

IMP User's Guide: How to Develop an Eclipse IDE for your Language using the IDE Meta-tooling Platform

May, 2008

| | |
|---|----|
| Introduction | 2 |
| IMP IDE Development Process Overview | 2 |
| Steps in Developing an IMP IDE | 6 |
| Setting Up a Plug-in Project | 6 |
| Describing a New Programming Language | 7 |
| Creating a Parsing Service and AST Representation | 10 |
| Using LPG in IMP | 10 |
| Other Approaches to Providing a Parser and AST Representation | 13 |
| Creating a Token Colorer | 14 |
| Creating a Tree-Model Builder and Label Provider..... | 17 |
| Creating an Outlining Service | 20 |
| Creating a Text-Folding Service | 20 |
| Creating a Reference-Resolution Service | 21 |
| Creating a Hyperlinking Service | 23 |
| Creating a Content-Proposer Service..... | 23 |
| Creating a Hover-Helper Service | 27 |
| Creating a Documentation-Provider Service..... | 30 |
| Creating an Incremental Builder and a Language Nature..... | 31 |
| Creating a Nature Enabler | 35 |
| Creating a "Poor-Man's Compiler" | 36 |
| Creating a Preferences Service and Preferences Page | 36 |
| Creating a Box Text-Formatting Specification | 36 |

Introduction

IMP—the IDE Meta-tooling Platform—is an Eclipse Technology project to support the development of richly featured, language-specific IDEs in Eclipse.

Prerequisites

In order to use IMP to develop an IDE you must have IMP installed in your IDE-development workspace. IMP is available through an Eclipse update site at

<http://downloads.eclipse.org/technology/imp/updates/>.

See the IMP installation instructions at

http://www.eclipse.org/imp/installation/installation_instructions.php

For information about additional sources of IMP-related features and plug-ins.

Other requirements for running IMP:

- JDK/JRE version 5.0 or later.
- Eclipse 3.2 at least; Eclipse 3.3 and 3.4 may be accommodated

[Regarding the compatibility of IMP code with different versions of Eclipse: the latest version of the IMP release is intended to be compatible with Eclipse versions 3.2, 3.3, and 3.4 M5 (although our experience with IMP on 3.3 and especially 3.4 is still limited). For IMP releases from Eclipse update sites, IMP-related features numbered 3.2.73 (or earlier in the 3.2.* line) will not work on Eclipse 3.3 or 3.4. Subsequent releases—which have been renumbered starting 0.1.*--should be suitable for use on Eclipse 3.3 and possibly Eclipse 3.4 M5.]

Terminology

The text below will refer to several of the classes generated by IMP by simplified names, such as the "Plugin" class or the "ParseController" class. As these classes are actually generated, their names will typically be prefixed by the name of the language to which they apply. Thus, if your language is called "MyLanguage", IMP by default will generate classes for you that are called "MyLanguagePlugin" and "MyLanguageParseController."

IMP IDE Development Process Overview

The Overall Process

The typical process of developing an IMP IDE can be divided into three phases:

1. Setting up a plug-in project and initiating the language
2. Providing an AST representation and implementations for basic parser services
3. Developing other IDE services for the editor, views, builders, and so on

The details of steps in each of these phases are explained in following sections of this manual—here we give just a brief overview.

The first phase is necessary to provide a context for IDE development and some basic information about the language that the IDE will support.

The second phase is to provide the language representation and processing technology that make the IDE "language specific" and provide a basis for many of the other IDE services. IMP provides tooling, based on the LPG parser generator, to support you in defining your language grammar, generating an AST, and implementing the parser services on which other IDE services depend. However, if you already have an AST representation and a parser, you can instead import them into IMP at this point.

Most of the IDE services depend on the AST representation (if not the parser directly) and so their development naturally follows phase 2. However, once the AST representation and parser are available, the choice of which IDE services to include and the order in which to develop them is largely up to you. Some of the IDE services are interdependent but many are independent. So, although we may address them here sequentially, their development is really only constrained in a few cases. We will point out specific dependencies in the relevant sections below.

Development of a Service

The development of an IMP IDE service typically follows one of a small number of patterns. All are initiated by running a "New" wizard for the particular service under development. (In the following the arrows "->" represent control and data flow between activities in the process.)

Case 1: Wizard -> skeleton implementation -> necessary customization

In the most common case, the wizard collects some basic information and generates a skeleton or partial implementation for the service. This may provide some nominal functionality for the service, but as a practical matter the service implementation will have to be manually elaborated and customized by further programming. IMP service implementations have been designed so that typically this customization is localized in one or two classes and involves simple kinds of programming, such as elaborating a switch statement or implementing a Visitor. Examples of services developed according to this pattern include the Token Colorer, Outliner, and Text Folder. (Some service implementations may be more complicated, depending on language semantics and structure and on the nature and degree of customization desired.)

Case 2: Wizard -> operational implementation -> optional customization

In a few cases, the information collected by the wizard is enough to generate an effective service implementation directly (sometimes relying on the prior implementation of other services). In these cases, further customization by programming is not generally required but is still possible if you want more highly customized behavior. Hover Help is an example of a service that is developed by this pattern.

Case 3: Wizard -> skeleton specification -> necessary customization -> operational implementation

Then there are other services for which the wizard generates a skeleton *specification* (rather than an implementation). Consequently, instead of customizing the service by programming, you customize it by filling out the specification. Operational service implementations are then generated automatically from the specifications.

Each of the services that is supported in this way has its own, domain-specific specification language. Examples include the LPG grammar specification language, the PrefSpecs language for specifying IDE preference fields and pages, and the Box text formatting language (among others). Each of these languages is supported by its own IMP IDE. Thus, for example, when you work with an LPG grammar specification, you do so in an LPG editor with outline view, hover help, content assist, and other features (plus additional documentation).

Each of these languages is also supported by a *builder*, which automatically generates Java code for the service implementations, according to what is written in the specifications. This generated code is not intended for further customization; modifications to the service should be made through updates to the specifications.

Case 4:

There are a small number of services that don't themselves require any additional development. Given other service implementations, these are available "for free" from the IMP framework. The language-specific editor is one of these, which only requires a parser to become operational at a basic level. (As more services are added, the editor grows in functionality.) Another is hyperlinking, which becomes operational once the reference resolution service is available. A built-in version of hover help also operates in this way.

The "IMP IDE Building" Perspective

IMP provides an "IMP IDE Building" perspective that you can use in developing your IMP-based IDE. This is a slight adaptation of the Java perspective, the main advantage of which is to present the IMP wizards in a way that makes them more accessible and suggests a plausible order for their use. However, we've also made all of the IMP features available through the Java perspective, so that can easily be used for IDE development as well.

Available Examples

To assist in the IDE development process, we provide numerous examples in this document. The IMP source release contains many examples that may provide guidance in IDE development. These fall into three main categories:

Functioning IMP IDEs: As noted above, IMP makes use of several custom, domain-specific languages for defining grammars and IDE services. These are supported by

functioning IMP IDEs that are integral to the IMP IDE development process and that we ourselves use for this purpose. The source code for these IDEs is available as part of the IMP release. Implementations of IMP IDEs are contained in the following projects:

- org.eclipse.imp.box (for the Box text formatting language)
- org.eclipse.imp.lpg.runtime (for the LPG grammar-specification language)
- org.eclipse.imp.prefspecs (for the PrefSpecs preferences specification language)

Other IMP IDEs are under development in various other projects in the IMP release (you can find most, if not all, of these by searching for “.g” files—their LPG grammar specifications). There is also an IMP IDE for the X10 programming language—a parallel extension of Java—which can be found on SourceForge (x10.sourceforge.net).

IDE Service Implementation Templates: The templates from which IDE service implementations are generated all contain example customizations for an example language “LEG”, the “little expression grammar.” The LEG grammar itself is used as an example in the LPG grammar templates, and most of the IMP service implementation templates contain representative code based on AST types that LPG would generate for this grammar. The LPG grammar templates for use with IMP are contained in the “templates” folder in the project org.eclipse.imp.lpg.metatooling. The IMP service-implementation templates are contained in the “templates” folder in the project org.eclipse.imp.metatooling.

The “LEG” IDE: The LEG example grammar and service implementations are complete enough that you can create a functioning IDE for LEG just by running IMP wizards without doing any additional programming or specification. To do this you must:

1. Create a plug-in project using the customary Eclipse wizard
2. Define the language using the New Programming Language Descriptor wizard (you can call the language anything you want)
3. Run the New LPG Grammar with Parser Wrapper for IMP wizard (be sure that the LPG builder runs to generate the parser and related implementation classes)
4. Run an assortment of IDE service wizards (just taking care to observe the few required dependencies among services)

(In the remainder of this manual you can find specific information about each wizard, the constraints among services, and so on.) This is essentially the same process that you would follow to develop a new IDE for a new language, minus the customizations. You can experiment with the grammar specification and service implementations to try out ideas and see how things work.

Technical note on management of plug-in extensions and dependencies

Many of the “new service” wizards will update the plug-in dependencies of the IDE project if necessary; details of these updates are generally not discussed here.

Also, many of the IMP wizards add extensions to the plugin.xml file for an IDE project. If one of these wizards is rerun it may add a second extension of the same type. This may not have a runtime impact (especially if the name of any associated implementation class remains unchanged), but the plugin.xml file can become cluttered.

Steps in Developing an IMP IDE

This section of the report addresses each of the steps in developing an IMP IDE, beginning with the creation of a project to contain the developed IDE. The steps are presented in a logical sequence, but (once the language is defined) the order of steps and even the choice of which steps to perform is largely up to the IDE developer.

Setting Up a Plug-in Project

An IMP IDE must be defined in a Eclipse plug-in project. Currently, each IMP IDE must be defined in a separate plug-in project, and the implementation of an IDE must be contained in a single plug-in project. (Some of these restrictions will be relaxed in the future.)

You can use an existing plug-in project for your IDE or create a new one using one of the Eclipse wizards or tools for this purpose. In the typical case, we use the New Plug-in Project wizard. On the first page of the wizard we

- Give the project a name (more on naming below) and use the default location
- Create a plug-in project that is a Java project (this is required) with separate source and output folders
- Select a target platform that has Eclipse version 3.2 and no selected OSGI framework

On the second page of the wizard we

- Use the default plug-in properties or update these as appropriate (for example, naming a provider)
- *Do not create an activator* (because IMP will do this; more below)
- Don't worry about the contributions to the UI
- Do not create a rich client application

Then we finish the wizard, ignoring the final page.

You can create a plug-in project by some other means, providing that you achieve more or less the same effects as the above process and address the concerns described below.

The IMP wizard will create a plug-in activator class. This is like an ordinary plug-in activator class except that it contains some additional members for IMP. If you let Eclipse create an activator class then you may end up with two activator classes (possibly in different packages) and this can lead to problems in later execution of the IDE. (If you do find that you have somehow created two activator classes, you can delete the non-IMP class. Just be sure that your MANIFEST.MF file refers to the activator class that you intend to use.)

A note about project naming: It's a good idea *not* to name your IDE development project after your language or, if you do, to give the project name in lower case. It's somewhat natural to name your project after your language, e.g., "LEG" for the LEG language. However, IMP by default creates packages for IDE service implementations that include the language name, but in lower case, e.g., "leg.parser." This leads to a

ResourceException that arises from a violation of an assertion on the Eclipse resource model. That is, no two resources can have the same name but with different cases. In this example, the model would contain a folder "LEG" for the project and a folder "leg" for the package. That's not allowed. So, if your language is "LEG", we suggest naming your project something like "LEG_IDE" or giving it a qualified name like "org.eclipse.LEG" (or using the lowercase version of the language name, "leg").

A note about the "New Project Wizard": You may come across an IMP wizard called "New Project Wizard." You should ignore it, it's not for creating new IDE-development projects (and it doesn't really work yet).

Describing a New Programming Language

The goal of this step is to create a descriptor for the language for which the IDE is intended. This descriptor will provide some basic meta-data about the language. Among other things, these data will tell your IMP-based language editor how to recognize source files in your language.

This step is accomplished by running the IMP "Programming language descriptor" wizard. To do this, invoke:

"File" -> "New" -> "Other" -> "Programming language descriptor"

An invocation of this wizard is shown in Figure 1

A note on navigating Eclipse menus for IMP wizards: There are a number of ways to navigate through the Eclipse menu hierarchy to get to specific IMP wizards. The above represent the menu path starting from "File" in the Java perspective. If you work in the IMP perspective, many IMP wizards will show up directly in the "New" menu (although you will have to traverse "Other" to get to others). In any perspective, Ctrl-N will open the "Select a wizard" dialog that will offer you the full "New" menu, including all IMP wizards. Also note that most of the IMP wizards will be found in the folder "IDE Language Support."

A note on the naming of IMP wizards: The names that appear in the menu for "New" wizards and the names that appear on wizards themselves (once opened) may vary. Usually they will obviously refer to the same concern. In this document we may likewise use slightly varying names to refer to the same wizard, depending on context. However, in all cases the particular wizard under discussion should be clear. (As in the following ...)



Figure 1: The "IMP Programming Language Description" wizard.

In the "IMP Programming Language Description" wizard, you must provide information for the following required fields:

- **Project:** Browse to or type-in the name of your IDE plug-in project
- **Language:** Enter the name of the language. This is the canonical name for your language, by which it will be identified to various IMP services. Within IMP it is case-insensitive (although case may matter elsewhere in Eclipse).
- **Description:** This field is for the entry of a user-readable, user-sensible description of the language. Although the description field is required, the description is not actually used by IMP. Enter whatever description you think is appropriate.
- **Extensions:** This field is for the entry of file-name extensions that may be used for source files in the language. Enter a non-empty, comma-separated list of extensions (omitting dots and without spaces, e.g., "c,h" for C-language source files). The IMP services defined for your language will be available for files with

these extensions. For instance, it is through these extensions that your language-specific editor can be opened automatically for files in the language.

You may also provide information in the following optional fields (note that these are generally not yet used by IMP):

- **Synonyms:** This is a comma-separated list of alternative names for your language
- **DerivedFrom:** This field should be the canonical name of the programming language, if any, from which the current language is derived. The "derived from" language, if any, should be one for which there is already a IMP IDE. It will be used within IMP for determining inheritance relationships when locating language services for a given language (for example, locating an appropriate editor if none is defined specifically for the derived language). If your language is derived from another IMP-supported language then this field should be filled in accordingly. If your language is not derived from another IMP-supported language then this field should be ignored.
- **Icon:** This should be the project-relative pathname of an icon that can be used in graphical interfaces to mark files in your language.
- **URL:** This is intended for the URL of a web page that contains information about your language.
- **Validator:** The "validator" field can be given the fully-qualified name of a class that can determine whether a given input file actually contains text in the given programming language. This class must extend the LanguageValidator class
- **Nature ID:** The nature ID is an Eclipse-internal identifier that is used to designate projects to which features of your IDE, notably the builder, may apply.

Hit "Finish" when you are done editing the various fields in the wizard. Finishing the wizard has two main effects:

One is to amend the projects "plugin.xml" file to contain an extension of the IMP-defined extension point `org.eclipse.imp.runtime.languageDescription`, which records the meta-data entered through the wizard.

The other is to create an IMP-suitable plug-in activator class (and to amend the projects MANIFEST.MF class to refer to this file).

Note on the "singleton" directive: When IMP updates the MANIFEST.MF file with the name of the activator class it also sets the "singleton" directive to true. However, this update is not always recognized by Eclipse, in which case the file will be marked with an error. This can be manually repaired by opening the MANIFEST.MF file, making an insignificant edit (such as adding and removing a blank), and saving the file, or by invoking the quick fix that is associated with the error marker.

Creating a Parsing Service and AST Representation

Most IMP IDE services rely on the ability to parse source-texts in the supported language and to operate on abstract syntax trees (AST) produced by this parsing. A parsing service and AST representation are thus fundamental to IMP-based IDEs.

There are two general ways in which parsing services and AST representations may be provided in an IMP-base IDE. One is to import existing, externally defined capabilities and adapt them to implement the parsing- and AST-related interfaces used by IMP. The other is to use facilities provided with IMP to define the language grammar and generate corresponding parser and AST representations. IMP provides the LPG LALR parser generator and supporting LPG IDE for this purpose.¹

Using LPG in IMP

Unless you already have an existing parser and AST representation, or are committed to using an alternative technology, we recommend that you use LPG to define the grammar for your language and generate a parsing service and AST representation. This is because LPG is a very flexible parser generator and because we have certain adaptations in place, such as base classes for some service implementations, which are customized to work with LPG-generated parsers and ASTs.

If you're going to use LPG in IMP to define a new language, then you should run the "New LPG Grammar with Parser Wrapper for IMP" wizard:

```
"File" -> "New" -> "IDE Language Support" -> "LPG" ->
  "New LPG Grammar with Parser Wrapper for IMP"
```

An example of the wizard is shown in Figure 2:

¹ LPG was developed independently of IMP and is available separately from it (see lpg.sourceforge.net). IMP includes a version of LPG and adaptive components that facilitate the use of LPG in IMP IDE development. This includes the LPG IDE, which was developed using IMP.

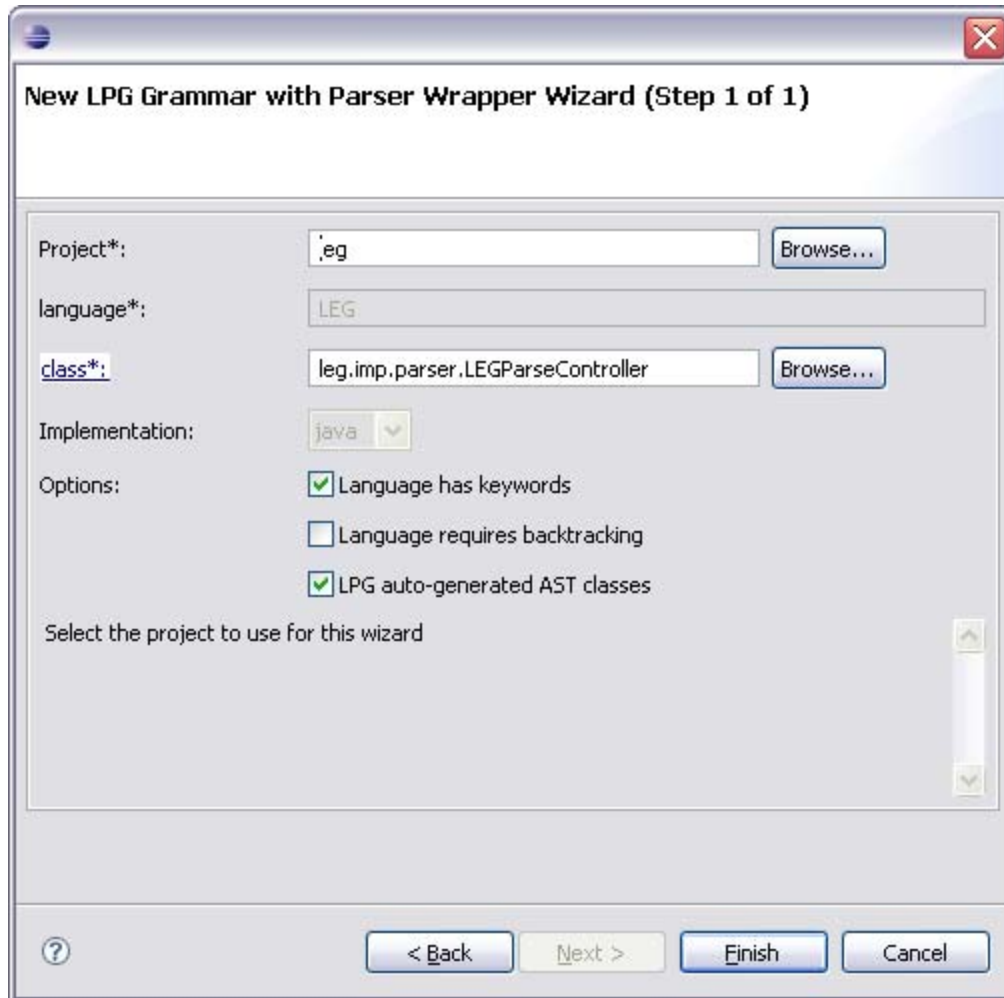


Figure 2: The "New LPG Grammar with Parser Wrapper Wizard"

When you run the wizard, be sure that the proper names are entered into the project and language fields. (These may be filled in automatically or you may have to enter them manually.) As these values are entered, reasonable default values will be entered into other fields.

- The "class" field represents the name of the Parse Controller (or "parser wrapper") class that will provide the main API through which other IMP services will access the parsing service. If you wish, edit this field to specify a non-default fully-qualified name for the Parse Controller.
- The implementation language is currently fixed at "Java"
- The various options fields should be checked according to the character of your language and what you want to generate:
 - Check "Language has keywords" if your language has keywords—this will control the generation of a separate keyword lexer
 - Check "Language requires backtracking" if your language requires backtracking—this will affect the type of parser generated by LPG

- Check "LPG auto-generated AST classes" if you want LPG to generate AST classes for your language. We strongly recommend this, since even a modest language (like LEG) can have dozens of AST node types.

Hit "Finish" when done editing.

Finishing the wizard has a number of effects. These include:

- Generating Java classes for the Parse Controller and Node Locator. These are instantiated from IMP templates and provide the main APIs through which other IMP services access the parser services.
- Generating LPG grammar specification files, including one ".g" file (a main grammar file) and possibly two ".gi" (grammar include) files (with separate lexer specifications)
- Updating the projects ".project" file to add the LPG nature and a "buildSpec" for the LPG builder. The LPG builder runs the LPG generator.

When the LPG nature and buildSpec are added, that should trigger the execution of the LPG builder, which is available through the LPG IDE, which is installed as part of IMP. The LPG builder will operate on the generated grammar files and in turn generate several Java classes to implement the parser for the grammar and, if the corresponding option was checked in the wizard, it will also generate classes for AST types and some related utilities into a separate, nested package.

You will also notice that, when you finish the wizard, the grammar files are opened in the LPG editor. This is typical of IMP wizards that generate customizable specifications or implementations: generated files are opened in their respective editors, and the cursor is placed at the point where customization should typically begin.

As mentioned above, the LPG grammar templates contain representative specifications for the LEG language, so at this point you have a working parser and AST representation for that grammar, whatever you may have called it. Unless you want to generate the LEG example, you will need to adapt or replace the grammar specifications so that they define your language rather than LEG. Each time the grammar files are updated the LPG builder should run, keeping the generated parser components up-to-date with your language definition. There is usually no need to modify any of the generated files—indeed, you should not do so.

Some instructions and examples for customizing an LPG grammar file are included in the generated grammar specifications. Additional documentation about LPG is available separately (see <http://www.eclipse.org/imp/documentation.php>) and some specific points are made elsewhere in this document where relevant.

The generated grammar files may be edited in any order. However, for purposes of generation, the general lexer depends on the keyword lexer and the parser depends on the general lexer.

Note on the LPG Grammar File Builder: If your language project is configured to include the LPG nature and builder (which should happen when you run the LPG wizard), then the builder should automatically run the LPG generator whenever a project build is triggered. It is also possible to run the LPG builder manually, as an external tool. The plug-in `lpg.runtime.java` contributes a launch configuration to the Eclipse “External Tools” run menu. To run the LPG builder manually as an external tool, select the grammar file on which you wish to run the tool and then invoke “lpg” from the External Tools run menu.

Note on the failure of the LPG builder to run automatically: For reasons that we have been unable to correct, the LPG generator will not run properly if the path name for the LPG runtime plug-in contains a blank (e.g., if it is under “Program Files”). More precisely, the LPG generator needs to access templates that are, by default, obtained from the `lpg.runtime.java` plug-in. If the pathname to this plug-in contains a blank, that blank is somewhere interpreted as a separator, thereby invalidating the argument in which the template location is passed to the generator. We have tried to avoid this by quoting the argument that contains the pathname in various places and in various ways, but we haven’t found anything that fixes the problem. You can avoid the problem by installing Eclipse in a location that doesn’t have any blanks in its pathname, by running a runtime-workbench to which you can provide templates through a workspace plug-in that doesn’t have any blanks in its pathname, or by manually running the LPG generator on your grammar files. At various times the author has done each of these. We regret the inconvenience and continue to work to try to fix the problem.

Note about compilation of generated parser classes: Your grammar may define a symbol that leads to the generation of an AST type with a name that duplicates one of the Java type names (such as “Number”). In that case, you will need to disambiguate references to your AST type in the generated parser—if you don’t, the Java compiler will report an error. You can effect this disambiguation by putting an import statement for your language’s type(s) into the generated parser. However, rather than edit the generated parser directly, the better way to do this is to put the import statement into the “Globals” section of the `.g` file for your language. That will cause the parser generator to add the import statement to the generated parser. (The “Globals” section already contain some other import statements, so you can easily see what needs to be done.)

Other Approaches to Providing a Parser and AST Representation

If you have your own parser: To work within IMP, any “hand written” or non-LPG-generated parser must at least implement the interfaces defined in the `org.eclipse.imp.parser` package of the `org.eclipse.imp.runtime` project. **[check on what else]**. These are fairly straightforward interfaces that may be relatively simple to implement using typical parsers from a variety of sources. If you are providing your own parser then you will presumably also be providing your own AST node types. The biggest challenge in integrating separately defined AST node types is making them work with the parser. So, if your AST node types are already integrated with your parser,

then that task is practically finished. Generally the IMP framework handles AST nodes as generic objects, so integrating any AST node types with the IMP framework is trivial.²

If you have your own AST representation: In some cases, you may be happy to use LPG to generate a parser, but you may want to use an AST representation that is developed in some other way. This was done in the X10 project, where the X10 language was defined as an extension of Java. LPG was used to generate the X10 parser, but Polyglot³ was used to define the AST node types (since Polyglot is designed to make this easy for languages that extend Java). The main concern in a case like this is integrating the generated parser with the separately produced AST node types. Depending on which comes first, the parser or node types, you may approach this in different ways. As noted above, since the IMP framework generally treats AST nodes as generic objects, integrating non-LPG-generated node types with the rest of an IMP IDE is trivial.⁴

If you have your own grammar: If you come to IMP with an existing grammar, if you can adapt that to the LPG notation, then you can use LPG to generate the parser and AST representation for your language. If you have a grammar and an existing parser and AST representation, then it may be easier to adapt the grammar and generate a parser and AST representation “native” to IMP than to try to adapt the parser and AST representation to fit the IMP interfaces.

Creating a Token Colorer

What the service does: The token coloring service (also known as “syntax highlighting”) enables the editor to specially “color” of text in various ways according to the type of its underlying token. This is also sometimes called “keyword coloring”, although the kinds of tokens for which text maybe colored are not restricted to keywords. Also, the notion of “color” can be construed to include any sort of text attribute. (For that matter, it’s possible to base the coloring not only on syntax but on semantic analysis.)

Prerequisites: Parse controller/Node locator and AST representation.

To run the wizard: IMP provides the IMP IDE Token Colorer wizard to support the implementation of this service:

"File" -> "New" -> "Editor Services" -> "Token colorer"

An example of this wizard is shown in Figure 3.

² In contrast to the situation with the IMP framework, which is language-independent, the various services that will be developed for the IDE will be language-dependent. However, these services are implemented after the AST representation is defined, and the internals of various service implementations can be programmed based on knowledge of specific AST node types.

³ <http://www.cs.cornell.edu/projects/polyglot/>

⁴ See footnote #2



Figure 3: The "IMP IDE Token Colorer" wizard.

This wizard is typical of many IMP wizards, where it is only necessary to specify the IDE project (from which the language can be determined) and the service implementation class. A default qualified name is filled in for the implementation class based on the language name and service type, but these can be changed.

What the wizard does:

- Generates an implementation skeleton for the service (by default called a "Token Colorer")
- Opens the generated skeleton in an editor
- Adds a "tokenColorer" extension to the plugin.xml file for the IDE project

To complete the service implementation:

The generated Token Colorer implementation will color a few of the token types found in the LEG language, such as keywords and identifiers. The TokenColorer class has two important methods. One is the constructor, where several instances of TextAttribute are

created for use with different token kinds. These attributes can be modified, for example, if you want to represent comments in dark green instead of dark red, and you can define whatever other colorings you want to use.

The other important method is `getColoring(..)`, shown in Figure 4, which returns the `TextAttribute` to be used to color a given token kind.

```
public TextAttribute getColoring(
    IParseController controller, Object o)
{
    IToken token = (IToken) o;
    if (token.getKind() == TK_EOF_TOKEN)
        return null;

    switch (token.getKind()) {
        // START_HERE
    case TK_IDENTIFIER:
        return identifierAttribute;
    case TK_NUMBER:
        return numberAttribute;
    case TK_DoubleLiteral:
        return doubleAttribute;
    default:
        if (((LEGParseController) controller).
            isKeyword(token.getKind()))
            return keywordAttribute;
        return super.getColoring(controller, token);
    }
}
```

Figure 4: Excerpt of the TokenColorer implementation.

This method illustrates several typical features of IMP service implementations: The customization is localized in one (or a small number) of classes and at one (or a small number) of spots, and the class file is opened in the editor with the cursor positioned at the starting point marked by the “// START_HERE” comment.

Note also that this method takes an `IParseController` and an `Object`. The `Object` represents the particular token to be colored. The `ITokenColorer` interface is written in terms of `Objects`, but within the language-specific implementation class, that `Object` can be cast to a known, language-specific `IToken` type. The switch statement can then be customized to return an appropriate text attribute for those token kinds that are to be colored. The Parse Controller not only provides a simplified test for keywords but also gives access to additional information about the program from which the token comes (e.g., making the whole token stream available). This can be used to gather additional information about the program if a more complicated approach to coloring is desired.

Creating a Tree-Model Builder and Label Provider

What the service does:

The Tree-Model Builder and Label Provider provide a way to create a labeled tree model based on an AST. The Tree-Model Builder builds a tree model in which nodes are constructed for selected ASTNode types; the Label Provider provides text and images for given object types, including AST node types.

The most immediate use for a Tree-Model Builder and Label Provider service is in constructing an outline view. Given a language-specific Tree-Model Builder and Label Provider, the IMP runtime will automatically populate an Outline View when files in the language are edited. But Tree-Model Builders and Label Providers can be used with any tree-structured view of the program (and Label Providers can be used apart from Tree-Model Builders as well).

Prerequisites: Parse controller/Node locator and AST representation.

To run the wizard:

IMP provides the "New Tree Model Builder" wizard to support the implementation of this service:

"File" -> "New" -> "Editor Services" -> "Tree Model Builder / Label Provider"

An example of this wizard is shown in Figure 5Figure 3.

This wizard combines the generation of a Tree-Model Builder and Label Provider since they are often used together and both will be used in creating a language-specific Outline View. However, you can choose to generate just one or the other of these services. As is typical with new service wizards in IMP, you must specify the name of the IDE project and give qualified names for the implementation classes.

An additional consideration with this wizard is that other classes may be affected. When a Label Provider is created, certain members must be added to the activator class for the IDE plug-in. This is only necessary the first time that the Label Provider is generated, so in that case the "Update plug-in activator?" checkbox should be checked. If the Label Provider is later regenerated, it is not necessary to update the activator class again, so the checkbox should be unchecked. (Regenerating with the checkbox checked may just create duplicate members in the activator class, which can be easily removed if necessary.)

Similarly, the generation of a new Label Provider causes certain String resources to be added to a resources class (e.g., for standard image names). This needs to be done just the first time a Label Provider is generated. (If the Label Provider is regenerated and the resources are updated a second time then some declarations may be duplicated, but these can easily be removed.)

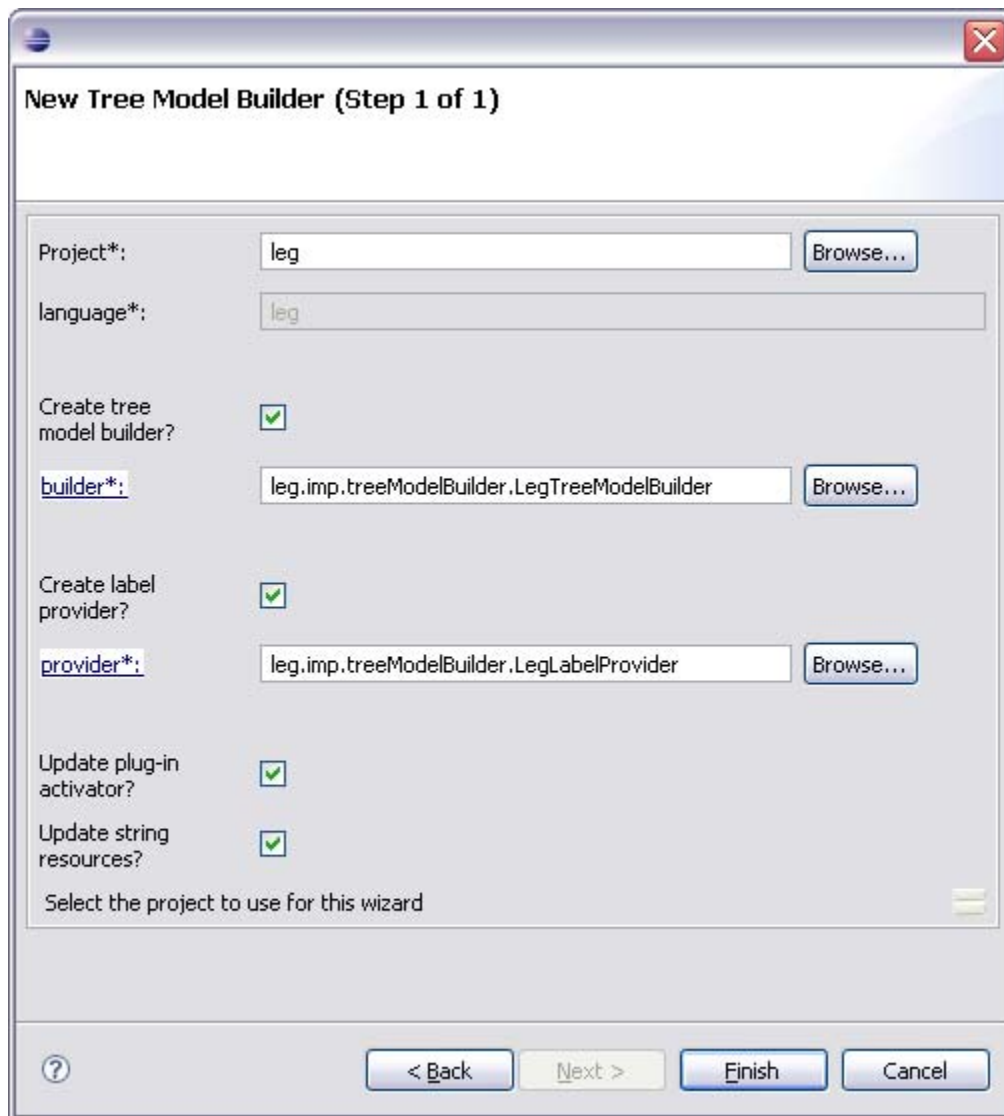


Figure 5. New Tree Model Builder

What the wizard does:

According to the settings in the wizard as described above:

- Generates an implementation skeleton for the Tree-Model Builder and Label-Provider services
- Opens generated implementation skeletons in an editor
- Updates the activator class for the plug-in with members related to label provision
- Creates an "icons" folder in the plug-in project, if there isn't one, and adds some standard image files to the folder
- Adds "modelTreeBuilder" and "labelProvider" extension to the plugin.xml file for the IDE project

To complete the service implementation:

The generated implementation skeletons for the Tree-Model Builder and Label-Provider services are very easy to complete. The Tree-Model Builder skeleton is shown in Figure 6 with example methods based on the LEG grammar. The language-specific customization that is required is to complete a Model Visitor by providing visit() and endVisit() methods to create, push, and pop items in, on, and from the tree model, where the items correspond to types of AST nodes that you want represented in the outline.

```
package leg.imp.treeModelBuilder;

import org.eclipse.imp.services.base.TreeModelBuilderBase;

import leg.imp.parser.Ast.*;

public class LegTreeModelBuilder extends TreeModelBuilderBase {
    @Override
    public void visitTree(Object root) {
        if (root == null) return;
        ASTNode rootNode= (ASTNode) root;
        LegModelVisitor visitor= new LegModelVisitor();
        rootNode.accept(visitor);
    }

    private class LegModelVisitor extends AbstractVisitor {
        private StringBuffer fRHSLabel;

        @Override
        public void unimplementedVisitor(String s) { }

        public boolean visit(block n) { pushSubItem(n); return true; }

        public void endVisit(block n) { popSubItem(); }

        public boolean visit(declarationStmt0 n) { createSubItem(n); return true; }

        // ...
    }
}
```

Figure 6. Tree-Model Builder implementation skeleton.

The implementation of the Label Provider is different. The wizard creates an “icons” folder in the IDE project (if one does not exist) and adds four “standard” images to it. In the generated label-provider skeleton, four constants are defined to refer to these images: `DEFAULT_IMAGE`, `FILE_IMAGE`, `FILE_WITH_WARNING_IMAGE`, and `FILE_WITH_ERROR_IMAGE`. These images can be replaced with custom images and new images can be added, with new constants defined to reference them.

The main functionality of the Label Provider is available through methods that return images or text when given an object (typically one that would represent a file or AST node). Within these methods the typical approach is to use a **switch** statement to branch on the type of the given object and to return a value (text or image) according to the object’s type.

The `getImage(Object)` method returns an `Image` for a given object. If the object is a file, then by default one of the file-related images is returned, depending on whether the file has associated errors or warnings. If the given object is an AST node, then the default image is returned. (This is a point at which distinct images could be returned for different types of AST nodes.)

The `getText(Object)` method returns some text, suitable for use as a label, for the given object. The example code in the generated skeleton obtains label text for AST nodes depending on the type of the node. For instance, for a declaration node, the text returned represent the type name and variable identifier. This method can be customized in many ways, for instance, in terms of the types of objects for which labels are returned, and in the method of computing labels based on the objects.

Creating an Outlining Service

As noted in the previous section, the most up-to-date way to create an Outline View for your IMP-based IDE is to create a Tree-Model Builder and Label Provider as described above. This is what we recommend that you do.

IMP previously supported the creation of an Outliner through the New Outliner wizard, which supported an alternative and less generally useful approach to building an outline tree. This wizard is still available, and Outliners created through this wizard are still supported in the IMP runtime. However, this approach should be considered deprecated.

Creating a Text-Folding Service

What the service does: The text-folding service gives you the ability to collapse regions of text in your language-specific editor. The foldable regions typically correspond to a particular kind of language construct (such as a method body or block) and are marked by little “+” or “-” annotations to the left of the text.

Prerequisites: Parse controller/Node locator and AST representation.

To run the wizard:

"File" -> "New" -> "IDE Language Support" -> "Editor Services" -> "Source folding updater"

This is a simple wizard with only two fields: one for you to select the IDE-development project, the other to give a qualified name to the class that will implement the service.

What the wizard does:

- Generates an implementation skeleton for the service
- Adds a "foldingUpdater" extension to the plugin.xml file for the IDE project

To complete the service implementation:

The Source Folding Updater wizard generates just a single implementation class, which will usually be all that is necessary for this service implementation. The main method of the Folding Updater class is `updateFoldingStructure(..)`. This method obtains the current AST for the file being edited and sends it a `FoldingVisitor` that is a specialization of the language-specific `AbstractVisitor`.⁵

The `FoldingVisitor` should contain a `visit(..)` method for each AST node type for which folding is desired. Each visit method should simply call one of the `makeAnnotation(..)` methods that are available within the Folding Updater, passing the given node (or, for some versions, offsets or tokens that may be based on the given AST node). The method `makeAnnotation(..)` makes an annotation that spans the represented text and enables folding in the source editor. The `visit(..)` methods should return true if folding is allowed for nested constructs and false if not. It may be necessary to add more complicated logic if more precise control of folding is desired, for example, if you wish to enable folding of top-level blocks but not nested blocks.

Creating a Reference-Resolution Service

What this service does: The reference-resolution service, given a reference to an entity in the language, returns a reference to the definition of that entity. This service is not visible to the user of an IMP-based IDE, but it is used in the implementation of other services, such as the hyperlinking service, that are user-visible.

⁵ The `AbstractVisitor` class is a language-specific "Visitor" implementation that contains "empty" `visit` and `endVisit` methods for each of the AST node types in the language. The example implementations of various IMP services assume the existence of such a class. These service implementations typically override the visitor methods for selected AST node types so as to take some appropriate action for those node types. LPG generates an `AbstractVisitor` class when it generates the AST node types for a given grammar.

Prerequisites: Parse controller/Node locator and AST representation.

To run the wizard:

"File" -> "New" -> "IDE Language Support" -> "Core Services" -> "Reference Resolver"

This is another simple wizard that just requires you to identify the IDE project and provide a qualified name for the reference-resolver implementation class.

What the wizard does:

- Generates an implementation skeleton for the service
- Opens this class in an editor
- Adds a "referenceResolvers" extension to the plugin.xml file for the IDE project

To complete the service implementation:

The customization of this service implementation for a particular language may be simple or complex, depending on language semantics and on the availability of information that can facilitate the resolution process.

The generated implementation skeleton contains a simple example implementation for reference resolution in LEG programs. The principle method in the implementation (which should suffice for most languages) is `getLinkTarget(..)`, which takes an Object that represents an AST node and an `IParseController` that provides the parser and additional information about the program and returns another Object that represents the AST node of the target. The implementation of this method for LEG is very simple because it can take advantage of symbol-table information that is provided by the parser. The implementation gets the parser from the parse controller, gets the symbol table from the parser, and looks up the given node in the symbol table. This illustrates how simple the Reference Resolver implementation can be if the required information is available.

(The LPG grammar specification that includes the LEG grammar also includes "action code" and supporting types that show how a symbol table can be built up during the parsing process.)

On the other hand, if information such as a symbol table is not available when reference resolution is needed then it will have to be computed at that time. In that case, some further analysis of your AST (or other program representation) may be necessary. The complexity of that analysis will depend on the semantics and structure of your language.

Another issue that arises when implementing a Reference Resolver is whether to filter the incoming AST node types. For instance, in the LEG Reference Resolver, a check is made as to whether the incoming node is an identifier—the only type of node that can serve as a reference in LEG. Only if the node is an identifier is its target looked up. An alternative approach is to attempt to find a target for AST nodes of any given type,

returning legitimate targets only for those nodes that have appropriate reference types. Either of these approaches can work—the choice between them may be based on convenience and efficiency (e.g., filtering before look-up if look-up may be costly).

Creating a Hyperlinking Service

What this service does: This service supports hyperlinking between regions of text in editors for the language. A common example is the ability to navigate from an identifier in a program to the declaration of that identifier. In the JDT's Java editor, CTRL-mouseover reveals hyperlink sources, and CTRL-clicking on a link source traverses the links, bringing up the target in an editor.

Prerequisites:

- Parse controller/Node locator and AST representation.
- Reference resolution service

To run the wizard: There is no wizard: given a reference-resolution service, the hyperlinking service functions automatically.

What the wizard does: There is no wizard to do anything.

To complete the service implementation: If you have a Reference Resolver then you're done. Hyperlinking is implemented by an "internal" class in the IMP runtime.

Creating a Content-Proposer Service

What this service does: A content proposer (which supports features like content-assist in editors) takes a prefix string and suggests terms that may represent more or less reasonable completion of that prefix in the context in which it occurs. For example, in the Java editor in the JDT, if you type the first few characters of an identifier and then type CTRL-Space, a list of visible identifiers appears, each beginning with the prefix to that point. If the prefix is also the prefix of a keyword or syntactic construct then those options are also shown. The proposals are available then for automatically completing the prefix text at that point in the editor (select one and hit return).

Prerequisites: Parse controller/Node locator and AST representation.

To run the wizard:

"File" -> "New" -> "IDE Language Support" -> "Editor Services" -> "Content proposer"

This is another simple wizard with only two fields: one for you to select the IDE-development project, the other to give a qualified name to the class that will implement the service.

What the wizard does:

- Generates an implementation skeleton for the service
- Opens this class in an editor
- Adds a “ContentProposer” extension to the plugin.xml file for the IDE project

To complete the service implementation:

The implementation for a Content Proposer for a typical programming language is one of the more complex of the IMP services. The generated implementation skeleton contains an example implementation based on LEG. The basic logic is straightforward, but it may require more elaboration of details than for many other services.

Figure 7 and Figure 8 sketch the Content Proposer implementation that is generated for LEG. As you would expect, it requires the ability to move between various representations of the program: source text, tokens, and AST nodes. For this reason, the LEG implementation depends on the parse controller, AST node types, token types, and “node locator” that are generated by LPG and the associated IMP wizard. It also makes use of the symbol table that is implemented through the action code in the LEG grammar specification. Of course, you can implement a content-proposer using language technology that is not based on LPG, but you may very well have to deal with analogous issues and require analogous capabilities.

The basic approach used with the LEG Content Proposer is to find the token that represents the text at the given offset, to determine the prefix within the text represented by the token up to the point of the offset, and then to find the AST node that contains the identified token. The type of the AST node determines whether and how the content proposals are calculated.

```

package leg.imp.contentProposer;

import leg.imp.parser.*; ...

public class LegContentProposer implements IContentProposer {

    // Return the variables that are visible at the point of the given ASTNode
    private HashMap getVisibleVariables(LegParser parser, ASTNode n) { ... }

    private String getVariableName(IAst decl) { ... }

    // Compute proposal text based on attributes of the given AST node
    // for a declaration type, depending on details of declaration node type
    private SourceProposal getDeclProposal(IAst decl, String prefix, int offset)
    {
        if (decl instanceof declaration) {
            // Compute source proposal based on declaration type and identifier
            ...
        } else if (decl instanceof functionDeclaration) {
            // Compute source proposal based on function declaration
            // type, identifier, and parameters
            ...
        }
        return null;
    }

    // Return a list of symbols in the given map that begin with the given prefix
    private ArrayList filterSymbols(HashMap in_symbols, String prefix) { ... }

    // Get the token in the given parse controller's token stream that represents
    // text at the given offset
    private IToken getToken(IParseController controller, int offset) { ... }

    // Within the text represented by the given token, get the prefix
    // that extends up to the given offset
    private String getPrefix(IToken token, int offset) { ...}
}

```

Figure 7: Content Proposer implementation example (part 1)

```

/**
 * Returns an array of content proposals applicable relative to the AST of the
 * given parse controller at the given text position.
 *
 * @param controller A parse controller from which the AST of the document
 *                  being edited can be obtained
 * @param int        The offset for which content proposals are sought
 * @param viewer     The viewer of the document; may be null
 *                  (provided if needed for further customization)
 * @return          An array of completion proposals applicable relative to the
 *                  AST of the given parse controller at the given position
 */
public ICompletionProposal[] getContentProposals(
    IParseController controller, int offset, ITextViewer viewer)
{
    ArrayList list = new ArrayList(); // a list of proposals.
    if (controller.getCurrentAst() != null) {
        // Get token and prefix string to be completed
        IToken token = getToken(controller, offset);
        String prefix = getPrefix(token, offset);
        // Get ASTNode corresponding to the token
        LegASTNodeLocator locator = new LegASTNodeLocator();
        ASTNode node = (ASTNode) locator.findNode(
            controller.getCurrentAst(), token.getStartOffset(), token.getEndOffset());
        // If the ASTNode has the proper type, compute completion proposals
        if (node != null && (node.getParent() instanceof Iexpression
            || node.getParent() instanceof assignmentStmt
            || node.getParent() instanceof BadAssignment))
        {
            HashMap symbols = getVisibleVariables(
                (LegParser) ((SimpleLPGParseController) controller).getParser(), node);
            ArrayList vars = filterSymbols(symbols, prefix);
            for (int i = 0; i < vars.size(); i++) {
                IAst decl = (IAst) vars.get(i);
                List.add(getDeclProposal(decl, prefix, offset));
            }
        } else
            list.add(new SourceProposal("no completion exists for that prefix", "", offset));
    } else
        list.add(new SourceProposal(
            "no info available due to Syntax error(s)", "", offset));
    return (ICompletionProposal[]) list.toArray(new ICompletionProposal[list.size()]);
}
}

```

Figure 8: Content Proposer implementation example (part 2)

For an appropriate AST node type, the variables visible at the given offset are found, those are filtered to match the prefix to that point, and then more complex proposals are constructed for those completions that may represent a declaration.

In the abstract, implementation of a Content Proposer is not especially difficult, but it may entail more detailed and extensive work with various program representations, and require more knowledge of program semantics than are needed to implement many other IMP IDE services.

Creating a Hover-Helper Service

What this service does: Hover help provides a pop-up with “help” text when you position the cursor over an appropriate piece of source code in the editor. For example, in the Java editor in the JDT, hover help provides information for identifiers, showing their declaration and additional information that may be obtained from their source text or JavaDoc.

Prerequisites: Parse controller/Node locator and AST representation. This is needed because the hover help is typically provided based on the AST node type that underlies the text over which the cursor is hovering.

Optional support: The IMP framework provides a base implementation of the hover-helper service that will work automatically (if not very informatively) without any additional support. However, the default implementation will take advantage of a reference-resolver and a documentation provider, if they are available. The reference resolver will be used automatically to find the target of references (e.g., the declarations of identifiers), which will then be used to determine help text for the reference source. The documentation provider will be used automatically to obtain help text for supported AST node types rather than using the nodes themselves for the text. Note that the base implementation of the hover-helper service will take advantage of these services whenever they become available, so they can be provided at any time and in any order. (The same can be true if a customized hover-helper service is implemented.)

To run the wizard:

First note that you don't have to run the wizard unless you are interested in creating a specially customized hover helper. You can get a significant degree of customization within the base hover-helper implementation just by providing a reference resolver or documentation provider.

You should run the wizard if you are interested in creating a specially customized hover helper. The wizard generates an implementation skeleton that you can reprogram however you wish (in contrast to the base implementation, which you can't reprogram). One reason you might want to do this is to make use of a particular reference resolver or documentation provider, in the event that more than one is available in your IDE. Also, in a generated implementation you have access to the parse controller and AST for the source text, which means that you can draw on additional context in formulating

your help text. Also, the wizard make it possible to opt out of the use of a reference resolver or documentation provider if, for some reason, you should want to do that.

To finally run the wizard:

"File" -> "New" -> "IDE Language Support" -> "Editor Services" -> "Content proposer"

The Hover Helper wizard is shown in Figure 9, with some selections already made.

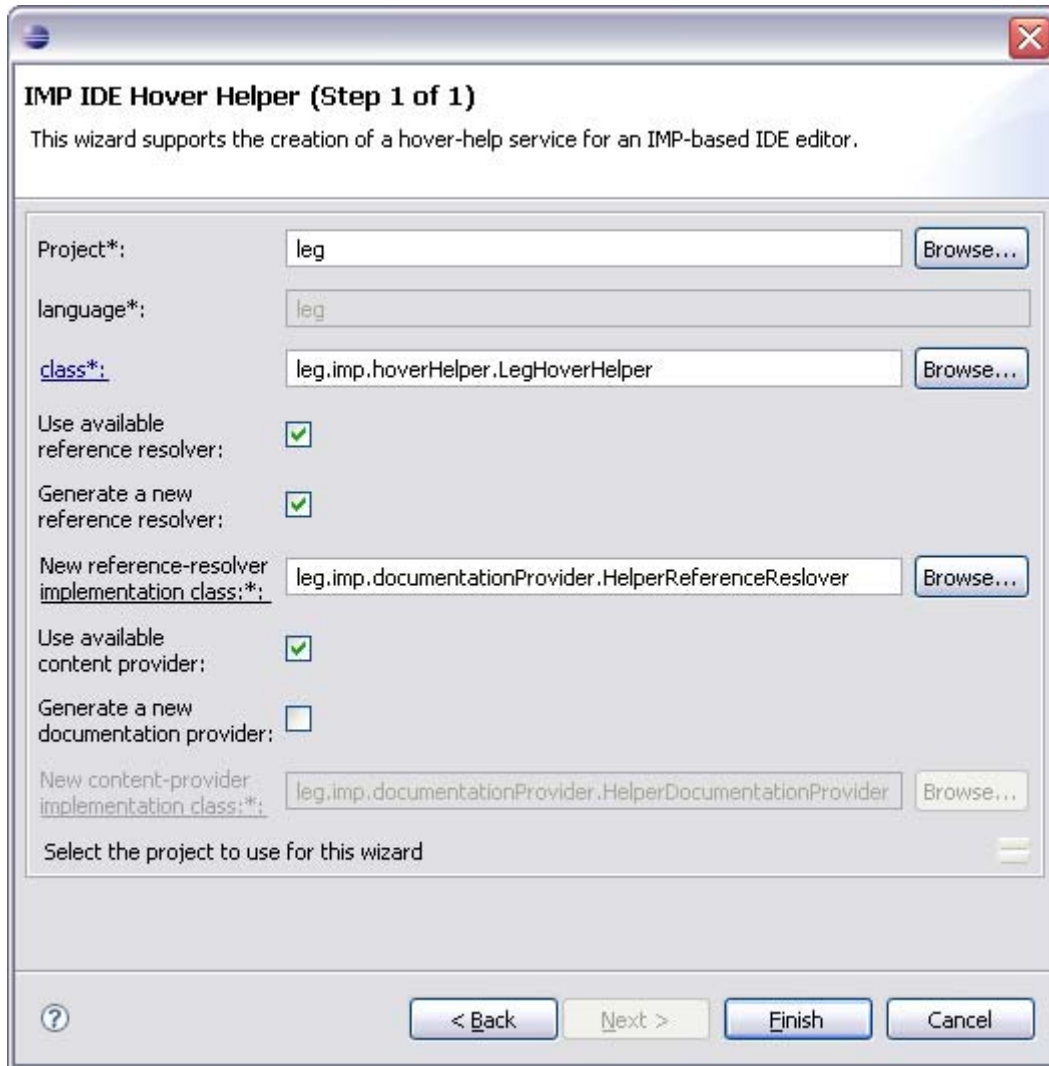


Figure 9: The IMP Hover Helper wizard.

As with other IMP wizards, it is necessary to identify the IDE project and provide the qualified name of an implementation class for the service. After that you have several options.

The first option is whether to use an available reference resolver. By default this is true. Note that this does not require that a reference resolver exist at this point, but it will set up the generated service implementation to use one if and when it can be found.

If the option to use an available reference resolver is checked, then the option to generate a reference resolver will be enabled. By default this is false. If you do not check this box, then you will not generate a reference resolver as part of the execution of this wizard. In this case, the assumption is that you already have a reference resolver that you can use or that you will create one in some other way. If you check this box, then a reference-resolver skeleton will be generated for you by this wizard (the same sort of skeleton as is generated by the New Reference Resolver wizard). In this case, the assumption is that you do not already have a reference resolver or, if you do, that you will take steps in the implementation of the hover-helper service to make use of the appropriate one. Note also that the "Reference resolver implementation class" field is only enabled when the "Generate a new references resolver" box is checked.

A similar set of options is available regarding the use and generation of a documentation provider. The details are entirely analogous.

What the wizard does:

Generates a Hover Helper implementation class that is parameterized according to settings in the wizard, adds an extension of the "hoverHelper" extension point that references this class, and opens the class in an editor.

Also does the analogous things for the Reference Resolver and Documentation Provider services, if those implementations are selected for generation by the wizard.

To complete the service implementation:

What you need to do to complete the implementation of the Hover Helper service depends on the options you have selected and the nature of any customizations you want.

If you have generated a new reference resolver, then you will need to complete the implementation of that, as described in the section "Creating a Reference-Resolution Service." If you do not have any other reference resolver, then this reference resolver will be used automatically by your hover helper. If you will end up with more than one reference resolver, then you will need to adapt the generated hover-helper implementation to select the appropriate reference resolver from among those available. Note that the reference resolver implementations will be extensions of the "org.eclipse.imp.runtime.referenceResolvers" extension point, and each extension (in this case, each reference resolver) should be given a unique id (as well as a possibly distinctive name) in its extension specification—that provides one way to sort out multiple extensions of a given point (in this case, multiple reference resolvers). Alternatively, since you control the implementation of a customized hover helper, you can make it explicitly dependent on a particular reference resolver that you may generate specifically for it.

If you have generated a new documentation provider, then you will need to complete the implementation of that, as described in the section “Creating a Documentation-Provider Service.” As with the reference resolver, if you have not created any other documentation provider, then this is the one that will be used automatically with your hover helper. If you have multiple documentation providers, then the implementation of your hover helper will have to sort them out somehow. “Note that documentation providers extend the “org.eclipse.imp.runtime.documentationProvider” extension point.”

Creating a Documentation-Provider Service

What this service does: A documentation provider provides some documentation (or other text) for a given program element (or other object). Probably the most familiar use for a documentation provider is in support of a hover helper, for which the documentation provider provides the text that is displayed by the helper. However, a documentation provider can be used to provide text for program elements (or other entities) in other contexts, as well.

To run the wizard:

"File" -> "New" -> "IDE Language Support" -> "Build Services" -> "Nature enabler"

This is another simple wizard in which it is only necessary to specify the IDE project and the qualified name of the service implementation class.

What the wizard does: Generates a Documentation Provider implementation class, adds an extension of the “documentationProvider” extension point that references this class, and opens the class in an editor.

To complete the service implementation:

The principal method in the generated Documentation Provider implementation skeleton is `getDocumentation(Object, ParseController)`, which takes (most importantly) an Object that is assumed to represent an AST node returns a String representing the “documentation” for that node, if any. The example code shows one case in which the documentation is determined by values associated with a node of a particular type, and other cases in which the documentation is determined based on the token type corresponding to the node type. In practice you can determine the values in any way that you can compute.

Note that in our example we do not do any “dereferencing” of nodes that can be interpreted as references. For example, when given a node corresponding to an identifier, we do not return text that describes the declaration of that identifier. That is because we have assumed that any such dereferencing of nodes of reference types has occurred before the Documentation Provider is called. (See the description and implementation of the Hover Helper service.) However, there may be situations in

which it makes sense to perform some dereferencing within the Documentation Provider itself.

Creating an Incremental Builder and a Language Nature

What this service does: Builders perform activities like running generators or compilers for files in a particular source language. An incremental builder runs in response to changes in source files (in particular to changes in the file resources that contain the source). The execution of incremental builders is managed automatically by Eclipse.

Natures are a meta-attribute of projects (represented in the .project file) that help in maintaining a mapping between a project and a feature or plug-in, in particular one that provides some tooling. For example, a project that is given the "Java nature" becomes recognizable as a "Java" project to the Java Development Tool plug-ins and, among other things, enables a Java builder to run automatically on the project.

Prerequisites:

There are no prerequisites on IDE services beyond the IDE plug-in class itself. However, the purpose of a builder is to run some other tool that does important work. So, the builder is only useful to the extent that there is some other tool for it to drive.

To run the wizard:

To invoke the Incremental Project Builder wizard, select

"File" -> "New" -> "IDE Language Support" -> "Build Services" -> "Incremental builder"

The wizard is shown in Figure 10 with some fields filled in.

The wizard page for the incremental-builder wizard is constructed from the extension point schema for the Eclipse "builders" extension point. Currently not all of the fields represented in that schema are used by IMP. The fields "hasNature", "isConfigurable", and "callOnEmptyDelta" are ignored by the wizard in generating the implementation skeleton. For now, they can be left blank (and any values provided will be ignored). In the future we may incorporate them.

The second "name" field and the "value" field are not used now and no use is planned for them. However, as a result of the way the schema is specified, some values must be given for these. We provide nonsense values that the wizard simply ignores. (You are free to provide custom nonsense value for your language.)

As is usual for IMP wizards, the IDE project must be specified and the name of the supported language is set on that basis. Also as usual, the IDE developer can give the qualified name of an implementation class, although we provide a (presumably) reasonable default value. Unlike most other wizards, this one exposes the "name" and

“id” fields which are used to name and identify the extensions within Eclipse. We do that for builders because a project may have more than one builder. The final field of interest is the checkbox “Add SMAP support.” SMAP support helps in maintaining the mapping from compiled files to source files for use with debuggers. Checking this box will enable this support for your language.



Figure 10: The IMP Incremental Project Builders wizard.

What the wizard does:

The wizard generates two classes: one that is an implementation for the language’s builder (as named in the wizard) and one that represents the language’s nature. The class generated for the nature is automatically put into the same package as the class for the builder. The wizard also adds three extensions to the IDE project. Two extend the Eclipse-defined “builders” and “natures” extension points, with the corresponding classes as implementations. The third extends the Eclipse-defined “markers” extension

point with a language-specific problem-marker type (the marker type is represented by a String value which is also represented in the builder class as a public String constant).

To complete the service implementation:

The Builder class is a skeleton that overrides or implements several methods defined in its abstract parent class, IMPBuilderBase:

```
getErrorMarker(..)
getInfoMarker(..)
getWarningMarker(..)
getPlugin(..)
isSourceFile(..)
isNonRootSourceFile(..)
isOutputFolder(..)
compile(..)
```

Most of these methods as generated will do reasonable things (although you may tailor any of them). The method `isNonRootSourceFile(..)` is used to identify files (like C/C++ ".h" files or LPG ".gi" files) that should not be compiled directly but that should nevertheless be processed for dependencies.

In any case you will have to edit the `compile(..)` method, which is really the main "build" method. This will have to call your "compiler" in an appropriate way for a given source file, retrieve any diagnostic messages that result from the building, and create problem markers for those. If your build activity runs in an external process, or uses `java.io` to create output files, you will also need to call `IResource.refresh()` on any folders that contain generated output files (assuming that they are somewhere in your workspace). IMP defines an interface, `IMessageHandler` (`org.eclipse.imp.parser.IMessageHandler`), to facilitate the reporting of error (and other) messages from compilers (or other builders) to interested services. IMP also provides an implementation of this interface, `MarkerCreator` (`org.eclipse.imp.builder`), that creates marker annotations for messages received and attaches those to a given source file.

Figure 11 below shows the implementation of a "compile" method that runs an LPG-generated `ParseController` in the role of the compiler and using `MarkerCreator` to handle parser error messages and create marker annotations on the given source file.

```

/**
 * This is an example "compiler", which simply uses the leg parse
 * controller to parse a file.
 *
 * TODO: remove or rename this method once an actual compiler
 * is being called.
 *
 * @param file    input source file
 * @param monitor progress monitor
 */
protected void runParserForCompiler(
    final IFile file, IProgressMonitor monitor)
{
    try {
        IParseController parseController = new LagParseController();

        MarkerCreator markerCreator = new MarkerCreator(
            file, parseController, PROBLEM_MARKER_ID);

        parseController.getAnnotationTypeInfo().
            addProblemMarkerType(getErrorMarkerID());

        ISourceProject sourceProject =
            ModelFactory.open(file.getProject());

        parseController.initialize(
            file.getProjectRelativePath(),
            sourceProject, markerCreator);

        String contents = BuilderUtils.getFileContents(file);
        parseController.parse(contents, false, monitor);
    } catch (ModelException e) {
        getPlugin().logException(
            "ModelException in runParserForCompiler", e);
    }
}

```

Figure 11: A substitute "compile" method

The generated nature class extends `org.eclipse.imp.bulder.ProjectNatureBase` and provides some simple method implementations and stubs. These can be used as-is without further modification unless more sophisticated control over building is required. (With a new nature this is not likely to be true initially.) If the checkbox "Add SMAP support" was checked in the Incremental Builder wizard, then some of these methods will be augmented with statements related to providing this support.

Creating a Nature Enabler

What this service does: A nature enabler is a utility that makes it easy for a person who is using an IMP-based IDE to associate the nature for that IDE with a project in which they wish to use the IDE. The main effect of the use of an IDE's nature enabler on a project is to enable the incremental builder from that IDE to run automatically on source files in the project. The IDE user could achieve the same effect in the target project by editing that project's .project file—the nature enabler automates this activity.

In a workspace where your IDE is installed, the nature enabler is available in the context menu when you right click on a project in the Package Explorer. It will say something like "Enable XXX builder", where "XXX" will be the name of your language.

Prerequisites: The generated nature-enabler class makes a call on the nature class that is generated along with the builder—so the New Nature Enabler wizard should not be run until after the New Incremental Builder wizard is run.

To run the wizard:

"File" -> "New" -> "IDE Language Support" -> "Build Services" -> "Incremental builder"

This is one of the simple variety of IMP wizards that just requires the name of the project and implementation class.

To complete the service implementation: There's nothing more to do.

Creating a “Poor-Man’s Compiler”

[To be done.]

Creating a Preferences Service and Preferences Page

Preferences-related services for IMP-based IDEs are supported by a preferences specification language (“PrefSpecs”) which is supported by its own IMP-based IDE that is part of the IMP release. There is a separate manual for the IMP preferences support and the PrefSpecs language. See the IMP documentation page on Eclipse.org (<http://www.eclipse.org/imp/documentation.php>) for a link to this document.

Creating a Box Text-Formatting Specification

Text formatting for editors in IMP-based IDEs is also supported by a special-purpose specification language (“Box”) with an IMP-based IDE that is part of the IMP release. There is a separate manual for text formatting and Box; see the IMP documentation page on Eclipse.org (<http://www.eclipse.org/imp/documentation.php>) for a link to this document.