IN DEVELOPMENT

The Java Tree Withers

Doug Lyon
Fairfield University, Fairfield, Connecticut

The Java report card: infrastructure gets a D, code reuse gets an F.

Computer languages are uniquely positioned as a case study in the development of man's most complex and powerful techno artifact—the programming language. The most popular of these languages is now Java.

In 15 years, my Java project has grown from 667 lines of code (LOC) to 633,436 LOC. During this time, I've had to struggle to keep up with API deprecations and defunct frameworks. Some code will no longer run. What will become of Java?

PREDICTION IS HARD, ESPECIALLY WHEN IT'S ABOUT THE FUTURE

If we establish a good reason for Java's ascendancy, perhaps we can come up with a prognosis for its future.

One of Java's key strengths is its portability. The Java virtual machine (JVM) is a high-performance, portable, and successful substrate. However, the tectonic shift in Java ownership (from Sun to Oracle) has caused aftershocks, destabilizing Java APIs. This has been exacerbated by neglect and deprecation. Deprecation warnings aren't generally deemed serious, as programs continue to run. However, sometimes entire JVMs are deprecated. The biggest threat to Java is Apple's banning of Java on the iPhone and the deprecation of the new JVM on the Mac OS.

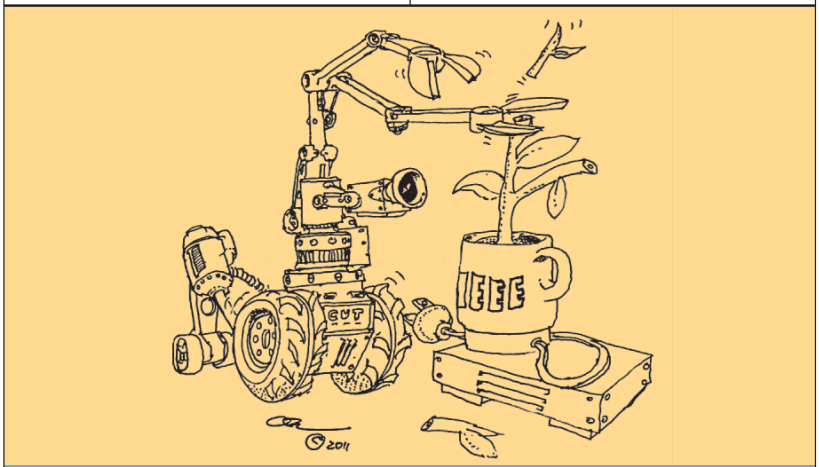

Figure 1. CutBot versus the Java tree. A tree grows from the cup of Java, only to be hacked by the merciless automation.

Java provides a way to isolate and manage API instability using JavaBeans. The bean embodies the promise of Java code-reuse via interchangeable parts—known as component software development (CSD).

Historically, the technology of interchangeable parts enabled an industrial revolution—among other things, it helped the Springfield Armory produce more than 1 million model 1861 rifles during the Civil War. However, JavaBeans never really caught on, leaving us with monolithic and brittle systems that shunned code reuse and encouraged ad hoc framework development. API complexity can reach a tipping point that causes frameworks to collapse under their own weight in a manner similar to punctuated equilibrium.

MEASURING CHANGE

The number of imports and deprecations represents the growth of the new and the withering of the

0018-9162/12/$31.00 © 2012 IEEE Published by the IEEE Computer Society

Table 1. Core Java API.*

Version	Number of imports	Number of deprecations	Release year
JDK1.1	415	41	1997
JDK1.2	1,108	99	1998
JDK1.3	22,545	967	2000
JDK1.5	49,471	1,285	2004
JDK1.6	90,788	1,868	2006

*JDK1.4 is missing as the source code is unavailable

Table 2. Endangered frameworks.

API	Last update
Java Sound	2004
JMF	2003
JAI	2008
JOGL	2008
Java 3D	2008

Table 3. Framework fossils.

Framework	Status
javax.comm	Replaced with RXTX
JNI	Replaced with JNA
QT4J	Dead
JVM on iPhone	Practically outlawed by Apple
JVM on Mac	Deprecated by Apple
FMJ/JMF	Replaced with JavaFX

old. Measurement of deprecations or imports is an easy one-liner in Unix:

```
grep import -d recurse . |
wc -l grep deprecated -d recurse . |
wc -l
```

Based on the information in Table 1, we can make the following observations:

- The number of imports is doubling with every major release.
- The 10-year compound annual growth rate of deprecations is 46 percent. On average, deprecations have doubled every 18 months.

Deprecations are the dark side to API growth and have a nonlinear cost. Interpackage association and framework complexity are functions of the number of imports. The bigger the building, the more it will cost to change the foundation.

Multimedia trench warfare

Developers create frameworks in a competitive environment. For example, while doing multimedia programming, I reviewed the manuscript for a book by Tom Maremaa and William Stewart of Apple Computer titled *QuickTime for Java* (Morgan Kaufmann, 1999). QT4J became the de facto standard for digitization of streaming video on the Mac (and it worked well). However, QT4J wouldn't work on Solaris (since Sun never licensed QuickTime). Apple officially deprecated QT4J in the same year the QuickTime book was published. QuickTime X no longer runs QT4J, and my QuickTime code is dead on that platform. There's no equivalent framework to take the place of QT4J on the Mac, only competing frameworks.

Java Media Framework can digitize video on Windows, but not the Mac. JMF "performance packs" enable fast execution on Windows, but not on the Mac. Worse, JMF first appeared in 1997 and hasn't been updated since 2004. In comparison, FMJ (Freedom for Media in Java) can't digitize video, at least, not on the Mac.

The chilling thought is that Oracle appears to have created a monolithic multimedia replacement for JMF, QT4J, and FMJ. The new technology is called JavaFX. For me, that's more than 6,000 LOC for JMF and another 1,000 LOC for QT4J—at least 7,000 LOC down the flusher. What do these changes cost?

THE COST OF DEPRECATION

Java is the most widely used programming platform on the planet. In 2005, Sun reported there were 4.5 million Java developers. Oracle says that number increased to 9 million by 2010. A global developer population and demographic survey published by the Evans Data Corp. showed that 61 percent of the world's programmers used Java in 2009 (http://tinyurl.com/yduglku). Deprecations impact an entire industry!

Suppose a deprecated LOC can be rewritten for just $10 worth of time (an optimistic assumption, considering testing, documentation, and maintenance). My QT4J-dependent code is 1,000 lines long, so that should cost only $10,000 worth of my time. Suppose 9 million programmers have, on average, 1,000 LOC to modify, at $10 per line. That cost would be $90 billion.

The entire software industry's output in 2008 was only $303 billion (http://tinyurl.com/45ck4v). Deprecation of 1,000 LOC per year,

per programmer, costs 30 percent of worldwide Java programmer productivity. By the way, JMF may be on the way out too—another 6,000 LOC for me.

Is 1,000 (or even 6,000) LOC per year per programmer realistic or optimistic? For example, deprecation of the JDK1.0 "handleEvent" method impacted millions of LOC, and even entire books (including some of my own).

I write in Java to obtain portability so that code can survive the transition from one OS to another. Now we face a different set of problems: code has trouble surviving from one JVM to another. The nice thing about the Java API is that if you don't like it, just wait two minutes—it will change.

WHERE DID ALL THE APIs GO? LONG TIME PASSING

The Java tree is withering and dying from neglect. Table 2 lists frameworks on the endangered list, and Table 3 lists frameworks that are no longer available. Add to that the 1,000 LOC from QT4J and the 6,000 LOC from JMF. And don't even get me started on the 62,000 LOC that depend on Java3D (if only someone would pay me $10 per LOC!).

The new normal means increased framework volatility and interlocking fragility. Framework chaotics require that programmers must adapt or die. I build a moat around core features, coding like I drive—defensively.

The alarming thing about seeing Java's infrastructure showing its age in this way is that it represents a declining Java programming civilization.

What will be the defining idea that will save or kill Java? Will JavaBeans return? Can private ownership and deployment of JavaBeans be the killer application? How can we monetize software components? How can we promote software reuse? Why is reusing software so much harder than rewriting it? How will we control complexity? Should we continue to follow along the Java path? Will Oracle lead Java down the path of salvation?

Oracle paid $7.4 billion (including debt) for Sun and discontinued the Sparc line. Oracle's middleware is built on Java, and they paid for Java ownership (they didn't spend all that money just for storage appliances and Solaris). In a recent webcast on Java strategy, Oracle claimed that it's dedicated to improving Java; however, Oracle has made little effort on frameworks beyond JavaFX. The emphasis of the presentation was on the core runtime, mobile platform/desktop convergence, and the Enterprise Edition (EE).

The trouble with JavaFX is that it's another very large, quickly growing, complex-looking technology. If JavaFX is the answer, then what was the question? How long will it be before JavaFX reaches its tipping point and needs to be thrown out in favor of a new API? Is this the new API life cycle? Java's reputation is at stake, and a reputation isn't a coin that's easily minted.

Without maintenance, the deterioration of the API infrastructure will be the defining idea of what it means to be a programmer of a declining language.

Doug Lyon *is chairman of the Computer Engineering Department at Fairfield University, Fairfield, Connecticut. Contact him at lyon@docjava.com.*

Editor: Chris Huntley, Fairfield University; chuntley@mail.fairfield.edu

Original copyrighted illustration by George Beker, whose iconic 1970s bot drawings were featured in ***Basic Computer Games*** and other publications. An e-book collection of Beker's drawings and observations is available at www.bekerbots.com. All revenue is donated to a nonprofit national children's literacy organization.