Rebuilding an Airliner in Flight

A Retrospective on Refactoring IBM Testarossa Production Compiler for Eclipse OMR

Matthew Gaudet
IBM Canada
magaudet@ca.ibm.com

Mark Stoodley
IBM Canada
mstoodle@ca.ibm.com

Abstract

Eclipse OMR is a new open source project created by refactoring the IBM J9 Java Virtual Machine to create a set of language agnostic components for building all kinds of language runtimes. This paper reflects on the effort, the successes, and the mistakes made while refactoring more than a million lines of code on the master development branch while 8 disparate production compiler products were under active development. Refactoring large scale projects like compilers and language runtimes can be done while shipping releases, but there are certainly challenges. We offer some recommendations for other projects considering this path.

Categories and Subject Descriptors D.2.7 [*Software Engineering*]: Distribution, Maintenance, and Enhancement—Restructuring, reverse engineering, and reengineering

Keywords Refactoring, compiler, retrospective

1. Introduction

Eclipse OMR is a set of reliable components for building high performance language runtimes. The project goal is to provide a common infrastructure layer that can be used by any language runtimes, establishing a lower cost path for other runtimes to leverage any of the deep investments that have been made in technologies such as concurrent garbage collection, JIT compilation and monitoring. Increased core technology sharing across currently disparate language runtime systems would provide greater leverage to improve core runtime technology with dividends paid in multiple runtime ecosystems. Furthermore, a widely shared common runtime technology layer could substantially accelerate innovation in cloud platforms whose capabilities are reliant on the level of support in the various language runtimes developers want to use on their platforms.

VMIL'16, October 31, 2016, Amsterdam, Netherlands
ACM. 978-1-4503-4645-0/16/10...$15.00
http://dx.doi.org/10.1145/2998415.2998419

The current set of OMR components were contributed by the IBM runtime technology team in the form of the J9 Java Virtual Machine as well as its Testarossa JIT compiler. J9 and Testarossa have formed the core of the IBM JDK for 11 years. OMR being built out of a pre-existing language runtime system allows us to produce high quality advanced components at a high velocity. Testarossa has historically be used in a number of compilation projects ranging from academic experimentation with dynamic languages to a production full system emulator to a binary re-optimizer to a dynamic Just In Time compiler for Java to static language compilers for Cobol, C, C++, and PL/X. This diversity of successful language compilation scenarios gave the team some confidence that the approach would work.

However, refactoring the Testarossa compiler technology for use in the OMR project had lots of challenges and resulted in successes but also some results that are hard to depict as "successful". This article summarizes some of our experiences in refactoring a large production compiler technology while under active development for multiple compiler products.

2. Development Requirements for OMR

OMR was developed under some very stringent requirements:

1. The Code must be Shippable: Development of OMR happened on the master development branch at the same time developers were actively writing new code. Multiple products shipped while the refactoring and rearchitecting to extract OMR was in progress.
2. The Code must not lose functionality: In addition to remaining shippable, we were not willing to disable functionality in the code to maintain our ability to ship.
3. The Code must not regress the many performance metrics regularly tracked during production compiler development: Testarossa-based products have stringent requirements not to regress compile time, code and compiler footprint, and various aspects of generated code performance across multiple languages, platforms, and com-

<table>
<tr><td>Java</td><td>COBOL</td><td>...</td><td>Proprietary</td><td colspan="2"></td></tr>
<tr><td colspan="4">IBM</td><td>Ruby</td><td>Python</td></tr>
<tr><td colspan="6">OMR</td></tr>
</table>

Table 1. Architecture Diagram for OMR Project as originally concieved.

pilation environments; refactoring work was held to the same bar as other development work.

The development of OMR proceeded along a number of fronts.

2.1 Preliminary Experimentation and Refactoring

Like many projects, OMR started as some experimentation. In the case of OMR, this was a very preliminary form of a JIT compiler for Python, quickly joined by a prototype for Ruby. Developing prototypes allowed our team to get a feeling for the code locations that were so specific to Java they could not be shared with other languages, as well as helping the team identify problems and challenges with how the OMR JIT would be consumed by language runtimes that were not designed from scratch to use the OMR JIT.

2.2 Development of an Organizating Theme and System

One fundamental question that confronted the OMR team was: How do we organize a codebase that was used in compilers for 8 products and generating code for four very different architectures. The codebase had a relatively common core, but some parts were more targeted to JIT compilation or specifically for static compilation, or in other places specialized to obey language rules or optimize language specific patterns. Over time, a plan emerged to refactor the codebase into layers, as described by Table 1.

There would be an *OMR* layer, which contained the core of the compiler technology but designed specifically to be language agnostic. All OMR based languages would build directly on top of that layer. Proprietary languages would build on top of another layer, simply called *IBM*, which was designed to share code across multiple proprietary products.

The refactoring challenge was then to divide each core Testarossa class into pieces falling into the appropriate layers: core language independent code in the OMR layer, code that could be shared across proprietary products in the IBM layer, and code specific to a particular language would reside in a piece used onto to build that particular compiler. These pieces would be arranged together into a C++ class hierarchy where each new layer was extending the capabilities of the lower layers. We called these extensible classes.

2.3 Design Iteration

While the most basic form of extensible classes worked well enough, it quickly became apparent that there were a number of pieces that were unwieldly. Over the course of months, our approach to providing extensible classes was refined step by step until we had shaved away most of the worst parts of the extensible class system. This was one instance of an ongoing process in OMR of constant iteration.

It is not typically possible to get all the elements of a particular design right initially, without broad discussion and examination. However, when working in a codebase with tens of developers across 8 disparate products, it's difficult to anticipate all things. Design iteration was very successful in this scenario, by getting the changes into the hands of developers where they can provide feedback for future work and cleanup.

Iteration was very successful approach from the point of view of the developers implementing the refactoring, but it was not always well received by the other groups of developers working in the same code base. Iterating on approaches to class extensibility, for example, was disruptive to other development work as the code base changed substantially and rapidly underfoot. Communication among the various teams was key and required constant reinforcement to head off problems.

2.4 Sometimes You Need A Drastic Pivot

The refactoring work of OMR highlighted some technical tensions in the Testarossa codebase. Some of the projects employed pragmatic design considerations that could make sense when the code base was completely proprietary, but that didn't correspond to the OMR team's original conception of a layered structuring of projects. Furthermore, the static language compilers built on top of Testarossa tended to use fairly different implementations of basic functionality due to the substantial differences in the semantics of these languages compared ot that of the Java language Testarossa was originally built for. Over time, it became clear that the small size of the OMR team combined with our design to open source meant that we'd have to make some drastic decisions.

Through candid discussion, it became clear that a substantial fraction of our static language compiler technology was going to remain proprietary in a way that was going to cause great difficulty if an open source project formed its foundation. Furthermore, in the time since the OMR project was initiated, the IBM strategy around the Java technology changed with the decision to open source the J9 Java Virtual Machine, leading to a significant Testarossa consumer being destined for open source. This change dramatically altered the work to refactor the code base and motivated a significant pivot in our approach: we decided to fork the open source languages, Java, and the language independent OMR

Java	Ruby	Python
OMR		

Table 2. Simplified architecture diagram

layer into a separate code base from the proprietary static language compilers and other proprietary compiler products.

Forking fairly significantly simplified our system architecture, as shown in Table 2.

This simplification of concerns propagated throughout the codebase: After the fork, the to-be-opened codebase dropped quickly from 1.2 million lines of code to 850,000.

The fork has given each set of projects the ability to much better control their technical destiny, at the cost of sacrificing some degree of code sharing, though sharing of effort is still occurring, where it makes sense.

3. Summary of Our Experience

The OMR project taught the team a large number of lessions about refactoring, large scale system design, teamwork as well as business and engineering tradeoffs. We summarize these lessons as

3.1 What Went Well

1. *Refactoring in Production* The disruptive change that the OMR project introduced into the Testarossa codebase was very hard on all developers. However, the choice to do the work in production had a number of very clear benefits:
   - The OMR project was forced to maintain a relatively low risk profile. We were working on the master development branch, directly alongside the developers shipping products. This approach was good for the business and the codebase, as changes were forced to maintain shippability, and there was no risk that a change in business strategy would leave us incapable of delivering products.
   - Because the work we were doing was happening in production, all OMR work was subject to the testing of the production runtimes built on top of Testarossa. This greatly aided Testarossa in maintaining high quality while its own language independent testing was under development, and in general has still held the code to a higher standard than otherwise would have been available given the resources available for development and testing.
   - By working in production, we were able to take an agile approach to refactoring. We were able to get and respond to feedback from developers on other projects, in some cases outright fixing problems we had not foreseen, in other cases assuaging concerns in other ways.
2. *Experience* Testarossa's history of being multilingual gave us confidence that we would be able to create the design structure we'd planned and be successful.
3. *Prototyping* Doing multiple prototype languages, Ruby, Python, and later SOM++ and JitBuilder have been excellent proving grounds for our approach, highlighting deficiencies, as well as giving us a measurable sense of progress.
4. *Agile Methods* The adoption of an agile project management process was very helpful to the OMR team. Our particular model involved giving everyone free license to create new work items, that were sized and prioritized at weekly meetings, then, team members were free to tackle any item available when they had cycles. This was particularly effective at reducing choice anxiety in newer team members and interns, giving them a great boost in productivity and huge amounts of self-direction.

3.2 What Went Poorly

3.2.1 Perfection is the Enemy of the Good

Inability to compromise even temporarily left certain tasks as insurmountable. The team has gotten better about this, but for a while, progress on a number of fronts languished while fears about regression in compile time performance ruled the roost. Fear of regressions also lead to technical decisions that harmed the software engineering rigor of the project. For example, due to the rigorous performance constraints on one of our supported platforms, we were forced to develop our own class model as well as support tooling, rather than using better supported C++ features such as virtual methods, and design patterns such as the CRTP. Some decisions on this road were made with insufficient empirical evidence, and insufficient consideration of tradeoffs.

Similarly, changes that touched delicate parts of the codebase were fraught. In retrospect, we should have spent more time rewriting troublesome classes than trying to mechanically tease apart multiple intersecting concerns in classes that had not been designed to be decomposed.

In hindsight, the very appealing conceptual reasons for sharing the Testarossa code base should have been evaluated in the context of the state of the actual code. This went too long unexamined and were ultimately responsible for far more consternation among teams than ever should have occurred.

Finally, coming from a part of IBM with relatively little direct open source software experience, we were also constantly confronted with our own fears and concerns around what would be an acceptable base on which to form an OSS community. Over time, we have come to believe that we would be better served working in the open, where we can get feedback from real potential consumers, rather than try-

ing to imagine and anticipate the needs and desires of a multitude of hypothetical consuming projects.

3.2.2 Riding on Coattails

The piggybacking on other languages for test kicked the development of stand-alone test too far down the road.

3.2.3 Internal Community Building

Iterative design while successful, leads to churn in the code base that wasn't always clearly communicated to other teams working on the code, leading to tensions. Part of the problem with community building was structural. The OMR team (those responsible for refactoring and re-architecting the project) were only a small part of the overall development team. While this choice seemed sensible early in the project, it did cause friction and led to teams not collaborating on design and structure as much as was probably necessary for a project of this magnitude. The refactoring work cut across code operating in 8 different compilers with sometimes very different expectations. It wasnt tractable for the small OMR team to grasp all the intersecting subtleties in many places, which left the OMR team moving code around more than rewriting to minimize impact as we worked to accomplish our goals. We were able to overcome the issue through open lines of communication and professionalism that helped to dispel issues in productive ways. Keeping people talking, even when disagreements were severe, was key to success.

4. Suggestions to Other Projects

For any project looking to follow in the OMR projects footprints, we offer the following advice:

1. *Absolutely do the work 'in production'*. Its difficult to estimate how long a branch would live, but it seems clear to us that fundamental refactoring work must occur in the same branch as other development work proceeds, especially if there is significant development velocity outside of refactoring activities.
2. *Be extremely careful about your 'absolutes'*. In our cases, some temporary regressions would have helped to unlock greater future potential. The challenge is scheduling the regressions and carefully planning the follow-on work to assuage concerns over the regression.
3. *Be agile* Use prototypes and be willing to revisit designs multiple times to let better solutions appear. Always question if operating conditions have changed and use opportunities to adjust your decisions appropriately.
4. *Do not just isolate the task to a subteam* Make the goal a priority for everyone working on the code base and spread the work and design effort across the team. When multiple teams share one code base, this recommendation can be extremely difficult to accomplish. But having at least some commitment from each team is better than leaving the work solely in the hands of one dedicated group.
5. *Always keep lines of communication open* It seems obvious, but in the thick of things, communication can very quickly break down and only concerted effort to keep discussions moving forward can avoid disaster.

5. Conclusion

The OMR project is proof that it is possible to do very large scale refactoring and re-architecting of a production language runtime, however, the process needs to be managed with careful attention paid to testing, agility, and communication. The OMR JIT technology was open sourced in September, 2016, having changed dramatically since the project inception years earlier yet still meeting the needs of all the products and open source projects that need to use this technology.