Plugin Writer's Guide

Author	Nick Battle
Date	03/10/22
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] https://microsoft.github.io/language-server-protocol/overviews/lsp/overview/ and https://microsoft.github.io/debug-adapter-protocol/overview

[5] Visual Studio Code, https://github.com/microsoft/vscode

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

0.3 Document History

Issue 0.1	02/10/22	First draft.

0.4 Copyright

1 Overview

This document describes how to write *Analysis Plugins* for the VDMJ LSP language server.

Section 1 gives an overview of the architecture into which plugins fit. Section 2 gives detailed information about how plugins interact with the language server. Section 3 walks through various common scenarios to describe the interaction of plugins with the server. 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 Interfaces

VDMJ offers language services independently of the means used to access those services. That means that VDMJ can be used from several different IDE environments.

The basic user interface is built in to VDMJ and offers a simple command line. But VDMJ can also be used via the LSP/DAP protocols [4] and so can be used by an IDE like Visual Studio Code [5][6]. To support this, VDMJ includes a “language server” that responds to LSP/DAP connections.

1.3 LSP Server Plugin Architecture

Services are added to the language server 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 and interpretation. 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 the language server, but additional plugins can be written independently and added to the language server environment easily. This document describes how to write such plugins.

2 Analysis Plugin Interfaces

This section describes how to build a new plugin for the VDMJ language server. The description is based around an example plugin that is provided with VDMJ.

2.1 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 the language server 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/lspplugin*. The entire suite can be compiled with the Maven command “mvn clean install”.

2.2 Plugin Configuration

All analysis plugins extend the abstract Java class *workspace.plugins.AnalysisPlugin*.

To configure the language server with the user supplied plugins to be used, the Java property *lspx.plugins* must be set to a comma-separated list of the class names of the plugins. The fundamental plugins are configured automatically, and *ahead* of all user supplied plugins.

The order of plugins, 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. CT plugin (combinatorial testing)
6. User supplied plugins in order of the *lspx.plugins* property.

To include just the example plugin, the property would be set as follows (using -D for the java command line):

```
-Dlspx.plugins=examples.ExamplePlugin
```

Naturally, as well as being listed in *lspx.plugins*, the 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/lspplugin/target/lspplugin-4.5.0-SNAPSHOT-221001.jar
```

The way of setting *lspx.plugins* and the classpath differs between IDEs. In VDM VSCode, they are set via the “Settings” page for the extension (which edits the .vscode/settings.json file).

2.3 Plugin Class Construction

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

```
public static <plugin class> factory(Dialect dialect)
```

If that method exists, it is passed the *dialect* of the language server, 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 important differences. Dialect subclasses be more efficient than testing the server dialect repeatedly. If this factory method does not exist, the default class constructor is used.

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

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

		case VDM_PP:
		case VDM_RT:
			return new ExamplePluginPR();

		default:
			Diag.error("Unsupported dialect " + dialect);
			throw new IllegalArgumentException(...);
	}
}
```

2.4 The Plugin Registry

Having constructed a plugin, the *registerPlugin* method of the *PluginRegistry* is called to register the plugin. It does the following:

```
public void registerPlugin(AnalysisPlugin plugin)
{
	plugins.put(plugin.getName(), plugin);
	plugin.init();
	Diag.config("Registered analysis plugin: %s", plugin.getName());
}
```

Note that this requires two methods to be implemented: *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 server, 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 2.5) and setting local fields.

After registration, your initialized plugin will be invoked when various things happen in the language server, 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.

2.5 Event Handling

One way for the language server to communicate with plugins is via *Events*. These are published by the language server when various protocol 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 two overloaded methods called *handleEvent*, which are passed either an *LSPEvent* or a *DAPEvent*. Events indicate that something has happened in the language server 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()
      {
            eventhub.register(InitializedEvent.class, this);
            eventhub.register(ChangeFileEvent.class, this);
            eventhub.register(CheckPrepareEvent.class, this);
            eventhub.register(CheckSyntaxEvent.class, this);
            this.dirty = false;
      }
```

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 if you wish. The listener interface defines two *handleEvent* methods.

Having registered for an *Event*, the appropriate *handleEvent* method is called whenever the event occurs. So for example, *ASTPlugin* has:

```
@Override
public RPCMessageList handleEvent(LSPEvent event) throws Exception
{
      if (event instanceof InitializedEvent)
      {
            return lspDynamicRegistrations();
      }
      else if (event instanceof ChangeFileEvent)
      {
            return didChange((ChangeFileEvent) event);
      }
      else ...
}
```

Notice that *handleEvent* returns an *RPCMessageList*, which is a list of *JSONObjects* that represent the response to the Client, if any (you can return null). The responses from every plugin that is registered for an *Event* are collected and sent back to the Client, along with any standard responses from the language server itself.

The *EventHub* calls each of the registered plugins, in the order of their registration, on the same thread – the main LSP or DAP listening thread. This means that if it takes a long time to process an event, you will be holding up the rest of the language server. In that case, you should consider using a background thread to perform the work. The language server includes a *CancellableThread* class that may help, but that is beyond the scope of this document.

In the example above, the *InitializedEvent* is sent when the language server is exchanging initialize messages with the Client. By handling this event, the *ASTPlugin* can add some dynamic registrations for services that it requires but which cannot be set via the standard initialize response.

The example plugin registers itself to receive every possible event type, and prints out the name of the event when it handles one:

```
@Override
public RPCMessageList handleEvent(LSPEvent event) throws Exception
{
      System.out.println("ExamplePluginSL got " + event);
      return null;
}
```

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

2.6 Client Capabilities

The IDE Client send its LSP capabilities to the language server as part of the “initialize” request. These are cached in the server and can be accessed from plugins via the *LSPWorkspaceManager*:

```
LSPWorkspaceManager manager = LSPWorkspaceManager.getInstance();
```

There are two methods on the manager to look at Client capabilities:

1. *boolean hasClientCapability(String dotName)*. This method takes a dot-format capability name (like “textDocument.synchronization.dynamicRegistration”) and returns a boolean, which is true if the capability exists and is “true”, else false.
2. *<T> T getClientCapability(String dotName)*. This method takes a dot-format capability name and returns the actual value of that capability, or null if it does not exist.

2.7 Server Startup

2.7.1 Setting Server Capabilities

When the language server starts, it has various LSP protocol exchanges with the Client (the IDE). These inform the server of the Client’s capabilities and allow the server to inform the Client of its own capabilities.

Plugins may want to extend the capabilities of the language server, so they need to be able to contribute to this exchange. To enable this, plugins can implement the following method:

```
@Override
public void setServerCapabilities(JSONObject capabilities)
```

The *JSONObject* passed is the server capabilities response that has been built so far. You will see that it includes capabilities supported by the fundamental plugins, as well as any plugins that are earlier than you in the plugin configuration (see 2.2). This object can be amended by the plugin to set capabilities that it provides itself.

2.7.2 Initialization Events

Two events are sent during startup:

1. InitializeEvent. This occurs when the server has received the “initialize” LSP request. All plugins have been initialized, and the Client capabilities are available for query. All of the files in the project will have been discovered and cached – including the reading of external file formats. The responses to the event are sent to the Client along with (and after) the “initialize” response.
2. InitializedEvent. This occurs when the server has received the “initialized” LSP request. The loaded files will have been type checked and any errors or warning notifications will be ready to send to the Client. Any responses to the event are sent along with (and after) any type checking errors.

2.8 Other Plugin Processing

2.8.1 Code Lenses

When a file is opened in the IDE, it is possible that the file includes *code lenses*, which are annotations that appear in the Editor as clickable items to perform useful code related activities. Code lenses can be provided by plugins, and when the IDE opens a file, the following method is called on

all plugins (the abstract version in *AnalysisPlugin* does nothing):

```
public JSONArray getCodeLenses(File file)
```

The *JSONArray* returned contains any code lenses that the plugin wishes to create for the file name passed. The *CodeLens* class contains useful methods for helping to create these.

For example, *ASTPlugin* and *TCPlugin* implement this method and return lenses that offer the “Launch | Debug” lenses that appear above executable functions or operations. The *ExamplePlugin* implements this method and adds a “Config” lens for explicit function definitions only. If you click the Config lens, VSCode will open the *launch.json* file in an editor window.

Note that the command sent back to the Client with a lens is IDE specific, so lenses would normally test the Client type, with a test like *isClientType("vscode")*, before returning a lens.

2.8.2 Plugin Commands

Plugins can also contribute commands that can be used in the “Debug Console” window (in VSCode) when an executable session is open. To do this, plugins implement this method:

```
public Command 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 a *Command* 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 order, see 2.2) 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 returns a *HelpList*, which is a *List* of Strings. The first word on each help line should match the command name that it describes – e.g. "print <exp> - evaluate an expression". These are displayed when the user enters the “help” command.

3 Server/Plugin Interactions

This section shows the sequence of manager/plugin interactions for a number of common scenarios, in the hope that this will clarify how plugins provide their services.

4 Example Plugin Functionality

...

A. Events

...