IMP Formatting: How to make a formatter

$Rev:$

Table of Contents

Introduction 1
- Architecture overview 2
- Knowing the Box language 3
- Prerequisites 4
- Step-by-step creation of a formatter 4
- See also 4

Set up the formatting specification editor 5
- Extend your parser to be able to recognize fragments 5
- Extend your parser with syntax for 'meta variables' 5
- Implementing IASTAdapter for your parser 6
- Bind the formattingSpecification extension point 7

Create a formatting specification 7
- Create a working copy 7
- Edit the formatting specification 8
- Test the formatting specification 9
- Parameterize the formatting specification 9

Finalize the formatting plugin 10
- Copy the formatting specification to the workspace 10
- Bind the formatting extension point 10

Using the formatter 10

Introduction

This document describes how to make an IMP based source code formatter for a programming language. It contains a brief motivation, an overview of the source formatting architecture, and a step-by-step manual.

Formatting source code is a simple but important tool for programmers. It has been shown that consistently formatted source code, with judiciously applied indentation, significantly improves code understanding, for novice as well as for advanced programmers. The book "Code Complete" by McConnell contains references to these studies. A source formatter is applied once when foreign code is imported into the programmers' workspace, or applied once in a while by the programmer while she is editing code. The IMP formatting framework naturally supports both use cases.

The construction of a formatter, and its maintenance, can be quite a bit of work. The IMP formatting plugin intends to help you create and maintain a formatter for your programming language, or your domain specific language. The "heavy lifting", i.e. the actual formatting of source code on a page of text, is done by a formatting language called "Box". Box can be learned in 5 minutes. But first, you need to supply the IMP formatting framework with a parser for your language. After that, you can start implementing a formatter using the IMP formatting specification editor. When the formatter is finished, you can include it as an Eclipse plugin and make it available in the editor for your language.

Note

Caveat Emptor; the IMP formatting tool is very new. It can be considered a prototype which still needs some work in terms of speed of the generated formatters and easy-of-use for the IDE builder. Box and the Box processing tools that are not new; they have been used extensively. Also, the formatter has some known issues concerning corner cases of dealing with source code comments and it always formats entire files at the moment.

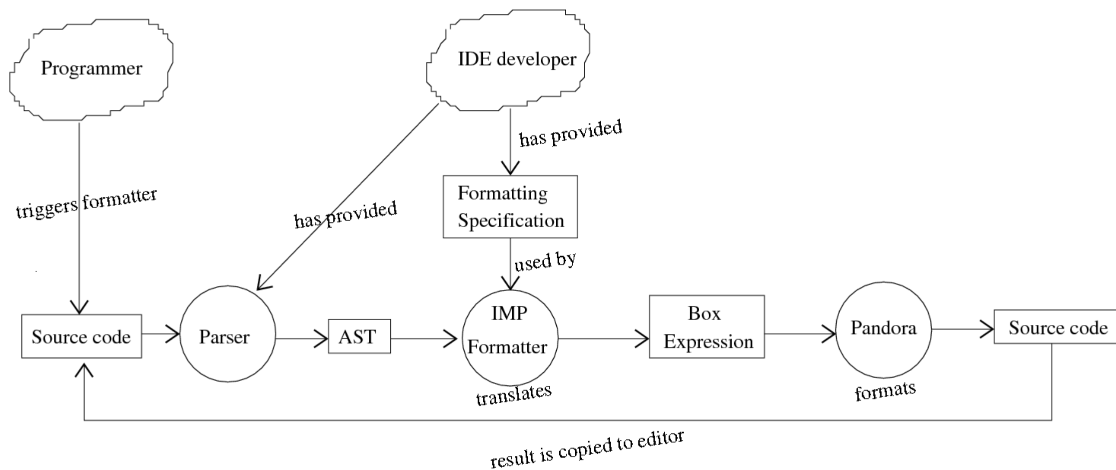
Architecture overview

After you are finished implementing an IMP-based formatter, the general control flow is this:

1. The programmer selects a piece of source code and chooses to format it using a menu option or a shortcut in Eclipse
2. The source code is parsed - using the parser you supplied - and the corresponding AST - your own AST class hierarchy - is constructed
3. The AST is translated to a Box expression using the formatting specification you have implemented
4. The Box expression is processed by **pandora**, a tool that formats the text and returns new source code
5. The new source code replaces the old source code in the editor

The following picture depicts the process.

Figure 1. Control and data flow of formatting a source file



There are three tools doing the work, the parser, the IMP formatter and Pandora.The IDE developer provides the parser. The IMP formatter is a generic component that is reused by every formatter. Pandora is also a generic component that is reused by every formatter. What the total effect on the source code actually is, is all controlled by the Formatting Specification, which is provided by the IDE developer.

Technically speaking, the IMP formatter is mainly a *pattern matcher* which applies the right Box expressions to the source code by matching AST patterns that are contained in the Formatting Specification. Pandora is a simple *constraint solver*, fitting text on a page while satisfying the boundaries of Box expressions.

Knowing the Box language

The Box language is central to all IMP based formatters. The following document, Formatting with Box and Pandora [???], is the primary online reference for this formatting language. Note that IMP currently supports the **G**, **H**, **V**, **HV**, **HOV**, **I**, **A** and **R** Box operators.

The Box language provides expressions for nesting two-dimensional boxes of text. These boxes never overlap and they never change the order of appearance of text from left to right. The Box operators try to fit characters on a limited two dimensional area in a very exact manner. Each Box operator has some *space options* (not required due to sensible defaults), which parameterize some of the behavior.

For a quick introduction, here are some example Box expressions:

- `H [ "a" "b" ]` will place a and b horizontally with a single space in between.
- `H hs=2 [ "a" "b" ]` will place a and b horizontally with two spaces in between.
- `V vs=3 [ "aa" "b"]` will place aa and b vertically with two empty lines between them. The members will be aligned at their left starting position.
- `HV [ "a" "b" "c" ]` will place the three members horizontally from left to right for as long as their is space on the page, or else members will flow to the next line.
- `HOV [ "a" "b" "c" ]` will place the three members horizontally if they will all fit on the remaining space, or else they will all be placed vertically.
- `I is=2 [ "foo" ]` will indent it's members by two spaces more than the current indentation level, but only if the box is in a context of a **V** box (or another vertical context). If the box contains more than one members it behaves as an **H** box.
- A (l,c,r) [R ["aaa" "bbbb" "c"] ["a" "bb" "ccc"]] will produce a table of three aligned columns. The first column left aligned, the second centered and the third right aligned.
- `G gs=2 op=H [ "1" "2" "3" "4" "5"]` is the grouping operator. This example translates away to `... H [ "1" "2" ] H ["3" "4"] H["5" ""] ...`
- Boxes can be arbitrarily nested, as in `V vs=2 ["begin" I hs=3 ["1" "2" ] I ["3"] HOV vs=8 [ "%% qwerty" "asdfg" "zxcvb" ] "end" ]`

The latter two operators, **A** and **G**, are advanced and sometimes tricky to get right because there are some (unchecked) restrictions to their use:

- The **A** and **R** operators need to be nested directly or you will get unexpected results. And, every **R** box needs exactly an equal number of arguments to the number of columns that the surrounding **A** box declares.
- The **G** operator will splice its resulting list of boxes into the list of boxes of its surrounding box operator!

  Example: `A (l,l) [ G op=R gs=2 [ "1" "2" "3" ] ]` will translate to the equivalent of: `A (l,l) [ R ["1" "2"] R ["3" ""] ]`
- Note how the **G** operator fills groups with empty strings ("") if the number of boxes it is applied to is not divisable by the group size.

Prerequisites

You need to install IMP and the Box tools. See http://www.eclipse.org/imp on how to the install IMP runtime. The (temporary) update site for the Box tools is: http://homepages.cwi.nl/~daybuild/releases/eclipse. In the near future, the Box tools will be distributed from the IMP update site directly.

Note

The Box tools are currently available only for linux and windows.

Note

Your users will have the same prerequisites.

Step-by-step creation of a formatter

Note

In this document we assume that you know about IMP extension points and Eclipse extension points in general. We also assume that you have already provided a language descriptor and a parse controller for your language to IMP.

The creation of a formatter has three phases, which each have a number of steps. Each phase is explained in detail in a section of this document and each step is explained in a corresponding subsection.

1. Set up the formatting specification editor
   a. Extend your parser to be able to recognize fragments
   b. Extend your parser with syntax for 'meta variables'
   c. Implementing IASTAdapter for your parser
   d. Bind the formattingSpecification extension point
2. Create a formatting specification
   a. edit the formatting specification
   b. test the formatting specification
   c. parameterize the formatting specification
3. Finalize the formatting plugin
   a. Copy the formatting specification to the workspace
   b. Bind the formatting extension point

See also

For more information on Box and formatting, have a look at these documents:

- Formatting with Box and Pandora [???]: basic manual for Box and Pandora (commandline usage)
- IMP Formatting: design: the technical documentation for the IMP formatting plugin

Set up the formatting specification editor

Note

In the current setup, a formatting specification will be edited in a *second level* Eclipse instance. This may change in the future.

The formatting specification editor is an editor for formatting rules. The editor knows about the object language that is being formatted, because you will provide it with a parser for *fragments* of the language. We assume that you have a plugin project for an existing parser for your language in your current workspace, and that you can start up a second level Eclipse (using the Run or Debug feature of Eclipse) to use it.

The pre-work you have to do now to provide a pattern parser to the formatting specification editor will safe you time later when editing the specification. Also, a pattern parser will probably be useful in the future for other features of your IDE.

Extend your parser to be able to recognize fragments

The formatting specification editor will allow you to declare formatting rules "construct by construct". For that you will write source code fragments. To be able to parse source code fragments, you should provide a parser (via the well known IParseController interface), that can accept *parts* of a source text (which we call *fragments*).

A normal parser would have only one start symbol (non-terminal), i.e. `CompilationUnit`. The parser you must construct, should be able to accept a limited amount of other start symbols.For example, when using LPG you can simply add non-terminals to the **%start** directive:

```
%Start
 CompilationUnit Statement Expression
%End
```

Or in SDF, this would look like:

```
context-free start-symbols
 CompilationUnit Statement Expression
```

Which non-terminals you choose will influence the way you will write formatting rules significantly:

- The more start symbols you have, the less formatting rules you have to write. This is because languages often reuse parts of a language. Say you forget to include a start symbol for Java expressions for example, but you do have a start symbol for statements, then you will have to write formatting rules for every kind of expression, combined with every kind of statement before you are finished. Expressions are part of statements, like the conditional expression of a while loop in Java. When you do include a start symbol for expressions, they can be formatted by independant formatting rules and you will not have to deal with the combinatorial explosion.
- The less start symbols you have, the less chance of syntactical ambiguity. So, do not add start non-terminals that overlap syntactically.

These two factors need to be balanced. Start with something simple.

Extend your parser with syntax for 'meta variables'

Now we can parse parts of a language, but we have no abstraction yet. Without abstraction you will have to write a formatting rule for every conceivable source fragment ever written or to be written. Example, you need to be able to write something like this:

```
if (<Expr>) {
  <StatementList>
}
```

Such a fragment, which we call a *pattern* because it contains placeholders, can be used to match arbitrary conditionals that only have a 'then' part in Java. We call particular instances of placeholders *meta variables*.They are called meta because they are not variables in the language itself, but variables about the syntax of the language.

You may choose your own syntax for the meta variables. You may choose which non-terminals to provide meta variables for. However, here are a number of advises:

- Start with providing meta variables for the non-terminals that are start non-terminals. Apart from the top one (i.e. CompilationUnit), you will need at least variables for all of these.
- Have meta variables for all classes of identifier tokens. You will definitely need to be able to abstract from particular names of source code entities, for obvious reasons.
- Choose a consistent umambiguous escape character for all meta variables. In the above example, we chose to wrap variable names in `<..>`. That would be a good choice for Java 1.4, but a bad one for Java 5 since these characters are now used for type parameters.
- Make sure that meta variables can be numbered or otherwise be made unique. Example: `<Statement.1>, <Statement.2>`. This is very necessary later on to get a correct formatter.

Implementing IASTAdapter for your parser

The source formatter works by pattern matching on AST's. To know about what AST format you are using, there is an IASTAdapter interface (from the org.eclipse.imp.runtime plugin) that you must implement. The method names should be self-explanatory and there is some javadoc documentation there too.

You do not have to implement all methods for the purpose of formatting. These are the relevant methods:

1. getChildren()
2. getOffset() and getLength()
3. getTypeOf()
4. isMetaVariable()

Note

The IASTAdapter interface has not stabilized yet, but these methods will be there in one form or another.

The semantics of how you implement IASTAdapter and the properties of the underlying AST representation you use are of great influence on the semantics (and correctness!) of the source formatting framework:

- getChildren() must return AST children in order of appearance in the source *text*.
- getChildren() must eventually (recursively) yield all visible parts of the source *text* (see getOffSet() and getLength())
- getOffset() returns the zero based character offset of the starting character of an AST node

- getLength() returns the number of characters that the AST node encompasses (including whitespace!)
- getTypeOf() returns a *canonical* name for a particular kind of AST node.

  Note

  The basic semantics of pattern matching is based on this method being correct. Pattern matching will continue matching the children of an AST node if and only if the type names of the pattern node and the object node are equal.

  Note

  Nodes that are semantically equivalent but syntactically different **MUST** have different type names, or else the syntax of the source code that is formatted might change during formatting.

- isMetaVariable() should return true if and only if the current AST node should be interpreted as a meta variable.

The author of the formatting framework recognizes that not every AST class hierarchy easily meets the above requirements. This is a fact of life; however when you do have an AST class hierarchy that meets the above requirements, many other IDE features - particularly the ones that manipulate source code - will be easier to implement.

Bind the formattingSpecification extension point

Now we you are ready to instantiate a formatting specification editor. Use the "New Formatting Specification" wizard from the IMP IDE building perspective, or simply bind the extension point from org.eclipse.imp.formatting. The extension point contains:

1. a reference to a formatting specification file (with the `.fsp` extension)
2. a reference to your IASTAdapter
3. a reference to an IParseController that controls the parser that can handle fragments and meta variables

When you use the wizard, a proper skeleton `.fsp` file will be generated for you. When you do it manually, please copy the contents of this template file and fill in the name of your language (essential):

```
<?xml version="1.0" encoding="UTF-8"?>
<formatter>
 <language>YourLanguageName</language>
 <rules/>
 <space-options/>
 <example/>
</formatter>
```

Create a formatting specification

Create a working copy

Because the formatting specification editor uses a parser that is in your workspace and not in your installed plugins, you have to start up a second level Eclipse. Also, you need to copy, or link the .fsp file somewhere in the second level workspace.

By double clicking on the .fsp file in your second level workspace, a specialized editor will be opened to allow you to specify a formatter for your language. There are three tabs to this editor:

1. The Rules tab, which contains a table of formatting rules
2. The Example tab, which contains a contiously formatted example source text
3. The Options tab, where you can declare global variables for Box space options

Edit the formatting specification

By adding rules in the Rules tab for every construct in the language, you increase the functionality of the formatter.

Note

A formatter is always complete, even though you have not produced rules for every language construct. The IMP formatter will choose a default layout for any construct you have not provided a rule for yet. It will guess a layout by looking at the shape of the AST's. However, the default is almost never very pretty. The main reason for the existence of default rules is correctness: they make sure all of the source text of the original program ends up in the formatted program.

The rule editor has three columns, only two of which are editable. You edit the second 'Box' column. The syntax of the text in that column is Box expressions. The first column will display an error message when you have typed a syntactically invalid Box expression.

Key point: Box expressions contain literal strings, for example: `H [ "if" ]` contains "if". The literal strings of a Box expression are parsed using your *fragment parser*. Here's an example:

```
V [ H [ "if" "("  "<Expression>" ")" "{" ] I [ "<StatementList>" ] "}"]
```

This example shows a Box expression, which embeds a Java fragment. This is the embedded fragment, which will be shown in the third (Preview) column:

```
if ( <Expression>) { <StatementList> }
```

The rule editor will not accept syntactically invalid embedded fragments.

Note

Technically speaking, the pair of the Box expression and the embedded fragment is called a *rule*. A rule fires on a part of an AST when the fragment pattern matches. When it fires, that part of the AST is translated to the corresponding Box expression. The embedded meta variables are replaced by the corresponding Box expressions for those parts of the program text. This is how a set of rules finally results in a total translation from the source text to a Box expression.

The rule editor has a number of buttons to help you get started. Use the "Add rule..." button first. It will allow you to type a program fragment, and it will generate a rule for you. This functionality uses a very simple tokenizer (separate each token by spaces), it adds the right quoting around the literals and provides a basic Box operator. For example, when you type: `1 == 2` in the popup dialog, it will generate a rule: `HOV [ "1" "==" "2" ]`.

The "Add rule" button simply adds an empty rule, in which you can start editing a new Box expression. Notice how the preview changes when you edit the rule, and how the first 'Status' column indicates that

there are problems. The preview column also applies the Box operator semantics, such that you can see how the fragment would look like in principle.

Note

The assumption of the Preview column is that there is always enough horizontal space, so you will not see the vertical effects of the **HOV** or **HV** boxes.

Nothing prohibits you from writing overlapping rules. I.e. you could have general rules for general formatting, and add special cases for special purposes. The rules are tried *in order*, so the first rule that matches takes precedence over the others. Take these two example rules for Java:

```
V [ H [ "if" "("  "<Expression>" ")" "{" ] "<StatementList>" "}"]        Preview:
---and---
V [ H [ "if" "(" "<Expression>" ")" ] I [ "<Statement>" ] ]               Preview:
```

The second rule overlaps with the first, because a block Statement is also a Statement. However, we do not want to indent the curly braces, and we want the first curly brace on the same line as the conditional expression. So, the first rule is a special case that takes precedence.

Key point: meta variable names better be unique for every pattern. What the variables are replaced by is identified by name, and not by position in the pattern, so if you use the same variable name twice, the formatter will replace the second occurrence by the Box expression for the first. The resulting source code will be broken. There is no static checker in the editor for this yet.

You can add so called separators to the rule table to logically group your rules. The separators have no semantics, they are there to keep the formatting specification readable. Finally the "Up" and "Down" button allow you to change the order of rules in order to influence their grouping, but more importantly their relative presedence.

Test the formatting specification

The Example tab can contain any syntactally valid fragment of source code in your programming language. When this tab is active and you press the "Format" button, the source text will be formatted using the current set of formatting rules and the current set of space options. Also, when the focus changes to the Example tab, the current set of rules and space options are applied.

You are advised to leave a small but complete example of a source text in this example tab. It will be saved with your formatting specification and can serve as a quick regression test when you add rules.

Parameterize the formatting specification

Box operators may have space options. A good example is the **I** box: `I is=4 ["a"]`. A consistent use of the space options is necessary to get a consistent source code formatter. The number 4 will appear in many rules that use indentation.

Instead you can define a global variable in the Options tab and use it anywhere there is a space option. All space option variable start with a '$' sign, which is enforced by the option table tab. All values of space options are non-negative integer numbers.

Example usage:

```
V [ H [ "if" "("  "<Expression>" ")" "{" ] I is=$indent [ "<StatementList>" ] "}"]
---and---
V [ H [ "if" "(" "<Expression>" ")" ] I is=$indent [ "<Statement>" ] ]
```

Note

There is a plan to expose these global parameters to the end user somehow, such that she can influence basic properties of the formatter. Another feature would be to include boolean parameters for rule activation. In this manner we can generate source formatting preferences with alternative pre-cooked formatting behavior straight from your formatting specification. If you want to help out, please contact us.

Finalize the formatting plugin

The `.fsp` file now contains many formatting rules, but it is in the second level workspace unless you used a softlink.

Copy the formatting specification to the workspace

Copy the `.fsp` file back to the first level workspace. Remember to put it where the formattingSpecification extension point expects it to be. You can look up this location in the `plugin.xml` file of your plugin project.

Bind the formatting extension point

Now you just need to tell the IMP editor to use this formatter. IMP is open to other implementations of formatters in principle. You now need to tell it to use the formatting functionality from org.eclipse.imp.formatting instead of something else. However, if you have used the 'New Formatting Specification' wizard, it will have provided the appropriate extension already. When you did not use the wizard, you still need to add the following extension (fill in your language name):

```
<extension point="org.eclipse.imp.runtime.formatter">
      <formatter
            class="org.eclipse.imp.formatting.SourceFormatter"
            language="YourLanguageName">
      </formatter>
</extension>
```

Using the formatter

IMP will automatically trigger the formatter when using the "Format Source Code" short-cut, or when you select "Format Source Code" from a menu. IMP knows to select the right formatting specification using the language name that is part of the formatting specification file. The first time that a formatter is being run, there may be a hick-up due to loading of the specification and parsing of the patterns.

Note

The current implementation formats entire source files. Future work includes formatting selected parts.