Leveraging Maven 2 for Agility

Timothy J. Andersen
Senior Software Analyst/Developer
Iowa Student Loan
West Des Moines, Iowa, USA
timander@gmail.com

Luke E. Amdor
Software Developer
Big Creek Software
Polk City, Iowa, USA
luke.amdor@gmail.com

Abstract

Software Developers at Iowa Student Loan® improved their technical approach for building Java [10] applications by switching from Apache Ant [4] to Apache Maven [6].

In 2003 we started using Ant. We had a lot of success with Ant and became very dependent upon it. At first, no one was comfortable switching to a new tool because Ant seemed to be working just fine.

In 2007, we decided to use Maven. Throughout this transition, it was necessary for us to overcome many obstacles including the learning curve and the infrastructure needs. This report illustrates the problems we had with our Ant builds and identifies why we decided to switch to Maven. Finally, it acknowledges the benefits we have realized and describes how Maven has augmented our agility.

1. Introduction

Iowa Student Loan was established in 1979 as a private, nonprofit corporation with a mission to help Iowa students and families obtain the resources necessary to succeed in postsecondary education. Since then, Iowa Student Loan has helped more than 260,000 students pay for college.

The company is based in West Des Moines, Iowa, and employs 315 employees. As of 2009, Iowa Student Loan manages more than $3.7 billion in student loans. The corporation is self-funded and self-capitalized. It is governed by a Board of Directors that, since its inception, have been appointed by the Governor of Iowa. Iowa Student Loan does not have shareholders nor does it receive state appropriations.

The software developers at Iowa Student Loan have been practicing Extreme Programming [1] since 2004. Ant was the tool of choice for build automation until we discovered Maven.

2. The Problems with our Ant builds

Even though we were able to build our products successfully with Ant, we encountered many problems that seemed unavoidable. The problems we faced included builds that took too long to run, complex builds, inconsistent build approaches, build logic duplication and overhead with managing dependencies.

2.1. Long-running builds

Long running builds were our most challenging problem and seemed to be the most difficult to solve. At the same time, we continued to accumulate more tests. One thing we tried was breaking test suites into smaller parts. We added a new Ant target called "quick-build" that would skip our slowest tests. This tactic improved our feedback cycle, but the continuous integration builds broke more often because all tests were not executed before we committed code to our Subversion [13] repository. To address the broken builds, we dedicated our "firefighter" (aka, odd-man out) to be the build master for the day. When the builds were all passing, the firefighter focused on making the builds run faster.

As our velocity declined, we explained the technical difficulty we had with our builds to our stakeholders. As a result, we decided to utilize one of our three pairs to improve our builds by making them run faster. We wrote the build times on the whiteboard and celebrated when someone did something to make a build run faster. Each iteration, we identified our longest running tests and re-worked them to provide the same feedback in less time. We used a hypersonic [8] in-memory database for our database tests. We replaced some of our browser/regression tests with faster running acceptance tests. We continued to make steady progress; but despite all of our efforts, we only shaved a few minutes off of our build time.

978-0-7695-3768-9/09 $25.00 © 2009 IEEE
DOI 10.1109/AGILE.2009.20

2.2. Complex builds

We came up with many interesting new ways to automate repetitive tasks. Ant was a great for this. The problem, however, was that each development team invented "clever tricks" to prepare the environment so the tests could be run. One example of a clever trick was an Ant task that used search-and-replace to set a configuration token from the value in a properties file. Another example was to copy a jar file to a location so a test could invoke a bat file in order to test an integration component.

2.3. Build logic duplication

We realized that duplication in our Ant builds was an issue, yet we did not have a tool to eliminate this duplication. We standardized multiple builds using Ant "includes" that invoked dependent builds, but it led to fragile builds.

We added code analysis tools to our Ant builds to provide metrics for an indication of code quality. We used PMD [12] to detect code duplication and Cobertura [7] to analyze test-coverage. We were happy to have this information, but were reluctant to add these tools to each of our Ant builds because they added a lot of duplication. It just seemed wrong to duplicate build logic in order to detect duplication. We resorted to documenting examples so they could be implemented by other teams.

2.4. Inconsistency

Learning how an unfamiliar project's build.xml worked was difficult and time consuming. We often implemented improved ways to automated tasks in Ant but these improvements wouldn't always get proliferated to other teams and other projects. The innovation was good, but the inconsistency was undesirable because of the learning premium when switching projects. We thought that we were feeling the growing pains of agile and started to wonder if we would be capable of achieving consistency across multiple projects while permitting freedom for each team to make decisions appropriate for their project's context.

In an attempt to provide consistency across projects, we wrote a JUnit [11] test that would search for all build.xml files and verify that certain Ant target names were declared. We documented the expected behavior for each common Ant task on the department Wiki. Our intent was to provide better consistency across projects by reducing unnecessary differences. The variances in the builds seemed wasteful to us because most differences only existed because of the preferences of the developers who worked the various projects.

2.5. Dependency management overhead

We kept all of the third-party libraries in a common location in our Subversion repository to avoid having additional copies for each project. For each third-party library, we manually configured IntelliJ IDEA [9] to attach the source code. We made sure we kept our IntelliJ IDEA dependencies synchronized with the dependencies declared in our Ant build by using a test to verify that they matched. Even though we could easily identify when they were out of sync, upgrading dependencies was a painful process. The overhead that we dealt with when we introduced a new dependency or upgraded an existing one was cumbersome, but it appeared to be unavoidable.

When we noticed the same code required in more than one project, we moved it to a shared module to avoid duplication and reinventing the wheel. Eventually, our shared code accumulated into a "big ball of mud" that we avoided changing, for fear of taking the responsibility for breaking another project's build. The general outlook on our shared code was that we would put up with what was already shared, but we felt like adding anything else to the pile would only make it worse. The primary reason we avoided splitting the "big ball of mud" into smaller modules was because the overhead of maintaining multiple shared modules was too cumbersome. Previous attempts had broken builds for several projects at once. We discussed strategies to break it apart, but we couldn't agree on simple things such as version numbering conventions. We developed a lot of pessimism towards sharing code.

3. Transition to Maven

Making the switch from Ant to Maven wasn't easy. Developers were familiar with Ant and didn't perceive the problems with our Ant builds as serious concerns. We first had to convince all developers that switching to Maven was a good idea before we could convince the rest of the organization that the learning curve and the investment in infrastructure would be worthwhile.

3.1. Appeal of Maven

The most attractive aspect of Maven was that it provided a way for us to simplify our builds. The standard build lifecycle provided the consistency we wanted. The use of a parent pom.xml provided a mechanism to reduce build logic duplication. Tools we used for code metrics such as Cobertura and PMD could be configured in one place instead of duplicated for each project. Additionally, the dependency management offered by Maven was amazing. Instead of explicitly including all of our transitive dependencies, Maven included them automatically.

3.2. Resistance

No one was ready to re-write the build in a different tool. At first glance, Maven seemed too magical because of the conventions. We initially felt like we had more control with Ant. Additionally, we would not only have to make a pom.xml file for our project, but we would also have to make one for each of our internal dependencies. This seemed like a daunting task. With a little convincing, we agreed to time-box an hour per day for introducing Maven. After a little more exposure, we realized Maven had the potential to help us solve many of our build problems.

3.3. Migration from Ant to Maven

We were fortunate that one of the developers on our team had prior experience using Maven 1. We started with Maven 2.0.7 in June of 2007. (As of June 2009 we are using Maven 2.0.9.) After a couple of weeks of “Mavenizing” our internal dependencies we had learned enough to use it on our project. It was a massive effort, but we took turns pairing on it and eventually got to the point where we were comfortable enough to delete the build.xml files.

3.4. Infrastructure Needs

After we had a few builds cut over to Maven, we realized it would be necessary to have a proxy repository solution so we didn't have to wait while downloading dependencies. We also wanted a place to share the code we built so it could be available as a binary dependency for other developers and to the build server. We selected Apache Archiva [5] (version 1.0 alpha 2) for our internal Maven repository server. (As of June 2009 we are using Archiva 1.1.3.) Initially, we ran Archiva on one of the developer's "cube stations" because it was just sitting there. (We write most of our code at pair-stations in our project rooms.) We eventually got a server specifically to host Archiva, which we could access remotely.

3.5. Organizational concerns

We had discussions with all areas and levels of our Information Technology department about how Maven impacted deployments. The standardization and consistency that Maven offered were important factors. Developers were able to convince decision makers with relative ease that Maven would improve our deployment process.

3.6. Explanation to business stakeholders

The business stakeholders were concerned that developers were playing with "shiny new toys" instead of working on features for their time-sensitive projects. From the non-technical perspective, the builds were already done and they perceived rewriting the builds using a different tool as a waste of time. Developers decided that the business stakeholders wouldn’t get to decide on whether or not we used Maven. Instead, the business stakeholders would have to trust the developers to make responsible decisions. We did our best to strengthen trust by explaining the benefits of using Maven in terms that mattered to the business stakeholders. To do this, we gave a presentation that explained the business benefits of Maven.

The main points were that Maven:

- provides standards and consistency
- reduces overhead
- makes developers more efficient

Developers explained that the benefits would eventually outweigh the costs; however, there would be an initial investment because of the learning curve. Even though developers did not permit the business stakeholders to make the decision, the communication and transparency helped to improve the relationship between developers and business stakeholders.

3.7. Learning curve

The biggest problem we had with adopting Maven was changing the way developers thought about the build. Many developers claimed that Ant is *easier*. I would argue that Maven is *simpler*, and Ant was only *easier* because we were forced to be more familiar with *how* the build worked. Using Maven, we were able to define *what* we were building, and Maven provided the build strategy. The shift from *how* to *what* seemed to be the thing that most developers initially struggled with when switching from Ant to Maven.

It took a long time for developers to get familiar with the Maven conventions and the build lifecycle. We held training sessions to show other project teams how to use Maven for their build and we pair-programmed across teams to share our newly obtained knowledge. On average, it took about three months for a developer to become proficient with Maven and another three months to master it. It was difficult for teams who didn't have a resident expert on Maven. We made mistakes during the learning process. We addressed these issues by switching team members and cross-team pairing.

4. Benefits

The benefits we *expected* to get from Maven were:

- consistency across projects
- simpler builds
- reduced build logic duplication
- improved dependency management

Maven did not disappoint. In fact, it exceeded our expectations for the problems we intended to solve. Our Maven builds were much simpler than our Ant builds and as a result they were easier to maintain. We greatly reduced the build logic duplication and the overhead of managing dependencies was drastically reduced. Builds were also much more consistent than we ever could have hoped for using Ant.

There were also some surprising benefits that we did not expect as a result of using Maven. The *unexpected* benefits were:

- faster builds
- cleaner code
- more cross-team collaboration

4.1. Social impact

What we noticed about our own social behavior was a much more optimistic outlook on shared code. When we adopted Maven, the overhead cost of introducing a new module was drastically reduced. Our fear towards the maintenance cost of multiple shared modules was diminished. We released our shared code and refactored [2] it into smaller focused modules without breaking all of the other projects. The pessimism towards shared code was replaced with excitement and can-do attitudes.

4.2. Unexpected benefits

The most rewarding outcome which none of us predicted was faster builds. Due to a combination of simplified releases, excellent dependency management, and a general sense of optimism, we extracted modules whenever/wherever we saw the opportunity. Because we could then declare a binary dependency, we didn't have to run all of the tests in one gargantuan build. The builds for the smaller extracted modules ran in much less time. The astonishing results we achieved via module extraction encouraged us to keep doing it. We encountered ugly code that was overly coupled and unnecessarily complex, but the motivation of a faster build by module extraction was enough to keep us on track to solve these problems once they were exposed. We cleaned up our code so it fit into the appropriate modules, which ultimately led to improved design. Maven allowed us to treat our internal dependencies the same way we treat third-party dependencies without the overhead cost associated with releasing, versioning, and upgrading. My project went from two large builds that shared code and took 30 and 40 minutes, to about 20 builds, most of which ran in less than five minutes, with the longest ones taking 10 and 15 minutes. Not only did we benefit from faster builds, we also benefited from better design. Maven provided the necessary tipping point for us to confront the challenges that lurked within our code.

5. Conclusion

Maven provided standards and a set of patterns in order to facilitate project management through reusable, common build strategies [3]. Iowa Student Loan has experienced tremendous success with Maven. It has given us the power to focus on our code instead of our builds. Our software is easier to change as a result of leveraging Maven for our builds.

If you have experienced similar problems such as long running builds, complex builds, inconsistency, build logic duplication and dependency management overhead, Maven is definitely worth a try. The initial learning curve is fairly steep, but the benefits are very rewarding.

6. References

[1] K. Beck, “Extreme Programming Explained: Embrace Change (Second Edition)” Addison-Wesley, ISBN 0321278658, November 2004.

[2] M. Fowler, K. Beck, J. Brant, W. Opdyke and D. Roberts, “Refactoring: Improving the Design of Existing Code”, Addison-Wesley, ISBN 0201485672, July 1999.

[3] V. Massol and J. Van Zyl, "Better Builds with Maven", Mergere Library Press. [PDF]. June 2006.

[4] Apache Ant http://ant.apache.org

[5] Apache Archiva http://archiva.apache.org

[6] Apache Maven http://maven.apache.org

[7] Cobertura http://cobertura.sourceforge.net

[8] Hypersonic http://hsqldb.org

[9] IntelliJ IDEA http://www.jetbrains.com/idea

[10] Java http://java.sun.com

[11] JUnit http://junit.org

[12] PMD http://pmd.sourceforge.net

[13] Subversion http://subversion.tigris.org