VDMJ Plugin Writer's Guide

Author	Nick Battle
Date	30/11/23
Issue	0.1

0 Document Control

0.1 Table of Contents

0.2 References

[1] Wikipedia entry for The Vienna Development Method, http://en.wikipedia.org/wiki/Vienna_Development_Method

[2] Wikipedia entry for Specification Languages, http://en.wikipedia.org/wiki/Specification_language

[3] VDMJ, https://github.com/nickbattle/vdmj

[4] VDM VSCode extension, https://github.com/overturetool/vdm-vscode

[5] VDMJ Annotations Guide

0.3 Document History

Issue 0.1	30/11/23	First draft.

0.4 Copyright

1 Overview

This document describes how to write *Analysis Plugins* for VDMJ[3]. The architecture of VDMJ plugins is very similar to that used in the LSP/DAP server [4].

Section 1 gives an overview of the architecture into which plugins fit. Section 2 walks through various common scenarios to describe the interaction of plugins with the main lifecycle. Section 3 gives detailed information about how to implement plugins. Section 4 gives some examples of what would be possible with plugins and how to achieve it.

1.1 VDMJ

VDMJ provides basic tool support for the VDM-SL, VDM++ and VDM-RT specification languages, written in Java [1][2][3]. It includes a parser, a type checker, an interpreter and debugger with coverage recording, a proof obligation generator, user definable annotations and a combinatorial test generator, as well as *JUnit* support for automatic testing.

1.2 VDMJ Plugin Architecture

Services are supported in VDMJ via a number of *Analysis Plugins*, which are responsible for all processing that relates to a particular *analysis*. An analysis is an independent aspect of the processing of a VDM specification. For example, the fundamental analyses cover parsing, type checking, interpretation and PO generation. But an analysis could also be a translation from VDM to another language, or a more advanced kind of type checking or testing.

Plugins for the fundamental analyses are built into VDMJ, but additional plugins can be written independently and added easily. This document describes how to write such plugins.

The basic architecture of the VDMJ plugin system is shown in Figure 1. A terminal console starts a Java main method, which uses lifecycle methods to interact with the plugins. The lifecycle constructs all the plugins at startup, some of which are built-in and some of which are user configured.

All plugins register with a *PluginRegistry* and optionally with an *EventHub*. The registry allows the lifecycle or plugins to make a request from all plugins – for example, the *getCommand* call illustrated in the figure allows a plugin to respond to commands typed in the console. The *EventHub* allows the lifecycle to inform plugins about lifecycle activity, where individual plugins choose to subscribe to particular event types. All plugins can raise messages for display on the console by responding to the lifecycle methods.

Plugins make use of VDMJ language services to actually do their processing (parsing, type checking etc). The lifecycle methods do not directly make calls to VDMJ, and generally have no idea what the plugins actually do.

So in effect, plugins “contribute” functionality to VDMJ. For example, when you see commands listed on the “help” output, all of those commands have been contributed from the plugins via the *getCommandHelp* method. Similarly, warning and error messages produced when checking the specification have all come from plugins.

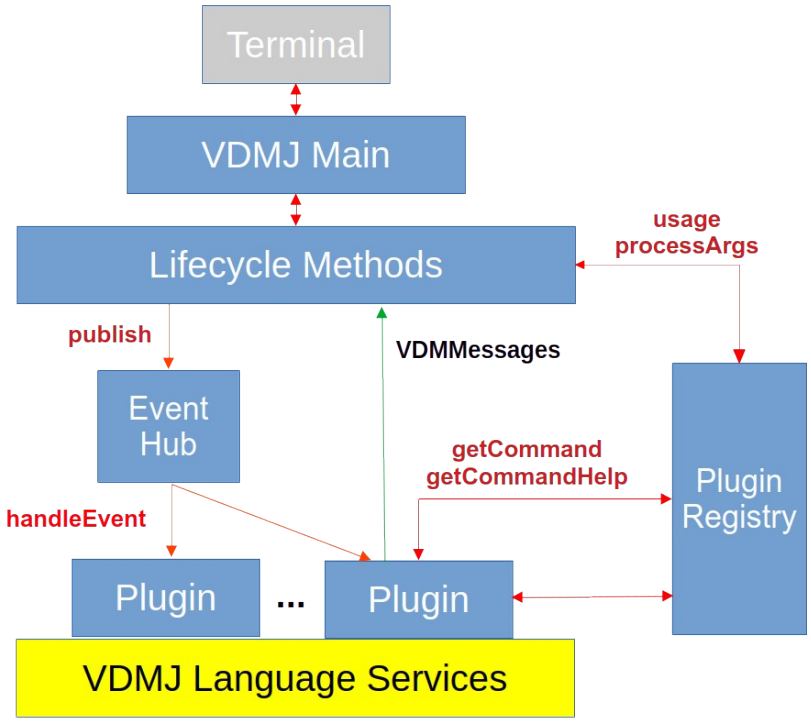

Figure 1: The VDMJ Plugin Architecture

2 VDMJ/Plugin Interactions

This section shows the sequence of lifecycle/plugin interactions for a number of common scenarios, in the hope that this will clarify how plugins provide their services. A more detailed description of how to write VDMJ plugins in Java is given in the next section.

2.1 VDMJ Lifecycle

VDMJ is initialized when the Java *main* method is started. The lifecycle methods then perform the following actions:

1. The VDMJ properties are initialized, using *vdmj.properties* if present in the classpath, else using -D options passed to Java.
2. The dialect of the project is determined, by looking for the *-vdmsl, -vdmpp* or *-vdmrt* command line options. This sets the global *Settings.dialect*.
3. The plugins are loaded. The built-in plugins come first: AST, TC, IN and PO. Then the classpath is searched for resource files called *vdmj.plugins*, which contain the class names of extra user plugins to include. Each loaded plugin is registered with the *PluginRegistry* and registers any events it wants to handle with the *EventHub*.
4. The Java command line switches are processed by calling each plugin's *processArgs* method via the *PluginRegistry*. Each plugin edits the *argv* array passed in, removing the options that it supports.
5. The remaining *argv* options after step 4 should only be filenames or directory names. These are searched to expand into the complete list of filenames to use. In the case of library files, these might be extracted from a library jar (eg. *stdlib.jar*).
6. The lifecycle uses the *EventHub* to publish events to parse, type check and initialize the specification. The events are sent to the plugins that register for each event type. If the handling of any event produces error responses from a plugin, the lifecycle stops. See below.
7. If the lifecycle successfully parses, checks and initializes the specification, the *StartConsoleEvent* is published to all plugins. The *INPlugin's* handler for this event interacts with the user. Other plugins can detect this event, but should not attempt to block the lifecycle.
8. The final response to the *StartConsoleEvent* indicates whether the lifecycle should reload the specification (eg. caused by the *reload* command, which goes back to step 5) or simply exit (eg. caused by the *quit* command).

2.2 Parsing, Checking and Initialization

At startup, the specification must be parsed, type checked and initialized, sending any warnings or errors to the console. The process is event driven, as follows:

1. The lifecycle calls c*heckAndInitFiles*, which handles the process.
2. A *CheckPrepareEvent* is published, which informs plugins that a parse/check is about to happen. They can use this to clear their state, in anticipation of more events. If any plugin returns messages, these are displayed on the console. If any are errors, a *CheckFailedEvent* is published and the lifecycle completes.
3. A *CheckSyntaxEvent* is published and the responses collected. Plugins are expected to parse the specification. If there are errors in the responses, the check is abandoned at this point: a *CheckFailedEvent* is published and the collected messages returned to the console. If there are no messages or only warnings, processing continues.
4. A *CheckTypeEvent* is published and the responses collected. Plugins are expected to type check the specification. If there are errors in the responses, the check is abandoned at this point: a *CheckFailedEvent* is published and the collected messages returned to the console. If

there are no messages or only warnings, processing continues.

5. A *CheckCompleteEvent* is published and the responses collected. Plugins are expected to initialize any runtime components that they have (e.g. the interpreter resets itself). If there are errors in the responses, the check is abandoned at this point: a *CheckFailedEvent* is published and the collected messages returned to the console.

Note that events are processed by plugins in priority order, with the built-in plugins coming first by default. So user-defined plugins can assume that (for example) the specification has been parsed by the AST plugin before it handles the *CheckSyntaxEvent*. The ordering of plugins is defined in section 3.2 .

2.3 An Interpreter Session is Started

Execution and debugging of a specification happens if the command line options indicate this is required (for example, *-i* is used). When an executable session is opened, the following occurs:

1. A *StartConsoleEvent* is published via the *EventHub*.
2. The *INPlugin* responds to this event and calls its *interpreterRun* method.
3. If the session is determined to be *interactive*, which the plugin previously determined by examination of the command line options passed (eg. *-i* or *-simulation*, but not *-e* or *-remote*), a *CommandReader* is started.
4. If an expression was passed via *-e* this is evaluated via the interpreter and the String value returned to the console.
5. If a *-remote* class was specified, this is instantiated and control is passed to the remote class.
6. If a *CommandReader* was started, this reads command lines from the console and passes them to the *PluginRegistry's getCommand* method. This forwards the request to each plugin's *getCommand* method, and any plugin that recognises the line typed can return an instance of an *AnalysisCommand* whose *run* method will do whatever is required.
7. If execution reaches a breakpoint, or a runtime exception occurs, control is passed to the *ConsoleDebugReader* via the *ConsoleDebugLink*. This allows the user to look at the environment and to step through the specification. The commands typed while debugging are *not* defined by the plugins.

Note that starting an executable session is unlike other lifecycle events in that one plugin in particular has to "take control". By default, this is the *INPlugin*. Other plugins can take control instead by registering themselves as the *INPlugin* and delegating calls to the real one, though this area should be more flexible. Similarly, debugging commands are unlike regular commands in that these are all handled implicitly rather than coming from a *getCommand* of a plugin. This should be improved to allow plugins to add their own special debugging commands.

2.4 Unrecognised Commands

If the user types an unknown command, or more likely mistypes a known one, the *getCommand* process will not find any applicable *AnalysisCommands*. In this case, a special command called an *ErrorCommand* is returned from *getCommand*. The run method of this command prints, "Unknown command <whatever>, try 'help'" to the console.

2.5 Help and Usage

Since the lifecycle does not know what the command line options or plugin commands actually do, it cannot provide help or usage information about them. This is therefore provided by the plugins.

When processing Java command line options via *processArgs*, any options remaining after all plugins have removed the ones they recognise must be invalid. But the lifecycle does not know what the valid options are, so in this case the *PluginRegistry* is used to call the *usage()* method of every plugin, which should print the list of command line options that it supports on the console.

Similarly, within a console session, if a *getCommand* fails, the user is prompted to try the *help* command, which uses the *PluginRegistry* to call the *getCommandHelp* method of each plugin. This method should print a list of commands supported by the plugin to the console.

3 Analysis Plugin Implementation

This section describes how to implement a new plugin for VDMJ. The description is based around an example *vdmjplugin* that is provided with VDMJ.

3.1 The Build Environment

VDMJ is written in Java, so it expects plugins to be written in Java also. You can choose the version of Java that you use, but it must be at least version 8 as VDMJ itself requires this.

The example plugin provided with VDMJ is built using Maven, which defines the dependencies required. But you may prefer to write your plugin using a different dependency management system.

The VDMJ source is available here [3]. The example plugin is in *examples/vdmjplugin*. The entire suite can be compiled with the Maven command “mvn clean install”.

3.2 Plugin Configuration

All analysis plugins extend the abstract Java class *com.fujitsu.vdmj.plugins.AnalysisPlugin*.

To configure VDMJ with the user supplied plugins to be used, a resource file called *vdmj.plugins* must be added to each plugin jar, containing the class name(s) of the plugin(s) in that jar. Alternatively, the *vdmj.plugins* Java property may be set, containing a CSV list of all the plugin class names to use globally. This overrides the resource files.

The order of plugin *initialization*, which may be significant (see below), is as follows:

1. AST plugin (parser)
2. TC plugin (type checker)
3. IN plugin (interpreter)
4. PO plugin (proof obligations)
5. User supplied plugins in order of the *vdmj.plugins* files discovered on the classpath, or provided via the property.

To include the example plugin, the resource file would contain the following line. See the example in *src/main/resources/vdmj.plugins*:

```
examples.vdmjplugin.ExamplePlugin
```

Naturally, as well as being listed in *vdmj.plugins*, plugin classes must also be available on the classpath. Typically, plugins are compiled into separate jars which are added to the classpath, for example (again, using the java command line).

```
-cp <etc>:examples/vdmjplugin/target/vdmjplugin-4.5.0.jar
```

3.3 Plugin Class Construction

When the VDMJ lifecycle starts, a single instance of each plugin is constructed. The construction of plugins looks for a *factory* method in each plugin class with this signature:

```
public static AnalysisPlugin factory(Dialect dialect)
```

If that method exists, it is passed the *dialect* of the specification, which allows the plugin to create a different variant for each dialect – remember VDM-SL specifications include modules, and other dialects include classes, so there are significant differences. Dialect subclasses can be more efficient than testing the specification dialect repeatedly.

The example plugin implements the factory method, returning one of three subclasses:

```
public static ExamplePlugin factory(Dialect dialect)
{
	switch (dialect)
	{
		case VDM_SL:
			return new ExamplePluginSL();

		case VDM_PP:
			return new ExamplePluginPP();

		case VDM_RT:
			return new ExamplePluginRT();

		default:
			throw new Exception("Unknown dialect: " + dialect);
	}
}
```

3.4 The Plugin Registry

Having constructed a plugin, the *registerPlugin* method of the *PluginRegistry* is called by the lifecycle to register it. The method does the following:

```
public void registerPlugin(AnalysisPlugin plugin)
{
	plugins.put(plugin.getName(), plugin);
	plugin.init();
}
```

Note that this requires two methods to be implemented by every plugin: *getName* and *init*.

The *String* name of your plugin can be used by other plugins to obtain services and data that you may provide. By convention, it is a short string that is also reflected in the class names of the plugin, but this is not a requirement. For example, the name of the parser plugin is “AST”, the type checker plugin is “TC” and the interpreter is “IN”. Plugin names must be unique in a running system, though note that a user plugin could replace a built-in plugin with the same name.

The *init* method, as the name suggests, should initialize your plugin. Typically, this will involve registering for various *events* with the *EventHub* (see 3.5) and setting local fields.

After registration, your initialized plugin will be invoked when various things happen in the lifecycle, either by events that it registered to receive, or by methods being called as discussed below.

Plugin instances can subsequently be found by calling the *getPlugin* method of the registry.

3.5 Event Handling

One way for the lifecycle to communicate with plugins is via *Events*. These are published by the lifecycle when various events occur; plugins subscribe to particular events of interest. An *EventHub* manages the plugin subscriptions and distributes *Events*.

If they register to receive events, plugins must implement the *EventListener* interface. This defines a method called *handleEvent*, which is passed an *Event*. Events indicate that something has happened in the lifecycle and carry information about that event.

For example, the *ASTPlugin* does the following in its *init* method:


```
public abstract class ASTPlugin
	extends AnalysisPlugin implements EventListener
{
	@Override
	public void init()
	{
		files = new Vector<File>();

		eventhub.register(CheckPrepareEvent.class, this);
		eventhub.register(CheckSyntaxEvent.class, this);
	}
```

The *eventhub* field is available to all plugins and is the same as *EventHub.getInstance()*. The register method is passed an *Event* subclass to subscribe to, and an *EventListener* to handle those events. The simplest design is for the *EventListener* to be implemented by the plugin itself, but you can use a separate listener object if you wish.

The *EventListener* defines a *getPriority()* method, which can optionally be used to order the dispatch of events to listeners. By default, the built-in plugins have highest priority and come first, with user plugins being passed events afterwards, but this can be altered by defining this method. The default priority of built-in plugins can be set via *vdmj.plugin.priority.<name>* properties. The “plugins” command lists plugins in priority order.

The *EventListener* interface defines the *handleEvent* method. Having registered for an *Event*, the plugin’s *handleEvent* method is called whenever the event occurs. So for example, *ASTPlugin* has:

```
@Override
public List<VDMMessage> handleEvent(AnalysisEvent event) throws Exception
{
	if (event instanceof CheckPrepareEvent)
	{
		CheckPrepareEvent pevent = (CheckPrepareEvent)event;
		files = pevent.getFiles();
		return syntaxPrepare();
	}
	else if (event instanceof CheckSyntaxEvent)
	{
		return syntaxCheck();
	}
	else
	{
		// Should never happen!
		throw new Exception("Unhandled event: " + event);
	}
}
```

Notice that *handleEvent* returns a *List<VDMMessage>*, which is a list of errors or warnings to display on the console, if any (you can return null).

The *EventHub* calls each of the registered plugins, in the order of their priority, on the same thread. This means that if it takes a long time to process an event, you will be holding up the rest of the lifecycle. In that case, you should consider using a background thread to perform the work, but that is beyond the scope of this document.

All *Event* objects contain a *properties* map (of String to Object), where arbitrary data can be saved. The same event object is passed to all plugins that handle that type, so this allows plugins later in the chain of processing to read properties that were set by earlier ones.

The example plugin handles *CheckPrepareEvents* and *CheckSyntaxEvents*. The first is actually ignored, though it could initialize the plugin in some way; the second generates an example message, which can be an error or a warning, depending on the upper/lower case form of the names of the functions in the specification (so this is like a style checking plugin, though the rules are not particularly sensible!).

A full list of Event types and when they are raised is given in Appendix A.

3.6 Plugin Commands

Plugins can contribute commands that can be used when an executable session is open. To do this, plugins implement this method:

```
public AnalysisCommand getCommand(String line)
```

This is called whenever the user types a line in the console. It is responsible for parsing the line and, if recognised, returning an *AnalysisCommand* subclass that implements the command via its *run* method. If the line is not recognised by the plugin, a null should be returned. If the line is recognised, but malformed, a Java *IllegalArgumentException* can be thrown, or an *ErrorCommand* object can be returned.

If multiple plugins recognise the same line, the last (in priority order, see 3.2) is used. Note that this means a user plugin can replace a standard command from one of the earlier built-in plugins.

If plugins contribute commands, they should also implement *getCommandHelp()*, which should return a *HelpList* of help lines for the commands supported.

3.7 Annotations

Annotations were introduced to VDMJ in version 4.3.0. Their use is described in [5].

New plugins may want to use this feature, either to perform extra plugin-specific processing for existing annotations or to add new annotation functionality linked to the plugin.

To add plugin-specific functionality to an existing plugin (like @*Override*), you typically use the *ClassMapper* to map the annotation from the source tree to your own tree – for example, you may map *TCOverrideAnnotation* to your own *XYZOverrideAnnotation,* and the configuration for this would be added to a tc-xyz.mapping file (see [5]). The mapped annotation would then automatically be present in the XYZ function and operation nodes of a VDM++ specification, and your processing of those elements should call your annotation (typically) before the node is processed and after it is processed.

You can also add an entirely new annotation via your plugin. In this case, you must define all of the plugin mappings for your annotation, from AST to your own plugin. So for example, if your XYZ plugin is based on the TC plugin and adds @Some, you would define *ASTSomeAnnotation*, *TCSomeAnnotation* and *XYZSomeAnnotation* classes and mappings.

If you do not use the *ClassMapper*, perhaps using visitors instead, your annotation will be invoked during earlier phases (eg. AST and TC) if it has mappings. You can then invoke (say) the TC annotation from your own analysis, but using a different method (ie. not *tcBefore* or *tcAfter*).

This is a complex area and should be regarded as experimental. Please contact the author, if you have complex requirements.

4 Example Plugin Functionality

Hopefully, it is clear how a simple plugin could work, responding to lifecycle events or providing *AnalysisCommands* for the console that add functionality. But this section contains a few examples in more detail, to illustrate what is possible using plugins.

4.1 Extra Type Checking

The “TC” plugin provides type checking for specifications. But it would be possible to add more sophisticated checks via a plugin:

- The plugin has a new name, perhaps “TCX”
- It is likely to want to register for the same events as the “TC” plugin in its *init* method. So it is invoked at each stage of the checking process. The *CheckSyntaxEvent* could be ignored, assuming it is not offering new syntax errors. Most of the work would be done via the *CheckTypeEvent,* or perhaps the *CheckCompleteEvent*.
- The “TC” plugin will have created a TC tree, which can be obtained from the TC plugin, via the *PluginRegistry*. The *getTC* method of that plugin will return a *TCModuleList* or a *TCClassList* for the specification. Note that this means you might want SL and PP variants of your plugin, with a factory method (see 3.3).
- The new plugin can process the TC tree and return any extra errors or warnings via the *handleEvent* response.

4.2 Additional or Replacement Commands

A plugin can contribute executable commands by implementing the *getCommand* and *getCommandHelp* methods. These will then appear in addition to the standard commands in the console when you enter “help”.

- *getCommand* is passed the line as typed by the user. You are responsible for parsing it and deciding whether you recognise the command.
- The *AnalysisCommand* can write to the console (stdout or stderr) via its return value or via the *PluginConsole* helper class. See the *ExampleCommand* class in the example plugin. Alternatively, VDMJ’s *Console* class can be used as normal.
- Remember to implement *getCommandHelp()* to describe your command as well.
- If you give a command the same name as an existing command, yours will replace the earlier one. This effectively means you can disable a built-in command via a plugin, or you can augment one by calling the original from your own code – to get the original, call the *getCommand* of the plugin that provides it.
- *AnalysisCommand* objects returned are stateful, so it is best to construct a new instance for every call of *getCommand*, rather than caching objects for reuse.

Note that when you are stopped at a breakpoint, the console is very restricted in what it can execute, and user-plugin commands are not available

4.3 Adding New Analysis Classes

The TC, IN and PO plugins are special, in that they define their own packages of Java classes that are specialized to the analysis they perform. So for example, when the TC plugin is processing a *CheckTypeEvent*, it takes the parsed tree from the AST plugin and then converts it into a tree of TC specialized classes using VDMJ’s *ClassMapper*, before actually performing the type check using the TC tree.

This technique is only really needed for very sophisticated analyses. Many simpler analyses can be performed on the AST using VDMJ's visitor framework.

But if this is being considered, it is likely that the plugin will be reacting to the four *Check*Event* events that are sent when a parse/check is performed. If your analysis needs a type clean tree, it will probably react to the *CheckCompleteEvent*. Or if your analysis contributes its own extra type checking, it will probably react to the *CheckTypeEvent*. The *CheckPrepareEvent* can be used to reset your converted tree before the new tree is available.

The TC, IN and PO plugins are good examples to follow.

4.4 Adding a Style or Spell Checker

Plugins can raise their own messages via the return value to *handleEvent*, so a plugin could process the AST (perhaps from the *ASTPlugin* or the *TCPlugin*) and raise messages relating to spelling or style, as the example does (crudely!). It is likely to do this via the *CheckPrepareEvent* or perhaps the *CheckCompleteEvent* if type information is needed. Note that style and spelling messages are likely to be warnings rather than errors, since the latter would prevent further processing of the specification.

5 Future Work

VDMJ is constantly being worked on, though we try to keep versions stable! The following are areas that need more work, or are in development (as of 4.5.0-SNAPSHOT):

- Better debug command integration via events or *getCommands*.
- Provision of a *MessageHub* to allow processing of other plugins' messages.

A. Events

The following events are available via the *EventHub*.

When an event is raised, a plugin that is registered for the event will receive a call to its *handleEvent* method. The return value of this call consists of *VDMMessages* for display on the console.

Event Class	When?
CheckPrepareEvent	Raised before the parse/type check process. It is used by all plugins involved in the parse/check process, which reset themselves in anticipation of the check process.
CheckSyntaxEvent	Raised to perform the syntax checking phase of a parse/check. The response messages will be displayed. It is primarily used by the AST plugin.
CheckTypeEvent	Raised to perform the type check phase of parse/check. The response messages will be displayed. It is primarily used by the TC plugin.
CheckCompleteEvent	Raised at the end of the parse/type checking process, when the specification is found to have no errors (warnings may be present). It is used by the IN, CT and PO plugins to reset themselves to the new checked TC tree.
CheckFailedEvent	Raised once during the parse/check process if the specification is found to have at least one error. It is not raised for warnings. The return value is displayed on the console.
StartConsoleEvent	Raised when the check phase is successful. This should cause the controlling plugin to start an interactive console, if needed.
ShutdownEvent	Raised at the end of the lifecycle. The return value is ignored.