# Adding support for a new language to an Eclipse based IDE

Position Paper for Eclipse Languages Workshop

Markus Schorn

October 5, 2005

I want to be able to easily add support for a new language to an IDE based on eclipse. When doing this today I have to (re-)write at least another editor, a call-tree view, type-hierarchy-view, a build-system, a navigation facility (F3) and code-completion. This is not only cumbersome for the developer, it is also hard to keep the different language specific implementations of basically the same tools in sync. They should behave the same way and look alike.

This paper presents the outline of a proven concept of a symbol representation shared between languages that allows for a common Outline View, Type Hierarchy, Call Tree, basic navigation and code completion. It does not deal with the editor and the build. In our eclipse-based product, Workbench, we can support at least C/C++, Assembler, Ada, Java, Cobol, Fortran, Perl, Makefile, Python, Shell Script via the same symbol representation and the same views.

Of course there are language specific features, that have to be implemented on a per-language basis. I believe that for instance refactoring should be done per language.

## *Symbol Representation*

Even if languages can differ from each other very much there is a common core that can be used for common tools. The key for an adequate representation is to avoid a hierarchy of interfaces that reflect the different kinds of symbols and how they are related for a specific language. Rather than that we are using a generic approach, where a symbol simply has a name and a kind and can be queried for all kinds of properties. For actual sample code, see the last section.

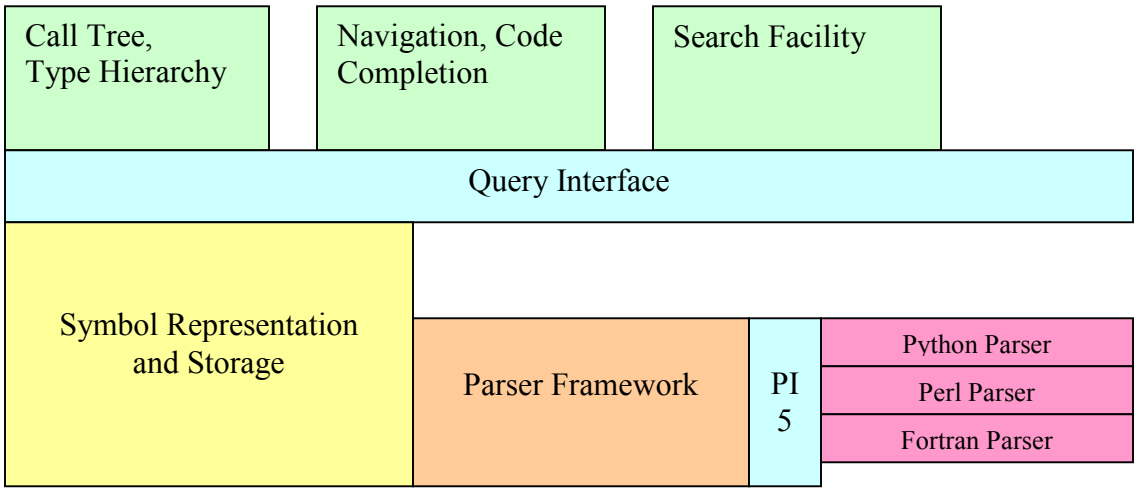
## *Parsing*

For each language you have to provide a scanner for various editor tasks that I do not discuss here. Furthermore you need a parser that supplies the symbols in the language independent format. The task of the parser can be limited to generating the symbols for a given list of files. All the handling of persisting the symbols and figuring out when to parse a file can be done by a parsing framework. The parser shall not be called explicitly to parse a single file, because a parser may need to make optimizations for parsing a whole bunch of files. The interface (PI5) between the parser and the framework allows for reporting symbol occurrences and defining relations between the symbols. Currently we use a C++-interface called PI4, which is also made public to our customers.

## Queries

All the information needed to build the tools shall be routed through a query interface, that dispatches the query to the correct source of symbols. This may also allow for tools that work cross-language.

## Architecture




## Code Fragment for Symbol Representation

```
/*
 * ISymbol.java
*/
package org.eclipse.symbol;

/**
 * @author markus.schorn@windriver.com
 */
public interface ISymbol {
    final static int SIGOPT_SIMPLE_NAME                 = 0x01;
    final static int SIGOPT_QUALIFIED_NAME              = 0x02;
    final static int SIGOPT_QUALIFY_GLOBAL_SCOPE        = 0x04;
    final static int SIGOPT_APPEND_RETURN_TYPE          = 0x08;
    final static int SIGOPT_PREPEND_RETURN_TYPE         = 0x10;
    final static int SIGOPT_APPEND_SIMPLE_QUALIFICATION = 0x20;
    final static int SIGOPT_APPEND_QUALIFICATION        = 0x40;

    final static int KIND_DECLARATION=  1;
    final static int KIND_REFERENCE=    2;


    /**
     * Returns an object describing the language the symbol belongs to.
     * @return one of LANGUAGE_...
     */
    ILanguage getLanguage();
```

```
/**
 * Returns the name of the symbol.
 */
String getName();

/**
 * Computes a signature containing parameter list, return value, etc. The
 * method exists for the purpose of displaying information.
 */
String getSignature();

/**
 * Computes a signature containing parameter list, return value, etc.
 * @param options a combination of SIGOPT_..., options to be defined
 */
String getSignature(int options);

/**
 * Returns the file this symbol belongs to.
 */
IFile getFile();


/**
 * In case the symbol is obtained from a source file this returns the
 * timestamp of the file when it was analyzed. May return 0L.
 */
long getTimestamp();

/**
 * Source range of the identifier of the symbol, may return null.
 */
ISourceRange getIDSourceRange();

/**
 * Classification of symbol-occurrence
 * @return one of KIND_...
 */
int getSymbolOccurrenceKind();

/**
 * Classification of symbols
 * @return one of SORT_...
 */
int getSymbolSort();


boolean isNamespace();

boolean isType();
boolean isAggregateType();
boolean isClass();
boolean isStruct();
boolean isUnion();
boolean isInterface();
boolean isTypedef();
boolean isEnumeration();

boolean isVariable();
boolean isEnumerator();
boolean isField();
boolean isLocalVariable();
```

```
    boolean isLabel();
    boolean isFunction();
    boolean isMethod();

    boolean isMacro();
    boolean isIncludeDirective();

    /**
     * For methods, only: The type of the this ptr, passed to the method.
     */
    String getTypeNameOfThisPtr();

    /**
     * For functions, returns the parameter list.
     */
    String[] getParameterTypes();

    // some modifiers
    boolean isConstructor();
    boolean isDestructor();
    boolean isStatic();
}



/**
 * A definition is considered a special case of a declaration. The same is true
 * for inheritance or friend declarations.
 */
public interface ISymbolDeclaration extends ISymbol {

    // accessibilities
    final static int ACCESS_PUBLIC         = 0;
    final static int ACCESS_PROTECTED      = 1;
    final static int ACCESS_DEFAULT        = 2;
    final static int ACCESS_PRIVATE        = 3;
    final static int ACCESS_NOT_APPLICABLE = 4;

    /**
     * Source range of the symbol, excluding comments, may return null.
     */
    ISourceRange getDeclarationRange();


    /**
     * For variables, fields, enumerators and typedefs:
     * Returns the type of the given entity.
     * The result may be null for other symbols.
     */
    String getTypeString();

    /**
     * For functions, methods: Returns the return type.
     * The result may be null for other symbols.
     */
    String getReturnTypeString();


    // various modifiers
    boolean isAnonymous();
    boolean isAbstractType();
    boolean isFriendDeclaration();
    boolean isDeclaration();
```

```
    boolean isDefinition();
    boolean isInheritanceDeclaration();
    boolean isInline();
    boolean isLocal();
    boolean isMutable();
    boolean isOverridden();
    boolean isOverrider();
    boolean isPureVirtual();
    boolean isRegister();
    boolean isSystemInclude();
    boolean isThreadLocal();
    boolean isTransient();
    boolean isVirtual();


    /**
     * Returns the accessibility of the symbol within its scope. One of
     * ACCESS_...
     */
    int getAccessibility();

    boolean isPublic();
    boolean isProtected();
    boolean isDefaultAccessible();
    boolean isPrivate();


    /**
     * Include directives may be transmitted even if they are located in
     * inactive code. You can test this property with this method.
     * @return whether symbol was parsed in active code or not.
     */
    boolean isFromActiveCode();

    /**
     * Returns whether this symbol declares itself as a member to the
     * enclosing scope.
     */
    boolean declaresMembership();


    /**
     * Returns symbols enclosed. Not all symbols necessarily support this
     * method. For instance the result of a CSearch may not be able to return
     * a result.
     */
    void acceptNestedSymbols(int options, IProgressMonitor monitor,
            ISymbolVisitor visitor) throws UnsupportedOperationException;

    /**
     * Returns the symbol enclosing. Not all symbols necessarily support this
     * method. For instance the result of a CSearch may not be able to return a
     * result. In case of top-level symbols null is returned.
     */
    ISymbol getEnclosingSymbol() throws UnsupportedOperationException;
}


public interface ISymbolReference extends ISymbol {
    final static int TRUE=  1;
    final static int FALSE= 0;
    final static int UNKNOWN= -1;
    final static int NOT_APPLICABLE= -2;
```

```
    /**
     * For references to variables it is nice to know whether the value of the
     * variable is read or not.
     * @return one of TRUE, FALSE, UNKNOWN, NOT_APPLICABLE.
     */
    int isReadAccess();

    /**
     * For references to variables it is nice to know whether the value of the
     * variable is being changed.
     * @return one of TRUE, FALSE, UNKNOWN, NOT_APPLICABLE.
     */
    int isWriteAccess();

    /**
     * For calls to methods it is nice to know whether the call is made via
     * a vtable or not.
     * @return one of TRUE, FALSE, UNKNOWN, NOT_APPLICABLE.
     */
    int isPolymorphicMethodCall();
}
```