IDebug: An Advanced Debugging Framework for Java*

Joseph R. Kiniry
Department of Computer Science,
California Institute of Technology,
Mailstop 256-80,
Pasadena, CA 91125

September, 1998

Abstract

The IDebug debugging framework is an advanced debugging framework for Java. This framework provides the standard core debugging and specification constructs such as assertions, debug levels and categories, stack traces, and specialized exceptions. Debugging functionality can be fine-tuned to a per-thread and/or a per-class basis, debugging contexts can be stored to and recovered from persistent storage, and several aspects of the debugging run-time are configurable at the meta-level. Additionally, the framework is designed for extensibility. Planned improvements include support for debugging distributed object systems via currying call stacks across virtual machine contexts and debug information logging with a variety of networking media including unicast, multicast, RMI, distributed events, and JavaSpaces. Finally, we are adding support for debugging mobile agent systems by providing mobile debug logs.

1 Introduction

Programming technologies have evolved greatly over the years. New programming models have emerged, new languages have gained popularity, new tools have been adopted, and yet several core debugging constructs have not changed. We believe that the two primary constructs for general debugging are the *execution trace* and the *assertion*.

1.1 Object-Oriented Debugging

Debugging object-oriented programs is not the same as debugging procedural ones. Because most object models enforce modularity and encapsulation, one must test both the implementation and the interface of a class.

*This document describes IDebug version 1.0

A specification of an interface is called a *Contract* [1, 2, 3, 4]. A class's contract specifies the externally visible behavior that a class guarantees.

Contracts are typically specified via three constructs: *preconditions*, *post-conditions*, and *invariants*. Using these three constructs, the safety properties of a class can be completely specified.

1.2 Debugging in Java

Surprisingly, given its popularity, the Java programming language provides very few built-in constructs for debugging classes.

Typically, a Java programmer relies upon language features and development tools for debugging. Java provides array bounds checking, static type checking, variable initialization testing, and exceptions to assist in code debugging. While programming environments provide sophisticated source-code debuggers, most developers seem fixated on using primitive `println`'s to debug their code.

Java is missing several traditional core debugging constructs, the most critical of which is *assertions*. Assertions are program statements of the form "at this point in execution the following *must* be true". They are used to specify predicates that must remain inviolate for a program to exhibit correct behavior. Typically, if an assertion is violated, a program is aborted. In object-oriented systems we often have options other than halting the program execution (e.g. throwing an exception).

1.3 Debugging Frameworks

A *framework* is a collection of classes that provides a unified model and interface to a specific piece of functionality. A framework in Java is typically implemented as a collection of classes organized into a *package*.

1.4 IDebug: A Debugging Framework

The IDebug debugging framework is implemented as a set of Java Beans (Java components) collected into the package `IDebug`. These classes can be used either (a) as "normal" classes with standard manual debugging techniques, or (b) as components within visual Java Bean programming tools.

This package provides many standard core debugging and specification constructs, such as the previously discussed assertions, debug levels and categories, call stack, and specialized exceptions. Debug levels permit a developer to assign a "depth" to debug statements. This creates a lattice of information that can be pruned at runtime according to the demands of the current execution. Call stack introspection is provided as part of the Java language specification. The IDebug framework uses the call stack to support a runtime user-configurable filter for debug messages based upon the current execution context of a thread. Finally, a set of specialized exceptions are provided for fine-tuning the debug process.

Additionally, the framework supports extensions for debugging distributed systems. One problem typical of debugging distributed systems is a loss of context when communication between two non-local entities takes place. E.g. When object A invokes a method m on object B, the thread within m does not have access to the call stack from the calling thread in A. Thus, the IDebug package supports what we call *call stack currying*. Information such as source object identity, calling thread call stack, and more is available to the debugging framework on both sides of a communication. Such information can be curried across arbitrary communication mediums (sockets, RMI, etc.).

The IDebug package is also being extended to support the debugging of mobile agent systems. Mobile agent architectures can support disconnected computing. For example, an object O can migrate from machine A to machine B, which might then becomes disconnected from the network (i.e. absolutely no communication can take place between B and A). Since B cannot communicate with A, and printing debugging information on B's display might not be useful or possible, B must log debugging information for later inspection. To support this functionality, the IDebug package will provide serializable debug logs. These logs can be carried around by a mobile object and inspected at a later time by the object's owner or developer.

The IDebug package is very configurable. Debugging functionality can be fine-tuned on a per-thread basis. Each thread can have its own debugging context. The context specifies the classes of interest to that particular thread. I.e. If a thread T specifies that it is interested in a class C but not a second class D, then debugging statements in C will be considered when T is inside of C, but debugging statements in D will be ignored at all times. These debugging contexts can be stored to and recovered from persistent storage. Thus, "named" special-purpose contexts can be created for reuse across a development team.

2 Requirements

In this section we will briefly present our project analysis including our project concept dictionary, a review of our requirements for the debugging package, and our goals.

2.1 Project Dictionaries

At the beginning of the project analysis phase, a dictionary of concepts was developed so that all designers, developers, and users would have a clear, common language. The dictionary of terms is included in Table 1 as well as in the design diagrams directory of the framework deliverable.

Next, we will consider the core, application, and innovative requirements we agreed upon before designing the IDebug framework.

Project Dictionary	
Assertion	An *assertion* is a *predicate* which states a logical sentence which evaluates to *true* or *false*. The assertion is typically embedded in program code. An error is indicated if, during program execution, the assertion evaluates to false. There are three main types of assertions (see below): *preconditions*, *postconditions*, and *invariants*.
Debug Context	A *debug context* is a debugging frame of reference. More specifically, each thread of control within a component can have an independent debug context. This context describes what types of debugging information are relevant to that specific thread.
Debug Semantics	*Debug semantics* are the runtime behavior of the debug package, as exhibited by its reactions to exceptions, the language and text of its output messages, etc.
Invariant	A condition that must be true at all stable points in program execution. There are several types of invariants. A *class invariant* is an assertion describing a property which holds for all instances of a class and, potentially, for all static calls to the class. A *loop invariant* for a given loop is an assertion that is true at the beginning of the loop and after each execution of the loop body.
Output Interface	The debug package's *output interface* is any legitimate output medium. Example output interfaces include the system console, a shell window, a GUI, etc.
Postcondition	A condition that must be true at the end of a section of code.
Precondition	A condition that must be true at the beginning of a section of code.
Predicate	Formally, a *predicate* is something that is affirmed or denied of the subject in a proposition in logic. In other words, it is a logical sentence that evaluates to a boolean within specific contexts.
Variant	A *variant* is a predicate that describes how state changes. A *loop variant* is an assertion that describes how the data in the loop condition is changed by the loop. Loop variants are used to check forward progress in the execution of loops (i.e. avoid infinite loops and other incorrect loop behavior).

Table 1: Project Dictionary

2.2 Core Requirements

We require that the IDebug framework support the following requirements. IDebug must, *at the minimum*:

1. *Provide an assertion mechanism.* Assertions are the core construct of any debugging system. Assertions can be intelligently inserted in program code and, if an assertion is violated, an error message is logged and a `AssertionFailedException` is thrown and the thread, and possibly the program halt.

2. *Support the output of debugging messages.* Printing miscellaneous debugging messages, perhaps outside the context of the primary interface of a component, is essential in a good debugging suite.

3. *Support multiple debugging levels.* Different types of errors, messages, and situations require different levels of response. An adequate debugging framework should not only support a set of debugging levels, but the set should be ordered so that user or developer-tunable filtering of debug output can take place[1].

4. *Complement the standard Java exception mechanism.* Since this is a debugging framework built for the Java language, it should work with, not against, the built in exception mechanisms. In particular, prudent use of exception types (`Runtime` verses `Throwable`) is necessary so that the framework is not overly intrusive to the developer[2].

5. *Work with all development environments.* IDebug must work with all development environments, from the most flashy IDE to the lowly CLI runtime. This means that IDebug must be implemented as "100% Pure Java"; no proprietary extensions or native code may be used.

2.3 Application Requirements

Because we build a wide array of Java applications and components, we believe that IDebug should support debugging all types of Java programs. This means that the framework must provide debug functionality that complements the following application types. Each type of application listed below is followed by a non-unique implication of that particular application assumption.

1. *Console-based applications.* Sometimes we will want to send messages to an output stream different from C's STDOUT or STDERR.

2. *Graphical user interface applications.* Occasionally, one wants to send debug messages to independent debugging windows or message sub-frames within a large application.

[1] A filtering mechanism could be used instead, though is usually more tedious for the tester.

[2] I.e. If all exceptions were `Runtime` exceptions, the developer would have to bracket nearly all code with `try-catch` blocks.

3. *Console-less applications.* If there is no output channel, logging debug messages for later retrieval is an excellent course of action.

4. *Independent components (e.g. beans, servlets, doclets, etc.).* Independent components should be able to maintain independent debugging semantics and contexts. Conversely, sometimes it is useful to have a compositional application share a debug context among its components.

5. *Mobile agent/object applications.* If an application has mobile sub-components, their debug contexts need to be mobile as well, and debugging message output and/or storage should be location-independent and/or location-aware.

6. *Distributed applications.* Distributed applications mean (at least) distributed control, distributed debugging context, and distributed debug messaging.

If a debug package were to support all of the above application types, we would consider it extremely powerful due to its flexibility.

2.4 Innovative Requirements

Finally, we wish to support a set of innovative debugging capabilities. While most of these goals are independent of the target language, they are facilitated by many of Java's more advanced features. The list of innovative requirements includes:

1. *Support categorized debugging.* Debugging messages, errors, warnings, etc. should not only have a value (the debug level), but they should have a debug category (a classification).

2. *Support per-class debugging.* A developer should be able to selectively turn debugging on or off at a per-class level.

3. *Have a configurable runtime.* We should not force developers to adopt our debugging semantics. New semantics (debug ranges, base categories, etc.) should be configurable at runtime.

4. *Support multiple output interfaces.* All debugging messages need not be sent to the same output channel. E.g. Consider messages generated by UNIX's `syslog` facility. Some messages are sent to the console, some are logged in a file, and some are sent directly to the system administrator via email.

5. *Support per-thread debugging.* Each thread within a runtime should be able to construct its own debugging context. More precisely, most of the above configurable options (debugging categories, classes, semantics, output interface, and level) should be configurable on a per-thread basis. Additionally, these options should be changeable at runtime.

6. *Support persistent debug contexts.* Once a debugging context is created, it should be possible to send it to persistent storage for later access. This way, debugging contexts can not only be shared across sets of components, but they can be shared across groups of developers.

If a debug framework were to support all of the above requirements, we would simply be amazed[3].

Now that we have a common vocabulary and understand the problem domain and the design goals, we'll consider a design for the debugging framework.

3 Design

We will not discuss the full design of IDebug here due to space considerations. Interested parties should download and consult the IDebug package or browse the information online via the IDebug release web page[4]. Only a few interesting and/or important design points are discussed below.

3.1 Context Configurability

As mentioned previously, debugging options should be configurable on a per-thread basis. On further consideration, we decided that two configurable settings should not be switchable at runtime: debug semantics and output interface.

The reason for this decision might not be immediately obvious, but consider the following two points:

- Debugging output might be queued due to the temporary unavailability of an output channel or user.
- Source code that uses a debugging package makes explicit assumptions about the semantics of the package. Meaning, while debugging semantics might be switchable at runtime by the *framework*, it is not (usually) switchable at runtime for the application *using* the framework.

Due to these factors, the configuration of debugging semantics and output interface is immutable. Meaning, once these options are set for a debugging context, they cannot be changed.

Note that a new context can be *created*. All the other flexibility mentioned in Section 2.4 is fully configurable at runtime on a per-thread basis.

Now, we'll briefly discuss the implementation and use of the IDebug framework, version 2.0.

[3]Hint: IDebug supports everything you have read so far.

[4]http://www.infospheres.caltech.edu/releases/

4 Implementation

IDebug is freely available from the KindSoftware's Open Source web pages[5].

4.1 Implementation Size and Performance

Implementation Summary (with test and example code)	
Total Number of Packages	2
Total Number of Classes	12
Total Number of KB of Java (includes code, documentation, and whitespace)	97.9KB
Total Number of KB of classfiles	
Independent class files	28.4KB
Jar (compressed) format	11.7KB
Total Number of Lines of Code[6]	2069
Total Number of Lines of Comments	1380
Comments/Code	67%

Table 2: Implementation Summary

The implementations of both versions of IDebug are summarized in Table 2.

IDebug Performance. We have not yet performed performance tests on the IDebug package. In general, its performance is entirely based upon the speed of the Java runtime's `Throwable.printStackTrace()` method and `Hashtable` and `StringBuffer` implementations, since these classes are at the core of the exception and assertion-handling mechanisms in IDebug.

A performance profile test of IDebug could reveal performance weaknesses. In general, any performance tuning would mean replacing data structures, rathan than changing core algorithms.

In general, performance is not an issue in debugging complex systems, especially distributed or object-oriented systems. We make this claim for two reasons:

First, the debugging phase of an implementation should be part of an ordered and reasoned test suite, and thus the use of the debugging framework should also be ordered and have reason. In other words, rarely will it be the case that all threads within a complex application will have all their debugging options turned on simultaneously.

Second, we believe that debugging statements should not be written by hand or statically inserted into program code. Debug code should be "tunable" at compile time, not just runtime, and thus debug framework performance should only matter for critical debug paths, of which there should be few.

[5]http://www.kindsoftware.com/products/opensource/

4.2 Framework Extensibility

The IDebug framework is extensible in two dimensions: debug semantics and output interfaces.

IDebug Framework Semantics. The semantics of the package can be changed by implementing new versions of DebugConstantsInterface. An example of such an extension is provided in the form of the `FrenchConstantsInterface` class in the `IDebug.examples` package. This class provides an implementation of `DebugConstantsInterface` that differs from the default implementation (`DefaultDebugConstants`) in two ways:

1. Debug levels range from 1 to 100 instead of 1 to 10,
2. Default debugging levels have been adjusted for this new granularity of debug levels, and
3. Default debug messages, categories, and documentation are provided in French.

IDebug Output Interfaces. New implementations of the `DebugOutput` interface can be designed to support sending debug messages to alternative output media/channels. As of version 2.0, the framework comes with two implementations: `ConsoleOutput`, which sends messages to the console of a Java runtime, and `WriterOutput`, which sends debug messages to a `Writer` which can be used as part of a normal `java.io` compositional data stream.

4.3 Complementary Tools

Static debugging statements clutter source code, increase object code size, and reduce execution speed. We have developed a application called JPP, the Java PreProcessor, that solves exactly this problem.

In short, JPP performs transformations of embedded program specification, in the form of design by contract[5] (DBC) predicates in documentation comments, into IDebug test code at compile time. Future versions of JPP will also perform code beautification, code standard conformance checking, code metric analysis, and documentation generation.

5 Conclusion

IDebug is the most advanced debugging framework available today. It is extremely configurable, supports a wide range of Java application types, and, because it is an open framework, is extensible by the developer.

Future Work. In fact, we encourage developers to extend IDebug. In particular, we are interested in hearing about (and including) alternative implementations of the `DebugOutput` interface and `DefaultDebugConstants`. We have come up with the following alternative ideas for output interfaces; perhaps your application could use one of these or one that we have not thought of:

- `DBOutput` — used to log debugging messages to a database via standard JDBC.
- `EventSourceOutput` — send messages to arbitrary listeners within a Java virtual machine (perhaps as part of a compositional Java Beans-based application).
- `FrameOutput` — to send debugging messages to an arbitrary frame within a larger GUI.
- `LogOutput` — to persistently log messages for off-line debugging.
- `MessageOutput` — send messages via a JMS-conformant messaging infrastructure to a/some remote objects.
- `RemoteEventSourceOutput` — to provide debugging messages as distributed events (perhaps as part of a Jini[7] application).
- `ScrollableWindowOutput` — display messages in an independent, scrollable window.
- `ServletLogOutput` — to persistently log messages via the servlet developers kit's debugging interface.
- `SpaceOutput` — store debugging events in a JavaSpace[6].

Finally, we are investigating integrating IDebug with Dan Zimmerman's ÜberNet distributed messaging infrastructure[8]. Our primary goal is to support the currying of call stacks across execution contexts. This would mean that assertions and exceptions on remote (receiver) machines would have access to the call stack of the sending thread.

Thanks. The author would like to thank the Infospheres Group for help with the initial problem analysis and early IDebug design. In particular, the comments of Mani Chandy, Dan Zimmerman, Wesley Tanaka, and Adam Rifkin were invaluable. Also, Nelson Minar used the first version of IDebug as part of his thesis work; his comments were very helpful. Matt Hanna helped review this technical report. Finally, I'd like to thank Ron Resnick, Mark Baker, Mary Baxter, and Cici Koenig for their general support and encouragement in all I do.

References

[1] Graham Hamilton, Michael L. Powell, and James G. Mitchell. Subcontract: A flexible base for distributed programming. In *Proceeings of the Fourteenth Symposium on Operating Systems Principles*. Sun Microsystems, Inc., December 1993.

[2] Richard Helm, Ian M. Holland, and Dipayan Gangopadhyay. Contracts: Specifying behavioral compositions in object-oriented systems. In *European Conference on Object-Oriented Programming/ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, volume 25/10 of *ACM SIGPLAN Notices*, pages 169–180. ACM SIGPLAN: Programming Languages, ACM Press and Addison–Wesley Publishing Company, October 1990.

[3] Ian M. Holland. Specifying reusable components using contracts. In *Proceedings of the Seventh Conference on Object Oriented Programming, Systems, Languages, and Applications*, pages 287–308. ACM SIGPLAN: Programming Languages, ACM Press and Addison–Wesley Publishing Company, 1992.

[4] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice–Hall, Inc., second edition, 1988.

[5] Bertrand Meyer. *Advances in Object-Oriented Software Engineering*, chapter Design by Contract. Prentice–Hall, Inc., 1992.

[6] Inc. Sun Microsystems. *JavaSpaces Specification*. Sun Microsystems, Inc., revision 1.0 beta edition, July 1998.

[7] Jim Waldo. *Jini Architecture Overview*. Sun Microsystems, Inc., 1998.

[8] Daniel M. Zimmerman. *ÜberNet: The Infospheres Network Layer User Guide*, 1.0a1 edition, February 1998.