

# Barbarians at the Gate

A report from the frontier where Open Source and Java meet

by Andrew Cowie, *Operational Dynamics*

## Introduction

'Open Source' means different things to different people. For some it's a business model. For others it's a way of collaborating. Some see a way of reducing costs. And some are out to change the world.

That may sound a touch dramatic, but it's a pretty accurate description of the landscape. So let's look at each of these and see if we can't point out something interesting and perhaps unexpected about how Java and Open Source are interacting.

## Open Source as collaboration

Although there has long been the gripe that Sun's Java implementation itself is not open source (not yet, anyway), huge amounts of work have been done on software that supports the construction of sophisticated applications and platforms that *is* open source, especially under the auspices of the Apache Foundation. Having started as a simple Java servlet runner called `Tomcat`, the Jakarta project has blossomed into a diverse group of frameworks for common tasks, web development, and enterprise applications.

Apache aren't the only ones. One of the amazing things that even most Linux and Open Source advocates don't realize is just how much open source activity based around Java. A huge percentage of the active projects listed at FreshMeat or hosted on SourceForge are written to run in Java VMs. You see the same level of collaboration in other dedicated communities including Codehaus and ObjectWeb, and, of course, the freight train that is the Eclipse Foundation.

When most people think about the products coming out of the Eclipse community, they think about the `Eclipse` Integrated Development Environment used to write and debug Java code. That's not wrong, *per se*, but it's easy to overlook the fact that from the beginning Eclipse was designed as a *platform* upon which applications can be built. Several major vendors have recently made decisions to base their products on top of the `Eclipse Rich Client Platform`. The next version of Lotus Notes is going to be built on this technology. This will of course be transparent to corporate users, but is great news for developers: as ever with open source, the work that Notes engineers put into improving the Eclipse platform for their own needs will in turn benefit everyone who uses Eclipse technologies.

All of these are examples of the essence of the open source movement: collaboration. But why do people do it?

## Open Source reducing costs

One of the common arguments advanced when explaining why organizations contribute to open source is to help reduce development costs. Part of what any company does add value – be this in terms of proprietary processes, specialized knowledge, or other unique abilities that give them their competitive advantage. At the same time, much of what has to be done in order to reach the point where they can provide that value is common ground they share with others in the industry; and that is where collaboration, even with competitors, makes sense. We frequently see this in infrastructure – do you *really* need to create your own Java logging framework? No, of course not – there are already several excellent ones available (even over and above that Java itself absorbed the open source originated `Log4j`). On the other hand, if you've had to go through the trouble of creating a library to do some set of complex yet common tasks that others likely have to grunt through as well, does it make sense to maintain that library yourself, or to work with others in the space to, thereby allowing you to take advantage of improvements and fixes that they might make and quite frequently enabling new functionality you wouldn't have been able to afford? Really, each of these

examples is nothing more than a joint venture, which is certainly a common enough structure in the business world. The next time you're arguing in favour of using an open source solution, you might bring this up.

And it's not just development costs. Not to be sneezed at is the fundamental premise that access to the code makes maintenance easier – and many cases is the only thing that makes it possible. Being able to see what is going on in an application or platform is often absolutely necessary to diagnose and troubleshoot problems. This is applicable up and down the IT stack, but is particularly relevant for Java developers trying to squeeze every last drop of performance out of a heavily loaded platform. If you're running an enterprise application server, the combination of `JBoss` and `Hibernate` on top of one of the open source databases is hard to beat. Being able to deploy on commodity hardware and leverage non-traditional architectures has led to a price/performance ratio that is hard to beat.

Just as important are infrastructure projects. Its easy to forget that tools which are a fundamental part of our daily work like `JUnit` and `Ant` are open source – and this is a case where the price is right indeed. Further up the stack are more comprehensive tools for a range of tasks – projects like `Maven` (a comprehensive build and deployment tool), continuous integration systems like `CruiseControl` and `BeetleJuice` which help automate the task of building and testing large software projects, and systems which bring knowledge together, like `Cargo`.

That last one bears a more detailed mention: One of my clients has me working on revamping the infrastructure they use to build their products and run functional tests across them. They're a Java shop, and so it's no surprise that their product, a rather large web application, is built in Java Servlets and JSP; since they target a wide range of enterprise customers they need to test their app in as many application server "containers" as possible.

Not terribly unusual, but when you're trying to run *automated* tests, it gets tricky. Although in theory one should be able to interchangeably use different app-servers, we all know that the different vendors (open source and commercial) who have implemented the Servlet, JSP, and J2EE specs have all done it differently. Even assuming the application you are testing doesn't use vendor specific extensions, you still have to deal with the problem of setting up, starting, and stopping the app-server containers themselves. And as you'd expect, each different app-server has a rather significantly different way of being configured and run.

That's where `Cargo`, a project hosted by Codehaus, comes in. Working together, groups from different environments have concentrated the knowledge of how to configure, start, and stop a wide range of different containers into a simple API that you can use from within a Java program or you can add to your `Ant` or `Maven` scripts. And this allows you to dramatically accelerate your testing.

Infrastructure, tools, automation, and testing. All of these taken together have reduced barriers to cross-platform development enabling Windows, Unix, and Linux and Mac developers to work together, helping broaden audience and usage while reducing cost and complexity.

## Open Source as business model

While some companies have leveraged Open Source as a way to reduce the cost of doing business, others have **chosen** Open Source as the *basis* for their business!

Object relational mappers are a necessary evil if you're forced to deal with a legacy SQL database system, but many developers, especially those working on small mobile or disconnected systems, have neither the resources nor the inclination to deal with such enormous complexity and in any case can't afford the footprint of such a combination on their devices. Enter `db4o`.

`db4objects`'s Java native embedded persistence solution, `db4o` is remarkably easy to use – developers can be up and running in less than 10 minutes. And that's not 10 minutes of figuring out how to install the thing, that's a few brief lines of code and you've got a complete database which stores your data in the object oriented form in which you work with it.

```
ObjectContainer db;
db = Db4o.openFile("warehouse.db");

// your code here. Perhaps...
Inventory a = new FastMovingItem();
q.setQuantity(400);

db.set(q);
db.commit();
```

And persisted! Queries are just as easy,

```
Inventory proto = new Inventory();
proto.setName("fruit");

List results = db.query(proto);
```

One of the great things about db4o is that it's native Java. You just feed it Java objects, and get Java objects back out again – no need to translate your data model to some third-party pseudo representation nor any requirement to write endless metadata in XML. It does a really straight forward job of just getting on with persisting Plain Old Java Objects. No mucking around with byte code enhancers (like JDO) and certainly none of the self-mutilation that goes with EJB.

db4o has proved really easy to use. You do have to wrap your head around the notion that you don't need to worry about foreign keys anymore – when you want a reference to another bit of data you just use a plain old Java variable ... because of course variables and instance fields are references in Java. Neat. The most remarkable part, though, has been the degree to which it *enables* object-oriented data models. For so long we've been forced to hamstring our domain object design in ways that are largely flat in order that we can shoehorn them into legacy rows-and-columns relational databases. With db4o, you can develop elaborate data models with complex object graphs and deep inheritance hierarchies and then persist that information without any fuss. Quite the productivity boost.

There have been some remarkable success stories with db4o. Intel is using it in chip fabrication plants; it's embedded in the robots Bosch manufactures for food preparation and consumer goods packaging; Boeing is using it for a new multi-mission aircraft and BMW have built it into their latest cars. Customers can't say enough about the short start-up times, zero administration, and low memory footprint as they enthuse about why they chose it for their products. Not bad for a database that comes in a 400kB jar file.

db4objects has also recently released a replication tool called dRS that allows you to exchange data not just with other db4o instances but with relational database systems as well. This makes it easy to design systems where a large enterprise data store contains information which workers need to access from disconnected devices. This is a common problem – a salesperson on the road needs customer data; someone working in a warehouse needs to be able to collect and propagate inventory information; medical devices need to gather and process vast amounts of data from a patient. In all these cases, mobile devices have limited resources and need to exchange information with larger systems. db4o excels at this.

Christof Wittig, CEO of db4objects, explains his company's choice to release their software this way: “We live in a post-materialist world. People don't have long lead times to do everything themselves. It's only through collaboration that we can empower each other to succeed.” Asked why they support open source use, “That's easy. We owe everything to the community. From the outset, the user and developer community has helped to shape and refine the product design and direction. We wouldn't have had that if we weren't open source. Even more importantly, our product is already easy to use; open source means that db4o is easy *try* and that in turn wins us customers.”

db4objects, Inc releases db4o under a dual licence model. Manufacturers can choose a commercial licence if doing proprietary embedded work for a surprisingly modest price per unit. And if you want to evaluate its use, are doing in-house work, or wish to use db4o in your own open source projects, then you can use db4o freely under the terms of the GPL.

Obviously we've been talking Java here, but developers required to do cross-platform development in heterogeneous environments will be interested to know that there is a .Net version of db4o available as well. The two are entirely binary compatible; a number of their customers are running db4o as the storage back end on Java application servers while writing their clients in little .Net programs. Yet another example of open source bringing communities together.

## Open Source to change the world

For most people, 'Open Source' means cost free. But a growing number of people have come to really believe in what Free Software is really about – freedom. Freedom to innovate. Freedom to modify and to help others by your work. It's the way the information age began.

Over the last few years an increasing amount of effort has been invested in what I'll term 'Free Java'. As ever, the word *free* is a tad inadequate; the open source movement encourages you to realize that their essential message is freedom as in *liberty* (to modify a program, to enhance it or reuse it as you see fit to redistribute) in addition to the notably useful property of being free as in *price* (though many of us gladly pay our distributor or vendor to package it all up for us and to provide us support).

Sun's Java VM is, of course, *not* free and open source software and worse, until recently their licence was such that even redistributing their Java binaries was forbidden. So, not unexpectedly, the free software community has been creating an implementation of Java which *does* give you those freedoms.

The GNU Classpath project is working towards providing a fully compatible set of class libraries and supports a dizzying array of implementations of Java Virtual Machines – from research VMs like Kaffe and SableVM to the quaint little JamVM to the rocketing CACAO and JikesRVM and the truly radical GCJ project.

GCJ deserves special mention. 5+ years ago, hackers at Cygnus (now a part of Red Hat) realized that one way they could look at Java would be to consider it a specifically defined subset of C++ (in the same sense that XML is a specifically defined subset of SGML). They reasoned that if they wrote a first stage compiler that would take Java in the front end and spit out the tree representation used by the gcc compiler internally, they could then take advantage of the tremendous power of the whole GCC suite behind it to optimize that code and link it to binary executables that would run on any of the many platforms that GCC already supports.

And ta-da, one of the consequences of the Free Java effort is that you can run Java programs on virtually every architecture out there. No longer are you limited to the few platforms that the major vendors create their runtimes for – a significant factor for embedded developers. Concurrent efforts to create a tight but highly performing garbage collector have meant that you can now create efficient and optimized programs with small footprints that {gasp} are written in Java.

Ahead of time compilation (AOT) is somewhat unusual for us to think about – we're quite used to the just in time compilation (JIT) that has been part of commercial VMs since the Java 1.1 days. JITs have a particular challenge though – they have to turn the intermediate representation (Java byte code from a `.class` file) and into native machine code fast enough that the user doesn't notice. That's a tall order, yet it's amazing how well modern VMs do at this! There is, nevertheless, a limitation at how good an optimization a JIT compiler can achieve because of the limited time it has available. There's also the question of how much computing power is taken to *do* the JIT compilation – horsepower that may not be available, especially on resource limited machines like small embedded devices. By doing the compilation to native machine code ahead of time, GCJ is not constrained at the need to provide near instantaneous compilation and has the luxury of being

able to try more in depth optimizations. The result is fast code and even better, there is no start up penalty when a Java program is launched. Instead of the overhead of instantiating a massive VM and struggling through the JIT compilation of an enormous number of core classes, the code is already native and so can immediately start executing – brilliant for small processes and desktop applications.

Red Hat in particular has invested significant effort into improving GCJ and the Classpath libraries to achieve these outcomes not just on small systems but on enterprise servers as well. This has had impressive results: major projects like `Eclipse` and `JOnAS` can be built to native resulting in *considerable* performance boosts. And if you're running the Fedora Core Linux distribution, then any time you install a Java library it will be compiled to native and transparently used to speed up your programs.

And there's even more to report from Free Java land. Recently a new project being incubated by the Apache Foundation has emerged on the scene. The effort, called `Harmony`, also aims to develop a fully compatible open source class library and VM implementation. It doesn't have much of a community behind it as yet but they have been getting impressive donations of code from several major players in the Java space.

So across the board, activity is up. Amazingly, some lament the bevy of choices maturing in the Java world, complaining that they have to invest effort into figuring out which solution is best for them. Ignoring for the moment that it has always taken hard work to figure out what the best course to take on an IT project will be, Java developers should be glad of all the competition. As all these projects jostle together, collaborating where they have common interest and competing where they think they can do better than the others, better platforms emerge, and that benefits all of us that write Java programs.

## Conclusion

All is not quiet on the Open Source front. From simple collaboration to unexpected innovation on platforms tiny or huge, Java stakes out a vibrant place in the open source pantheon.

Of course, we can't end an a discussion about Open Source and Java without at least briefly touching on the strong signals coming out of Sun Microsystems that they will, in the not too distant future, open their Java implementation. It remains to be seen whether Sun will choose a licence that will *actually* qualify under the Open Source Definition, but early indications are that they have listened to people across the FOSS community and know they have to go all the way.

And then watch at amazement at the hordes who will be using Open Source. Keep watch at the gates!

## Author



Andrew Cowie is a long time Unix and Linux user & advocate, but somewhat unusually was an infantry officer in the Canadian army, having graduated from Royal Military College with a degree in engineering physics. He saw service across North America and a peacekeeping tour in Bosnia. He later ran operations for an SMS company in Manhattan and was a part of recovering the firm after the Sept 11 attacks.

Now based in Sydney, Andrew runs [Operational Dynamics](#), a consultancy helping clients worldwide with crisis management: crisis prevention, proactively evaluating organizations for resiliency and disaster preparedness; change management, helping plan and execute major upgrades to their critical infrastructure; and troubleshooting, working with people to solve difficult problems in uncertain and complex environments, be they financial or technical ... at levels ranging from the board suite to the engineers doing hands-on implementation.

Andrew is a frequent conference speaker, presenting about increasing the level of professionalism in the operations world.

On the technical side, Andrew has extensive experience as a Unix/Linux sysadmin, Java developer, and has long been an Open Source advocate. Most recently, he has been contributing to the GNOME project by maintaining the `java-gnome` bindings allowing you to write GTK programs from Java.

You can reach him at [andrew@operationaldynamics.com](mailto:andrew@operationaldynamics.com) or as AfC on [irc.freenode.net](irc://irc.freenode.net) or [irc.gimp.net](irc://irc.gimp.net)