Paper Plain English Java Debugger

A user-friendly debugger and visual system inspection tool

Ahmet Emre Ünal
Computer Sciences, Faculty of Engineering
Özyeğin University
İstanbul, Turkey
Emre.unal@ozu.edu.tr

Çelebi Murat
Computer Sciences, Faculty of Engineering
Özyeğin University
İstanbul, Turkey
Celebi.murat@ozu.edu.tr

Barlas Karahocaoğlu
Computer Sciences, Graduate School of Engineering
Özyeğin University
İstanbul, Turkey
Barlas.karahocaoglu@ozu.edu.tr

Abstract— Efficient usage of debugging tools has been proposed as an essential part of the cost/time efficient software development and effective learning process [1]. However, efficient usage of those tools is not fulfilled by most of the developers. This is because those tools' lack of simplicity and lack of having a user-friendly interface. This paper provides a solution on the need of an easy-to-use and easy-to-understand beginner level debugger to aid learning process, especially for students, by providing a plug-in for Eclipse Integrated Development Environment (IDE) for Java language that helps developers to understand the background operations in a simple textual and visual based way.

I. Introduction

With the advance of computer technology world's interest trend bends over software engineering and development. Day by day universities try to advance their Computer Sciences education system to catch that trend. Students faced with new languages and environments. The fundamentals of software development are mostly independent of the language and the environment. Software development highly depends on the one's understanding of the background processes, method calls, call stacks etc. Instead of memorizing code blocks, to develop code one needs to no what will happen for each specific line. This is much experience and knowledge to be expected from freshmen, but still needed. There are debugging tools for this purpose but not used efficiently because of their lack of simplicity and lack of having an user-friendly interface. Majority of the developers still uses textually contiguous program slices to debug while not using the tools [2]. Like and more than the senior developers, especially computer programming students (mostly freshman) are the worst users of those tools.

Our main goal was to address that issue and to improve the freshman's understanding of the background of the computer program development, so that they will not undershoot their goals. This paper focuses on the need of an aid for the beginner Java courses by providing a plug-in that helps students to understand the background operations in a simple textual and visual based way. An efficient, in terms of assisting learning process, debugging and inspection tool.

The rest of the paper has the following organization.
*Java language, Eclipse IDE and Debugging Concept are introduced
*Background, investigated domain, challenges and utilized techniques described;
*Problem statement presented;
*Solution approach given;
*Evaluation of the overall process;
*Results & Future Work
*References

A. *Java Language*

Java is a computer programming language that is concurrent, class-based, object-oriented, and specifically designed to have as few implementation dependencies as possible. It is intended to let application developers "write once, run anywhere" (WORA), meaning that code that runs on one platform does not need to be recompiled to run on another Java is, as of 2012, one of the most popular programming languages in use, particularly for client-server web applications, with a reported 9 million developers (langpop.com). [Wikipedia]

B. *Eclipse Integreated Development Environment*

Eclipse is an integrated development environment (IDE). It contains a base workspace and an extensible plug-in system for customizing the environment. Written mostly in Java, Eclipse can be used to develop applications in Java. By means of various plug-ins, Eclipse may also be used to develop applications in other programming languages: Ada, C, C++, COBOL, Fortran, Haskell, JavaScript, Lasso, Perl, PHP, Python, R, Ruby, Scala, Clojure, Groovy, Scheme, and Erlang. [Wikipedia]

C. Debugging

There are two ways to write error-free programs;
Only the third one works.

- Perlis
(Epigrams on Programming, 1982)

Program errors are results of inappropriate actions of developers, during development, which causes failures. The physical manifestations of errors in the program text are known as faults or bugs [3]. We can realize the presence of errors if we can find an error-revealing test case that causes the program to fail. Errors are inevitable facts of software development because of errors are human sourced and developers are humans.

Testing is a process to find error-revealing test cases that causes the program to fail in the input space. Debugging is a process of locating and repairing the faults that propagates to failures. The difference between testing and debugging is that testing reveals is there a failure or not while debugging finds and fixes the faults responsible for that failure.

II. Background

There are debugging tools for nearly every Integrated Development Environment (IDE)., but they are complex programs. The challenge was to make this debugging information as clear as possible, so that developer can understand and interpret the given information.

To overcome that challenge we investigates popular Languages and IDEs to provide a solution with the broadest range, in other words to maximize the target users. We decided to develop an extended debugger translator plug-in for Java with Eclipse. These two combinations are the widest used combinations for the programming education, since our real targets are students.

We investigated two domains. First was the Eclipse's debugger, second was the Eclipse's plug-in development process. We planned to extract debugger state and information, interpret that data and view in both visual and textual base. We provided that functionality as a plug-in.

Making the program flexible enough to be useful was the most challenging part of this task. Due to the quality, or the lack of it, of student-written code, a strict set of pre-defined, but well-explained, statement templates were doomed to fail. To achieve a useful balance, the statement structure requirements, as well as the thoroughness of explanations, were relaxed to fit the basic structure of simple statements.

The simplest way to do this matching was to use detailed regular expressions that match statements to their appropriate explanation generation engines. These engines, again through the use of regular expressions, try to make sense out of these codes and try to present the student a simple but useful explanation.

III. Problem Statement

We aimed to make programming students to better understand the way programs work in the background so they can write codes easily by knowing how they work and we want them to better identify the faults that propagated to failures by providing a more understandable debugging and program state information.

In order to be a good developer one must understand the background operations, what is happening in the runtime, which state the program fails, what are the variables etc. The problem is freshmen students were not capable of having what was expected. There are debuggers to address this issue. When developer is not sure about what is happening, he or she can use debugging tools to understand the system behavior. However, it is not an easy thing to use debuggers. They are complicated, technical and they only, if you know how to use, give you appropriate answers. They don't explain a lot nor give much clue.

While programing/teaching websites like CodeSchool.com & CodeAcademy.com went mainstream, they don't offer true flexibility. Most of the time, they show and explain pre-defined sets of statements, aimed to give a perspective to a student. Yet, because of the aforementioned barriers of a student's mind, flexibility (in the teacher's, in this case the website's, part) becomes a sought-after quality. The closest to what we want was in PythonTutor.com, but because that website creates just a crude representation of the memory layout of a program, and nothing more, we were left unsatisfied.

Current mainstream programming languages (Java, in this case) hide most of the technical details. A lot of programming students whom aren't into programming, memorize commonly used Java statements, and try to write a half-baked program from their memorizations. This approach, inevitably, is doomed. To avoid this memorization, teachers put a lot of effort into explaining the meaning of such statements - what a 'method call' means, for example.

IV. Solution Approach

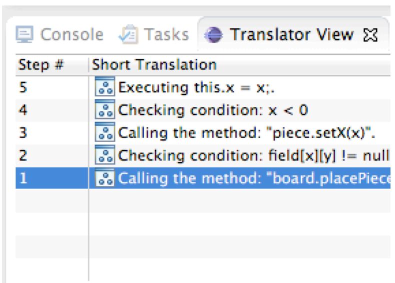
A lot of effort goes into breaking down the barriers of a student's mind. Yet, no one, other than the student, can bring those barriers down without huge efforts. This is the philosophy behind the Plain English Java Debugger (PEJD). PEJD aims to let students breathe programming, one line at a time, at their own pace.

We developed a plug-in for Eclipse IDE's Debugger perspective where students can easily monitor the states, variables and stacks in both textual and visual way(see Fig1-5). When using PEJD, users can set breakpoints or use automatic breakpoints at each line. Users can set stepping time between respective lines and skip lines with buttons (see Fig.3. and Fig. 4.).

A. PEJD uses the information from Eclipse Debugger.

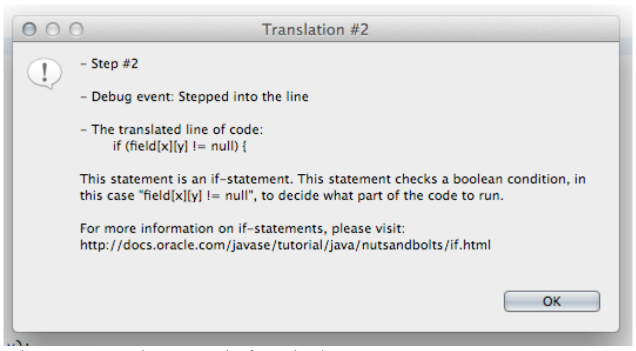
PEJD was designed to provide understandable information on dynamic inspection and must be flexible and efficient in order to prevent slowing down the system. Since PEJD uses performance proven Eclipse Debugger data, there is no probe effect introduced to the system at all. PEJD simply reads Eclipse Debugger data and state.

When PEJD obtains the debugger state, it interprets that state data to an easy to understand textual component. This state data can have, declarations, instantiations, method calls/returns, variable assignations etc. Interpreted data is show in an embedded tab at the Eclipse "show view"(see Fig. 1). Embedded tab is interactive, where each row represents relative debugger state and when clicked provides additional information about the general behavior of that action (see Fig. 2). Explanation that provided by PEJD may not be sufficient, so there are links to the community web sites provided for that specific action/error (see Fig. 2.).



Step #	Short Translation
5	Executing this.x = x;.
4	Checking condition: x < 0
3	Calling the method: "piece.setX(x)".
2	Checking condition: field[x][y] != null
1	Calling the method: "board.placePiece

Fig. 1. Translator View.



- Step #2

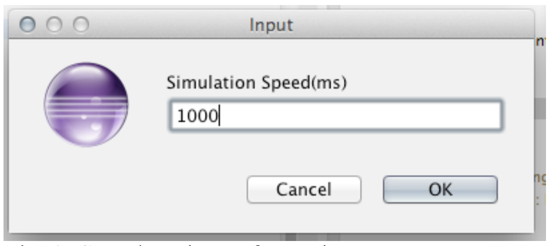
- Debug event: Stepped into the line

- The translated line of code:
    if (field[x][y] != null) {

This statement is an if-statement. This statement checks a boolean condition, in this case "field[x][y] != null", to decide what part of the code to run.

For more information on if-statements, please visit:
http://docs.oracle.com/javase/tutorial/java/nutsandbolts/if.html

Fig. 2. Sample extra info window.



Simulation Speed(ms)
1000

Fig. 3. Speed settings of stepping.

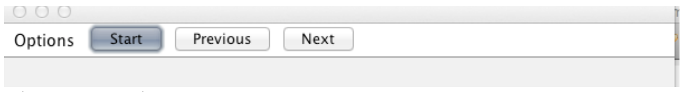
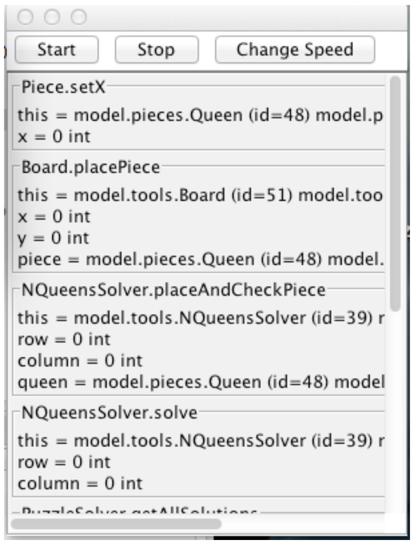


Fig. 4. Stepping Buttons.



```
Piece.setX
this = model.pieces.Queen (id=48) model.p
x = 0 int
Board.placePiece
this = model.tools.Board (id=51) model.too
x = 0 int
y = 0 int
piece = model.pieces.Queen (id=48) model.
NQueensSolver.placeAndCheckPiece
this = model.tools.NQueensSolver (id=39) r
row = 0 int
column = 0 int
queen = model.pieces.Queen (id=48) model
NQueensSolver.solve
this = model.tools.NQueensSolver (id=39) r
row = 0 int
column = 0 int
```

Fig. 5. Graphical stack representation.

PEJD was designed using Model View Control principles. We have thirty-four classes divided to four main packages, which are "actions", "simulationModule", "translationModule" and "views". Base classes of the plugin are under the "views" package. First class that is initialized at the beginning of the plug-in intialization is the "TranslatorView" class inside the "translatorView" package, which then initializes the "TranslatorViewModel" class.

"TranslatorViewModel" class is the bridge between the view part of the program and the backend processes of the program (translation and simulation). It notifies the view part of the program according to changes made in the debugger, like starting the debugger or stepping into a line.

Inside the "translationModule" package we have classes that provides translation for each processed line, as the name suggests. We have different classes for each statement type that we process.

"simulationModule" package includes the stack memory simulation part of the program. There are Actions and ActionListeners for opening the view in plug-in, for example, changing the speed of the simulation.

Last, we have "actions" package. This package contains 3 classes that listens the debugger for break points and other events like stepping actions

V. Evaluation

PEJD's biggest strength is its flexibility - the ability to adapt to large-scale, student-level programs, without any loss in effectiveness. When a student wants to learn what a specific piece of code (or a program) does, instead of trying to piece together half-learned code snippets together in his/her mind, he/she can load it in Eclipse, run PEJD and have it explained to him/her in a human-readable language. PEJD is flexible enough to explain the basics of most of the statements a Java 101 student may encounter. With the open-source nature of the plug-in, we wanted to provide the foundations and the inspiration for future programmers to take on and expand this beautiful tool.

Another benefit is for the other side of the programming learning process, Academicians. When debugger is not efficiently used, students suffer from malfunctioning codes and seek help from each other, TAs and Professors, but this is inefficient in terms of academic resources and when they give up, results in plagiarism. PEJD addresses that secondary issue by enabling students to understand the steps of their code by themselves.

VI. Relevant Work

A. *WhyLine*

The Whyline is a prototype Interrogative Debugging interface for the Alice programming environment developed in Carnegie Mellon University by Andrew J. Ko and Brad A. Myers. Interrogative Debugging is a new debugging paradigm in which programmers can ask why did and even why didn't questions directly about their program's runtime failures. The Whyline visualizes answers in terms of runtime events directly relevant to a programmer's question. Comparisons of identical debugging scenarios from user tests with and without the Whyline showed that the Whyline reduced debugging time by nearly a factor of 8, and helped programmers to complete 40% more tasks [6]. The Whyline was not designed to work with Java, thus this reduces its target by a large factor. The way Whyline works is a great fit for students, where they can “ask” why their program fails.

B. *Deet*

Deet (Desktop Error Elimination Tool) is a simple debugger for ANSI C and Java developed by David R. Hanson and Jeffrey L. Korn. It differs from conventional debuggers in that it is machine-independent, graphical, programmable, distributed, extensible, and small. Deet provides both textual and graphical interactive interface where users can debug code with click of their mouse [7].

VII. Conlusion and Future Work

The best way of learning is relies on understanding. Academic staff is the key component of that process. PEJD may be considered like a personal Teaching Assistant. It assists academic staff by providing dynamic program state information to students whenever, wherever needed thus reducing the percentage of office hours visits.

PEJD provides step-by-step real time data of; call stacks, instantiations etc.

Since Eclipse supports many languages, PEJD can be extended to be used by those languages also. To enable organic evolution of PEJD, we open sourced it. It is publically available under Ahmet Emre Ünal’s gitHub repository [4].

PEJD can adopt Interrogative Debugging concept to enable users asking questions. When this functionality and PEJD’s solution of providing explanations and relevant community support links combined, there is a great chance of a powerful combination.

There is a great improvement space on visual interaction. Visual representation is one thing; visual interaction is a whole another thing. PEJD have a sort of visual interaction but not a real one, where users can click on the stacks, see the real time method calls, passing arguments; like watching a video game. People feel more comfortable, more accepting, more secure and less judgmental when they have visual interaction. Thus adopting/improving such functionality for PEJD will be a huge and an important step.

References

[1] Min Xie, Bo Yang, “A study of the effect of imperfect debugging on software development cost” Software Engineering, IEEE Transactions on, vol. 26 issue. 5, pp. 471-473, May 2003

[2] Mark Weiser, “Programmers use slices when debugging”, Communications of ACM, vol. 25 issue. 7, pp 446

[3] IEEE Standard Glossary of Software Engineering Terminology, IEEE Std. 729, 1983

[4] Ahmet Emre Ünal GitHub Repository, https://github.com/aemreunal/PlainEnglishJavaDebugger

[5] Hialal Agrawal, “Towards automatic debugging of computer programs PhD thesis, Purdue University, 1991

[6] Adrew J. Ko, Brad A Myers, “Designing the whyline: a debugging interface for asking question about program behaviour”, Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, pp. 151-158, 2004

[7] David R. Hanson, Jeffrey L. Korn, “A Simple and Extensible graphical Debugger”, Proceedings of the USENIX Annual Technical Conference, pp. 173-184