Plugin Writer's Guide	
Author	Nick Battle
Date	02/10/22
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 (with arbitrary precision arithmetic), a debugger, a proof obligation generator and a combinatorial test generator with coverage recording, as well as *JUnit* support for automatic testing and user definable annotations.

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].

1.3 LSP Server Plugin Architecture

LSP and DAP protocol services are offered by VDMJ to allow tools which support these protocols to use VDMJ language and debugging services. The functionality is built into an “LSP Server”, which listens for socket connections for both LSP and DAP protocols, translating protocol requests to and from VDMJ language service calls.

Language services are added to the LSP Server via a number of *Analysis Plugins*, which are responsible for all communication 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 LSP Server, but additional plugins can be written independently and added to the LSP 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*.

2.2 Plugin Configuration

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

To inform the language server of the plugins to be used, the Java property *lspx.plugins* must be set to a comma-separated list of the class names of the plugins to use. The fundamental plugins are added automatically, and *ahead* of all user added plugins. The order of plugins may be significant (see below).

Naturally, as well as being listed in the property, the classes must also be available on the classpath. Typically, plugins are compiled into jars which are added.

The way of setting *lspx.plugins* and the classpath differs between IDEs. In VDM VSCode, they can be set via the settings for the extension.

2.3 Plugin Construction

When the language server starts, a single instance of each plugin in *lspx.plugins* will be constructed and registered with the *PluginRegistry*. The construction of plugins looks for a method in your class with this signature:

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

If that method exists, it is passed the *dialect* of the language server, which allows the plugin to create a slightly different variant for each. This might be more efficient than testing the server dialect repeatedly. If this factory method does not exist, the default class constructor is used.

The *PluginRegistry* does the following to register the plugin:

```
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 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”, and the type checker plugin is “TC”.

The *init* method, as the name suggests, should initialize your plugin. Typically, this will involve registering for various *events* with the *EventHub*. See below.

You could create a plugin that only implemented *getName* and *init*, but it would not take part in any language processing subsequently.

After registration, your initialized plugin will be invoked when various things happen in the language server, either by events being sent or methods being called, as discussed below.

2.4 Event Handling

Plugins can implement a method called *handleEvent*, which is passed either an *LSPEvent* or a *DAPEvent*. Events indicate that something has happened in the language server and the plugins register to be informed by calling the *register* method of the *EventHub* class.

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

```
@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* value is available to all plugins, and is the same as *EventHub.getInstance()*. The register method is passed an *Event* class, and an *EventListener,* though by convention the plugin implements the *EventListener* interface itself, which just defines the *handleEvent* methods.

Having registered for an *Event*, the *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 the *handleEvent* returns an *RPCMessageList*, which is a list of *JSONObjects* that represent the response to the Client, if any. 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.

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.

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

2.5 Server Startup Processing

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

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

```
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).

3 Server/Plugin Interactions

…

4 Example Plugins

A. Events

...