LSP Plugin Writer's Guide

Author	Nick Battle
Date	22/04/23
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] 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

[7] VDMJ Annotations Guide

0.3 Document History

Issue 0.1	22/04/23	First draft.

0.4 Copyright

1 Overview

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

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 server. 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. Lastly, Section 5 briefly describes how the language server is likely to change in future releases.

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.

A 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.

The basic architecture of the VDMJ language server is shown in Figure 1. An IDE “Client” connects to a JSON RPC server, which uses a *Workspace Manager* to handle RPC calls, one for LSP, one for DAP. The Workspace Manager constructs a number of 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 Workspace Managers to make a request from all plugins – for example, the *getCommand* call illustrated in the Figure allows plugins to respond to commands in the debug console. The *EventHub* allows the Workspace Managers to inform plugins about activity from the Client, where individual plugins choose to subscribe to particular event types. All plugins can raise messages for display on the Client by sending them to a *MessageHub*, which the Workspace Managers query.

Plugins make use of VDMJ language services to actually do their processing (parsing, type checking etc). The Workspace Managers should not directly make calls to VDMJ[1].

So in effect, plugins “contribute” functionality to the language server. For example, when you open a debug console and you see commands on the “help” output, all of those commands have been contributed from one or more plugins. Similarly, when you see code lenses appear in an editor window, these have also been contributed by one or more plugins.

[1] though at release 4.5.0 there are still some direct calls that have yet to be migrated to the appropriate plugin.

Plugins are also able to “react” to actions that the user performs or provokes. So for example, every time you edit a file, messages are sent to the language server which checks the syntax as you type. But it would also be possible for other plugins to react to those same events and (say) notice how long you have waited since typing and provoke a type check if you have stopped typing for more than a few seconds.

Plugins can also react to new protocol extensions that are not part of the standard LSP/DAP protocol. These messages will raise special events that a plugin can react to, sending back the new responses. In this way, plugins can be used to extend the functionality of the tool considerably.

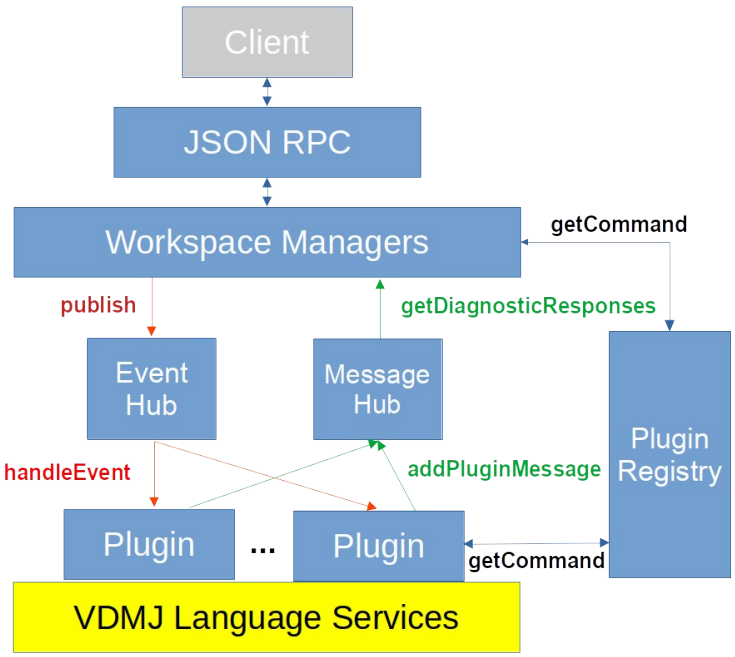

Figure 1: LSP Server Architecture

2 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. A more detailed description of how to write plugins in Java is given in the next section.

2.1 Server Initialization

The language server is typically initialized when the IDE opens a VDM project and perhaps opens the first file. The LSP protocol [4] defines a number of interactions at initialization, which causes the following:

1. The JSON RPC server starts and listens for connections.
2. The IDE sends the LSP “initialize” request, which is sent to the *LSPWorkspaceManager*, passing the Client’s capabilities and the root location of the project.
3. The Workspace Manager’s construction registers the built-in plugins with the *PluginRegistry.* This calls their *init* methods, which may register the plugins to receive various event types with the *EventHub*.
4. The Client capabilities and project root are stored in the Workspace Manager.
5. The *LSPXWorkspaceManager* is called to register user-defined plugins with the *PluginRegistry*, and they may also register with the *EventHub*.
6. The project files are (recursively) loaded into Workspace Manager memory. This includes the generation of any external format files, and respects the *vdmignore* list or *ordering* files. Each loaded file is also added to the *MessageHub* to allow diagnostic reporting.
7. The server capabilities for the built-in plugins is prepared and then the *setServerCapabilities* method of each plugin is called, via the *PluginRegistry*, to allow them to add or change settings.
8. An *InitializeEvent* is published, and any responses collected.
9. The final server capabilities and any responses to the event are returned to the IDE.
10. The IDE should respond with an “initialized” LSP message, which is dispatched to the Workspace Manager.
11. The memory-loaded project files are parsed and type checked. See 2.2 . This may include adding error and warning messages to the *MessageHub*.
12. An *InitializedEvent* is published, and any responses collected. This can be used by plugins to make dynamic registrations for LSP features that they offer.
13. Any warning or error messages from the parse/check (added to the *MessageHub*), along with any responses to the event are returned to the IDE.
14. The system is then quiet until the user does something in the IDE.

2.2 Parsing and Checking

At several points while interacting with the IDE, the currently loaded specification must be re-parsed and type checked, sending any warnings or errors to the IDE as notifications. The process is event driven, as follows:

1. Some other processing calls the *checkLoadedFiles* method of the *LSPWorkspaceManager*.
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. Typically they will call the *MessageHub clearPluginMessages* method to prepare for rebuilding their own diagnostic messages. If any plugin raises errors, a *CheckFailedEvent* is published and the

collected messages returned to the IDE via the *MessageHub*.

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 IDE via the *MessageHub*. 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 IDE via the *MessageHub*. 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 IDE via the *MessageHub*.
6. Any messages collected so far by the *MessageHub* are returned to the IDE as notifications.
7. A "slsp/checked" notification is also sent to the IDE.

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 discussed more in section 3.2 .

2.3 Opening a File

It may be that the server is initialized on the opening of the first source file in the IDE, in which case the processing above occurs as beforehand, but the following events occur when an existing source file is opened in the IDE:

1. A “textDocument/didOpen” LSP notification is sent to the language server, which is handled by the *LSPWorkspaceManager*. Notifications do not expect responses.
2. The file is checked against any *vdmignore* list.
3. The Workspace Manager notes that the file is open and whether it is a new file (whether it was loaded at startup, or a newly created file).
4. An *OpenFileEvent* is published via the *EventHub.* The responses are ignored, because the open file LSP message is a notification.
5. If the file is new, the project is re-parsed and checked, and any warnings or errors sent to the Client. See 2.2 .
6. The built-in plugins will have registered a code lens capability with the Client, which will send a “textDocument/codeLens” request, if it also supports lenses. This is handled by the *LSPWorkspaceManager*.
7. Responses to a code lens request are generated by publishing a *CodeLensEvent* to ask each plugin whether it wants to contribute lenses for the open file. The AST and TC plugins will contribute their “Launch | Debug” lenses, but there may be others.
8. The system is then quiet until the user does something else in the IDE.

2.4 Editor Changes are Made

When the user updates a file, even if those changes have not been saved, the “edits” are sent to the language server so that it can keep its in-memory version of the file up to date. The following occurs:

1. When you update a file, the Client sends "textDocument/didChange" notifications to the language server, which are handled by the *LSPWorkspaceManager*.

2. The Workspace Manager updates the in-memory version of the file, respecting the *vdmignore* list (so you can edit any file, but only VDM changes get processed further).
3. If you are editing an external format generated file, an IDE warning notification is sent.
4. A *ChangeFileEvent* is published. This can be handled by any plugin, but the AST plugin will use it to re-parse the updated in-memory file and send back syntax errors and warnings via the *MessageHub*, so that these appear on the fly.

2.5 A File is Created, Deleted or Saved

Edits made in an open editor are sent to the language server, as discussed above, but this is not the same as the file changes being made permanently on disk. When the user saves the changes, or a new file is created or deleted, the following events occur:

1. The built-in plugins register for “changedWatchedFiles” LSP events at startup. The Client will send these whenever filesystem changes are noticed within the project.
2. A file save or new file create/delete sends a "workspace/didChangeWatchedFiles" LSP notification. The notification is either of type “create”, “change” or “delete” and includes folders as well as files.
3. The server goes through all of the changes received and, depending on what changes, decides whether afterwards it needs to do nothing, to just re-check the in-memory specification or completely reload and re-check. See 2.2 .

Note that there are currently no events published for watched file changes. This may be added in a future release.

2.6 An Interpreter Session is Opened

Execution and debugging of a specification uses the DAP protocol rather than the LSP protocol, and there is a *DAPWorkspaceManager* that handles these messages. But the basic architecture of the DAP interactions is the same as for LSP. When an executable session is opened, the following occurs:

1. A DAP “initialize” request is sent to the language server. As with LSP, this include the Client’s debugging capabilities and these are saved by the *DAPWorkspaceManager*.
2. The DAP capabilities for the built-in plugins is prepared and then the *setDAPCapabilities* method of each plugin is called, via the *PluginRegistry*, to allow them to add or change settings.
3. A *DAPInitializeEvent* is published, and any responses saved.
4. An “initialized” DAP event is returned to the Client, along with any responses to the event above.
5. The Client may then send DAP requests to set breakpoints, if these have been set in the IDE. These are passed to the VDMJ interpreter.
6. The Client will then send a DAP “launch” command, which indicates whether the session is for debugging or just execution, along with other settings. These are saved by the Workspace Manager.
7. The Client should then send a “configurationDone” DAP notification. This is the indication that everything is set up and VDMJ can initialize the specification. Before this is done, a *DAPConfigDoneEvent* is published.
8. The “IN” plugin is obtained from the PluginRegistry, and this provides a VDMJ *Interpreter* instance. The specification is then initialized using a background thread.

9. The initialization thread will send DAP events to print the VDM banner on the debug console, and then run the *init* method of the *Interpreter*. When the initialization completes, the duration of the initialization is printed and the language server waits for the user to type something on the console.
10. (If a command was sent with the “launch” request, this will be executed immediately, rather than waiting for the user).
11. When the user enters a line on the console, an “evaluate” request is sent to the language server. The line typed is passed to all plugins, via the *PluginRegistry*, to call their *getCommand* methods. If a plugin recognises the line, it returns an *AnalysisCommand* object. If there are usage errors, it can return an *ErrorCommand* or just throw an exception.
12. A *DAPEvaluateEvent* is published, with details of the *AnalysisCommand* being executed.
13. The *AnalysisCommand’s run* method is invoked by the Workspace Manager. This is responsible for doing whatever the command intends, and may use other plugins or the Workspace Manager to achieve this.
14. The responses from the *AnalysisCommand run* are sent back to the Client, which typically displays them on the console.

Note that the DAP protocol is less well integrated with the plugin architecture, with fewer events being published for example. This will be improved in future to allow tighter integration of plugins with executions.

2.7 Unrecognised LSP/DAP Messages

It is possible that a new IDE feature wants to send a new message to the language server, which is not supported by the LSP or DAP protocols. This already happens with the *VDM VSCode* extension [6], where several non-standard messages are sent via LSP.

Typically, new messages will be handled by new plugins in the language server. So to connect the two, the *LSPXWorkspaceManager* and *DAPXWorkspaceManager* publish events, *UnknownMethodEvent* and *UnknownCommandEvent* for LSP and DAP respectively[2]. Plugins can register for these events and check for the new message/command names that they support.

[2] LSP calls its RPC names “methods”, while DAP calls them “commands”.

3 Analysis Plugin Implementation

This section describes how to implement a new plugin for the VDMJ language server. The description is based around an example plugin 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 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”.

3.2 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, a Java resource file called *lspx.plugins* must be set to a list of the class names of the plugins. Alternatively, a Java property of the same name can provide a CSV list of plugin class names. The fundamental plugins are configured automatically, and *ahead* of all user supplied plugins.

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. CT plugin (combinatorial testing)
6. User supplied plugins in order of the *lspx.plugins* resource files on the classpath.

To include the example plugin, the resource file would contain one line (see src/main/resources/lspx.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.jar
```

The way of setting 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).

3.3 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 significant differences. Dialect subclasses can 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(...);
	}
}
```

3.4 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 3.5) 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.

3.5 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 *EventListener* defines a method called *getPriority()*. This can optionally be used to order plugin registrations for the same event – so if both AST and a user plugin register for the XYZ event, the order of the event publication (below) will respect their priorities. By default, the built-in plugins come first and user plugins come afterwards.

The *EventListener* 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). In addition, error or warning messages can be raised for events by setting them on the *MessageHub* (below). 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 (particularly, anything from the *MessageHub*).

The *EventHub* calls each of the registered plugins, in the order of their priority, 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.

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.

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);
      ...
```

The example plugin handles *CodeLensEvents, CheckCompleteEvents* and *CheckFailedEvents*. The first generates a dummy code lens and returns the necessary response; the second generates an example warning message, which is raised by calling the *addPluginMessages* method of the *MessageHub*; the third event is sent when there were errors in the check process, and is used to examine and replace those errors in the *MessageHub* with dummy messages.

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

3.6 Message Handling

Diagnostic messages are displayed on the GUI as “squiggly lines” as well as being listed in the *Problems* view. But the GUI does not know about the collection of plugins that produced these errors; it only knows about files that have a list of errors to display. This means that something must act to collect diagnostic messages from all plugins and allow them to be re-grouped by file. This is handled by the *MessageHub*, which is a singleton class.

Plugins may either raise or clear diagnostic messages via the *MessageHub* without affecting any messages that have been raised by other plugins. The workspace manager then uses the *MessageHub* to collect all of the messages raised by all plugins for each particular file in the source.

So typically, plugins catch the *CheckPrepareEvent* and use that to clear their messages from all files. Then they process the specification using the other check events and re-populate the hub with any new messages. If any plugin raises *VDMErrors*, a *CheckFailedEvent* is published instead of *CheckCompleteEvent*, which allows plugins that did not raise the error(s) to react. Finally, at the end of the check process, the workspace manager uses the hub to collect messages from each plugin for each source file, and reports them to the GUI.

3.7 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.

3.8 Server Startup

3.8.1 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 3.2). This object can be amended by the plugin to set capabilities that it provides itself.

3.8.2 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.

3.9 Other Plugin Processing

3.9.1 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, a *CodeLensEvent* is published via the *EventHub.* By convention, this event is handled by a method called *getCodeLenses(File)* in each plugin. 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.

3.9.2 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 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 returns a *HelpList*, which is passed a list of Strings. The first word on each help line must match the command name that it describes – e.g. "print <exp> - evaluate an expression". These are displayed when the user enters the “help” command.

3.9.3 Annotations

Annotations were introduced to VDMJ in version 4.3.0. Their use is described in [7].

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 [7]). 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 IDE 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 plugin, via the *PluginRegistry*. The *getTC* method 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 *addPluginMessages* methods of the *MessageHub*.

4.2 Additional Code Lenses

The built-in plugins contribute the “Launch | Debug” lenses that you see in the Editor view of Visual Studio Code. But extra lenses can be supplied by a plugin, either at the same locations or totally separate locations in a file.

- Note that lenses are very Client specific, because they send back abstract “commands” that the Client should execute, but the meaning of the commands is not defined by LSP.
- Any user plugin can handle *CodeLenEvents* to provide lenses.
- There is a *CodeLens* abstract class that has helper methods for creating lens responses. This also includes methods to allow you to check the Client type.
- You are likely to want to locate your lenses at a position in the file that corresponds to the location of a VDM construct – like a definition or expression type. To find these locations, you are likely to want to get the parsed/checked tree from the “AST” or “TC” plugins, via the *PluginRegistry*.

4.3 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. See the *ExampleCommand* class in the example plugin. Alternatively, VDMJ’s *Console* class will have been redirected to the LSP client too.
- Remember to implement *getCommandHelp* as well. The help strings passed to the *HelpList* constructor must start with the command name.

- 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.
- If you override the *notWhileRunning* method, and return false, you will be allowed to run your command even while the system is evaluating an expression. Use this with care!
- If you implement the *InitRunnable* interface, you will be able to use your command in the "command" argument in "launch.json". The interface requires you to implement some methods that will be called in this case, and which allow the result to be formatted as you require.
- If you implement the *ScriptRunnable* interface, you will be able to include your command in scripts (ie. so that the "script" command can execute them). In this case you have to provide a *scriptRun* method as well as the regular *run* method.

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.

4.4 Adding New Capabilities

You can turn on LSP/DAP server capabilities in a plugin and then respond to the IDE messages they provoke.

- The *setServerCapabilities* method in your plugin can add or amend the settings added by previous plugins (typically, built-in plugins).
- If the plugin enables (say) "codeActionProvider", and the Client also supports this, then the IDE will start sending "textDocument/codeAction" requests. These are not handled by the built-in plugins, so an *UnknownMethodEvent* will be published.
- If your plugin registers to handle *UnknownMethodEvents*, it will receive these notifications and (after checking that the method is "textDocument/codeAction"!) can proceed to implement the response to those messages.
- The IDE should then do whatever you requested (usually refactorings, in this case).

4.5 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.

4.6 Adding a Style or Spell Checker

Plugins can raise their own messages via the *MessageHub*, so a plugin could process the AST (perhaps from the ASTPlugin or the TCPlugin) and raise messages relating to spelling or style. 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.

4.7 Enhancing Other Plugin's Messages

A plugin can look at other plugin's messages via the *MessageHub.getPluginMessages* method. The map returned (of files to messages) is a reference to the data used by the *MessageHub*, which means that the existing messages can be changed, added to or removed.

That is an unusual thing to do, but it might be useful in some cases for a plugin to enhance an existing message, either with more information about the problem, or perhaps replacing one message with another (possibly at a different location).

5 Future Work

The language server 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 DAP protocol integration via events.
- More consistent use of LSP events.
- Plugins that include Client side components too.
- Move some internals from the Workspace Managers into plugins.

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 may be sent back to the Client, or it may be ignored, depending on the event type.

Event Class	When?
InitializeEvent	Raised when an LSP “initialize” message is received at the server startup. It is sent after the server capabilities have been calculated, but before they are sent to the Client. The return value is sent along with the server capabilities.
InitializedEvent	Raised when an LSP “initialized” message is received. The event is sent after the parse/check that is performed, and the return value is added to any responses to the Client.
OpenFileEvent	Raised when a VDM file is opened, except for *vdmignore* files or those on a “dot” path. Also sent for new files.
ChangeFileEvent	Raised when the user types a key in an editor window. The in-memory copy of the file is updated with the change, and then the event is raised. It is primarily used by the ASTPlugin to parse the specification as you type. The return value is ignored.
SaveFileEvent	Raised when a VDM file is saved – though not if the server supports *changedWatchedFiles*.
CloseFileEvent	Raised when a specification file, which was previously open, is closed. Note, this is not raised for files that are ignored or within “dot” folders. The response is ignored.
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 sent to the Client. It is primarily used by the AST plugin.
CheckTypeEvent	Raised to perform the type check phase of parse/check. The response messages will be sent to the Client. 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. The return value is added to the responses to the Client.
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 added to the responses to the Client.
DAPInitializeEvent	Raised at the end of a DAP “initialize” event. The Client capabilities are set in the *DAPWorkspaceManager*, and the server capabilities have been calculated. The response is sent back to the Client, along with the server capabilities.

DAPLaunchEvent	Raised when a DAP “launch” command is received at the start of a session. The launch arguments will be available and the VDMJ properties will have been set. The return value is ignored.
DAPConfigDoneEvent	Raised when the DAP “configurationDone” message is received. Any breakpoints will have been set, but the interpreter initialization has not been started.
DAPBeforeEvaluateEvent	Raised when and evaluate request is received, either immediately after initialization (with a launch command) or via a console command. Note that the command may not be a “print” command (the event carries the details). If any responses are DAP “failures”, the evaluation will be aborted with the message given. This can be used by plugins to veto an evaluation, because they are busy (e.g. running CT traces).
DAPEvaluateEvent	Raised after the DAPBeforeEvaluateEvent, and immediately before the expression is actually evaluated. The response is ignored.
DAPTerminateEvent	Raised when a DAP “terminate” message is received, if if the “restart” flag is set. The event is sent after all threads have been terminated, after the breakpoints have been cleared, and after VDMJ properties have been restored. The return value is ignored.
DAPDisconnectEvent	Raised when a DAP “disconnect” message is received. The event is sent after all threads have been terminated, after the breakpoints have been cleared, and after VDMJ properties have been restored. The return value is ignored.
ShutdownEvent	Raised when an LSP “shutdown” request is received. The event is sent early in the process, so the external format files will still exist and the *PluginRegistry* and *EventHub* will not have been reset. The return value is ignored.
UnknownCommandEvent	Raised when a DAP message is received with an unknown “command” field. This usually indicates a protocol extension. The return value is sent back to the Client.
UnknownMethodEvent	Raised when an LSP message is received with an unknown “method” field. This usually indicates a protocol extension. The return value is sent back to the Client.
UnknownTranslationEvent	Raised when an “slsp/translate” LSPX method is received, but with an unknown target language. This usually indicates a protocol extension, but definitely for a translation. The return value is sent back to the Client.