One area we could help is the code writing layer of DLTK. We have direct experience in this area from JBuilder. I will explain the basic evolution and concept of Jam/Jom which is our two way code writing API we developed. I believe a similar approach for DLTK is warranted.

When we looked at providing refactoring support for JBuilder we needed two basic underlying abilities. One was a code generation framework. We had an existing code generation framework call Jot. We also needed some kind of cache of relationship information between classes (i.e. Class A has a dependency on Class B).

We created a repository that determined this information from our build dependency files and the .class files (we eventually supplemented this with an identifier cache – basically equivalent to the index searching mechanism in DLTK).

This repository provided enough information to do find references and global refactorings like rename. We chose not to use Jot for the refactorings due to limitations and fragility. For the first pass of refactorings we ended up creating a simple group editing/undo framework and used AST's directly (somewhat similar to Eclipse does for JDT) along with the repository to give us the basic reference information.

Based on the fact that Jot wasn't useful for refactoring we eventually decided to replace Jot altogether with a new API that would make code introspection and re-writing much easier. While designing this new API we also implemented our second generation of refactorings with it and redesigned our EJB designer with it as well.

One of the fundamental problems in Jot was that logical notions such as Class and Method were mixed up with physical notions such as compilation units. We decided early on that these should be treated separately with some sort of bridge between the two. Jam became the read only logical abstraction and Jom the read/write dealing with source files. Jam was optimized for fast look up of class/method level abstractions and Jom was optimized for single source file manipulation (optimized with respect to simplicity in the API and speed of execution). It was a little hard to conceptualize at first but as we progressed the effectiveness of this choice became obvious.

It is very easy to look up classes, methods and fields with Jam without any knowledge of where that object exists in source or class and then ask for the associated Jom when required. Jam was built on top our repository (JBuilders repository is equivalent to DLTK's search index engine, although DLTK seems faster than JBuilders repository).

Jam stands for Java Abstraction Model and Jom is the Java Object Model. Jam is a read only layer that provides logical information about Java Classes. It doesn't deal with source files or anything finer grained than a field or method. The API was designed to be very easy and transparent to use. It directly modeled the basic java constructs of Class, Method, Field, and Parameter. There were some other powerful abstractions such as JamType, which held fully qualified type information and was a light weight means of passing class and method signatures.

Jom was the complete representation of a single source file. A typical scenario is to use Jam to find a class or method and then ask the Jam for the associated Jom (i.e. you ask a JamClass for its JomClass or a JamMethod for its JomMethod). Jom was basically a wrapper around the AST but provided a uniform API for reading and modifying the AST.

I compared the usage of Jam/Jom with the dom in Eclipse and Jam/Jom was generally easier to use and understand (I believe we could improve on Jom as well). I did a sample refactoring in

Eclipse and compared them to the same refactoring using Jam/Jom. Here is a small excerpt from it.

//Here is an example where we are taking a declaration in an inner block and moving it to the outer block. The first is Eclipse ast and the second is using Jam/Jom (both without any error checking)

//ECLIPSE –

```
List statements = outerBlock.statements();
VariableDeclarationFragment frag = ast.newVariableDeclarationFragment();
frag.setName(ast.newSimpleName(variableName));
VariableDeclarationStatement varDecl =
ast.newVariableDeclarationStatement(frag);
ASTNode newType = ASTNode.copySubtree(ast, declarationStatement.getType());
varDecl.setType((Type)newType);
statements.add(0, varDecl);
```

//Jom

```
JomVariable newVar = JomFactory.createJomVariable(var.getName(), type);
varsBlock = var.getEnclosingBlock();
JomBlock outsideBlock = varsBlock.getEnclosingBlock();
outsideBlock.insertBefore(varsBlock, newVar);
```

Jom tends to be somewhat simpler and easier to understand for someone not familiar with working with AST's, but doesn't preclude working directly with AST's

Mark Howe