Exploring the Architecture of Javelin with a Dependency Structure Matrix

How a DSM-based analysis would fare when applied to Javelin

by Tim Hanson and Neeraj Sangal

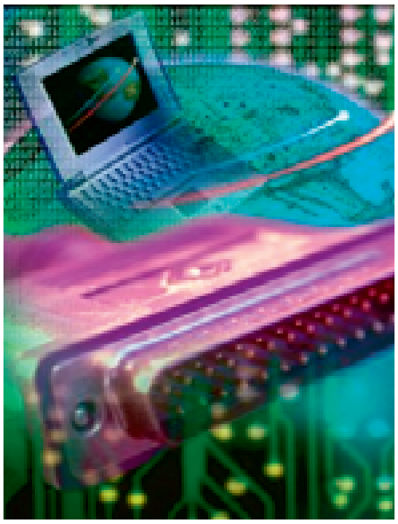
Javelin is a compiler framework written entirely in Java. It lets developers support new languages or extend current languages. It also provides interfaces to the parsing and compilation process. For instance, an IDE's editor can take advantage of Javelin to support syntax highlighting, code completions, and refactoring. Javelin could be extended to compile languages like XSLT, XQuery, and PHP. It already contains a Java compiler and an XML Schema compiler, which is fully integrated into BEA Workshop and provides support for various editing and compiling tasks. The JSP compiler also uses the Javelin framework.

However, this article isn't about Javelin. It's about the *architecture* of Javelin. We decided to explore the architecture of Javelin using its inter-module dependencies. For purposes of this analysis, all we needed was the Javelin jar file. Given that Javelin has more than 800 classes and thousands of dependencies between these classes, we needed a representation that could scale up while allowing us to dig into any specific dependencies.

Our choice for this analysis was to use a Dependency Structure Matrix (DSM) to represent the architecture. This powerful new approach (see http://sdg.lcs.mit.edu/pubs/2005/oopsla05-dsm.pdf) has recently been introduced for specifying software architectures but it's been built on ideas that have actually been around for a long time. Historically, DSMs have been used to represent complex product development processes. The new approach has two key elements: (1) A precisely ordered hierarchical decomposition and (2) explicit control over allowed and disallowed dependencies between the subsystems. DSM provides a compact representation that can easily scale up to tens of thousands of classes, whereas conventional box-and-arrow diagrams become unusable in systems composed of even a few hundred classes.

Tim Hanson is the Javelin compiler architect at BEA Systems. Tim developed much of BEA's Java compiler - one of the earliest 1.5-compliant implementations. He has written numerous other compilers, including a CORBA/IDL compiler while at IBM, and an XQuery compiler.

tim.hanson@bea.com



We used Lattix LDM for this analysis. Lattix LDM recently won the Jolt award for design and modeling tools, a category that has traditionally been dominated by UML tools. We wanted to see how a DSM-based analysis would fare when applied to Javelin.

Our goal for this project was to discover the architecture and identify the undesirable couplings that might have crept into the architecture, since it's undergone development over the last few years. Ultimately, we want to improve the modularity of the architecture so that it's easier to understand and maintain.

An Initial DSM for Javelin

We loaded the file, javelinx.jar, into Lattix LDM. The initial picture represents the decomposition of the jar organized as a tree of packages and classes. You can expand each level of the tree to display the next level of decomposition. At any given level of expansion the matrix shows you the dependencies between the subsystems. You can select any cell of the matrix to see the exact dependencies between any two of the subsystems.

The initial DSM, in Figure 1, shows a decomposition based on a package structure. We expanded the *javelin* and the *com.bea* packages to reveal the packages inside them. Note that when a package contains classes and packages, Lattix LDM creates a partition called "*" that contains all the classes at the root level of that package.

To see the dependence of one subsystem on another just read down the column. By definition the DSM is a square matrix where the rows and columns are numbered to correspond to a subsystem. For instance, the selected cell in Figure 1 shows the dependence of the source subsystem, *javelin.java,* on the target subsystem, *com.bea.languages.java*. In Lattix LDM, as you select a cell, a usage pane on the right shows you the exact dependence of each class in the source on classes in the target.

Reorganizing the DSM To Represent the Javelin Architecture

Trying to make sense of all these dependencies at first appears daunting. However, the use of partitioning algorithms can quickly bring order to it. Partitioning can be applied to any subsystem of the tree and it reorders the subsystems, which are directly within the selected subsystem. Partitioning is a precise mathematical algorithm, which returns a block triangular matrix. This means that those subsystems, which are used more by others, are moved closer to the bottom and those subsystems, which tend to use other subsystems more are at the top. For instance, a utility package would tend to move to the bottom because it's more likely to be used by other packages. Conversely, a package called client is more likely to be at the top because it's going to make use of the services offered by other packages.

The interesting cases are the ones where the abstraction is blurred. What if the interface depends on the implementation? What if the compiler depends on the client? Those cases require untangling of the code. This can sometimes be as easy as moving a few files around but more generally requires code refactoring to achieve the separation of abstractions. The purpose of architecture discovery is to decompose the problem into clean abstractions. The initial step is to start with a decomposition that uses packages; we partition them to see if reordering them based on their dependencies will reveal the structure.

Note that to come up with this structure we partitioned many of the subsystems. Partitioning the *$root* node (the top level) moved the *javelin* package above the *com* package. This conforms to our expectations because *com.bea* and *com.sun* packages contain the interface while *javelin* is the implementation of that interface. We expanded *com.bea* to reveal *com.bea.languages* and *com.bea.compiler*. These were, in turn, expanded to reveal their subsystems. Paritioning *com.bea* moved *com.bea.compiler* underneath *com.bea.languages*, once again confirming our expections that *com.bea.compiler* contains the general API extended by *com.bea.languages*. Finally, we partitioned *com.bea.compiler*, *com.bea.languages*, and *javelin*. This reordering based on partitioning gave us a sense of which subsystems underlie which other subsystems. It also showed us that many of the subsystems were coupled. Figure 2 shows the results of partitioning these selected subsystems.

Recognizing that Javelin had an infrastructure built around the Java compiler we created an appropriate abstraction to reflect it. We also recognized that there were a few subsystems that were either obsolete or simply reflected abstractions built on top of other systems. We moved them out to the top level. Having made these manipulations, we came up with the conceptual picture in Figure 3.

The conceptual diagram is derived from the matrix view to reflect the decomposition of Javelin. We used the layout further to show both "horizontal" and "vertical" splitting. The horizontal splitting is the splitting up into layers. For instance, the *API* is layered to be underneath *javelin* to indicate that we expect *javelin* to depend on the *API* but not vice versa. Similarly, we use vertical splitting to indicate independent components. For instance, *com.sun.mirror.apt* and *com.bea* in the *API* are split up to indicate that these two have no dependencies on each other.

Identifying the Improvements

Examining the DSM corresponding to the conceptual architecture then let us determine the dependencies that were the targets for elimination through refactoring.

Area 1 In Figure 4 shows the dependence of the *API* on *javelin* while Area 3 shows the dependence of *javelin* on the *API*. Normally, we expect the implementation to depend on the interface, not the other way around. As a result, we would target the dependencies in Area 1 for removal through refactoring.

Most of these references come from the fact that the original interfaces to the Java typesystem were removed from the public API and refactored into *javelin.java.typesystem*. References to these newly private classes were never fully removed from the public API.

Area 2 shows the dependence of *com.bea.compiler* on *com.bea.languages*. Once again we expect the *com.bea.languages* to depend on *com.bea.compiler*,

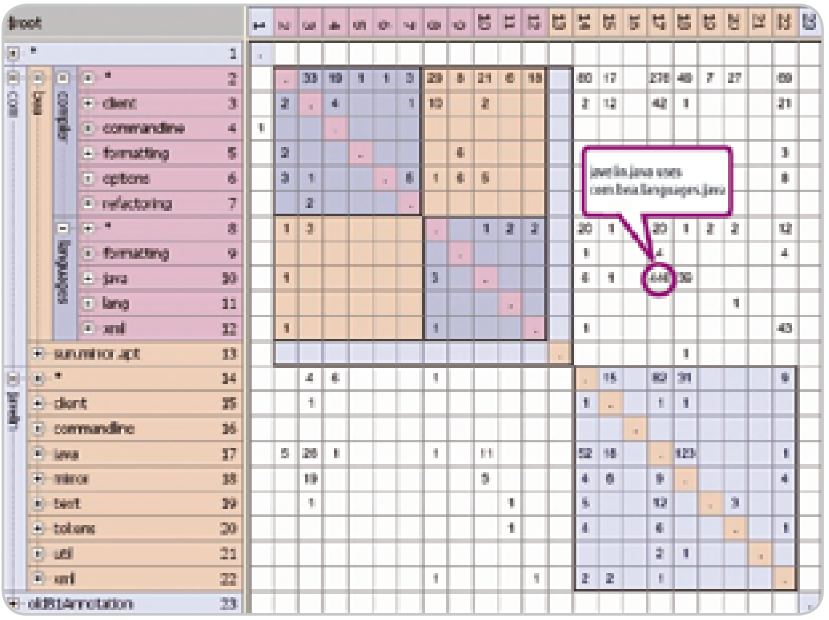


Figure 1 The initial DSM

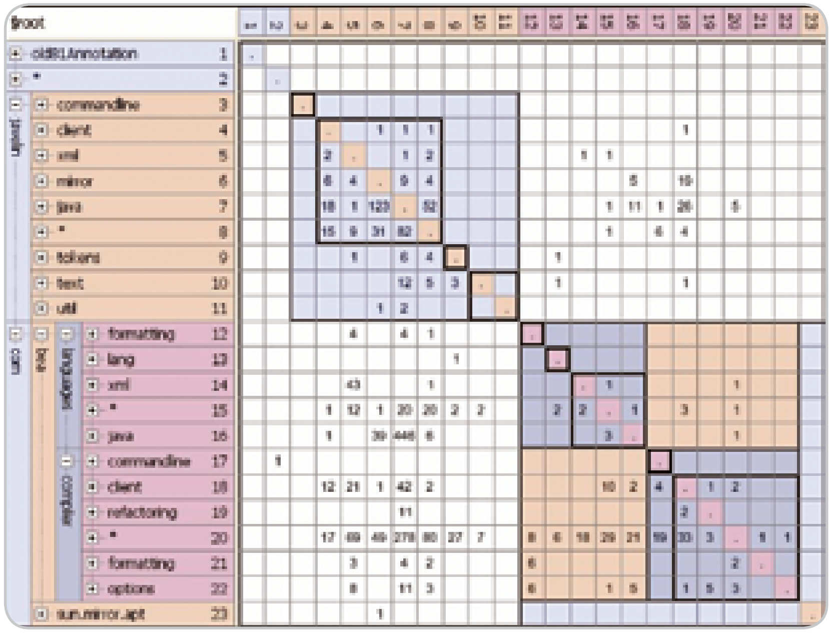


Figure 2 DSM after partitioning certain subsystems

Neeraj Sangal is president of Lattix Inc., which specializes in software architecture management solutions and services. He has analyzed many large proprietary and open source systems. Previously, Sangal was president of Tendril Software that pioneered model-driven Enterprise Java Beans development and synchronized UML models for Java. Prior to Tendril, he managed a distributed development organization at Hewlett Packard.

neeraj.sangal@lattix.com

not the other way around. As a result, we'd like to get rid of the dependences in this area.

Notice also that *javelin.** and *javelin.java* are tightly coupled together. This is an implementation artifact that arose because the team that wrote the Java compiler wrote the compiler framework. We never tried to decouple these, but that seems like a mistake. If the Java language can't be implemented using only the bare compiler framework, how can we expect another language to be implemented? In particular, we never distinguished a generic project from a Java project. The base project class has all the methods to get and set the classpath, as well as find a Java type by its qualified name. This should be refactored to separate the notion of a generic project from a Java project.

The dependency of *javelin.xml* on *javelin.java* tells us that the implementation of our XML schema compiler depends on the **implementation** of our Java compiler because it needs to generate Java bindings to access XML instances. This tells us that our API isn't sufficient to meet our client's needs. This is an obvious area for refactoring. Any features needed by the XML Schema compiler should be put in the interface layer. Intuitively, we know that that's possible because the JSP compiler, which also generates Java code, uses only the public API. This sort of interface creeping occurred primarily because the same team developed both parts of the code.

Finally, we haven't yet looked inside *javelin.java*. This is the heart of the infrastructure and needs a more in-depth examination. It also shows the power of the approach. The DSM lets us systematically analyze our system in a top-down fashion. For larger projects, this means that different groups can focus on different parts of their system while maintaining the overall architecture.

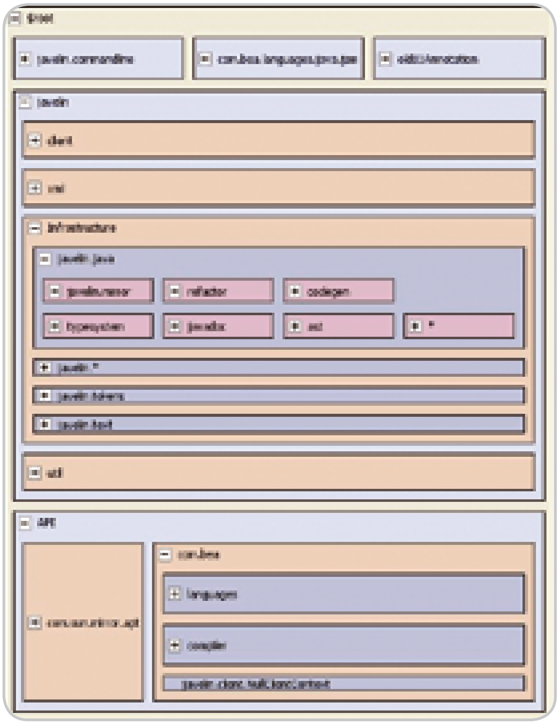


Figure 3 Javelin's conceptual architecture

Conclusion

Once we finish our analysis we'll create design rules that enforce the architecture so future developers don't end up creating unnecessary coupling. On a DSM this is represented by annotating the cells to indicate those cells where dependencies are allowed and those where they aren't. Once these rules are in place, they can be verified as part of the regular builds.

We note that we were able to identify undesirable couplings very quickly. There are more than 800 classes in the Javelin jar file, which have nearly 80,000 dependencies among them. The power of the DSM is that it could incorporate our architecture knowledge and we could winnow down the dependencies that we needed to worry about to a tiny number.

We were also pleased with the matrix representation. The conventional visualizations for representing architectures have typically used box-and-arrow diagrams. These diagrams get cluttered quickly and simply don't scale when systems get large. The matrix representation scaled easily, letting us create new abstractions and try out what-if scenarios by moving classes and packages. All of the changes that we made to the architecture were maintained in a WorkList that could then become a task list for refactoring.

We urge you to try this new approach on your own software to see how easy and effective it is.

Useful Links

You can learn about BEA's Javelin compiler from http://dev2dev.bea.com/wlworkshop/javelin/index.html

You can download and try Lattix LDM from http://www.lattix.com

You can learn more about DSMs from the following URLs:

- http://www.dsmweb.org
- http://sdg.lcs.mit.edu/pubs/2005/oopsla05-dsm.pdf
- http://www.lattix.com/technology/whatisdsm.htm

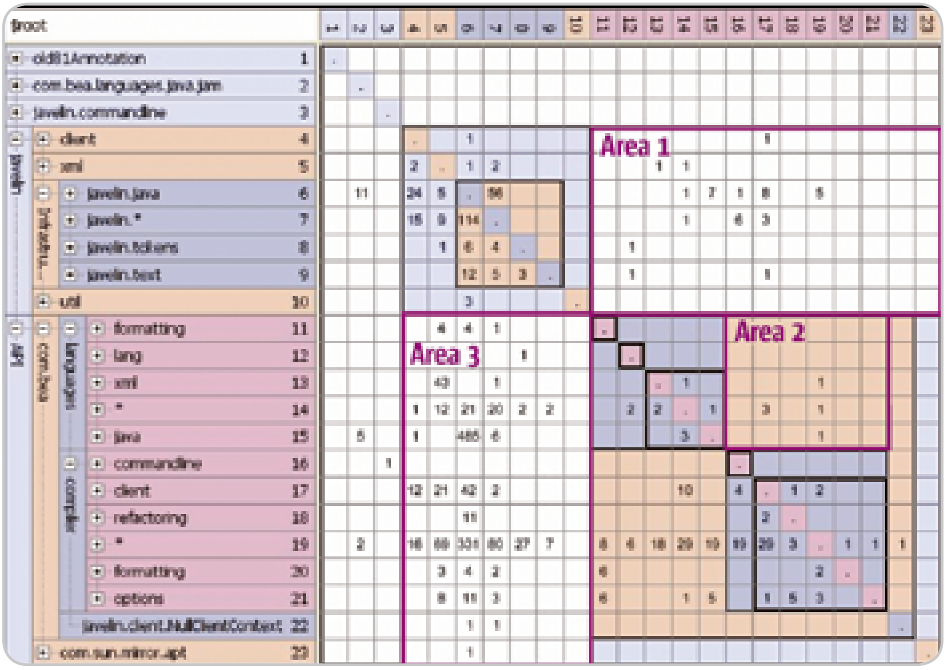


Figure 4 DSM corresponding to the conceptual architecture