Must Java development be so slow?

An "Oberon-inspired" Java environment might have some answers

E x t e n d e d   A b s t r a c t

Albrecht Wöß
Johannes Kepler University Linz, Systemsoftware Group
Altenbergerstr. 69, 4040 Linz, Austria
woess@ssw.uni-linz.ac.at
http://www.ssw.uni-linz.ac.at

ABSTRACT

Starting a new VM each time an application is executed, forces numerous classes to be loaded multiple times. This overhead significantly slows down the development of Java software. So don't do it! We plan to eliminate these bottlenecks with an open Java environment where only one VM hosts all applications and classes are loaded only once.
Furthermore we want to add a powerful textual user interface that allows command execution at a mouse-click in any text, supports literate programming, and thus highly enhances Java programming, development and maintenance of Java software.

1. PURPOSE

As one can deduce from the title of this poster presentation, which just poses a question without giving a definite answer, this piece of research is at a very early stage. This is at the same time the main reason for it to go into a poster session. We like to expose our intentions to public discussion as soon as possible, because we expect them to raise controversy, and we want to ensure that we are going into the right direction.

2. INTRODUCTION

In 1986 Niklaus Wirth set out to build an operating system and compiler from scratch. The result of this effort was the Oberon System [7] [6]. This operating system and software development environment has some unique features that distinguish it from most other state-of-the-art IDEs. The most interesting features are: (a) commands which provide programs with multiple entry points, (b) dynamic loading of modules which stay resident until they are explicitly unloaded again, and (c) a powerful textual user interface based on commands and embedded objects.

3. GOALS & CONTRIBUTIONS

What we intend to do is to build a Java Development (and maybe Execution) Environment that brings some of the features of the Oberon System into the Java World. Our main goal is, as stated in the abstract, to avoid unnecessary reloading of classes, but there are other "goodies" in Oberon that we miss when programming in Java.

3.1 Class Loading and Modularity

We started out by considering the way Java executes programs and loads classes. First we did not want to have a separate virtual machine (VM) running for every executing program (= OS-process based multi-tasking). When using a single multi-tasking VM the usual approach is to use separate class loaders for each application (e.g. [2] uses a Java-process based approach). Thereby one achieves a maximum amount of safety, since each application will be provided its own private copy of each class, because class loaders introduce additional namespaces (besides packages), and thereby let one and the same class appear as different types to different applications (for more information about Java class loading see [4]). We did not like this approach either, because it requires many classes to be loaded multiple times (over 200 for a "Hello World"-program, more than 1000 for a very simple Swing-application). We wanted to ensure that each class is loaded only once and can then be used by all applications. This is one way to reduce class loading effort and thereby start-up time. [1] employs an interesting concept of sharing program code completely among applications but still strictly separating them by replicating their static fields, and thus prohibiting communication between them.

This separation, however, is still not in the spirit of the Oberon system, where maximum extensibility, cooperation and integration of different programs are achieved by complete sharing of all loaded modules (including their state). That is why we call it an "open" environment.

A consequence of the extended module sharing is that classes are *not* unloaded automatically when they are no longer used by any program. We want to keep them in memory until they are explicitly unloaded. When a program terminates the classes it has used stay in memory and can thus be "recycled" by a next program without further need to reload them. Only when a developer wants to replace an erroneous version of a class with a new one, she would explicitly unload the old instance. When the class is needed again, the new version will be loaded.

Since reusing classes that are already loaded does not call the static initializers, our environment will require a different programming style for certain classes. Class state that should not be shared between applications has to be separated (and initialized) explicitly by the programmer. On the other hand, our approach allows the programmer to share class state between applications, which is not possible in other Java environments.

Regarding classes as modules that provide certain functionality, allows the realization of minimal systems, meaning that e.g. a word-processor does not have to load a module (class) for displaying images unless it actually needs to process an image. If that image processing module happens to be loaded, e.g. because an image viewer has been running before, no more loading needs to take place, and no more memory will be wasted by a second copy of the same module.

3.2 Commands

We plan to introduce a (to Java) new concept of program entry point. Like Oberon we would like to allow a class to provide more than just one entry point. Currently Java program execution can only start with the *main*-method of a class. We want to provide a user with the possibility to directly invoke not only the *main*-method, but other methods, too. These methods will be called "commands" and separate the "unit of compilation (and loading)" (= class) from the "unit of action" (= command). Whether only static methods will qualify or if we will even add some kind of scripting language that supports object creation and then invocation of instance methods, is yet to be determined.

3.3 Text

We also work to supply the Java world with an equivalently powerful concept of text as introduced by Oberon. The notion of text, which most current development tools employ, is a sequence of ASCII- or Unicode-characters. This not only limits the expressiveness of source code, but also decreases turnaround time in the development process. When text is no longer just a sequence of characters, but one of objects, a whole new way of writing source code becomes feasible.

Combining the new text model with the above mentioned commands allows command names (like *MyClass.foo*, *MyClass.bar*, ...) to serve as "textual buttons" that invoke the respective command when the user clicks on them in a text. This make it possible for a user to write command menus using a simple text editor and store them in text files. In order to open the menu, the user simply opens the file in the editor and clicks on the command names. Thus the user has to type the commands only once(!), which will have some implications: First of all it saves a developer from going out of her way to implement a GUI to use a class' functionality. Secondly each user can create her individual tool text in a blink of an eye. Thirdly, instead of being forced to retype the same commands again and again at command prompts, one will save a lot of typing effort this way. This will further lead to more descriptive command names like e.g. *Files.Directory*, *System.ListClasses*, etc. instead of the widespread, but still rather cryptic abbreviations like e.g. *ls*, *rm*, *mkdir*. One can provide help texts, tutorials, etc. that not only describe what the user should do, but provide the directly clickable commands along the way. And don't forget, one can easily enlarge the pool of available commands just by writing another "command class". These commands will then be immediately available to any user after a few key types and a mouse click.

3.4 Text Elements

Literate Programming as introduced by [3] can be highly enhanced by the use of so called Text Elements in your source code [5]. These can be folding elements that allow parts of the code to be collapsed into a comment. Cascading such elements allows replication of the stepwise refinement process of software development. Graphic elements can be used to improve the explicability of comments, showing UML or other diagrams, screenshots and the like. Link and Mark elements support hypertext-like connections between parts of the document. All these text elements are inserted directly into the source code and thus allow a developer to read the code selectively, navigating from a method call directly to its definition, zooming into procedures, and so on. An integrated compiler ignores the elements and just regards the actual program code, so that no separate conversion from the "annotated" code into a compiler compatible version is necessary.

A tool shall be implemented that allows automated generation of WEB-like [3] (or other forms of) printable program documentation from such Text Elements as they are already available in Oberon. Note that this approach does not require the use of any special documentation language.

4. CONCLUSION

Oberon is a remarkable programming environment and operating system, that - in our opinion - lacks the attention and appreciation it deserves. Bringing the "good stuff" of Oberon into the vast Java universe, might help spread some of the fundamental concepts of Oberon among many software developers worldwide. This, we believe, will be a huge benefit for all resulting software products.

Since Oberon supplies not only an IDE, but also an operating system along the way and the areas of application differ widely (single-user for Oberon, worldwide multi-user for Java), one important aspect of this PhD thesis will be to evaluate the chances of the Oberon approach in the Java universe. For example versioning is a crucial issue here. When reusing a loaded class, one must make sure that it is actually the class one is looking for. Since class names do not suffice to distinguish java types, this might become tricky.

There are many more issues that remain to be investigated, and some, which we are still unaware of, may - hopefully - come up at the poster discussions.

5. REFERENCES

[1] Czajkowski, G. *Application Isolation in the Java™ Virtual Machine*, in Proceedings of OOPSLA '00 (Minneapolis MN, October 2000), ACM Press, 354-366.

[2] Gorrie, L. *Echidna - a Free Multitask System in Java™*, http://www.javagroup.org/echidna .

[3] Knuth, D., and Levy, S. *The CWEB System of Structured Documentation*, Addison-Wesley, 1993.

[4] Liang, S., and Bracha, G. *Dynamic Class Loading in the Java™ Virtual Machine,* in Proceedings of OOPSLA '98 (Vancouver BC, October 1998), ACM Press, 36-44.

[5] Mössenböck, H., and Koskimies, K. *Active Text for Structuring and Understanding Source Code,* SOFTWARE - Practice and Experience, 26(7), 1996, 833-850.

[6] ETH Oberon home page, http://www.oberon.ethz.ch .

[7] Wirth, N., and Gutknecht, J. *Project Oberon - The Design of an Operating System and Compiler*, ACM Press, Addison-Wesley, 1992.