WRITING OUTPUT PLUGINS FOR PLATYPUS

v. 0.2.0
REV 1

This document is an adjunct to the information found in the Platypus Architecture Manual, entitled Anatomy of the Platypus, available here: http://platypus.pz.org/ArchPlatypus-v0.2.x.pdf It assumes familiarity with that document.

December 2009

CONTENTS

BASIC ARCHITECTURE

Platypus output plugins are the workhorses of the system. They convert a data structure containing document information and resources plus a linked list of text and command tokens into the final document.

The plugin (from here on, the word plugin will refer to an output plugin unless an input plugin is explicitly specified.) is a JAR file whose name and location are known to Platypus via the configuration file, **config.properties**. This file relies on a coding scheme explained in the Architecture Manual to uncover the name of the plugin as well as its location in the event it is not located in the default directory (PLATYPUS_HOME/plugin).

Once the plugin is located, it is loaded by a Platypus class loader. It is then passed the previously mentioned data and it receives control from Platypus. When the output plugin exits, control returns to Platypus, which performs any of its own clean up and exits. Platypus has no awareness whatever of the plugins operations and must not be counted on to perform any action. This brings us to the first requirement of plugin development:

⏩ **Requirement 1**: Plugins must handle all set-up, processing, shut-down and contingencies. These include:
- all exceptions and error conditions
- all file I/O including opening and closing the output file

Of these aspects, the question of exceptions is particularly important, as the stack dump caused by uncaught exceptions reflects Platypus's stack, not that of the plugin. As a result, it can be nearly impossible to diagnose exceptions that occur in the plugin.

Because the plugin is a black box to Platypus, no help from Platypus can be expected. All that Platypus can provide is the data structures passed to the plugin.

WHAT PLATYPUS DOES WHEN IT STARTS A PLUGIN

The initial class of a plugin is called **Start.class**. It implements the Pluggable interface, which is defined in **/interfaces/Pluggable.java**. Platypus first calls the Start() constructor with no arguments. This method can be used for initialization or other activities. When the constructor returns, Platypus then calls process(), at which point it transfers command to the plugin. The process() method is defined via the Pluggable interface as:

```
public void process( GDD gdd, final CommandLineArgs clArgs );
```

These data structures are examined in the next section.

All processing done by the output plugin must derive from the process() method. As can be seen in the PDF plugin, this processing can be very extensive and involve numerous other packages including third-party libraries. The techniques for using these libraries and how to call them are all demonstrated in the PDF plugin and discussed later in this document.

⏩ **Requirement 2**: Plugins must have a Start class that implements the Pluggable interface and contains a public, no-argument constructor (which is called first by Platypus) and the process() method required by the Pluggable interface.

When plugin processing has finished, the plugin needs only to return from the process() method. Note that process() is declared as void, so Platypus does not verify any return code or take any action deriving from events that might have occurred in plugin processing.

WHAT PLATYPUS PASSES TO A PLUGIN

Of the two data structures passed to a plugin's process method, the GDD is by far the most important. This section examines its contents. It then briefly examines the second argument, which contains the command line.

THE GLOBAL DOCUMENT DATA (GDD)

The GDD is incarnated in **GDD.java**. This structure contains environmental data, configuration data, all literals, a logger, the list of tokens, and some macro look-up tables. These are discussed in the order of their importance next.

• TokenList inputTokens (see **TokenList.java**): This is an ArrayList of all the tokens Platypus generated when parsing the input document. In the event this item is null, or empty (or GDD is null), an error message should be returned immediately and process() should return. The tokens consist of text, and optionally commands and comments. Note: a file consisting entirely of commands could conceivably contain text, as some commands (symbols and system macros represent text). The types of possible tokens are discussed in the Architecture Manual in several places. When processing of inputTokens is complete, the output should do its shutdown routine and return to Platypus.

• commandTable (see **CommandTable.java**) contains all the commands supported by Platypus. It is highly likely that a given plugin will not support all the tokens that a text file might include. However, it must be able to handle them all. Unimplemented commands should generate a warning using the logger in GDD. The command should then be ignored. This brings us to a new requirement:

⏩ **Guideline** All valid Platypus commands should be handled correctly by the plugin. Any commands that cannot be handled by the plugin should result in a warning to the user and the command should be ignored.

The commandTable is loaded with all the commands found in **Commands.properties** and the symbols found in **Symbols.list** (which is located in PLATYPUS_HOME/config). The format for Commands.properties is explained in the Architecture Manual and in the heading comments of the file itself.

• Literals lits (see **Literals.java**) contains key-value pairs for all the literals used in Platypus processing in a look up table (except for diagnostic logging messages and error messages that precede the loading of the literals file—all of which are strictly in English.) The contents of the Literals file are loaded from the Platypus.properties file in the PLATYPUS_HOME/config directory. The purpose of using literals that must be looked up is to provide for eventual translation of these literals into other languages. Note: the Literals file also contains an English-language description of every command. Commands can be looked up in the file via their root. Further note: If a plugin looks up a literal that is not found in the file, Literals.java returns a single space. This approach enables processing to proceed. Surely an invalid look up in the literals table is not a user error, so no user-facing error is generated. The idea behind this informs the next requirement.

⏩ **Guideline** When an error occurs, respond with a solution that provides (in order) the least disruption and the least surprise. A recoverable error should never stop the plugin from functioning. However, no part of this rule can be used to justify generating an invalid output file. All output files should be valid or else be erased.

• Environment entries:

- **GDD.homeDirectory** (a String) contains the contents of PLATYPUS_HOME

- Verbosity switches: clVerbose (named after "command line Verbose") and clVVberbose ("very verbose") determine the amount of data given to the user. The latter command can print extensive data, including a list of all the tokens passed to the plugin. Each token is accompanied by an explanation.

- sysStrings (see **SystemStrings.java**) contains macros (string substitutions) for various system parameters. These currently include the following self-explanatory items. Note that system strings are set by Platypus. Users cannot set them directly, but they can read them at any time.

`_commandLine`	→	the command line with which Platypus was called
`_format`		the output format (PDF, HTML, etc.)
`_version`		the version number of Platypus (such as: 0.2.0)

-userStrings(see **UserStrings.java**) contains a hash map of user-defined macros.

• logger (see **LogFormatter.java**, in part). This is the principal logger for all Platypus activity, including that of the plugin. Well-behaved plugins log an entry every time they reach a milestone. Which milestones to record depends on the plugin, but a good guideline is to log any item that represents a completed activity. The ultimate purpose of this logging is for user information when the –verbose switch is set. To see what the PDF plugin logs and how, run any PDF job with the –verbose switch. Note that in the output, the date/time is captured by the log formatter and need not be entered.

The GDD contains a few other minor fields that are of little use to most plugins.

COMMAND-LINE ARGUMENT CLASS

The second parameter passed to the plugin is the active instance of **CommandLineArgs.java**. Its use might be deprecated in future releases.

PLUGIN OPERATION

[This section details how plugins work in general and uses the PDF plugin as the basis of various explanations. The architecture presented here is known to work, but its use is not required. Other designs that achieve the same ends are both possible and desirable.]

As mentioned earlier, the plugin accepts a token stream and generates a final document. The process generally has several components: the token processor, the command executor, and the document generator. The latter two components key off of the input tokens, whether they are commands or text.

TOKEN PROCESSOR

The tokens are read in and a switch statement dispatches them to the appropriate handlers. In almost all cases, this process occurs in the **Start.java** class of the plugin.

The token stream has been designed to reduce the amount of look ahead that needs to occur for immediate processing. The notable exception is the case of forward references, where it is generally necessary for the plugin to emulate document generation to determine page numbers or names of forward references. To facilitate this process, document generators should have some sort of accurate simulation mode.

Because the token list (see **TokenList.java**) is a Java ArrayList, it is possible to inject tokens into the token stream at any point. This is how macro resolution is performed and surely will be the mechanism for handling scripting features when they are implemented. Plugins should feel free to inject tokens into the token list as required for their own needs. However, because the –vverbose options dumps the token list when the plugin returns, the tokens should use the current Platypus token formats.

COMMAND EXECUTOR

The command tokens in the token stream are known to be valid Platypus commands. However, not all valid Platypus commands can be handled by all output plugins. One way to resolve this is to create a command table for the plugin. When a command token is read from the token list, it is looked up in the plugin's command table. If it is found there, the appropriate processing takes place; if not, a warning is emitted via the logger in GDD.

This warning should use the logWarning() method in the GDD logger and it should provide detailed information about which command is not supported. An example of this is shown in errMsgUnrecognizedCommand() in Start.java of the PDF plugin.

Guideline When an error occurs, try to give the user sufficient information to enable correction. This should include: the file, the line, the erroneous command/item, and an error message. It is frequently useful to also state how the plugin handled the problem.

The command table in the plugin could be modeled on **PdfCommandTable.java** in the PDF plugin. The table is a hash map, in which the command root serves as the lookup key. The value consists of an object with the **OutputCommandable** interface—the command-processing class.

This interface defines the command implementations and requires a process() command that accepts an Output-Contextable object (discussed shortly), the token (which contains all parameter data, if the token has parameters), and the token number. The process() method is called and that is how the command is executed.

Note that, each command is loaded separately into the command table when the table is initialized. Symbols (which Platypus treats as commands) are also entered as commands. The symbols are read from a plugin-specific configuration file that explains how each symbol should be handled. (See **PdfSymbolsTable.java**)

WHAT CHANGES WHEN A COMMAND EXECUTES?

Document commands are of several kinds: those that emit text, those that force an action on the document, and those that change a setting that will be reflected later in the document's development. Most commands fall into the last two categories. Let's look at these in greater detail.

- Text-emitting commands (such as symbols and macro expansions) primarily determine what text needs to be output and then interact with the document generator to output the text correctly.

- Commands that force document actions generally invoke actions in the document generator. They often trigger new settings in a data structure that holds all the state data for the document. In the PDF plugin, this structure is the **PdfData.java** class. A typical command is `[pg]`, which forces a new page. In the PDF plugin, this forces the document generator to output any text that has been processed but not output. It then skips a line (or not) depending on the settings for end-of-paragraph. And it prepares to (optionally) indent the next paragraph if there is more text to follow.

- Commands that change a setting that will later be reflected in the document. Many commands create changes that do not take effect immediately, but instead are delayed in their execution until a future event occurs. For example, the [pagesize: command changes the size of the following page. Encountering this command, Platypus resets the page size in **PdfData**. Whenever the document generator triggers a new page, it will read the page size from **PdfData** and update the document's new page to the specified size. (Similar delayed activity occurs with changes of margins, fonts, paragraph formatting, etc.)

Every format has different constraints on what kind of implementation a given command requires. Being able to characterize the commands into these categories helps design the plugin efficiently.

THE OUTPUTCOMMANDABLE OBJECT

All commands rely on an **OutputCommandable** object, found in the command table, to handle their processing. This object has a process() method that accepts an output-context object, the command token with its associated parameters, and the token number in the input token list.

The output-context object is the document state container. In the PDF plugin, this is the PdfData object. It contains several important entries, besides the document-specific items (such as page size, margins, font info, etc.). These include a reference to the document generator (discussed in the next section) and a reference to the GDD that was passed to the plugin. This means that the output-context container can access any item the plugin needs to generate a correct document.

⏩**Guideline** The plugin should favor rely on only a single document-data container that can access all the fields and data items needed for document generation. The only other item it should need for proper execution of a command or output of text is are related to the token being processed.

In all cases in which a value is set as the result of a command, a check is first made to see whether the value is different from the current setting. If the new value is the same as the old value, the update is not performed. This is an important diagnostic tool. Currently, Platypus enables users to dump select internal variables to see their settings. These variables, stored in the output-context container hold the file and line number when the field was changed from its default. If the new value is the same as the old value, the absence of an update means that the file and line # data are not updated. As a result, the user sees in the dump of variables only the point at which the variable was changed to its current value.

⏩**Guideline** The output-context container should keep track of the file number and line number of all changes and present this data to the user when the [dump: command appears. These numbers should be updated only when the variables change. That is, a command that attempts to set an item to its existing value should not result in an update to the file and line numbers for that item.

DOCUMENT GENERATOR

The document generator is the class that handles all the generation of document data and writes it out to the document. In the PDF plugin, this class is **PdfOutfile.java**. In all but the most trivial plugins, this will be a large complex class.

It needs to handle certain key functions:

- Opening the output file. This should be done only when there is text to output, not before. (So called, lazy execution.) This prevents a file from being created only to discover that it contains no text or text-generating commands. Frequently, prior to the opening there will be a large amount of configuration data that is set (such as page size, margins, etc.) and held in the output-context container and then consulted at opening. For all such parameters, there should be intelligent defaults that are used and these defaults should be available to the user in the plugin documentation. (In the PDF plugin, these defaults are all specified in the **DefaultValues.java** class.)

- Output of text. Text output is surprisingly complicated. It requires specification of the text (which can include symbols) and the font to use. In addition, text-level features such as bold, italic, strikethrough, underline, etc. have to be maintained by the document generator, so that the text is formatted correctly.

- End of paragraph and sections. The end of paragraph triggers a sequence of activities that relate to spacing and how the next paragraph begins, if there is one. Likewise, end of sections (such as chapters, for example) can involve a complex sequence of events (such as skipping to a new page, creating a title, changing the number of columns, etc.)

- Page-related output. As discussed previously, pages have distinct sizes and shapes. The printing area is usually smaller than the full page and delimited by margins. Within the printable area, text can appear in columns, which are managed by the document generator. When a page is complete, some documents place a footer and/or header in the top or bottom margin, requiring the ability to write outside of the printing area.

- Closing the output file. Closing a file means flushing all unwritten text, ending the current paragraph, and ending the current section. The close routine should make sure the document is actually open; if it is not, the document generator can assume that no output occurred. This should result in an error or warning to the user.

⏩ **Requirement** Platypus must not allow the generation of a structurally invalid output file. If the document generator recognizes that a file will be structurally invalid, it should close the output file, erase it, and issue an error message. A structurally invalid file is one that is internally damaged so that it cannot be properly accessed by standard document readers or one that is readable but contains incomplete or invalid content.

The functions listed above are highly intertwined: a single command can result in text output, the end of a paragraph, a new column on a new page in a new section. Consequently, the document generator has many moving parts that need to be coordinated.

ADDITIONAL CAPABILITIES AND REQUIREMENTS

This section describes various capabilities or requirements that do not fit easily in the previous sections.

CONTROLLING THE PLATYPUS PARSER

Plugins can control some aspects of the Platypus parser's behavior. This is done via settings in the **Config.properties** configuration file in PLATYPUS_HOME/config. The key settings are show next. The xxx in the key field should be replaced by the format for the given plugin (for PDF, for example, xxx = pdf). The settings as shown here are the default values.

```
pi.out.xxx.platyparse=yes
pi.out.xxx.keep_comments=no
pi.out.xxx.passthrough_escape_char=no
pi.out.xxx.process_replaced_commands=no
```

The first (platyparse) determines whether Platypus should parse the input or whether it should be passed untouched line-by-line to the output plugin. In almost all cases, the setting is yes. (This setting might be removed in future versions, so if you use the 'no' option, please let the Platypus team know.) The default is yes.

The second setting (keep_comments) determines whether comments are stripped off or passed to the output plugin. This setting is "no" by default. However, the listing plugin, which prints comments (plus text and commands), sets this to "yes". Plugins that want to pass parameters to the plugin from the Platypus file in a way that is not recognized by the parser could do so using this option. Simply use specifically marked comments as means of passing items to the plugin and set this switch to "yes" so that the items make it to their destination.

The third setting (passthrough_escape_char) tells Platypus whether an escape character preceding a [or] should be passed through to the output plugin. This is useful only in plugins that need to know whether a [in text was escaped or not. The listing plugin sets this switch to "yes" but almost all other plugins will want to set it to "no".

The fourth setting (process_replaced_commands) determines whether the Platypus parser informs the plugin about command substitution. Commands such as `[fsize:` are short-hand aliases for, in this case, `[font|size:` This setting determines whether the output plugin is made aware of the original command alias or just the resolved command. Currently, only the listing plugin sets this value to "yes." Note: commands that are aliases for other commands can be identified in the **Commands.properties** configuration file, because they are all flagged with an `r` (for replace) as the first character in the value. (See the entry for `[fsize:` .)

THE [DUMP: COMMAND

Platypus provides a diagnostic command, `[dump:`, which enables a user to print out a group of associated variables. See the User Guide for a description of its various options. This command should be implemented by all output plugins, as it is part of the core contract with the Platypus user, namely, ease of use. If you have employed other typesetting software, you know how few good diagnostic tools are provided and how frustrating this can be.

⏩ **Requirement** The `[dump:` command must be implemented in some form in all plugins. All fields that a user might need should be dumpable using one or more of the options listed for this command.

INDEX(CLASSES MARKED IN BOLD)