lot of programmers spend much time debugging during their development time. Therefore, they need an effective debugger to help them debugging in less efforts.

The basic functions of a debugger are tracing the program step by step, stopping and continuing at breakpoints, and printing the variable values. Besides these basic functions, some debuggers (e.g.[2, 4, 10]) provide advanced functions such as data breakpoint or reverse execution.

Debuggers are hard to build because they depend heavily on the underlying capabilities of operation systems. Figure 2-1 shows the typical debugger architecture.

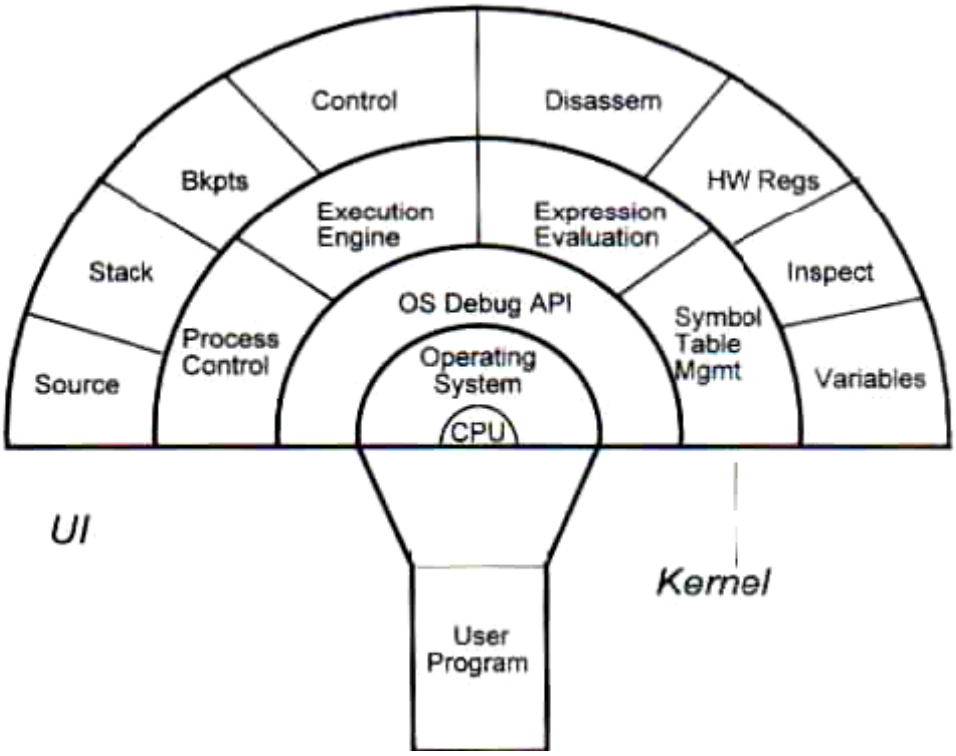


Figure 2-1 : One typical debugger architecture [18]

In the following, each surveyed debugger is brief presented.

### 2.1.1 GDB

The GDB is the GNU Project debugger, which allows programmers to see what is going on “inside” a program while it executes -- or what the program was doing at the moment it crashes [19]. The debugged program can be written in non-interpreted languages such as C, C++, Pascal, or Perl. GDB can run on most popular UNIX and Microsoft Windows variants[19].

There are still several basic debuggers in text mode like GDB, for example: dbx, JDB. Typically, well-known debuggers such as GDB, DBX, JDB only provide a basic set of debugging commands. They serve as a basis for most GUI debuggers, so

it is difficult to use these debuggers to make complex query. A typical debugging process always needs to repeat similar steps such as set breakpoints and print variable values until the cause of defect is found.

### 2.1.2 DDD

DDD (Data Display Debugger) is a front-end to a command-line debugger [22] such as GDB, DBX, WDB, Ladebug, JDB, XDB, the Perl debugger, the bash debugger, or the Python debugger. It is served to display graphically and debug data structures without adding additional code. It is a free alternative to commercial debuggers that are bundled with the integrated development environment.

There are two biggest advantages of DDD. One of the advantages is its ability to send commands to GDB by menus instead of typing as shown in Figure 2-2. Another is its ability to graphically display the debugged program’s data by the data window and allow the user change layout like compact layout as shown in Figure 2-3. However, it has only one layout to illustrate data structure such as graph, tree, or linked-list. It is almost limited to illustrate structures or classes with boxes and the pointer with arrow as well as plotting array data using the plotting program gnuplot.

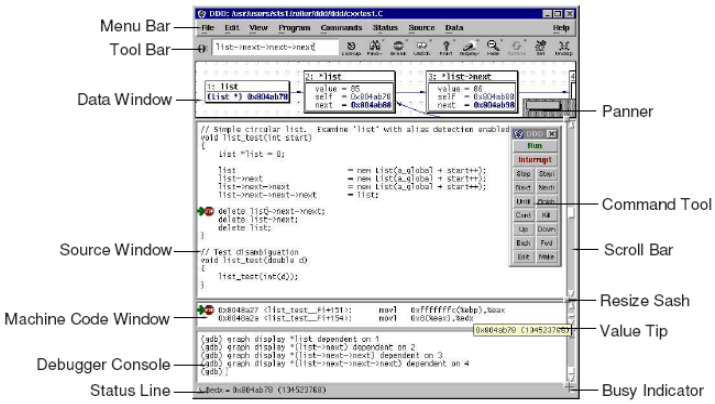


Figure 2-2 : the Data Display Debugger [22]

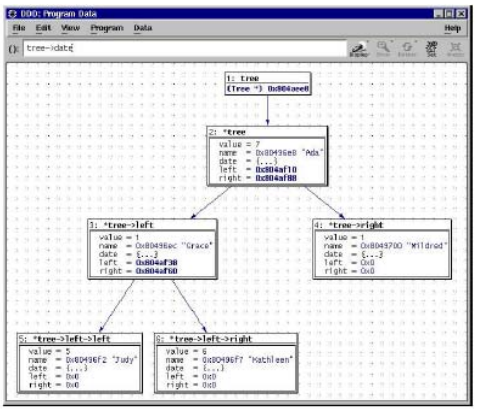

Figure 2-3 : compact layout in DDD [22]

### 2.1.3 JDB

Our system is implemented in java. In java, the typical debugger is JDB. It is a simple command-line debugger for Java classes. It is a demonstration of the Java Platform Debugger Architecture that provides inspection and debugging of a local or remote Java Virtual Machine.

Just like GDB in text mode, it provides a basic set of debugging commands. So it is difficult to use these debuggers to make complex query.

### 2.1.4 ODB

ODB (the Omniscient Debugger) [10] as the Figure 2-4 shows records all program state over the entire program in order to allow the user to explore the desired history of that program runs. ODB supports stepping the program backwards and forwards through the execution history. ODB shows the threads in the program, their stack, the tree of executed methods, and a treelike view of selected objects and

objects they refer to.

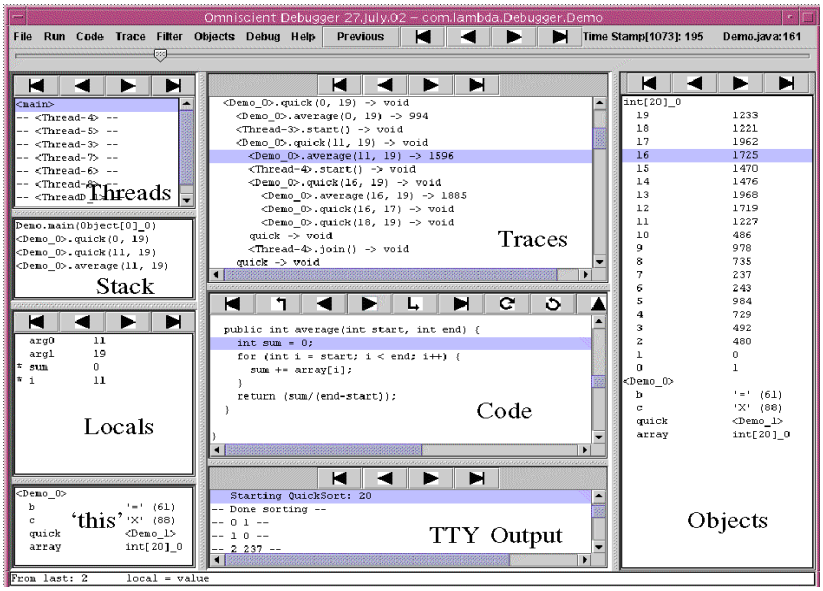


Figure 2-4 : ODB [10]

Although ODB make it easy and efficient to debug the unknown program, there may be a little problem about performance as what Bil Lewis wrote. Performance is not an issue for bugs which generate less than 10 million events. For programs that depend on the garbage collection, this is an issue [10].

## 2.2 Software Visualization

According to the “A Principled Taxonomy of Software Visualization” [17], Software Visualization is defined as the use of the crafts of typography, graphic design, animation, and cinematography with modern human-computer interaction technology to facilitate both the human understanding and effective use of computer software [17]. Software visualization is an aid for program understanding, debugging and teaching of algorithms. A typical approach to understanding the

behaviors of a program is to trace the program step by step through a debugger. During the tracing, a programmer builds up a mental flowchart of a program execution. It may hardly work in a large program. If there is a software visualization tool, a programmer can easily and quickly understand a program since it is much easier for humans to comprehend a concept via pictures like an old saying - “ A picture is worth a thousand words”

According to the definition of software visualization, program animation is a kind of the software visualization that is concerned with dynamic graphical displays of a program’s execution. In other words, program animation means a serial of graphical representations of a data structure at continuous state. It is used to show the behaviors of a program. Some program animation systems are used for teaching, but some are for programming debugging. Some program animation systems are designed for specific purpose, but some are for general purpose. Some program animation systems execute without the user’s extra efforts, but some may need users to configure or implement animation.

In the following section will describe some surveyed systems briefly.

### 2.2.1 Lens

The Lens [14, 15] seeks to bridge the two domain of program visualization and algorithm animation. Lens provides application-specific animation views for

debugging purpose. Len is integrated with a system debugger to promote iterative design and exploration [15]. It extends XTango [20] by combining algorithm animation into the dbx source-level debugger.

Lens allows programmers to build rapidly (in minutes) algorithm animation-style program views without requiring any sophisticated graphics knowledge and without textual coding [15]. Creating visualization from Lens requires some extra work. Before running program animation, a user needs to create graphical objects (lines, rectangles, text and object arrays, for instance) and chooses one of the six action commands (such as Move, Fill, Exchange...and so on) that apply these objects to the source code. The actions are defined by a graphical editor, as shown in Figure 2-5. Therefore, Lens has some limited execution control facilities.

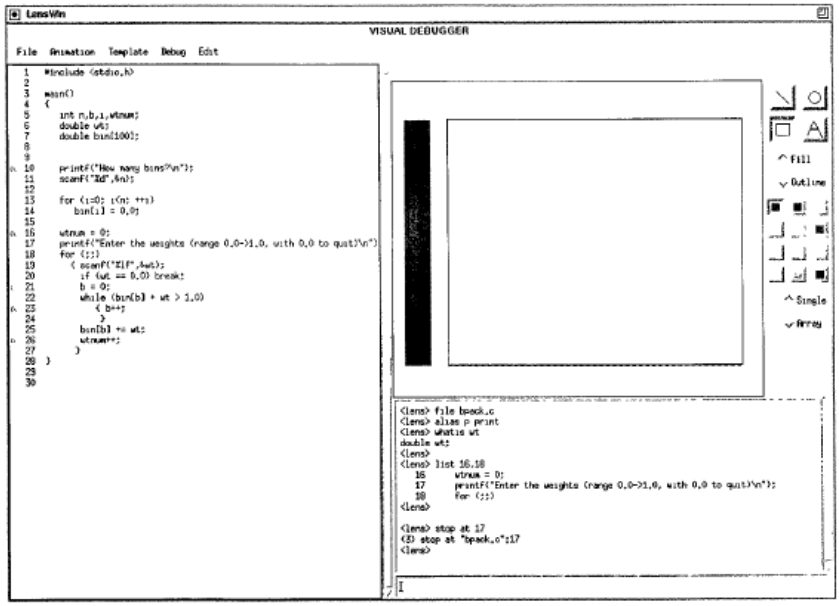

Figure 2-5 : Lens display presented to programmer. [15].
The left section shows source with animation annotations, and the right section contains the graphical editor and debugger command window.

### 2.2.2 Leonardo

The *Leonardo* software visualization environment allows the users to edit, compile, execute and animate general-purpose C programs[4]. Visualization rules are written in the logic visualization language *Alpha* embedded in the C program as comments. Programmers can create many types of visualizations containing geometric primitives or graphs by *Alpha*. However, before executing the *Leonardo*, users need to learn the *Alpha* language. Because the system is based on a virtual CPU and the Alpha-based declarative approach, reversible execution can work without extra effort.

### 2.2.3 JAN

JAN[11] is an animation system for program understanding rather than debugging. The animation is based on UML including object and sequence diagrams. Besides, the animation can be customized by annotating the code with visualization directives and by interactively adapting the visual appearance to the user's personal taste[11]. All annotations are written in the form of Javadoc comments with special tags (e.g. @show, @component). So a programmer has to design how to animate the program during and after program development. However, it is hard for UML to

describe dynamically the internal behaviors of a program such as an assignment statement.

### 2.2.4 JaVis

The JaVis[12] is developed for visualizing and debugging concurrent java programs. Like JAN, the visualization is based on UML. The JaVis system includes two parts. The fist part is tracing component implemented by JDI to generate traces from running program and detect deadlock. The second part is UML-based visualization component to visualize the traces and the deadlock events.

The Java Platform Debugger Architecture provides the Java Debug Interface implemented by each Java Virtual Machine. The JDI allows to collect debugging and tracing information from a running Java program without modifying the source code[12]. After being collected, the traces are analyzed to detect deadlocks. The deadlock is visualized with Unified Modeling Language (UML) collaboration diagram and the traces are visualized with UML sequence diagram. These diagrams are generated by the Together Open API.

The standard for visual modeling of object-oriented systems is the UML[12]. Therefore, the user easily understands the visualization meanings without extra work.

### 2.2.5 Jeliot 3

Jeliot 3 is a program visualization tool that is designed to help beginners to learn Java program. Jeliot 3 extends Jeliot 2000 to visualize object oriented concepts. The key feature of Jeliot is the fully or semi-automatic visualization of the data and control flows [13]. Unlike the Lens, the Jeliot 3 does not let the user create any graphic objects and design the animation action before animation executing. The Jeliot 3 allows user to animate general-purpose Java programs, but the data structure of the execution program does not support the complex type such as two-dimension array. Besides, users can tune animation speed to let it slower or faster.

The animation frame in Jeliot 3 is divided into four parts as shown in Figure 2-6. Therefore, the appearance of the visualizations is close to the Java language specification. It is indeed helpful to learn Java program, but not to fit for understanding the behavior of the program graphically. As the user interface in Jeliot 3 is shown in Figure 2-7, all the graphical presentation is still text-base description.

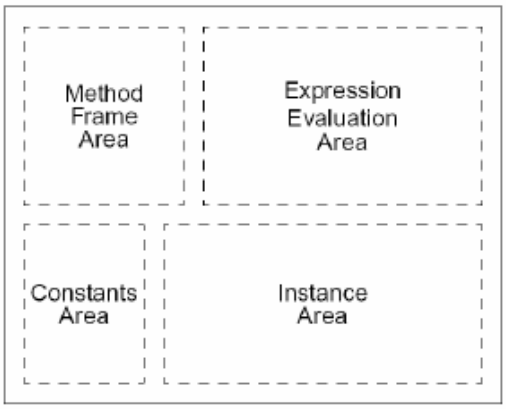


Figure 2-6 : the structure of the animation frame in Jeliot 3[13]

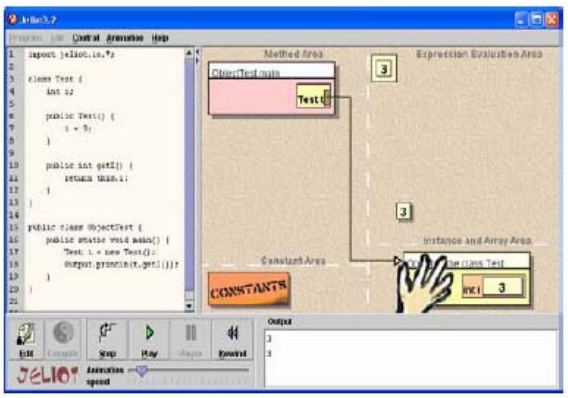

Figure 2-7 : the user interface in Jeliot 3[13]

### 2.2.6 JIVE

A novel approach to the runtime visualization and analysis of objected-oriented programs is presented and illustrated through a prototype system called JIVE: Java Interactive Visualization Environment. JIVE is implemented using the Java Platform Debugger Architecture. [7, 8] JIVE supports multiple views of execution, so the current execution state of the program is represented in a variety of diagram forms for better clarity. The diagram forms include the UML sequence diagram, tree-like diagram. Besides, the JIVE supports backward execution. Visualization can be executed backward through the execution history stored into a database. So the JIVE does not allow the user to change program input or variables' value.

### 2.2.7 JSAVE

JSAVE is an algorithm visualization tool for the Java Collection Framework

that addresses both the requirements for effective visualization tools and the deficiencies that have prevented their widespread use. [21] JSAVE provides a set of classes that generate their own visualization. It may implement each data structure being used. Therefore, it maybe limited to basic data structures and do not work with the data structures that have not been implemented in the visualization library. The user does not need to implement and design how to visualize an algorithm.

All visualizations within JSAVE can be saved as XML scripts. These scripts can then be loaded within the JSAVE Visualizer, allowing them to be viewed as visualizations no differently than if they were being generated by a live program. [21]

### 2.2.8 Zeus

The Zeus algorithm animation system is noteworthy for its use of objects, strong-typing, parallelism, and graphical development of views[3]. Zeus is implemented by Modula-2 code and primarily designed for teaching and instruction. The Zeus provides facilities for users to view and interact with an animation, and for programmers to develop such animations. The Zeus allows user change a data item’s representation, once the animation is stopped. All of the other views are updated following the change, and it dose so with the changed data if the program proceeds. Before a program animating, a programmer has to design how to animate a program.

Therefore, one animation is designed for one program.

## 2.3 XML

Extensible Markup Language (XML) is a general-purpose markup language for creating special-purpose markup languages to describe data. XML provides a description to characterize a tree-based structure as text-based information. XML tags are defined by users through XML schema.

XML schema is a language for describing the structure and constraining the contents of XML documents. A number of standard and proprietary XML schema languages have emerged for the purpose of formally expressing such schemas such as DTD and XML Schema.

Document Type Definition (DTD) is one kind of XML schema. DTD is used to define legal building blocks of an XML document and the document structure with a list of legal elements. The main building blocks of both XML and HTML documents are tags like <body>....</body>. All XML documents are made up by the following simple building blocks: Elements, Attributes, Entities, PCDATA, and CDATA.

The DTD we design is only described in two building blocks, element and PCDATA. Therefore, the following is a brief explanation of the two building blocks:

Element

Element declarations name the allowable set of elements within the document,

and specify whether and how declared elements and order of character data may be contained within each element. The declaration format is as following:

<!ELEMENT element_name (element_or_data)>

element_name : the tag name of XML element

element_or_data : declared elements or data

PCDATA

PCDATA means parsed character data. Think of character data as the text found between the start tag and the end tag of an XML element. PCDATA is text that will be parsed by a parser. Elements with only character data are declared with #PCDATA inside parentheses. The description and examples shown in following:

**<!ELEMENT element-name (#PCDATA)>**

One example:

<!ELEMENT from (#PCDATA)>

XML shows:

<from> Taipei </from>

# 2.4 Related Technology

## 2.4.1 JPDA

Java Platform Debugger Architecture (JPDA) is the debugging support for the Java 2 Platform. JPDA provides the infrastructure to build end-user debugger applications. JPDA consists of three layered modules. Details of the modular structure of JPDA are discussed below. Figure 2-8 shows the relationship of the

modules.

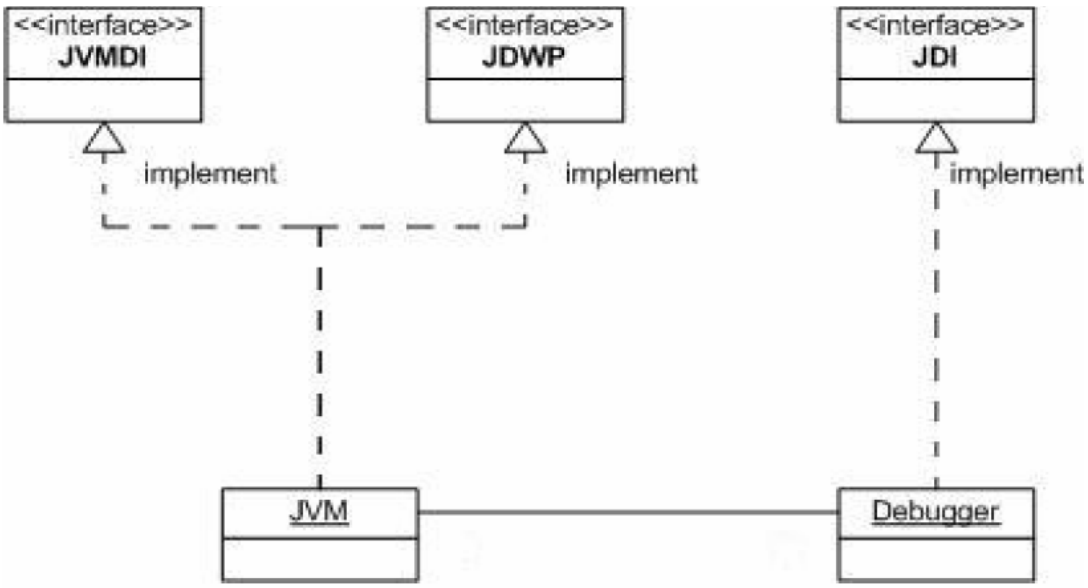


Figure 2-8 : relations of these models

• JVMDI: Java Virtual Machine Debug Interface. It is a low-level native interface and defines the services a Java virtual machine must provide for debugging.

• JDWP: Java Debug Wire Protocol. It defines the format of information and requests transferred between the debugger process and the debuggee process.

• JDI: Java Debug Interface. It is a high-level Java programming language interface including support for remote debugging. Following are packages defined in JDI.

**JVMDI Interface**

It is a native interface to be implemented by any JVM (Java Virtual Machine) builders, supplied as a native shared library (jdwp.so, jdwp.dll, ...). It defines the services of JVM which must be implemented for debugging. It includes requests (for

example, to ask information of current stack), actions (for example, setting a breakpoint), and notification (for example, when a breakpoint has been hit). So, to implement a JVM, typically written in C, programmers should include the following header file in this source code.

#include <jvmdi.h>

There are JVMDI functions can be accessed through a function table. The JVMDI function table can be obtained through the JNI GetEnv function. For example, the following code access the function table. JVMDI_Interface_1 *jvmdi;

... (*jvm)->GetEnv(jvm, &jvmdi, JVMDI_VERSION_1);

**JDWP Interface**

The Java Debug Wire Protocol (JDWP) describes the format of debugging information and requests between a debuggee and a debugger. In JPDA, there is a communication channel between the front-end (in the debugger process) and the back-end (in the debuggee process) - the format of data flowing on this channel is described by JDWP. In the implementation of JPDA, the implementation of the back-end serves at the debuggee side of this channel, and the implementation of the front-end (a Java programming language component of the J2SE SDK, located in tools.jar) serves at the debugger side.

**JDI Interface**

A 100% Java interface. It defines information and requests at a user code level.

We can use JDI to implement a debugger and it will automatically work with the JVM on the platforms that Sun supports. It can also work with another company's JVM those implements JDWP and JVMDI.

The Java Debug Interface (JDI) provides a pure Java programming language interface for debugging Java applications. It is typically implemented by a debugger-like application (e.g., IDE, debugger, tracer, monitoring tool ...).

### 2.4.2 JXDB

The Java Architecture for XML Binding (JAXB), available in the Java Web Services Developer Pack developed by Sun Microsystems, is part of the integrated toolkit for Web services development. JAXB provides a fast and convenient way to bind between XML schemas and representations in java codes. A Java class name is derived from the XML element name, or specified by a binding customization. JAXB also provides a way generate XML schema from Java objects. JAXB makes it easy for programmers to handle XML data in Java applications. Figure 2-9 shows the JAXB binding process. As part of this process, JAXB provides methods for unmarshalling XML documents into Java objects and marshalling Java objects into XML documents.

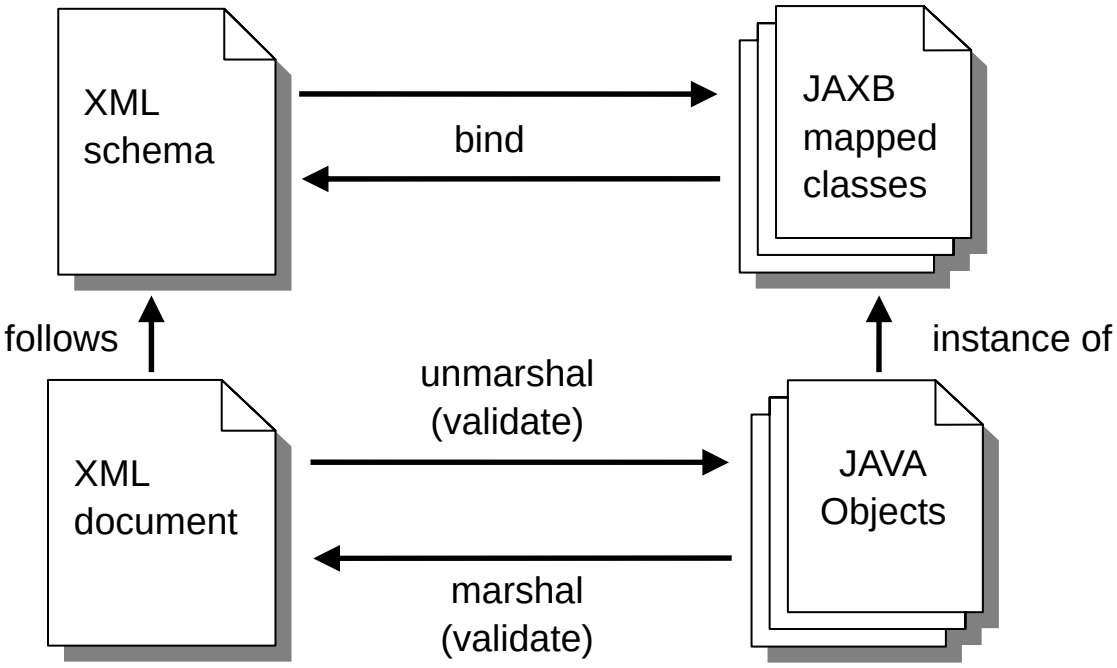


Figure 2-9 : JAXB Binding Process [1]