Continuous Refactoring of a Java Server Application

DI Siegfried Goeschl
IT20one GmbH, Austria
siegfried.goeschl@it20one.at

Abstract

The Invoice20one Server is a server application written in Java providing digital signatures for electronic invoices. The application design was based on the service architecture of the Jakarta Turbine framework. Extending the product portfolio required a continuous refactoring of the existing code base, which resulted in a light-weight Avalon container now part of the Jakarta Fulcrum project. This paper describes the problems encountered with Jakarta Turbine, the existing Avalon containers, the refactoring process and the Maven build tool being used to provide automated building, regression testing, generation of software metric reports and project documentation.

1. Introduction

IT20one GmbH is an IT company based in Vienna, Austria, focusing on Electronic Bill Presentment and Payment. The main product is the Invoice20one Enterprise Server, an add-on for commercial billing systems providing electronic invoices with digital signatures. The Invoice20one Server is implemented in Java and consists of 17 subprojects with 25,000 non-comment code statements altogether.

The initial development started in 2001, and the Jakarta Turbine was used as the foundation of the Invoice20one Enterprise Server. Jakarta Turbine provides a rich set of functionality for web applications based on an embedded service framework where most of the functionality is exposed as services. Following this design paradigm, the Invoice20one Enterprise Server consists of a set of services instantiated at runtime by the Turbine service framework.

2. Avalon, Turbine and Fulcrum

Due to the long development history as a web application framework, Turbine has its own share of shortcomings.

- Turbine is monolithic software depending on many external libraries and services to be available.
- There is no way to release new versions of Turbine services independent of Turbine's distributable.
- The Turbine service framework does not support service reconfiguration, nested service lookups or encryption of configuration files.

Since these shortcomings were well-known by the Jakarta community, the sister-project Fulcrum was founded. Fulcrum provides a decoupled service component repository for Turbine using an Avalon service container.

An Avalon service container implements the IOC (inversion of control) pattern [3] by describing which services are needed by which components, and the container is responsible for hooking it all together. The Avalon container creates all the services, wires them together by setting the necessary properties, and determines when methods are invoked. An Avalon container requires all services to support the Avalon Lifecycle Management Specification - a set of interfaces governing the lifecycle of a service. The lifecycle can be roughly categorized as incarnation, reconfiguration and decommission of an Avalon service.

This approach sparked a few different implementations of Jakarta Avalon containers such as ECM, Phoenix, Fortress or Merlin. The Excalibur Component Manager (ECM) was the first incarnation of an Avalon container. It didn't have many advanced features but was easy to embed into existing applications. Soon an improved Avalon container called Phoenix was under development. The Phoenix container was designed as a microkernel for deploying and running server applications. In the summer of 2003 the ECM container was superseded by the Fortress container providing many more features, e.g. instrumentation support and lifecycle extensions. While
Fortress grew from ECM, Merlin grew from Phoenix - an advanced Avalon container especially suited for application servers. The following multiple efforts to coordinate and unify the Avalon development failed. This resulted in the official closure of the Merlin project, a new fork of Merlin called Metro and a fork of Phoenix named Loom.

The aftermath of these so-called "Avalonic Wars" resulted in a Fulcrum project relying on two obsolete Avalon containers (ECM, Merlin). The dependence on two obsolete Avalon containers effectively stalled any further development of the project.

3. The Goals of Refactoring Project

In the autumn of 2004 the product portfolio was to be extended with three new applications, and Turbine's design limitations became more severe. This situation yielded to an analysis of the anticipated versus the implemented usage of the Turbine application framework. It turned out that only a small subset of Turbine's functionality was used, i.e. the embedded service framework, a couple of Turbine services and more than twenty services developed in-house.

Based on those findings, a refactoring project was started focusing on the following objectives

- using Jakarta Fulcrum and an Avalon container
- migrating the reusable Turbine services
- improving the software quality of the refactored services
- providing a flexible way to produce internal and external documentation

4. Choosing an Avalon Container

Choosing an Avalon container was a challenging task. The ECM container had long been abandoned whereas Merlin had not been finished which resulted in poor documentation for both of them. It was decided to use Merlin in favor of deprecated ECM, but Merlin turned out to be the wrong choice. The Merlin container is technically an application server making many assumptions about the environment and depending on thirty-eight external libraries. In short Merlin is not particularly suitable for being embedded in light-weight applications.

To overcome the problem it was then decided to write yet another Avalon service framework implementation (YAAFI) fulfilling the following requirements

- light-weight implementation providing a restricted functionality compared with existing Avalon containers, e.g. only supporting singletons [6] as service implementation
- minimal dependence on external libraries
- support of crucial in-house requirements, e.g. automatic service reconfiguration based on changes of configuration files and transparent decryption of configuration files

The code base was contributed to the Jakarta community and is now part of the Fulcrum project. The implementation (April 2004) resulted in 45 Java classes consisting of 2.360 non-comment code statements and 43 unit tests methods providing 63% statement coverage.

5. Migration of Turbine Services

The main task was the migration of existing Turbine services without the breaking the code base. This task was further complicated by rewriting existing services to fulfill new requirements, providing technical documentation for development partners and last but not least significantly improving the test coverage of the migrated services.

During initial planning it was decided to implement the following prerequisites

- Provision of a fully automated build and test environment. As such an environment Maven was used, the Jakarta project build and comprehension tool.
- Provision of a Turbine service running the YAAFI service framework to enable incremental refactoring. The caller of the service is not aware of the service framework managing it, thus providing container transparency.
- Use of an embedded relational database to provide a sandbox for each developer. Having an embedded database makes regression testing easier since the build is self-contained. HSQLDB was chosen and embedded in an Avalon service which is now also part of the Jakarta Fulcrum project.

6. The Maven Build Tool

A Daily and Build Smoke test [10] becomes crucial in order to avoid a Big Bang Integration [1]. For this task the Maven build tool was used. Maven [8] was initially developed as a tool for building Jakarta Turbine but matured into an open source software
engineering platform. Maven’s core functionality is building, testing, documenting and deploying Java applications. A Maven project stores its metadata in the XML Project Object Model (POM). The POM contains, among many other things, the location of the source files, the name and version of the distributable and library dependencies.

The functionality of Maven is implemented in terms of plug-ins accessing the project metadata of the POM. The Maven plug-ins are written in Jelly [5], an XML-based scripting language, which allows rapid development of new plug-ins. The fact that the plug-ins are decoupled from the core application (as shown in Figure 1) and that they can be easily written by the Maven community is one of the most valuable aspects of Maven.

![Maven Architecture](image)

**Figure 1 - Maven Architecture**

Assuming that some functionality is not found within the approximately eighty installed plug-ins, it can be easily added and contributed back to the community. Having more plug-ins makes using Maven more attractive thereby increasing the user base, which in turn provides additional plug-ins.

One of Maven’s key features is the creation of a comprehensive HTML project website based on XML data from the POM, user-supplied documentation and reports of the various Maven plug-ins. Consequently, many Maven plug-ins generate XML reports to capture various aspects of the software development and quality assurance to be incorporated into the project website.

During the refactoring process the following Maven plug-ins were particularly relevant:

- The Site plug-in creates a comprehensive project website with links to the user-supplied documentation written in XDOC and the various Maven reports.

- The JXR plug-in creates HTML pages of the Java source code including links to referenced classes and syntax coloring. This plug-in provides the foundation of many other Maven plug-ins because the HTML pages allow linking of a report incident and the corresponding source code, e.g. the CheckStyle plug-in links a coding style violation with the offending source code.

- The JDepend plug-in captures project metrics and project dependencies such as afferent/efferent coupling, abstractness and dependency cycles. Having an overview of the dependencies allows detecting design short-comings.

- The JavaNCCS plug-in reports simple software metrics such as the number of packages, number of classes, number of methods, cyclomatic complexity [9], amount of comments and non-commenting source statements.

- The CheckStyle plug-in checks coding convention violations which are defined in a XML file.

- The PMD [7] plug-in is a static source code analyzer to detect source code deficiencies and code duplication.

- The JUnit plug-in runs the JUnit regression tests [4] whereas passing the tests is a prerequisite of a successful build.

- The JCoverage plug-in is a coverage profiler to measure the code coverage of the regression tests.

- The Eclipse plug-in creates a project file for the Eclipse IDE based on Maven’s POM

- The MultiProject generates a dependency convergence report which is an invaluable tool to detect the usage of different versions of the same external library.

- The Dashboard plug-in aggregates the findings of Maven reports and provides an overview of multiple projects.

- The NSIS (Nullsoft Scriptable Install System) plug-in creates an installer for the application being tested.

### 7. The Refactoring Process

The first step is migrating and/or rewriting a Turbine service to become an Avalon service using the Eclipse IDE. Usually one Maven project per Avalon
service is created to keep the projects manageable. In parallel, the JUnit test suite is migrated and extended. After a successful test run within the Eclipse IDE, the corresponding Maven project is created manually.

The Maven build and generated HTML reports act as acceptance test before adding the new project to the automated build environment. In order to accept a new service, it is expected to pass all JUnit tests, have no coding style violations, to have statement coverage of at least seventy percent and technical documentation regarding the service.

For the integration the Named Stable Bases pattern [2] is used consisting of

- A nightly build guaranteed to compile, pass the JUNIT regression tests. In addition the nightly build creates the HTML project reports and website.
- A two-weekly test build considered stable enough for testing - the test build is delivered as Windows installer to ensure that the deployment works as well.

The nightly build results in a comprehensive overview of the project's health, quite literally the heartbeat [11] of the project. The generated website is accessible within the company's intranet. Anyone using a web browser can view the result of the daily build thereby increasing the transparency of the development work.

The generated Dashboard report shown in Figure 2 gives detailed information of the number of JUnit tests and their test coverage. Looking at "it20one-digistamp-service" reveals that no unit tests were executed. The reason why the unit tests were omitted is that the service accesses a remote server over the Internet and would break the build if the server couldn't be contacted.

![Table: Maven Dashboard Report](image)

**Figure 2 - Maven Dashboard Report**

The Dependency Convergence report shown in Figure 3 indicates that two projects use the HSQLDB library with conflicting versions. Such an issue needs to be addressed before releasing a final version.

![Table: Dependency Convergence](image)

**Figure 3 - Dependency Convergence Report**

8. The Documentation Trap

Documenting a software project is a tedious and time-consuming task leading to the so-called documentation trap. The documentation trap consists of obsolete documents obfuscated by proprietary document formats, buried in the depths of a version control system and long forgotten by everyone.

What are the reasons for being caught in the documentation trap? Documents are often written using Microsoft Word and are difficult to maintain due to the binary document format and the lack of visual differentiation tools. Furthermore Microsoft Word is not commonly available on Linux and Apple platforms, which forces the developers to use multiple operating systems.

Using a document format which does not separate content from formatting information makes writing documentation easier but distributing the documents using a different target format (e.g. HTML, PDF) and/or channel (development partners, resellers) a lot harder. Enforcing a consistent corporate layout is difficult if each document uses a customized document layout.

Last but not least, project documents are hard to find if they are scattered around multiple directories. Consequently they tend to be out of date or plainly forgotten.

One of Maven's key features is the creation of a comprehensive HTML documentation based on Maven plug-ins and user-supplied documentation. Technically the HTML creation uses an intermediary XML format called XDOC to be transformed into HTML. This transformation enforces a separation of content and final layout and allows the generation of additional document formats.

A project's documentation can be divided into internal and external documentation. The internal documentation is mostly used by developers, management, the testing and operations team whereas
external documentation is made accessible to development partners, resellers and actual users. The Maven build tool provides two plug-ins focused on creating external documentation. The PDF plug-in transforms the user-supplied documentation into a PDF document, thus providing the previously mentioned internal documentation. The SDocBook plug-in transforms a DocBook XML document into PDF suitable for publishing. Using DocBook is also a common option for publishing whole books, e.g. the "Version Control with Subversion" book is written in DocBook and the sources can be downloaded from http://svnbook.red-bean.com/.

7. Conclusion

Over the last eight months fifteen Avalon services were migrated or newly written. The implementation including the YAAFI service container consists of roughly 8,000 non-comment code statements and achieves test coverage of more than seventy percent on the average. The Avalon services are now used in all of the four products thus providing a high degree of reuse.

But the continuous refactoring of the Invoice2One Server was not an outright success story. On the one hand the initial requirements regarding software reuse, quality and documentation were fulfilled. On the other hand the schedule was not met due to the unplanned development work on the YAAFI service container and the feature creep during the refactoring project.

The high-frequency integration being used requires a carefully crafted build and test environment so that the software will never be allowed to disintegrate. Setting up such an environment is not trivial but Maven proved invaluable for this task. The nightly Maven build generates the HTML reports necessary for improving the quality of the ongoing software development. In addition, the general availability of the generated project website over the intranet greatly improved the project visibility.

Last but not least, writing and contributing an Avalon service container was a humble effort to support the work done by Jakarta's Open Source contributors and committers.

8. References