Axel Uhl, Johannes Link

Java Applied to Logistics for Power Plant Construction

Abstract

This paper describes the use of Java for developing a mission critical application in ABB's power generation segment. It presents the rationale for choosing Java, the experience made with both Java itself and diverse tools, how the logistics challenge was approached with Java, and issues to be considered when building multi-tier internet applications. As a conclusion some suggestions are made about where further research efforts for Java and OOP in general could be necessary.

1 Introduction

ABB is a global company operating in about 140 countries around the world. A very important business segment of ABB is *power generation*. One of the artifacts of the power generation segment is turnkey power plants – fossile, hydro and nuclear. In a segment-wide effort a vision was crafted on how speed, quality and efficiency could be increased in turnkey plant construction. During this process, logistics was identified as a central point for optimization.

Several weak spots in the current logistical process were identified:

- the flow of information is too slow,
- data is inconsistent or missing,
- current systems are far too heterogeneous.

These problems prevent the coordinating manager(s) from getting a comprehensive and up-to-date overview about what's going on. A solution was needed that would harmonize the existing systems, speed up the communication between all process participants, check information consistency and provide an easy-to-use GUI. Therefore, a project called *World Wide Logistics System (WWLS)* was launched.

After evaluating different technologies with respect to the project goals (*Lotus Notes*, *Microsoft Access* and a proprietary *C++*-implementation) we finally decided in favor of Java as development platform. The main reasons were the good library support for any kind of online application, the clear language structure and the expected industry support for the Java environment as a whole. Portability was deemed important as well since in an environment with lots of communication partners no a priori predictions about platforms can be made.

The remainder of this article is structured as follows: Section 2 gives an overview of our experiences with applying Java to an industrial problem. It explains the problem we had to solve and how we did this. In section 3 some light will be shed on the software process and its peculiarities, benefits and challenges in connection with Java. Our conclusions are wrapped up in section 4.

2 Logistics with Java

This section will explain our thoughts on multi-tiered architectures, their realization as Java applets or applications and related security aspects.

2.1 Three-Tier Architecture: Scaling the Client

The first application in the WWLS project we had to build was a supporting tool for the packing firms. They have to exchange information on the packing state of items and packaging orders with the head office. They did that on paper and by exchanging floppy disks so far which for obvious reasons is slow and error-prone.

The chosen solution to these problems is a three-tier architecture, connecting the packing companies with the head office to allow online downloads of packaging orders and uploads of part and packing information.

However, the clean three-tier approach does not work for all distributed multi-tier applications. Especially when the communication link between the client and the middle tier is unreliable or extremely slow there might be problems when executing the complete business logic on the middle tier. A thin client usually handles nothing more than visualization and data entry; running all process-relevant or transactional steps on the middle tier requires heavy communication back and forth. Thus the link from the client to the middle tier becomes a mission critical bottleneck.

Furthermore, using an unmodified three-tier approach would have forced us to keep the packing site online during the whole data entry process. Entering thousands of delivery parts and their pertinent packing information, though, can take several days. In most non-US countries dial-up links to Internet providers are charged by the minute, therefore avoiding long lasting online connections is an important economical factor.

Instead, smart replication strategies are required to keep the "air time" as short as possible. This of course can only be achieved by replicating considerable parts of the business logic to the client as well. This in turn causes a rather fat client as compared to a GUI-only frontend. All consistency checks have to be done offline. Parts of the replication and locking strategy have to be implemented on the client. Even some aspects of the workflow and the processing rules must be present at the client.

Obviously, a distinction is required between three-tier architectures with a tight coupling between client and middle tier and those architectures with a rather loose coupling between those two tiers. A loose coupling requires replication, not only of data but also of parts of the business logic, whereas tightly coupled systems can be implemented with a much simpler approach, assuming there's a stable and fast enough connection between all tiers. Loose coupling, though, is the preferred approach in environments where links are unreliable or where the ability to work in an offline mode is mission critical for the application.

This kind of replication task with all its problems and challenges like locking strategies, conflict resolution, fallback strategies for long down times, smooth transition between online and offline modes etc. should be addressed as a topic of basic research. It is inherent to many multi-tier architectures, but right now we don't see any standards or products emerging in this area.

2.2 Applets and Applications

Java has for quite some time been regarded as a technology to vamp up web pages with embedded applets. Their functionality reached from animated images to tree controls for navigating a web site. As web browsers enhanced and stabilized their Java support, more and more small business applications were deployed through applets running in browsers. This trend has raised a few problems: Whereas thin frontends can be easily deployed through a browser, full-blown applications must often go far beyond what a browser environment will allow them to do. Therefore, instead of being shipped as an applet an increasing number of Java applications are now delivered as standalone applications. The reasons are manifold. In particular, disadvantages of the applet approach are:

- restrictions on file access and socket communication imposed by the browser's security manager
- performance bottleneck through repeated download of applet bytecode
- incompatibilities due to varying level of Java support across different browsers

In our case we were initially seduced by the simplicity of the browser-based deployment of applets. Eventually we realized all the aforementioned issues and switched over to delivering a standalone application.

2.3 Internet and Security

When clients communicate with the middle tier across public Internet connections, encryption and authentication play an important role. This becomes obvious when thinking about mission-critical transactions being performed across such links. Neither does a corporation want others to tamper with their business data, nor does it want others to read the contents of any of its business transactions.

Java's *Remote Method Invocation (RMI)* ([RMI98]) architecture enabled us to transparently integrate these two aspects with our system. Since we based the communication between client and middle tier on RMI, we were able to use its so-called *socket factory* interface[1]: We implemeted an SSL-like protocol, wrapped it with a standard socket interface and wrote a socket factory returning this type of socket instead of the standard Java sockets. This way remote method calls use encrypted and authenticated sockets instead of the standard ones for parameter and result passing. This solution can be integrated into any application using RMI.

3 The Software Process Around Java

Whereas in the previous section we described application specific issues, this section focuses on problems arising during the development process.

3.1 Using an "Internet Language"

One of the downsides of using Java, a language whose evolution is driven and accelerated by the Internet community, are its rapid change cycles. The term "Internet-year" has become

[1]an abstract factory class which internally handles the creation of new client and server sockets

popular for describing a time span somewhere between a week and a month, demonstrating the vast velocity "the web" is moving forward with. In such an environment, where a standard is pushed ahead so massively, two things are hard to achieve: good quality and reuse. Before a component is fully documented, tested, debugged and deployed it has often become obsolete, either because the underlying platform changed in an incompatible way or because somebody else already came up with a component doing the job. This drives component vendors into faster and faster development cycles with quality as the first casualty.

But worse than the decreasing robustness of components and products in the Internet marketplace is the fact that reuse becomes close to impossible, in particular for those kinds of components heavily dependent on user interface aspects of an application or the platform itself. Examples are components that were developed to take care of deficencies in the Java Abstract Windowing Toolkit (AWT) became obsolete due to the advent of the *Java Foundation Classes* library (JFC). The same holds for components being based on particular GUI styles and fashions which might be obsolete with the next release of Microsoft Office.

The hope remains that, as Sun announced earlier this year, after the release of the Java Development Kit (JDK) 1.2 at least the Java platform itself will stabilize, so that developers don't have to hassle with changing APIs, changing package names and changing language specifications.

3.2 Configuration Management, Deployment and Update

Since Java is based on the notion of source code files[2], virtually any file-based configuration management tool can be used with it. Initially we started with a low-end RCS-based solution [Tichy85], putting some shell scripts on top of RCS to allow easy file replication to different user directories. Like many things that start out little this solution grew into a highly customized set of software process support tools written in Java, which we call the *Xtools*. Three major features of the Xtools should be mentioned here:

Changesets: In order to help developers in keeping the archive consistent and compilable we implemented the notion of *changesets* [Dart90], [LDC+89] which describe a transaction of related check-ins and check-outs leading to a consistent and compilable state of the application.

Metrics: Based on the knowledge about *releases* (frozen and named states of the whole archive) an extensible set of software metrics [Lore94], [Camp94] considering the whole project history can be computed, thus displaying crucial trends.

Automated Updates as described below.

3.2.1 Automated Updates

There are off-the-shelf products offering installation support for Java applications (e.g. *InstallShield* or *InstallAnywhere*). They also allow you to more or less automate the setup of a Java runtime environment. But they don't support updating an already installed application. Reinstalling, though, causes long download times, where most of the data is redundant in that it didn't change since the last installation.

[2]as opposed to e.g. Smalltalk's *image approach*

Other products specialize in online updates, like *Marimba Castanet* or *Microsoft Channels*. With those, the client can update an installation by the click of a button which delivers the advantage of an applet deployed through the browser to a standalone application. The problem with these tools is that they are hard to integrate with the development process. It causes too much effort to feed those tools with the information about what changed since the last release manually. Best would be a coupling with the software revision control system in use.

We tackled this by developing an integration of a little update utility with the Xtools. The update client is deployed with the application and accesses update information on the server which, in turn, is automatically generated out of the Xtools log files. Thus, the client has a similar one-button update feature as with tools like Castanet. Moreover, it is fully integrated with the development cycle.

3.3 Documentation

The Java development environment inherently fosters keeping the code documentation up to date. With its `javadoc` tool it is possible to generate HTML documentation from the source code, given that the developers used some simple tags to annotate class and method comments. The results are impressive. With some additional effort for documenting the overall architecture and the package contexts we were able to have a complete online documentation of the full system in a very useful format. This *single source approach* for code documentation is nothing new. Tools like *Together C++* have used them for some years already. But with Java this approach is designed into the standard environment and therefore accepted in a much broader community. We see this as an important contribution to software quality.

3.4 Testing

Testing is a very important issue for the quality of software. Although this has been widely acknowledged as a fact [Siegel96], only mature software companies have an established "testing culture" within their development process. There exist, of course, many tools for supporting and facilitating the test of Java programs:

- Tools to statically test the code for conformance to some well-known programming guidelines (e.g. Parasoft's *CodeWizard for Java*)
- Tools for recording and replaying GUI interactions (e.g. Sun's *JavaStar* [SunTest])
- Tools for specifying and executing unit-based test cases and test suits (e.g. Sun's *JavaSpec* [SunTest])
- Tools for coverage analysis (e.g. SUN's *JavaScope* [SunTest])

Since the introduction of *systematic* testing into a software process cannot be done in a single shot, we decided to take a pragmatic way as a start which did not require the purchase of tools in the first place: We adopted Kent Beck's testing framework for Smalltalk (described in [Beck94]) to Java.

This framework facilitates the writing of unit-based test cases and their combination into test suites. Test suites – or test cases – can then be executed; the results of these program runs - including execution times, unfulfilled postconditions as well as uncaught exceptions - are returned as an object and can be printed and/or stored to disk. To be as close as possible to Kent Beck's Smalltalk implementation we used Java's reflection API for important pieces in our adaption. Extensions to the framework - like semi-automatic input data generation or a GUI for creating test cases - should be easy to implement.

3.5 Development Tool Integration

Looking at the tool environments we see a heavy lack of integration: Standalone design tools that do not fit in with IDEs; IDEs that have no open interface to configuration management systems and standard editors; debuggers that cannot be separated from their IDE; configuration management tools that are either incredibly expensive or not powerful enough to meet advanced needs.

The fact that most of these tools just refuse to integrate with one another makes development more awkward and annoying than it has to be. What developers need are software development tools that behave like components in the true sense of the word, i.e. they are open and flexible, allow plug&play and are freely composable. Starting low and cheap - e.g. by using Emacs and RCS - up to combining expensive and comprehensive CASE tools and IDEs, software developers love to customize their environments according to their requirements.

With the advent of object-oriented middleware technologies like CORBA most technological problems on the way to a standard for the integration of software development tools could be solved. The real challenge seems to lie in bringing vendors, universities and other providers of software development tools together to form that standard and to create and adapt tools to it.

4 Conclusion and Outlook

In this paper we presented how a mission critical problem – material logistics during power plant construction – was tackled with Java. We found that Java is a powerful general purpose programming platform with some essential advantages when compared to other object-oriented languages. We presented a light-weight approach towards testing and showed how an elegant design in JDK could be used to transparently add a security layer to RMI. However, there are a few drawbacks which aggravate a Java programmer's life:

- In many areas Java is not stable yet. This requires a considerable amount of work for adapting an application to upcoming JDK changes and prevents reuse of many components.
- Design and programming tools are not open enough to be used as components for a pluggable and scalable software development environment. This issue is not limited to Java and should be tackled by "open" tools committed to standardized APIs for inter-tool collaboration.

Another topic we discussed is the different kinds of multi-tier architectures. When stable and fast internet connections cannot be guaranteed, the standard three-tier approach must be adapted. Modifications are required for replication, locking, conflict resolution and off-line modes. There is no common architecture yet which is able to handle all the described problems.

We expect that the Java environment will stabilize during 1998. Moreover, we hope that computer science and software industry will soon come up with major suggestions or even solutions for an "open software tools standard" and alternative multi-tier architectures.

References

[Beck94] Kent Beck, *Simple Smalltalk Testing*, Smalltalk Report, Vol.4, No.3, 1994, `http://www.armaties.com/testfram.htm`

[Camp94] M. Campanai, P. Nesi, *Supporting Object-Oriented Design with Metrics*, Proceedings of TOOLS 94, 1994

[Dart90] Susan Dart, *Spectrum of Functionality in Configuration Management Systems*, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, Technical Report CMU/SEI-90-TR-11, `http://www.sei.cmu.edu/activities/legacy/scm/tech_rep/TR11_90/3.3.1_ChangeSet.html`

[LDC+89] Anund Lie, Tor M. Didriksen, Reidar Conradi, EvenAndrè Karlsson, Svein O. Hallsteinsen, and Per Holager, *Change oriented versioning*, In C. Ghezzi and J. A. McDermid, editors, Proc. 2nd European Software Engineering Conference, volume 387 of Lecture Notes in Computer Science, pages 191–202. Springer Verlag, September 1989.

[Lore94] M. Lorenz, J. Kidd, Object-Oriented Software Metrics, Prentice Hall Object-Oriented Series, Englewood Cliffs, N.J., 1994.

[RMI98] Sun Microsystems, *Java Remote Method Invocation*, `http://java.sun.com/products/rmi`

[Siegel96] Shel Siegel, *Object Oriented Software Testing - A Hierarchical Approach*, Wiley & Sons, 1996.

[SunTest] Sun Microsystems, *SunTest - JavaStar, JavaSpec, JavaLoad & JavaScope*, `http://www.suntest.com/suntest/tools/TestingTools.html`

[Tichy85] Walter F. Tichy, *RCS – A System for Version Control*, Software–Practice & Experience 15, 7 (July 1985), 637-654

Authors
Dipl.-Inform. Uhl, Axel
Dipl.-Inform. Link, Johannes
ABB Corporate Research Center, Postfach 101332
69003 Heidelberg
Tel.: 06221 / 59{6302|6255}, Fax: 06221 / 596253
e-mail: {uhl|link}@decrc.abb.de