Qt vs. Java

A Comparison of Qt and Java for Large-Scale, Industrial-Strength GUI Development

Matthias Kalle Dalheimer
Klarälvdalens Datakonsult AB

kalle@klaralvdalens-datakonsult.se

This white paper compares C++/Qt with Java/AWT/Swing for developing large-scale, real-world software with graphical user interfaces. References are made to independent reports that examine various aspects of the two toolsets.

1. What Do We Compare?

When selecting an environment for a large software development project, there are many aspects that must be considered. The programming language is one of the most significant aspects, since its choice has considerable impact on what other options are available. For example, in a GUI development project, developers will need a GUI library that provides ready-made user interface components, for example, buttons and menus. Since the selection of the GUI library itself has a large impact on the development of a project, it is not uncommon for the GUI library to be chosen first, with the programming language being determined by the languages for which the library is available. Usually, there is only one language per library.

Other software components like database access libraries or communication libraries must also be taken into consideration, but they rarely have such a strong impact on the overall design as the GUI libraries.

In this white paper, the objective is to compare the C++/Qt environment with the Java/AWT/Swing environment. In order to do this in the most useful way, we will begin by comparing the programming languages involved, i.e., C++ and Java, and then compare the two GUI libraries, Qt for C++ and AWT/Swing for Java.

2. Comparing C++ and Java

When discussing the various benefits and drawbacks of particular programming languages, the debate often degenerates into arguments that are based on personal experience and preference rather than any objective criteria. Personal preferences and experience should be taken into account when selecting a programming language for a project, but because it is subjective, it cannot be considered here. Instead we will look at issues such as programmer-efficiency, runtime-efficiency and memory-efficiency since these can be quantified and have been examined in scientifically conducted research, although we will also incorporate information based on the practical exerience of projects that have been implemented in our own company.

2.1. Programmer-efficiency

Programmer-efficiency describes how efficiently (i.e. how quickly and accurately) a programmer with a given degree of experience and knowledge can implement a certain set of requirements in a particular programming language, including debugging and project setup time. Since developer salaries are one of the primary cost factors for any programming project, programmer-efficiency greatly affects the

cost-efficiency of the project. To some extent, programmer-efficiency is also determined by the tools available.

The main design goal of Java is increased programmer-efficiency compared to other general-purpose programming languages, rather than increased memory- or runtime-efficiency.

Java has several features designed to make it more programmer-efficient. For example, unlike C++ (or C), the programmer does not have to explicitly "free" (give back) allocated memory resources to the operating system. Freeing unused memory (garbage collection) is handled automatically by the Java runtime system, at the expense of memory- and runtime-efficiency (see below). This liberates the programmer from the burden of keeping track of allocated memory, a tedious task that is a major cause of bugs. This feature alone should significantly increase the programmer-efficiency of Java programmers, compared to C++ (or C) programmers.

Research shows that in practice, garbage collection and other Java features, do not have a major influence on the programmer-efficiency. One of the classic software estimation models, Barry Boehm's CoCoMo[1] predicts the cost and schedule of a software project using *cost drivers* which take into account variables like the general experience of a programmers, the experience with the programming language in question, the targeted reliability of the program, etc. Boehm writes that *the amount of effort per source statement was highly independent of the language level*. Other research, for example, *A method of programming measurement and estimation* by C.E. Walston and C.P. Felix of IBM[2,] points in the same direction.

Both the reports cited here pre-date the advent of Java by many years, although they seem to reveal a general principle that the sophistication of a general-purpose programming language has, compared with other aspects, like the experience of the developers, no significant influence on the overall project costs.

There is more recent research that explicitly includes Java and which supports this hypothesis. In *An empirical comparison of C, C++, Java, Perl, Python, Rexx, and Tcl*[3], Lutz Prechelt of the University of Karlsruhe, describes an experiment he conducted in which computer science students were assigned a particular design and development task and asked to implement the specification provided in any of the languages C, C++, or Java which they could freely choose according to their personal preferences (the other languages were examined in a different part of the research project). The data gathered shows almost the same results for C++ and Java (with C running third in most aspects). This is also backed up by our own experience: if programmers can choose their favorite programming language (which is usually the one they have most experience of), programmers with the same level of experience (measured for example, in years of programming experience in general) achieve about the same programmer-efficiency. Another interesting aspect that we noted (but which is not yet supported by any formal

research) is that less experienced developers seem to achieve somewhat better results with Java, medium-experienced developers achieve about the same results with both programming languages, and experienced developers achieve better results with C++. These findings could be due to better tools being available for C++; nevertheless this is an aspect that must be taken into account.

An interesting way to quantify programmer-efficiency is the Function Point method developed by Capers Jones. Function points are a software metric that only depend on the functionality, not on the implementation. Working from the function points, it is possible to compute the lines of code needed per function point as well as the *language level* which describes how many function points can be implemented in a certain amount of time. Intriguingly, both the values for the lines of code per function point and the language level are identical for C++ and Java (6 for the language level, compared with C's 3.5 and Tcl's 5, and 53 for the lines of code per function point, compared with C's 91 and Tcl's 64).

In conclusion: both research and practice contradict the claim that Java programmers achieve a higher programmer-efficiency than C++ programmers.

2.2. Runtime-efficiency

We have seen that Java's programmer-efficiency appears to be illusory. We will now examine its runtime efficiency.

Again, Prechelt provides useful data. The amount of data he provides is huge, but he arrives at the conclusion that "a Java program must be expected to run at least 1.22 times as long as a C/C++ program". Note that he says *at least*; the average runtime of Java programs is even longer. Our own experience shows that Java programs tend to run about 2-3 times as long than their equivalent C/C++ programs for the same task. Not surprisingly, Java loses even more ground when the tasks are CPU-bound.

When it comes to programs with a graphical user interface, the increased latency of Java programs is worse than the runtime performance hit. Usability studies show that users do not care about whether a long running task takes, say, two or three minutes, but they do care when a program does not show an immediate reaction to their interaction, for example when they press a button. These studies show that the limit of what a user accepts before they consider a program to be "unresponsive" can be as little as 0.7 seconds. We'll return to this issue when we compare graphical user interfaces in Java and C++ programs.

An explanation about why Java programs are slower than C++ is in order. C++ programs are compiled by the C++ compiler into a binary format that can be executed directly by the CPU; the whole program execution thus takes place in

hardware. (This is an oversimplification since most modern CPUs execute microcode, but this does not affect the issues discussed here.) On the other hand, the Java compiler compiles the source code into "bytecode" which is not executed directly by the CPU, but rather by another piece of software, the Java Virtual Machine (JVM). The JVM in turn, runs on the CPU. The execution of the bytecode of a Java program does not take place in (fast) hardware, but instead in (much slower) software emulation.

Work has been undertaken to develop "Just in Time" (JIT) compilers to address Java's runtime efficiency problem, but no universal solution has yet emerged.

It is the semi-interpreted nature of Java programs that makes the "compile once, run anywhere" approach of Java possible in the first place. Once a Java program is compiled into bytecode, it can be executed on any platform which has a JVM. In practice, this is not always the case, because of implementation differences in different JVMs, and because of the necessity to sometimes use native, non-Java code, usually written in C or C++, together with Java programs.

But is the use of platform-independent bytecode the right approach for cross-platform applications? With a good cross-platform toolkit like Qt and good compilers on the various platforms, programmers can achieve almost the same by compiling their source code once for each platform: "write once, compile everywhere". It can be argued that for this to work, developers need access to all the platforms they want to support, while with Java, in theory at least, developers only need access to one platform running the Java development tools and a JVM. In practice, no responsible software manufacturer will ever certify their software for a platform the software hasn't been tested on, so they would still need access to all the relevant platforms.

The question arises why it should be necessary to run the Java Virtual Machine in software; if a program can be implemented in software, it should also be possible to have hardware implement the same functionality.

This is what the Java designers had in mind when they developed the language; they assumed that the performance penalty would disappear as soon as Java CPUs that implement the JVM in hardware would become available. But after five years, such Java CPUs have not become generally available. There are design studies and also some working prototypes, but it will still be a long time before it is possible to order a Java CPU.

2.3. Memory-efficiency

Java and C++ take completely different approaches to memory management. In C++, all memory management must be done explicitly by the programmer, i.e. the programmer is responsible for allocating and de-allocating memory as necessary. If the programmer forgets to de-allocate allocated memory, they have created a

"memory leak". If such a leak only occurs once during the runtime of an application it may not be a problem, since the operating system will reclaim all the memory once the application stops running. But if the memory leak recurs, (e.g. each time the user invokes a certain functionality) then the memory requirements of the running program will grow over time, eventually consuming all the computer's available memory and possibly crashing the machine.

Java automatically de-allocates (frees) unused memory. The programmer allocates memory, and the JVM keeps track of all the allocated memory blocks and the references to them. As soon as a memory block is no longer referenced, it can be reclaimed. This is done in a process called "garbage collection" in which the JVM periodically checks all the allocated memory blocks, and removes any which are no longer referred to.

Garbage collection is very convenient, but the trade offs are greater memory consumption and slower runtime speed.. With C++, the programmer can (and should) delete blocks of memory as soon as they are no longer required. With Java, blocks are not deleted until the next garbage collection run, and this depends on the implementation on the JVM being used. Prechtelt provides figures which state that *on average (...) and with a confidence of 80%, the Java programs consume at least 32 MB (or 297%) more memory than the C/C++ programs (...)*. In addition to the higher memory requirements, the garbage collection process itself requires processing power which is consequently not available to the actual application functionality, leading to slower overall runtimes. Since the garbage collector runs periodically, it can occasionally lead to Java programs "freezing" for a few seconds. The best JVM implementations keep the occurrence of such freezes to a minimum, but the freezes have not been eliminated entirely.

When dealing with external programs and devices, for example, during I/O or when interacting with a database, it is usually desirable to close the file or database connection as soon as it is no longer required. Using C++'s destructors, this happens as soon as the programmer calls delete. In Java, closing may not occur until the next garbage collecting sweep, which at best may tie up resources unnecessarily, and at worst risks the open resources ending up in an inconsistent state.

The fact that Java programs keep memory blocks around longer than is strictly necessary is especially problematic for embedded devices where memory is often at a premium. It is no coincidence that there is (at the time of writing) no complete implementation of the Java platform for embedded devices, only partial implementations that implement a subset.

The main reason why garbage collection is more expensive than explicit memory management by the programmer is that with the Java scheme, information is lost. In a C++ program, the programmer knows both where their memory blocks are (by storing pointers to them) and knows when they are not needed any longer. In a Java

program, the latter information is not available to the JVM (even though it is known to the programmer), and thus the JVM has to manually find unreferenced blocks. A Java programmer can make use of their knowledge of when a memory block is not needed any longer by deleting all references that are still around and triggering garbage collection manually, but this requires as much effort on the part of the programmer as with the explicit memory management in C++, and still the JVM has to look at each block during garbage collection to determine which ones are no longer used.

Technically, there is nothing that prevents the implementation and use of garbage collection in C++ programs, and there are commercial programs and libraries available that offer this. But because of the disadvantages mentioned above, few C++ programmers make use of this. The Qt toolkit takes a more efficient approach to easing the memory management task for its programmers: when an object is deleted, all dependant objects are automatically deleted too. Qt's approach does not interfere with the programmer's freedom to delete manually when they wish to.

Because manual memory management burdens programmers, C and C++ have been accused of being prone to generate unstable, bug-ridden software. Although the danger of producing memory corruption (which typically leads to program crashes) is certainly higher with C and C++, good education, tools and experience can greatly reduce the risks. Memory management can be learned like anything else, and there are a large number of tools available, both commercial and open source, that help programmers ensure that there are no memory errors in the program; for example, Insure++ by Parasoft, Purify by Rational and the open source Electric Fence. C++'s flexible memory management system also makes it possible to write custom memory profilers that are adapted to whichever type of application a programmer writes.

To sum up this discussion, we have found C++ to provide much better runtime- and memory-efficiency than Java, while having comparable programmer-efficiency.

2.4. Available libraries and tools

The Java platform includes an impressive number of packages that provide hundreds of classes for all kinds of purposes, including graphical user interfaces, security, networking and other tasks. This is certainly an advantage of the Java platform. For each package available on the Java platform, there is at least one corresponding library for C++, although it can be difficult to assemble the various libraries that would be needed for a C++ project and make them all work together correctly.

However, this strength of Java is also one of its weaknesses. It becomes increasingly difficult for the individual programmer to find their way through the huge APIs. For any given task, you can be almost certain that somewhere, there is

functionality that would accomplish the task or at least help with its implementation. But it can be very difficult to find the right package and the right class. Also, with an increasing number of packages, the size of the Java platform has increased considerably. This has led to subsets e.g., for embedded systems, but with a subset, the advantage of having everything readily available disappears. As an aside, the size of the Java platform makes it almost impossible for smaller manufacturers to ship a Java system independent from Sun Microsystems, Java's inventor, and this reduces competition.

If Java has an advantage on the side of available libraries, C++ clearly has an advantage when it comes to available tools. Because of the considerable maturity of the C and C++ family of languages, many tools for all aspects of application development have been developed, including: design, debugging, and profiling tools. While there are Java tools appearing all the time, they seldom measure up to their C++ counterparts. This is often even the case with tools with the same functionality coming from the same manufacturer; compare, for example, Rational's Quantify, a profiler for Java and for C/C++.

The most important tool any developer of a compiled language uses, is still the compiler. C++ has the advantage of having compilers that are clearly superior in execution speed. In order to be able to ship their compilers (and other tools) on various platforms, vendors tend to implement their Java tools in Java itself, with all the aforementioned memory and efficiency problems. There are a few Java compilers written in a native language like C (for example, IBM's Jikes), but these are the exception, and seldom used.

3. Comparing AWT/Swing and Qt

So far, we have compared the programming language Java and the programming language C++. But as we discussed at the beginning of this article, the programming language is only one of the aspects to consider in GUI development. We will now compare the packages for GUI development that are shipped with Java, i.e. AWT and Swing, with the cross-platform GUI toolkit, Qt, from the Norwegian supplier, Trolltech. We have confined the comparision on the C++ side to the Qt GUI toolkit, since unlike MFC (Microsoft Foundation Classes) and similar toolkits, Qt runs on all 32-bit Windows platforms (apart from NT 3.5x), most Unix variants, including Linux, Solaris, AIX and Mac OS X, and embedded platforms, making C++ with Qt the closest match to Java with AWT and Swing.

3.1. Properties of AWT, Swing, and Qt

AWT ("Abstract Windowing Toolkit") was shipped with the very first version of Java. It uses native code (e.g. the Win32 API on Windows and the Motif library on Unix) for the GUI components and implements a portable wrapper around these. This approach means that an AWT program will look and behave differently when run on a different platform, since it is the platform that actually draws and handles the GUI components. This seems to contradict Java's cross-platform philosophy and may be due to the the initial AWT version being reputedly developed in under fourteen days.

Because of these and a number of other problems with the AWT, it has since been augmented by the Swing toolkit. Swing relies on the AWT (and consequently on the native libraries) only for very basic things like creating rectangular windows, handling events and executing primitive drawing operations. Everything else is handled within Swing, including all the drawing of the GUI components. This does away with the problem of applications looking and behaving differently on different platforms. Unfortunately, because Swing is mostly implemented in Java itself, it lacks efficiency. As a result, Swing programs are not only slow when performing computations, but also when drawing and handling the user interface, leading to poor responsiveness. As mentioned earlier, poor responsiveness is one of the things that users are least willing to tolerate in a GUI application. On today's standard commodity hardware, it is not unusual to be able to watch how a Swing button is redrawn when the mouse is pressed over it. While this situation will surely improve with faster hardware, this does not address the fundamental problem that complex user interfaces developed with Swing are inherently slow.

The Qt toolkit follows a similar approach; like Swing, it only relies on the native libraries only for very basic things and handles the drawing of GUI components itself. This brings Qt the same advantages as Swing (for example, applications look and behave the same on different platforms), but since Qt is entirely implemented in C++ and thus compiled to native code; it does not have Swing's efficiency problems. User interfaces written with Qt are typically very fast; because of Qt's smart use of caching techniques, they are sometimes even faster than comparable programs written using only the native libraries. Theoretically, an optimal native program should always be at least as fast as an equivalent optimal Qt program; however, making a native program optimal is much more difficult and requires more programming skills than making a Qt program optimal.

Both Qt and Swing employ a styling technique that lets programs display in any one of a number of styles, independent of the platform they are running on. This is possible because both Qt and Swing handle the drawing themselves and can draw GUI elements in whichever style is desired. Qt even ships with a style that emulates the default look-and-feel in Swing programs, along with styles that emulate the

Win32 look-and-feel, the Motif look-and-feel, and—in the Macintosh version— the MacOS X Aqua style.

3.2. Programming Paradigms In Qt and Swing

While programming APIs to some extent are a matter of the programmers' personal taste, there are some APIs that lend themselves to simple, short, and elegant application code far more readily than others. Below we provide two code snippets, the first using Java/Swing, the second using C++/Qt; both snippets insert a number of items in a hierarchical tree view GUI component. Swing code:

```
...
DefaultMutableTreeNode root = new DefaultMutableTreeNode( "Root" );
DefaultMutableTreeNode child1 = new DefaultMutableTreeNode( "Child 1"
);
DefaultMutableTreeNode child2 = new DefaultMutableTreeNode( "Child 2"
);
DefaultTreeModel model = new DefaultTreeModel( root );
JTree tree = new JTree( model );
model.insertNodeInto( child1, root, 0 );
model.insertNodeInto( child2, root, 1 );
...
```

The same code using Qt:

```
...
QListView* tree = new QListView;
QListViewItem* root = new QListViewItem( tree, "Root" );
QListViewItem* child1 = new QListViewItem( root, "Child 1" );
QListViewItem* child2 = new QListViewItem( root, "Child 2" );
...
```

As you can see, the Qt code is considerably more intuitive. This is because Swing enforces the use of a Model-View-Controller architecture (MVC) while Qt supports, but does not enforce, such an approach. Comparing the code for creating a table with data or other complex GUI components leads to the same results.

Another aspect to consider, is how the various GUI toolkits relate user interaction (like clicking on an item in the tree views created above) with program functionality (executing a certain function or method). Syntactically, this looks entirely different in Java/Swing and C++/Qt, but the underlying concepts are the same, and it is difficult to says whether the Swing example,

```
...
tree.addTreeSelectionListener( handler );
...
```

or the Qt example,

```
...
connect( tree, SIGNAL( itemSelected( QListViewItem* ) ),
         handler, SLOT( handlerMethod( QListViewItem* ) ) );
...
```

leads to more elegant or more robust code. At first sight, the Swing example looks simpler, but the Qt code is more flexible. Qt lets programmers give the handler method any name they like, while Swing forces it to be called `valueChanged()` (which is why it is not explicitly named in the Swing example above). Qt also makes it simple to connect an event (a *signal* in Qt terminology) to any number of handlers (*slots*).

To sum up this section, both Java/AWT/Swing and C++/Qt support the development of sophisticated user interfaces. Swing user interfaces are invariably hampered by Java's general problems with runtime- and memory-efficiency.

4. Conclusion

We have compared the two development platforms Java/AWT/Swing and C++/Qt regarding their suitability for efficiently developing high-performance, user-friendly applications with graphical user interfaces. While the Java-based platform only manages to achieve comparable programmer efficiency to that of the C++/Qt platform it is clearly inferior when it comes to runtime and memory efficiency. C++ also benefits from better tools than those available for Java.

When it comes to the GUI libraries, Swing and Qt, the poor runtime-efficiency of Java programs is clearly evident, making the Java/Swing platform unsuitable for many GUI development efforts, even though the programming experience is comparable. Since Qt does not enforce particular programming paradigms as Swing does with the Model-View-Controller paradigm, Qt programmers often achieve more concise code.

Both independent academic research and industrial experience demonstrate that the hype favouring Java is mostly unjustified, and that the C++/Qt combination is superior. This is mainly due to the runtime and memory efficiency problems in Java programs (which are especially striking when using the Swing GUI toolkit) and Java's failure to deliver increased programmer efficiency. In various programming projects we have been involved in, junior-level programmers learnt Java faster, but more experienced and senior-level programmers (which are usually in charge of the application design and the implementation of the critical parts of an application) achieved better and faster results using C++.

Java/Swing may be appropriate for certain projects, especially those without GUIs or with limited GUI functionality. C++/Qt is an overall superior solution, particularly for GUI applications.

References

1. *Software Engineering Economics*, by Barry Boehm, Prentice Hall.

2. *A method of programming measurement and estimation*by C. E. Walston and C. P. Fexix, IBM.
3. *An empirical comparison of C, C++, Java, Perl, Python, Rexx, and Tcl* by Lutz Prechelt, University of Karlsruhe.