EXPERIENCES FROM A LARGE COMMERCIAL OBJECT-ORIENTED SOFTWARE PROJECT USING EIFFEL AND JAVA

David Oppenheim
Optimation Software Engineering, Melbourne

In November of 1994, Optimation embarked on a major research and development project to enhance its existing systems testing product. One of the key goals was the need for a GUI for the product, and portability was paramount. Optimation was also looking for a higher quality, more productive development environment than traditional 3GLs. The Eiffel language environment was selected. After a year of development, the project switched to Java, when Java was in its infancy as a commercial offering. This paper discusses the background, goals, development and outcomes of this project. It will discuss the reasons for selecting Eiffel, the reasons for switching to Java and the advantages and disadvantages of both technologies. While the discussion focuses primarily on the software engineering of the project, it also includes commercial and technical content.

1. THE JOB AT HAND

Optimation is a medium-sized company specialising in performance, communications and systems software. One of Optimation’s major product lines is a software family called TermSim used for carrying out testing. In 1994 we commenced a project to add a major new component to the product. Until then, TermSim ran as text-mode Unix programs. We wanted to add a Graphical User Interface (GUI) to make using the product more productive to use, easier for users (especially first-time users), and to give more real-time performance metrics via dials, graphs and the like.

The design we settled on utilised the existing programs (written in C) as background "engines", since these had extensive R&D effort and worked well - we saw no value in rewriting them in any new language or environment. The design called for the new GUI components to be integrated with the background C programs via TCP/IP inter-program communications. This had the extra benefit of allowing the components to run in a distributed processing environment, making good use of multiple machines.

The design added real-time control and feedback of the testing tools, rather like the dashboard of a car. Graphs and dials could show the value of any run-time performance statistic, and the performance engineer running the test could control and change parameters of the test as it progressed. In addition, there was significant innovation in tying it all together, and providing integrated functions such as a combined WYSIWYG editor, script compiler, and run-time debugger.

2. REQUIREMENTS FOR THE DEVELOPMENT TEAM

Having specified the project requirements, and done the high-level design, the next step was to identify the needs of the development team. We needed:

• A design methodology and/or tools
• A productive language - the fewer lines written the better
• A productive development environment - editor, debugger, etc

A key requirement was portability of our code. This was 1994-5, so to us this meant portability of a GUI on Unixes, hence at least portability between OpenLook and Motif. Because TermSim is far beyond the capabilities of a Microsoft operating system, we did not have immediate requirements for portability to Windows 3.11, 95 or NT, but we certainly were looking to that for future projects.
Of course portability in 1994 (prior to Java) meant that, with luck, to port you "simply" recompiled, then you could start testing whether your code worked on some new platform.

Another requirement was that we didn't want to reinvent wheels - we needed rich libraries, such as GUI, communications, data structures, etc.

The senior people at Optimization carried out a lengthy market search for development tools, languages, libraries, and environments. We chose ISE Eiffel first and foremost for its portable GUI libraries (Eiffel Vision). These promised run-time portability between different window systems on the same platform (e.g. OpenLook and Motif on Sun), and re-compile only to cater for different platforms (e.g. Microsoft Windows on Intel).

A secondary reason for choosing Eiffel was the excellence of the Eiffel language. In these respects, Eiffel stood head-and-shoulders above any competition. A subsidiary benefit was that Eiffel is Object Oriented - perhaps some would have made this a mandatory requirement, but we did not see this as an essential criterion for our choice.

We chose to use a modified BON method for design.

Next, we spent an intense period reading, and learning Eiffel and its libraries. And then much effort went into detailed planning & designing prior to the coding and testing phases.

We bought a range of hardware for development and testing on different platforms. This included a high-end Sun, a Silicon Graphics, and an HP workstation. Later, and IBM RS6000 running AIX was added. These machines entailed a significant amount of systems administration to set up the development environment in a seamless manner across all the machines.

3. HOW DID WE GO WITH EIFFEL?

After a year of intensive development TermSim was mostly finished. But it wasn't quite shippable - the 90 / 10 rule for remaining functionality versus effort seemed to apply. It looked like as much work to still to do as had been done.

Productivity had been better than if we'd used C and we were glad we hadn't seriously contemplated C++, but productivity hadn't been as high as hoped. Compounding this were two issues - quite a number of bugs in the Eiffel Vision GUI class libraries, and glacial compilation speed of the ISE Eiffel compiler.

We'd also found support for ISE Eiffel from the vendor (ISE) fairly so-so. Then, ISE dropped OpenLook support from Eiffel Vision, rather undermining our primary criterion of portability for choosing Eiffel.

Technically, we were still Eiffel language enthusiasts, but our management consciences (and hip-pockets) said it was time for a project checkpoint and reassessment.

4. WHAT'S THIS THING CALLED JAVA?

So, towards the end of 1995, we paused the project, and looked at the various possibilities for finishing it. First choice was a focussed, as limited as possible plan to finish the project in Eiffel.

Then we looked again at the market for alternatives. We looked inquisitively at Sun's just-announced Java. It was unclear what it could do, and what it was for. We experimented. We also looked at other possibilities, including TCL/TK.

We found Java gave new meaning to "portable" code -- we wrote and compiled a few small text and GUI test programs, and ran them on several different platforms. Note carefully - we ran them on the other platforms - no recompiling. The same object code ran across multiple platforms. Moreover, it did the right things! This was hugely convincing after Eiffel gremlins re portability.

The Java language - like Eiffel - was a clean, simple Object Oriented language. It wasn't as good as the Eiffel language, but it was a lot better than C (and we won't mention the horrors of trying to do good software engineering in C++).

We took the brave decision - it was December 1995 - to go with Java. We recognised it might become a dead-end, in which case we wouldn't write other products in Java. But two major vendors - Sun and Oracle - were committed to Java, and a few others were thinking about it.

5. HOW DOES JAVA COMPARE TO EIFFEL?

We've been programming in Java for 80 weeks (Java and Internet time moves in days and weeks, other things in months and years - this
means we are senior citizens in the Java community), what do we think?

Well, first let's look at the strictly technical issues and compare the languages.

They are both modern, Object Oriented languages. Eiffel compiles (usually to C, then to object code), Java semi-compiles to machine-independent byte-code and then runs in a virtual machine interpreter (which is free-standing, or can be part of a Web browser - thus spawning the world of Java applets).

Java has no multiple inheritance - Eiffel has, and has it very nicely sorted out. We think multiple inheritance is good for modelling the real world. In fact the real world is even more complicated than the tree- or graph-structure implied by inheritance: perhaps a many-dimensioned space (colour, temperature, number of wheels, tax rate, ...) is a closer match to reality. Some (e.g. Smalltalk programmers) would argue you don't need multiple inheritance, some would argue it's better and simpler to not have it. We think there is some truth to this, but it'd be better to have multiple inheritance and use it sparingly and well than to be forced to not use it because the Java inventors didn't have time to learn from Eiffel how to implement it properly.

Java does have something that can be used as a form of multiple inheritance, called interfaces. Our experience in both Eiffel and Java has taught us that Java interfaces can give much of what you would use Eiffel multiple inheritance for, often Java interfaces provide a clean and simple implementation.

Java has no assertions (especially pre- and post-conditions) as found in Eiffel. We, like almost everyone who has used assertions in Eiffel, found them to be very useful in improving programmer productivity and code quality. We have toyed with the idea of using the C preprocessor cpp (or emacs) on our Java code, but please don't tell James Gosling.

Java's instance variables are unusable:

- They break polymorphism ("shadowing")
- They are settable (without going through class's defined interface)
- Java's instance variables are syntactically different from parameterless methods

Java has class variables - variables that exist once per class rather than being per object. This has pluses and minuses. It's tempting (especially for ex-C-coders) to write Java that looks like an ordinary 3GL rather than well-designed OO code. On the positive side, class variables are useful for utility subroutines (better than multiple inheritance). They are also useful for per-class variables (e.g. preferences).

Java is more dynamic in binding - so there is less compile-time checking, more run-time errors if you are careless. Of course this can be as much a feature as a bug - classes are loaded into the running program dynamically when they are needed, so doing away with the need to re-link programs when subroutines change.

Java isn't as simple and elegant and engineered as Eiffel, but it's not bad compared to C++. It's easy to write awful code in Java, it's easy to write good code in Eiffel.

The senior technical staff at Optimization have been writing system and communications software in various languages for 15 years. We think that when the world went C in the '80's instead of PL/1, Pascal or Modula, our profession took a step back towards assembler. Java gets us back from C to about the same degree of engineering and productivity as those languages.

6. WHAT REALLY MATTERS OVER AND ABOVE THESE TECHNICAL DETAILS

But none of the above technical comparisons really matter much in the bigger picture - where the issue is, for companies and managers, to have programs written on-time, on-budget, and with good quality. For programmers, the issue is to get satisfaction from the creative work they do (rather than have to battle obstacles put in their way like buggy compilers).

What does matter is that Java really is write once run anywhere - this changes a lot of things. First and foremost, developers can devote more of their resources to creative work, rather than porting. The style of the Java development community is also different - the community is in charge and everyone knows all about the technology (and many are contributing) because it is all published (unlike the legacy Windows platform). Java's write once run anywhere property makes software writing, building, and shipping much easier. It fuels the Internet explosion - developers sell or buy software on the net (like CP/M and DOS in the early '80's). Lots of new businesses and business methods are able to spring up on the
Web - using new software. The user is back in charge.

Interestingly, we at Optimation have found that nothing else is keeping up with the pace of new stuff in the Java world. This is actually delivering - very rapidly - Bertrand Meyer's dream of software re-use.

Java might not be the best language, but it's almost the best. And not even the most perfect language available is going to make all the software writers in the world do good design and write clean, bug-free code, and test it completely.

We have also found Java to be fairly reliable - compile-time, virtual machine, and class libraries. Given that we have deliberately tended towards the bleeding edge of Java revisions, we are pleased to be able to report that we can happily run Java code for serious production use. The Java compilers, virtual machine environments, and class libraries aren't bug-free, but with not too much effort you can avoid the bleeding edge bugs and trust Java.

7. OTHER EIFFEL AND JAVA COMPARISONS

The base Eiffel libraries are well designed and work well.

Java's libraries are fairly well designed (some better than others). There were (are) some bugs (especially the GUI toolkit AWT, but in our experience AWT is much better than Eiffel Vision). The Java virtual machine has some bugs too, but we are satisfied that work is progressing apace at JavaSoft and elsewhere.

ISE's Eiffel compiler is very slow - we had to resort to overnight batch compiles for TermSim.

There are few useful debuggers, class browsers, and IDE's for Java, except some stuff on the Windows 95 platform, but nothing like the maturity of serious OO (yet).

Most vendors don't know if/how/when they support Java.

Finally - and most pertinent for CMG - there is very little Java experience in the world - compare for instance 20 years of Pick - e.g. performance questions (and mythical answers) are on everyone's lips.

8. MANAGEMENT LESSONS

We have found that all the standard IT project management lessons apply in spades to Object Oriented projects. They move fast, and you need to be sure they move fast in the right direction. There are also a lot of detailed, often new (with new jargon) technical areas.

Just like assembler and Cobol, you need a good requirements specification, good functional specification, and then you must design, design, design.

If Object Oriented projects move fast, Java moves even faster. There is a wealth of new information appearing (mainly on the Internet) daily, with major new areas (such as new security class libraries, new GUI and 3D class libraries) to learn every month.

We think RAD is not yet doable (Java Beans is perhaps about to change this). But make stepping stones in your project, aim at them religiously, and beware developer-inspired or JavaSoft-inspired function creep.

An important lesson we learned at Optimation is to budget time for systems administration, tools research, and especially keeping up-to-date with the Java world.

9. MYTHS RE JAVA

Here are some myths regarding Java (and Eiffel):

- Java is only good for applets
- Java isn't a standard, or Microsoft could take control/usurp it, ..
- Java will cure all your programming ills. Eiffel will.
- Java programs run slow
- Java is fast

The last two warrant some comment : our experience is that Java is like every other new language/environment/set of libraries, especially those like Pick and Smalltalk that are interpretive. Java is new, and raw. There are indeed some parts of it that are slow, but in general it performs acceptably (surprisingly so in some environments, e.g. Project Rescue on a 386). It is, currently, not wise to use Java for CPU-intensive work, such as weather forecasting or encryption.

The biggest issue, though, is particularly pertinent to CMG - there are little or no hard, scientific facts and only a small amount of
"collective wisdom" regarding good and bad uses of Java, since it is so new. There are just about no useful performance measurement tools or performance meters in Java environments.

10. CONCLUSION
The bottom line is: our software is now complete, and entering new revisions, and new fields of endeavour as well. Java got the job done for us, and gave us productivity, quality, and especially portability. We are enthusiastic adopters of Java.

11. REFERENCES

"Object Oriented Software Construction", Bertrand Meyer
Read it as your OO primer, even if you then fall from grace and use Java or C++

TermSim system testing software by Optimation
www.optimization.com.au

"Introducing Eiffel to a Commercial Software Supplier", Andrew Blucher, Monash University
A description of the first year of the TermSim project at Optimation

Avoiding the Second historic Mistake", Bertrand Meyer, truths and myths about Java

JavaSoft's home page - the start of a wealth of information
http://www.javasoft.com

Australian Java Users' Group home page
http://sunsite.anu.edu.au/ajug/

“Object-Oriented Applications”, Prentice Hall
An informative and entertaining set of case studies