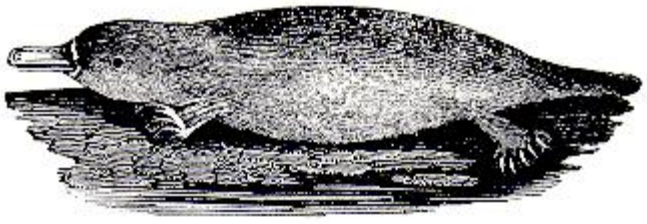
ANATOMY OF THE PLATYPUS



The Architecture and Design of Platypus 0.2.x

December 2009

TABLE OF CONTENTS

INTRODUCTION 3

OVERVIEW 3

DELIVERABLE 3

ARCHITECTURE 4

SETUP AND CONFIGURATION 4

A1. SET UP THE LITERALS 4

A2. SET UP THE GDD 5

A3. PROCESSING THE COMMAND LINE 6

A4. PROCESSING THE CONFIGURATION FILE 6

A5. LOCATING THE OUTPUT PLUGIN 6

PARSING THE INPUT FILE 7

B1. TOKEN TYPES 7

B2. GETTING LINES OF TEXT 7

B3. SETTING UP THE COMMAND TABLE 8

B4. PARSING 9

B5. THE CONTENTS OF TOKENS 9

B6. COMMAND ALIAS AND SUBSTITUTION 10

B7. COMMAND TOKEN PARAMETERS 11

B8. FURTHER NOTES ON TOKENS 12

INPUT PLUGINS 12

OUTPUT PLUGINS 12

D1. LISTING PLUGIN 13

D2. PDF PLUGIN 13

IMPLEMENTATION NOTES 16

INDEX OF JAVA CLASSES 17

INTRODUCTION

Platypus is a software package that converts a UTF-8 file containing text and possibly embedded formatting commands into typeset-quality documents in a variety of output formats. In this way, it is similar to TeX and LaTeX, troff and groff, and to Lout. It has several compelling advantages over those systems. These advantages are discussed at the project's principal website at http://platypus.pz.org.

Platypus is written primarily in Java and requires the Java runtime to run. The currently available release of Platypus is v. 0.2.0, which ships as a single JAR file and requires Java 6 SE (or Java 1.6) to run. This version of Java is required due to the upcoming incorporation of a scripting language as a macro language for Platypus.

OVERVIEW

Platypus uses a simple three-part processing pipeline: the Platypus document is read and parsed by the Platypus core. The parsed result is then modified by input plugins. This modified token stream is then passed to an output plugin for conversion into a typeset-quality document. So, parser→ input plugin → output plugin.

Currently, Platypus has not implemented any input plugins. However, these are foreseen to include straight-forward functionality, such as:

- converting markdown into the corresponding Platypus commands;
- converting " and ' into properly angled quotation marks;
- converting double spaces after a period into single spaces;

When implemented, input plugins will be chainable, so that multiple filters and conversions will be possible. (While not currently part of the project plan, it is possible that one day these input plugins will be able to convert files in non-Platypus formats—such as TeX/LaTeX, XML, or FOP—into Platypus token streams.)

Output plugins are expected to represent the bulk of the codebase for Platypus. Currently, two output plugins—PDF and listing files in HTML—are available. An HTML plugin is in design. The PDF and HTML plugins are the principal plugins for the majority of users of Platypus. Additional plugins that generate Microsoft Word, and other formats are anticipated.

DELIVERABLE

Platypus 0.2.x ships as a zip file that expands into a directory (the Platypus home directory), which contains the Platypus JAR, as well as subdirectories for: configuration files (/config); licenses; fonts (/fonts); vendor libraries (/lib), and plugins (/plugins). A future subdirectory is anticipated for dictionaries. The user must set an environment variable, PLATYPUS_HOME, to point to the Platypus home directory.

ARCHITECTURE

The Platypus architecture is based around a Platypus core that performs setup and configuration, and then parses the input file. Then input and output plugins are applied to the token stream (technically a linked list) generated by the core. Via these plugins, an output document is generated.

Technically speaking, the Platypus core hands off a data structure (the Global Document Data or GDD) that contains not only the token stream, but other data items that were collected during the core processing. These items include:

- command-line parameters
- the contents of the configuration files
- the contents of the literals file
- the Platypus logger
- various minor data fields

When the Platypus core loads an output plugin, it as passes the GDD to the plugin, and then waits for the plugin to return. When the output plugin returns, Platypus exits. By definition, then, plugins perform their own set-up and their own error handling.

The rest of this document examines the four principal Platypus processes — setup and configuration, parsing, input plugins, and output plugins. It also concludes with a brief discussion of the coding approach used in the project. Thereafter are appendices that explain various aspects in detail.

SETUP AND CONFIGURATION

Setup and configuration of Platypus consists of six steps, all called from main() in **Platypus.java**:

- Set up the literals
- Set up the GDD
- Process the command line
- Process the configuration file
- Locate the output plugin

These are discussed next.

A1. SET UP THE LITERALS

Platypus places all literals in a resource file, in anticipation of someday offering the software in multiple languages. The file, which defaults to *Platypus.properties* is located in the /config directory under the Platypus home directory. It use the format of Java resource bundles, namely single-line entries of the form:

```
abc=xyz
```

Setting up the literals, consists of locating the literals file, reading it in, and placing the entries as key-value pairs into a hashMap. All literals operations are handled in **PropertyFile.java.**

If an error occurs prior to successful retrieval of the literals file, the error messages are perforce hard-coded in English.

While not currently implemented, once literals files have been translated into other languages, a convention for specifying which language's literals to use in Platypus messages will be devised, probably using a command-line option to specify the language.

A2. SET UP THE GDD

The GDD, as explained earlier, is primarily a container for data items that are needed during the parsing and output cycles. It is found in **GDD.java**. If it singletons were easier to use in testing, GDD would be implemented as a singleton. However, as it is now, it's not — Platypus simply uses just one instance of it.

Setting up GDD comprises several steps:

1) Setting up the default Platypus logger. This logger uses custom formatter (see **LogFormatter.java**), which outputs the log message to the console as a single line preceded only by the severity level.

2) Setting up a hash table (a TreeMap) of system strings (see **SystemStrings.java**). This map contains various strings that are defined and updated by Platypus and are used for printing debug info to the console and for the upcoming macro/scripting language. An example would be the format of the output document, or the full command line, etc.

   (Eventually a similar structure will be set up for user-defined strings, so that users can define macros to replace repeating text or complex command sequences.)

3) Getting the Platypus home directory location from the environment (see **PlatypusHomeDirectory.java**)

4) Extracting the user environment strings from the user system

GDD setup also creates several empty data structures that are used in later steps:

- A command table, in which all legal Platypus commands are stored for reference use by the parser. (See **CommandTable.java**)
- A structure containing all the lines of text read from the Platypus document. (See LineList.java)
- A list of the parsed input tokens. (See **TokenList.java**)

- Various minor fields that control some aspects of parsing

A3. PROCESSING THE COMMAND LINE

The command-line processing is called from Platypus.java. It relies on the command-line interface provided by Apache Commons CLI (see http://commons.apache.org/cli/).

Processing including validation of options and error reporting is occurs in **CommandLineArgs.java**. The CLI package expects that all command-line options are preceded by a hyphenated option, whereas Platypus views the first two entries on the command line (if they're not hyphenated) as the input and ouput file names. Consequently, the CLI processing injects –inputFile and –outputFile in front of these arguments (respectively) prior to processing.

v. 0.2.0 adds a feature whereby a single file name (therefore, input file) on the command line defaults to an ouput of a PDF file, whose name defaults to inputfile with its extension replaced by .pdf.

A4. PROCESSING THE CONFIGURATION FILE

The configuration file is a property file just as the literals file is. (See **PropertyFile.java**) It contains configuration information that locates plugins for input and output and controls options for the parser and for the plugins themselves.

The configuration file's name defaults to Config.properties and is located in the /config directory below the Platypus home directory. A different file name or location can be specified on the command line, using the –config option.

Once the configuration file is read and parsed, it is stored in the GDD. If no configuration file is found by Platypus, the program exits with an error message.

A5. LOCATING THE OUTPUT PLUGIN

The name of the output plugin is computed as the prefix `pi.out` (pi = plug-in) followed by the format of the output. So the name of the config file entry for the default PDF plugin is pi.out.pdf. In the default config file that ships with Platypus, this plugin is called pdf.jar. So, the complete entry in the config file is:

```
pi.out.pdf=pdf.jar
```

By convention, this points to the file pdf.jar in the \plugin directory under PLATYPUS_HOME.

If the plugin JAR is located anywhere else, then its location must be specified using the config entry

```
pi.out.pdf.location=other_location
```

The –format option on the command line allows the user to specify a different plugin than the default for the output file's extension. This option is used in generating HTML listings of Platypus files. Without this capability, Platypus would search for pi.out.html.location. But listing files are generated using `–format listing`, command line option, which tells Platypus the plugin is located in the config file via `pi.out.listing.location`.

Once the name of the plugin JAR is located, it is loaded and the Start method is called. This is explained later.

PARSING THE INPUT FILE

Parsing consists of breaking the input file into chunks that can be processed further. The largest chunk handled by Platypus is a single line. Block comments are the only elements that can span more than a single line and they are handled specially. All commands must start and finish within the same line of text, or an error is generated.

In the event that a command were to accidentally span more than one line, the entire command would be treated as text and output as text to the token stream. This action illustrates a key principle of Platypus, namely that a pure text file (with no Platypus commands) can be run through Platypus and generate a sensible output. In this way, someone could use Platypus simply to convert text files to PDF or HTML and if a sequence of characters appeared to be a Platypus command, but in fact was not one, it would be output correctly as text. Note that most other typesetting command languages, such as TeX/Latex cannot do this.

B1. TOKEN TYPES

There are only a few kinds of tokens (see **TokenType.java**) that Platypus identifies in an input file:

- Text — anything that is not one of the tokens below.
- Comments — these can be line comments or block comments
- Commands — There are three kinds of commands, that vary only in their syntax
    - Simple commands (they have no parameters)
    - Single commands (they have a single parameter)
    - Compound commands (they have one or more parameters)
- Symbols — these are actually commands, but are so numerous that they are treated as a category apart. They include all symbols and foreign characters.

Eventually, Platypus will add additional tokens. See section B8.

Note: the input file can contain macros. These macros, which currently are simple string replacements, are processed in the output plug-in and generate a separate token: macro_text. It is processed as regular text and distinguished from regular text only so that the function that dumps tokens (via the –vverbose option) can identify for the user what was input text and what was the result of macro look-ups.

B2. GETTING LINES OF TEXT

The **Infile.java** class handles the I/O for the input file. It breaks up the input file into a set of InputLines (see **InputLine.java**), which consist of a structure with three fields: the file number, the line number in the file, and the

content of the line. (The file number, hitherto undiscussed, simply lays groundwork for an upcoming feature: the ability to include other text and Platypus files in the current file. At that point, the need to track which file a given input line came from will be necessary. For the nonce, the file number is set to 0.)

The individual lines are read, a CR/LF is appended, which guarantees that we have an ending token for the line parsing, and then the InputLines are placed into an ArrayList (see **LineList.java**).

B3. SETTING UP THE COMMAND TABLE

The Command Table holds all implemented Platypus commands in a HashMap (see **CommandTable.java**). These commands are read from a PropertyFile, called Commands.properties (in the /config directory). That file has the following format:

```
1. [font|size=vn
2. [fsize:=r [font|size:
3. [+i]=0n
4. [-i]=0n
5. [indent:=vn
6. [leading:=vn
7. [align:=sn
```

The format, expressed syntactically, is: Command-root=command-type|allowed in code? |substitution string

Command types include:

- 0 = simple command (example, see line 3 above),
- s = single command taking a string (line 7),
- v= single command taking a value, such as a number and units (see lines 1, 5, and 6),
- r = replacement command, that is, it's the syntax for an alias to another command (see line 2).

These command types have corresponding classes that process them, entitled **Command0.java**, **CommandS.java**, **CommandV.java**, and **CommandR.java** (respectively).

The allowed-in-code filed is either a y or an n for yes/no respecitively. It indicates which (few) commands are interpreted in code. (Code listings might heavily use [ ], and so only a minimum of formatting commands are allowed in code blocks.)

The entry for CommandR entries is followed by the subsitute root that should replace the specified root. So, on line 2, the root `[fsize:` should be replaced with `[font|size:`.

The parser identifies commands by comparing the root of commands it encounters in the input lines with command roots in the command table. When a match is made, the command is parsed further, using one of the four Command classes listed above. If no match is made, Platypus assumes that the token is text that happens to have the syntax that looks like a command, but is not one. (This is a generous interpretation. Many times it will be an error.) It issues a warning measure. By converting the unrecognized command to text, two benefits accrue: 1) if it is text, all is well; 2) if it's a malformed command and the user did not see the error message when Platypus was run, the odd entry in the document will alert him to the error and enable him to locate it in the text file.

The root of a command is the part of the command sufficient for Platypus to recognize which command it is. Here are the various kinds of command Platypus uses, with their root underlined:

Simple commands (i.e., with no parameters): [pg] note: the new page command

Single commands (1 parameter): [fsize:15pt] note: set font size to 15 points

Compound command (1+ parameters): [font|size:15pt|face:Arial] note:set font to 15 point Arial

The entries in the Command Table, use a key consisting of the command root and a value consisting of an item with the Commandable interface. (see **interfaces.Commandable.java**). Both Commands and Symbols use the Commandable interface.

Commands (see **Command.java**) are an abstract class that is implemented by numerous classes in the **command** package.

B4. PARSING

The bulk of the parsing is done by **PlatypusParser.java**. It is called from **Platypus.java** and passed the list of lines. Before parsing, the parser checks the configuration file to make sure that this output plugin wants Platypus to do the parsing. The default is for Platypus to parse. However, this option enables future plugins to do their own parsing of the input file, should they want to.

The lines are parsed individually in units called segments. Segments consist of portions of the line that are either text or not text. So, for example, a line is parsed by having all the text up to the first command written out as one text token. Then, if a command follows, that is parsed as another segment, and so forth.

Every line will have at minimum a CR/LF, which is converted into a token by the parser. So, the only lines that do not result in tokens are comment lines in which case the CR/LF is ignored. However, the configuration file allows the output plugin to request that comments be preserved. (This option is used, for example, in the listing plugin.) This ability also enables a future output plugin to extend the Platypus capabilities by passing values or special commands via comments.

B5. THE CONTENTS OF TOKENS

Tokens are defined in **Token.java**. They contain the following fields:

- fileNumber
- lineNumber
- token type
- token subtype
- parameter

The first three fields have previously been explained. The token subtype is used to identify the command when the token type is `TokenType.COMMAND`. Here are two examples of the list of tokens generated by the following compound command. (Note: the fileNumber and lineNumber fields are ignored for this example.)

Single command: `[leading:12pt]` results in a single token:

```
Token type = COMMAND
Subtype = COMMAND_LEADING
Content = leading:12pt
Parameter
       Type: integer
       value: 12
       Unit: pts
```

Compound command: `[font|size:12pt|face:arial]` results in four tokens:

```
Type = COMPOUND_COMMAND
Subtype = COMMAND_FONT_FAMILY
Content = [font|size:12pt|face:arial]


Type = COMMAND
Subtype = COMMAND_FONT_SIZE
Content = size:12pt
Parameter
       Type: integer
       value: 12
       unit: pt

Type = COMMAND
Subtype:  COMMAND_FONT_FACE
Content = face:arial
Parameter
       Type: string
       Value: arial

Type = COMPOUND_COMMAND_END
Subtype = COMMAND_FONT_FAMILY
```

As can be seen, a compound command starts with a single token to mark the beginning of the string of individual commands, and it completes with a matching end-of-compound-command. This marking of beginning and end is important, because compound commands sometimes need to be performed together as an atomic operation.

For example, consider the compound command for a URL, where, as in HTML, the URL is buried under some text such as “click here”. The actual URL and the URL text need to be handled in the same processing cycle so that they can be implemented correctly. It would not work to first have a single command for a URL. Then later a command for the URL text. HTML, for one, needs both pieces of data. So, to provide multiple parameters in a single command, compound commands are processed as one atomic operation.

B6. COMMAND ALIAS AND SUBSTITUTION

Platypus uses command aliases to provide ease of use, where convenient. For example, the command `[font|size:12pt]` (sets font size to 12 points) Is a lot to type. So, Platypus offers this shortcut `[fsize:12pt]`. This shortcut is the substitute command discussed in section B5 and is processed via the CommandR class. As can be seen in line 2 in section B3, the command root and its replacement root are specified in the command file that is loaded into the command table.

By having single commands be abbreviations for other commands, it's easy to add new shorthand commands without worrying that they will disturb the processing. The aliases are simply added to the command file. Nothing else needs to be done.

Note: In the COMMAND tokens, the Content field provides the actual command as specified by the user in the text, without the enclosing [and] characters.

B6A. HOW COMMAND SUBSTITUTION IS PERFORMED IN THE PARSER

The substitution occurs in process() found in **CommandR.java** and is moderately complex. Here is the logic using `[fsize:12pt]` as the alias command and `[font|size:12pt]` as the real command.

The parser comes across the alias command in the input file. It extracts the root of the command, looks it up in the command table. There it finds, the entry that maps the alias to the real command:

```
[fsize:=r [font|size:
```

The r after the = sign tells the parser to replace the alias root with the text that follows after the space to the right of the r. When the CommandR class for this command is created, it extracts the root from this replacement text (in this case, it's the entire replacement text) using the method extractReplacementRoot() and stores the result in the `replacement` instance field.

When the parser finds the alias command in text, it looks it up, finds the corresponding CommandR class and calls the process() method, passing in the GDD, the context that gives the input line and the starting point for the parse, the token list to add the token to, and the in-code boolean. This method does the heavy lifting.

Process creates a new ad hoc parse context, containing the real root for the alias and adjusts the parsing location to point to the real root. It then looks up the command in the command table using the real root. It then calls the process() function on that command passing it the adjusted parsing context. When that call to the real command's process() function returns, the old parsing context is reinstated, with the parsing point adjusted, and the alias's process() exits.

The state of the parser upon successful operation is that the original parsing context is preserved, and the parse point is advanced just past the alias command and its associated parameters, if any.

B7. COMMAND TOKEN PARAMETERS

The Parameter field in tokens is a data structure that holds all possible variants of a parameter (see **CommandParameter.java**). Any given command can have 0 or 1 parameters. The Parameter consists of the following fields:

- amount
- unit
- errorCode
- charsParsed
- string

These indicate the amount, the units of measure, an error code, if any, the number of chars parsed by the parser for this parameter, and a string, for cases where the parameter is a string. In the event of an error, the unit will be `UnitType.ERR0R` (see **UnitType.java**) and the error code will appear in errorCode.

Note: the charsParsed field is likely to be removed shortly, once it becomes clear that there are no unexpected needs for it.

B8. FURTHER NOTES ON TOKENS

New types of tokens will need to be added in upcoming versions of Platypus:

- A token that identifies whether the token stream contains any forward references. If so, the output plugin will almost surely have to do two passes: one to resolve the page numbers (and/or figure numbers, etc.), the second to actually generate the document. This token will be placed at the beginning of the file.

- A token that marks the start and end of scripting language.

- A token marking end of input file. This token is needed only in one circumstance: when the input comes from stdin. Using stdin will allow users to pipe data to Platypus. So as not to wait for more data that might not be forthcoming, piped Platypus files will need to end with some EOF command, which will be implemented either as a command (which seems a bit unnatural) or as a new kind of status token.

INPUT PLUGINS

Input plugins have not been implemented at this point, but they are expected to have the same basic structure as the output plugins described in the next section.

Like output plugins, input plugins will be located using the configuration file.

OUTPUT PLUGINS

Output plugins are located by the configuration file, with additional specification optionally provided on the command line. They consist of a JAR file that must have a **Start.java** class in it. This Start class is the one called by Platypus, so it represents the main line of the plugin.

The Start class (see **Start.java**) implements the Pluggable interface (see **interfaces.Pluggable.java**), which simply specifies that there is one function entitled process(), which is handed a copy of the GDD and of the command line.

What is done in the output plugin is not monitored by Platypus, so it is up to plugins to handle all their own I/O (including closing the output file) as well as issue the appropriate error warnings and handle all plugin exceptions.

Platypus currently ships with two plugins that illustrate how plugins can work. They are the listing and PDF plugins. The former is a simple tutorial implementation, the latter a complex difficult plugin.

D1. LISTING PLUGIN

This plugin (see **plugin.listing.Start.java**), while needed by Platypus for publishing listings of Platypus files on the website, is primarily provided as a working example of a simple plugin. Tokens are read in and an HTML listing of the Platypus file is generated. This listing has color syntax highlighting and it identifies various Platypus elements in the file (commands, text, comments, etc.) if the cursor is held over them in the browser.

The implementation is unremarkable, consisting of an implementation of the Strategy pattern to account for the basic types of tokens, which are then written using colors to an output HTML file.

D2. PDF PLUGIN

The PDF plugin (see **plugin.pdf.Start.java** et seq.) produces a PDF file using the FOSS iText library (www.lowagie.com/itext).

It loads a command table (a HashMap) with all valid commands supported in PDF. The key is the command token type and the value is the routine that processes that command. (Likewise for symbols.) Since only valid Platypus commands are passed to the plugin, any commands not found in the lookup table during processing generate an info message, saying the command is not implemented for this output format. Under PDF, the number of these commands should be very small.

The plugin reads the tokens sequentially and processes them sequentially, with two important exceptions. The first, not yet implemented, will occur when Platypus can pass the plugin a token alerting it that the Platypus file contains forward references. This will require the plugin to do two passes through the tokens: first to resolve the forward references, second to generate the PDF file.

The other non-sequential process is the actual generation of the PDF file via iText. The mechanism for supporting columnar text in iText requires writing the text out in chunks and then rendering those chunks to the PDF file at various moments. This approach represents a deliberate design choice. It means relying on iText to provide most of the handling of the layout of text and graphics, as guided by the command in the Platypus file. The benefits of this are that it is easy to implement Platypus features quickly, because iText provides a rich selection of high-level methods. The downside is that the fine control over text placement is difficult and, at times, impossible to exert.

The alternative would be to use iText only for its primitives and then work at a much lower level of PDF construction. This approach is inherently much, much slower to implement, although the control over extremely precise placement is now possible.

The current approach was done in the name of expediency and because of the generally high quality of the iText documents. At some point, however, it's certainly possible that the PDF plugin will need to be rewritten to access that higher resolution placement.

D2.1 IMPLEMENTATION DETAILS OF THE PDF PLUGIN

D2.1.1 EOL/EOPARAGRAPH HANDLING

CR/LFs are converted into a single space, unless they are the first character of the input line (in other words a blank line). When they are the first character, this tells Platypus that the previous paragraph is ended. Platypus then outputs the paragraph, skips the number of lines specified by [paraskip:…], (default = 1), and indents the start of the next paragraph by the number of spaces specified by [indent:…] (default = 0). This blank line is the only way to indicate end of paragraph, except for the end of input file which forces an end of paragraph.

Sometimes, a user might want to have an EOL, without forcing a new paragraph, as in the case of a list (see example below). The command [] (opening and closing bracket) is used to indicate this:

Things to buy:[]
bananas[]
paper towels[]
apple juice[]

The[] simply serves as a CR with a single LF. An [] occurring on an empty line ends the previous line of text and forces a blank line (without doing the processing for a new paragraph).

Consequently, the following three lines are equivalent when paraskip=1 and paraindent=0:

When in the course of human events

When in the course of human events
[]

When in the course of human events[]
[]

Implementation details

This turns out to be very complex because of various implications. In the Platypus implementation, the end of an input line generates a [cr] token. If the end of line is the first (and therefore only) token in the line, Platypus generates a [CR] token, which means new paragraph. Users could enter these tokens, but they should not as there are numerous assumptions made when these tokens are encountered. (Eventually, logic will be added to prevent these tokens from being entered by a user). The one EOL token the user can enter is the [] command.

Let's examine the special cases:

- A line consisting only of an []. This line actually is converted into [][cr]. Since the [cr] becomes a space (except when EOL treatment = hard, when it becomes an EOL), this pair of tokens would introduce a new space at the start of the next line of text. So, in the PDF plugin parser, if a [] is followed by a [cr], the [cr] is not processed.

- A line consisting of only commands. So a line that consists only of [fsize:12pt] becomes [fsize:12pt][cr]. Again, we don't likely want a space inserted in this case. Such a line might seem rare, but in some cases it is desirable to string together a lengthy set of formatting commands. We would not then want to generate a spurious

space that would appear before the first text. So, when a [cr] is encountered, we check whether the previous tokens in the line are all commands. If they are, no space is generated. If any is text a space (or a hard EOL if EOL treatment is hard) is generated.

IMPLEMENTATION NOTES

One of the overarching goals of Platypus is to produce a software product characterized by good design and superior implementation. The specific set of goals for the implementation consists of (starting with the highest priority):

- Reliable code

To meet this goal, the following attributes are viewed as indispensable:

- Clear Code
- Testable Code

Consequently, we frequently write code to heighten its testability.

We also find the following precepts of good OO implementation to be very useful and so we used throughout:

- Dependency injection
- Primitive avoidance (the opposite of primitive obsession)
- Moderate to liberal use of comments
- Current, well-maintained documentation

These points are of course guiding principles that are secondary to the earlier goals and attributes of code.

Further documentation, as well as the code and binaries, can be found at the project home site at:
http://platypus.pz.org

INDEX OF JAVA CLASSES

Command.java 9

Command0.java 8

Commandable.java 9

CommandLineArgs.java 6

CommandParameter.java 11

CommandR.java 8

extractReplacementRoot() 11

process() 11

CommandS.java 8

CommandTable.java 5, 8

CommandV.java 8

GDD.java 5

Infile.java 7

InputLine.java 7

LineList.java 8

listing.Start.java 13

LogFormatter.java 5

pdf.Start.java 13

Platypus.java 4

calls PlatypusParser 9

PlatypusHomeDirectory.java 5

PlatypusParser.java 9

Pluggable.java 12

PropertyFile.java 5, 6

Start.java 12

SystemStrings.java 5

Token.java 9

TokenList.java 5

TokenType.java 7

UnitType.java 12