A principled approach to software Engineering Education, or Java considered Harmful

Edmond Schonberg and Robert Dewar
Adacore Inc, 104 5th Avenue, NYC 10011; email: Schonberg@gnat.com, dewar@gnat.com

Abstract

We examine the use of Java as a first programming language, in the light of well-established principles of software engineering, and the increasing concern with correctness, performance, and maintainability. We argue that Java is markedly inferior to Ada or C++ as a language for introductory Computer Science courses, and that its widespread use in the training of tomorrow's software engineers is counterproductive.

Keywords: Software Engineering, Education, Java, Ada.

1 Introduction

It is a well-established fact (first discussed by E.Dijkstra) that the programming language in which programmers receive their first instruction has a large impact on their programming habits. Current instruction in Computer Science (see for example ACM's Computing Curricula 2005 [1] minimizes the teaching of multiple programming languages, which makes the impact of the first language even more critical. Java is more and more the language of choice for introductory programming courses. We argue that this is a poor choice.: we examine the drawbacks of Java as a teaching language under four headings, which following Koolhas et al are conveniently labelled small, medium, large, and extra large [4]. Before delving into the details, let us establish the limits of our arguments:

a) we consider that programming will retain a central role in all software construction: there is no automatic programming machinery in sight that will make programming a secondary activity.

b) Programming remains a demanding intellectual discipline. The separation between "designers" and "programmers" attempts to create hierarchy of skills (and salaries!) but this separation is artificial and counterproductive: software authors (to coin a term) must have a rigorous training that includes solid foundations in software engineering. We are particularly concerned with safety- and security-critical systems, that present considerable engineering challenges.

c) We do not debate the importance of Java in today's software industry, and do not discuss the merits of the language in its industrial and commercial applications: our concern is with the training of soft3are engineers.

d) One of the fundamental skills of a good software engineer is the ability to zoom, that is to say to change the focus of his activity from the very large (software architecture) to the very small (efficiency of generated code, cost of synchronization, etc.). The education he receives must develop this ability, and the languages in which he is taught plays a vital role in this. This argues for the use of a wide-spectrum language from the beginning.

2 Programming in the small

This is the realm of algorithmic analysis: the programmer must be able to estimate reliably the performance of code, in terms of time and space. The disadvantages of Java in this respect are several:

a) The Java virtual machine hides the real architecture.. The JVM is basically a simple stack machine, which makes it easy to port, but it includes some complex operations whose cost will vary from target to target. The use of just-in-time compiling to speed up critical paths makes the performance of a Java program even harder to estimate. It is certainly the case that for many applications (in particular Web programming) a casual approach to performance is acceptable. For safety-critical systems and real-time systems this is not sufficient.

b) Most critically, garbage-collection adds a hard-to-quantify cost to Java programs. Furthermore, the presence of the garbage collector encourages what we might call a profligate style of programming, where objects are created freely for the simplest of computations. For example, object-oriented methodologies encourage the "boxing' of atomic values (transforming an int to an Integer object, for example). So as to honour the concept that "everything is an object". As a result, the simplest computation will involve the dynamic creation of heap-allocated objects, and it will become impossible to estimate the time behaviour of code. This attempt at unification is inspired by Smalltalk, but it is interesting to note that Eiffel abandoned this unified model early in its design [5] and that C++, like Ada, sensibly maintains a clear distinction between elementary values and composite ones. The difficulties of analyzing the performance of large Java systems is vividly described in [7].

3 Programming in the medium

This is the realm of abstraction and encapsulation. In Java (and to a large extent in C++) the fundamental concept is the class, which we must contrast with the various type

constructors in Ada. We include in this category the primitives for concurrent programming.

It is well-known that when designing new abstractions (software components) composition is more important, and used more often than inheritance. Yet in Java composition can only be obtained by delegation, that is to say by embedding a pointer to an object inside of another one. By contrast, in Ada (and to some extent in C++) composition is obtained through aggregation, subtyping, and unions (free in C++, discriminated in Ada). As a result Java design leads to a proliferation of objects, heavy use of dynamic storage, and structures that pointer-heavy and therefore wasteful of storage. The impact of this on performance is well described by Mitchell et all [6]..

3.1 Concurrent Programming

Concurrency is an aspect of Java that is decidedly low-level:

a) Synchronization is per-method, and there is no direct thread-to-thread communication, except through shared memory.

b) Distributed locking operations make it harder to formalize concurrent behaviour, and the suspend/resume mechanisms are notoriously error-prone (race conditions, deadlock).

c) The semantics of priorities and the queueing regime are not defined precisely enough to guarantee real-time behaviour.

Concurrency is much better taught with tasks and protected objects, as presented in [3]: standard concurrent paradigms (producer-consumer, mailboxes, semaphores, broadcasting, etc.) are easily constructed with them. Finally, the use of the Ravenscar profile (part of the Ada 2005 standard) allows the construction of concurrent programs with fully deterministic behaviour.

4 Programming in the large

The only program-structuring mechanism of Java is the class. This the most glaring deficiency of Java from the point of view of software engineering: there is no proper separation between specification and implementation, and there is no mechanism for hierarchical composition of components.

a) The separation between specification and implementation is not just a matter of information hiding (which is handled by public/private dictions in all languages of interest): it is of the greatest importance in the simultaneous development of large systems. In Ada, design starts with package specifications. Once these are agreed upon, development of client code can proceed independently of the implementation of these specifications. Finally, the separation between specification and body simplifies incremental recompilation: a client need not be recompiled when changes in the implementation of a package do not affect its specification.

b) The class is too small a unit out of which to design systems, but there is no grouping mechanism that allows the semantically coherent aggregation of classes. The Java notion of a package is more akin to that of a library of weakly related components. In contrast, the Ada package provides a mechanism for type aggregation, a visible dependency graph through context clauses, and a flexible model of system extensibility through child units.

c) Java cannot deal with subtyping independently of inheritance: there is a conceptual confusion between subtyping as enrichment (the usual notion of inheritance0 and subtyping as subsetting. This is a problem with all O-O methodologies, and is not just a philosophical issue, but one with pedagogical import (see e.g. the discussions around the circle-ellipse relation: which should be considered a subtype of the other?[8]).

After the concept of package, the most important contribution of Ada to software engineering is the notion of constraint (including constraints on scalar types). This notion has no analogue in other languages. Constraints are of course a simple but powerful example of program assertions: they define behaviour more precisely, and they can be checked statically by the compiler, or enforced dynamically. In either case they pin down the semantics of the program in ways that are not available in other languages.

5 Programming in the very large

Most large software systems today combine components that are themselves aggregates (subsystems) consisting of a number of packages or classes, often written in different languages. For the most part the mechanisms for assembling these components are embedded in an Interactive Development Environment (IDE), of which Eclipse is a well-known example. At this level it would appear that the choice of language plays a smaller role, but there are two areas in which Ada presents definite advantages: interfacing with other languages, and static program analysis.

a) Ada formalizes the description of components that may be written in other languages, by means of pragmas (Import, Export, and Convention). The Ada library provides data conversion routines to transform e.g. Ada self-describing strings into C zero-terminated strings, and Fortran numeric, character, and logical types into the corresponding Ada types. These pragmas and library routines allow the Ada compiler to verify the type coherence of a program that has foreign language components. By contrast, the JNI mechanism in Java, and the mechanisms provided by other languages, lose most type checking in the presence of components written in other languages.

b) Larger and faster machines make whole-program analysis possible over programs with tens and hundreds of thousands of source lines. This makes it possible to detect programming defects at compile time (uninitialized variables, race conditions, constraint violations, etc) that

are beyond the reach of unit by unit compilation. However, the power of static analysis depends on the richness of information available to the analyzer, and to a large extent this depends on the richness of the type system of the language. In this context it is useful to think of a compiler as simple theorem prover: every diagnosed error is a proof that a certain invariant is violated somewhere. As with any deductive system, the richer the set of axioms, the more interesting the proofs that can be derived from it. In this sense, redundancy within the program text is beneficial, because it makes it possible to check for consistency. Programmers often regard Ada as too verbose, and balk at the substantial declarative machinery that they have to use, but these declarations are precisely what makes Ada compilers so much more precise in their diagnostics.

There is a continuum between type checking as performed by a compiler, ambitious static analysis as performed by a tool such as SoftCheck's Inspector [7], and program verification as obtained with SPARK [2]. However, what makes Inspector and SPARK possible (and what makes the error messages of a good Ada compiler so precise) is the strong static typing model of Ada. No other language today has a typing system that is rich enough to support such tools. The quality of diagnostics produced by these tools is particularly valuable for beginners, and is a revelation for programmers coming from other languages.

6 Conclusions

We can summarize the shortcomings of Java as an introductory programming language as follows:

a) Java hinders the understanding of code performance.

b) Java design methodologies lead to a proliferation of objects, heavy use of dynamic storage, and data structures that are pointer-heavy and thus wasteful.

c) The Java model of concurrency is low-level and error-prone, and the garbage-collected environmemt prevents its use in real-time applications.

d) The fundamental separation between specification and implementation is absent in Java, hampering good software engineering development practices.

e) Without the notion of constraint, Java has no way of specifying useful invariants to describe program behaviour.

Some of these deficiencies also apply to C++ as an introductory language, even though as a wide-spectrum language it does satisfy the concern with performance analysis described in section 3. There is no further need to enumerate the reasons for the superiority of Ada over either Java or C++ as an introductory programming language.

References.

[1] ACM Computing Curricula 2005. At http://www.acm.org/education/curric_vols, 2006.

[2] John Barnes: High Integrity Software: The Spark Approach to software Safety and Integrity. Addison-Wesley 2003

[3] Alan Burns and Andy Wellings: Concurrent and Real-time Programmming in ada 2005 Cambridge University Press, 2006

[4] Rem Koolhas et al: S M L XL Monacelli Press, 1997

[5] Bertrand Meyer: Object-Oriented Software Construction, Prentice Hall 1997

[6] Nick Mitchell, Gary Sevitsky, Harini Srinivasan : The diary of a Datum: an approach to analyzing runtime complexity in Framework-based applications in Workshop on Library-centric software design, OOPSLA 2005

[7] SoftCheck: Inspector: http://en.wikipedia.org/wiki/SofCheck_Inspector

[8] Wikipedia : Circle-ellipse problem, in

http://en.wikipedia.org/wiki/Circle-ellipse_problem