LSP Design Specification

Author	Nick Battle
Date	23/04/24
Issue	0.4

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] Overture, https://github.com/overturetool/overture

[5] https://microsoft.github.io/language-server-protocol/overviews/lsp/overview/ and https://microsoft.github.io/debug-adapter-protocol/overview

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

[7] LSP4J, https://github.com/eclipse/lsp4j

[8] VDMJ Design Specification, https://github.com/nickbattle/vdmj

[9] Neovim editor, https://neovim.io/

[10] LSP4E, https://projects.eclipse.org/projects/technology.lsp4e

0.3. Document History

Issue 0.1	18/04/22	First draft.
Issue 0.2	10/01/23	Updated for DAPX
Issue 0.3	20/02/23	Small updates for 2023.
Issue 0.4	23/04/24	Updated for LSP/DAPPlugins

0.4. Copyright

1. Overview

This design describes the Language Server Protocol and Debug Adapter Protocol support in the VDMJ tool.

Section 1 gives an overview of the architecture into which the LSP/DAP server fits and the Java package structure. Section 2 gives detailed information about each package. Section 3 walks through various common scenarios to describe the interaction of the internals.

1.1. VDMJ

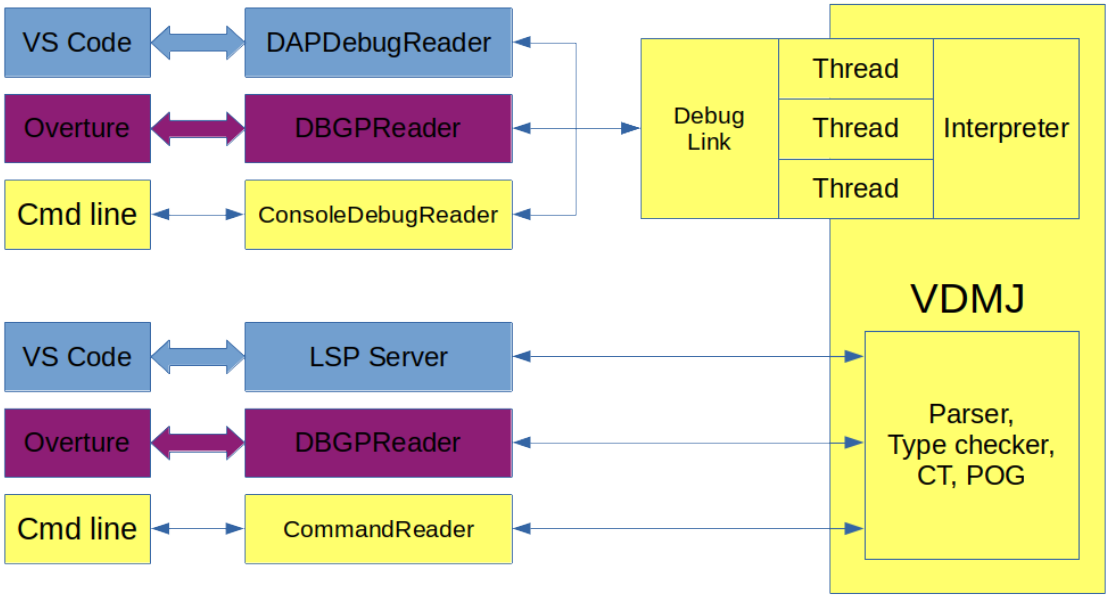
VDMJ provides basic tool support for the VDM-SL, VDM++ and VDM-RT specification languages, written in Java [1][2][3]. It includes a parser, a type checker, an interpreter (with arbitrary precision arithmetic), a debugger, a proof obligation generator and a combinatorial test generator with coverage recording, 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 interface is built in to VDMJ and offers a simple command line. But VDMJ is also used by the Overture project [4], which adds a graphical Eclipse IDE interface. It can also be used via the LSP/DAP protocols [5] and so can be used by an IDE like VS Code [6].

The following diagram illustrates the tool-independent core VDMJ functions (yellow) and how they are accessed by three different user interfaces.



The thin arrows indicate a direct API connection, while the thicker coloured arrows indicate a remote control socket protocol.

This design document discusses the internal design of the blue LSP and DAP boxes and their interaction with the VS Code Client and the VDMJ language services.

1.3. LSP and DAP Protocols

To quote from [5], regarding LSP:

> *The idea behind a Language Server is to provide the language-specific smarts inside a server that can communicate with development tooling over a protocol that enables inter-process communication.*
>
> *The idea behind the Language Server Protocol (LSP) is to standardize the protocol for how tools and servers communicate, so a single Language Server can be re-used in multiple development tools, and tools can support languages with minimal effort.*

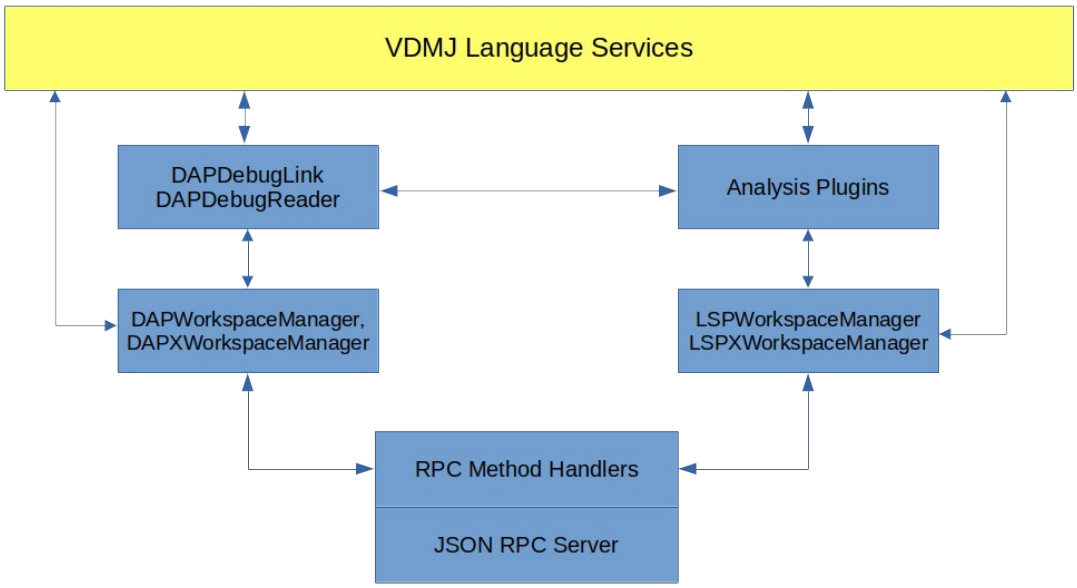
And regarding the DAP:

> *It takes a significant effort to implement the UI for a new debugger… Typically this work must be repeated for each development tool, as each tool uses different APIs for implementing its user interface. This results in lots of duplicated functionality (and implementation)… The idea behind the Debug Adapter Protocol is to standardize an abstract protocol for how a development tool communicates with concrete debuggers.*

1.4. LSP and DAP Server 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.

The overall architecture of the server is as follows, where the blue boxes implement the LSP and DAP parts of the diagram above:



The JSON RPC server and method handlers are responsible for communication with the Client. They have no knowledge of the language environment. JSON data structures are decoded and encoded at this point, so that the layers above operate with normal Java data types.

The plugins understand how to implement each of the LSP and DAP methods that come from the Client. The *LSPPlugin* loads and caches all of the VDM source files for the project; other plugins handle extended protocol requests for non-standard features, such as PO generation and

combinatorial testing. The *DAPPlugin* handles everything to do with the DAP protocol, when the user is in an execute/debugging session.

The debug link and reader classes interact with the VDMJ interpreter when breakpoints or exceptions are encountered.

Analysis plugins hide the detail of the VDMJ analysis data structures from the LSP/DAP plugins. For example, depending on the VDM dialect, the specification may be represented as a list of classes or a list of modules. The plugins hide this difference and allow the *LSPPlugin* to (say) locate a definition, regardless of the dialect.

1.5. LSP Package Overview

The implementation is divided into the following Java packages.

Packages	
json	Classes that implement JSON parsing and the JSON Server
rpc	Classes that implement JSON RPC
lsp	LSP support classes
dap	DAP support classes
vdmj	VDMJ language interaction classes
workspace	The plugin registry and event and message hubs

The *json* package contains classes to represent JSON data structures and read or write such structures to a communication channel. An abstract *JSONServer* class implements the basic functionality to read JSON messages from and write JSON messages to a pair of streams.

The *rpc* package contains classes to receive and dispatch a JSON RPC call. This includes *RPCRequest* and *RPCResponse* classes which extend *JSONObject*, as well as a dispatcher which routes RPC method invocations to a set of registered *RPCHandler* classes.

The *lsp* package mainly has RPC handlers for the various LSP protocol messages. Handlers are in sub-packages that reflect their LSP method name structure, like “textdocument/completion”.

The *dap* package is similar to *lsp,* but for the DAP protocol. One difference is that the DAP RPC messages are *not quite* standard JSON RPC messages and so do not use the *rpc* classes.

The *vdmj* package includes classes that interact directly with VDMJ language services. For example, there are visitors which search for definitions within the specification, and there are link classes that allow the DAP processing to connect to the VDMJ runtime during debugging. There is a sub-package called *commands* which contains classes to implement commands which can be entered at the command line during an execution/debugging session.

The *workspace* package includes the *PluginRegistry, EventHub* and *MessageHub* classes. There are three sub-packages: *plugins* which contains all of the *AnalysisPlugins*; *events* which contains LSP and DAP events; and *lenses* which contains the built-in code lenses.

1.5.1. Comments

A natural question to ask is why so much of this is implemented in the VDMJ LSP project, when the *lsp4j* [7] project provides an LSP/DAP implementation already. The answer is partly historical, partly convenience and partly a matter of licence concerns.

The JSON handling and JSON RPC layers were created while experimenting with the LSP protocol. JSON and JSON RPC are extremely simple to implement (by design), so this was a realistic way to start the development, and meant that the internals were transparent while debugging.

The LSP and DAP processing was then relatively easy to add on top of the JSON layer. Much of this

would have had to be implemented under *lsp4j* anyway because the framework cannot know what services the language server should provide. This meant that the implementation could concentrate on just the LSP methods that the server intended to provide. By contrast, there is a lot of code within *lsp4j* that supports functionality that is not relevant for VDMJ.

The *lsp4j* project is designed to be used from within the Eclipse IDE and therefore has multiple dependencies on other projects, such as GSON and Guava, as well as core Eclipse libraries. These can presumably be unpicked, but by contrast the VDMJ LSP service is a single Java project. The main *lsp4j* jar alone is about 788Kb in size, whereas the VDMJ LSP jar (including everything) is about 188Kb.

Lastly, there is some concern about the licencing of all third party products and their dependencies, and whether they are compatible with the GPL licence that is used for VDMJ. The *lsp4j* project is released under the Eclipse Distribution Licence or the Eclipse Public Licence. These are thought to give licence compatibility issues with GPL. By contrast, the VDMJ LSP project is locally owned and can be licensed as we choose.

So the VDMJ LSP implementation is as small and fast as possible, its internals are transparent, it has a small footprint, no external dependencies and does not cause any licensing issues.

2. Package Detail

This section gives more detail about the Java classes in the packages within LSP.

2.1. json

Class Summary	
JSONArray	A representation of a JSON array
JSONObject	A map representation of a JSON object
JSONReader	A class to read a JSON stream into a JSONObject
JSONWriter	A class to write a JSONObject as to a stream as JSON
JSONServer	An abstract base class for writing JSON servers (eg. an RPC server)

The *json* package contains classes to read and write JSON streams via Java objects.

All top-level JSON data is held as a *JSONObject*, which is a map from simple key Strings to Java types. Arrays within JSON are held as *JSONArray* objects, and sub-objects are naturally held as *JSONObjects* too. The primitive types are as follows:

JSON-Java Type Mapping	
String	java.lang.String
Real	java.lang.Double or double
Integer	java.lang.Long or long
Boolean	java.lang.Boolean or boolean
Null	A null pointer
{ Object }	*JSONObject* (Map of String to Java Object)
[Array]	*JSONArray* (Vector of Java Object)

JSONObject and *JSONArray* have a *getPath* method that allows convenient access to a field deeply nested within a tree of objects and arrays. For example:

```
String name = jsonObject.getPath("result.[0].traces.[2].name")
```

This gets the “result” field from the *JSONObject*, which it expects to be an array; then it gets the [0] item, which it expects to be an object; then it gets the “traces” field of that object, which is expected to be an array; then it gets the [2] item of that array, which is expected to be an object; and lastly the “name” field of that is returned. If any of these type assumptions are false, or the fields do not exist, the *getPath* method returns null.

JSONReader and *JSONWriter* read and write (respectively) *JSONObjects* from and to a Java Reader or Writer. Note that this means the character encoding of the underlying stream is the responsibility of the provider of the Reader/Writer. In the case of an LSP/DAP server, the underlying stream is always UTF-8 encoded.

A *JSONServer* is constructed with a pair of Java Streams (not Readers) and has messages to read a *JSONObject* from or write a *JSONObject* to them. The server includes the common convention for messages to be prefixed with headers as follows:

```
Content-Length: <decimal length in bytes><CRLF>
[Content-Type: <encoding><CRLF>]
<CRLF>
<JSON data in encoding>
```


The Content-Type header is optional. By default the encoding is assumed to be UTF-8.

2.1.1. Comments

There are some generic getter methods in *JSONObject* and *JSONArray* to allow the classes to be used more conveniently (avoiding the need for excessive casting). It might be possible to have methods that are even easier to use, with a bit of thought.

2.2. rpc

The *rpc* package defines classes for representing and dispatching JSON RPC requests to a set of registered handlers for each method. In combination with the *JSONServer* class (above) this can implement a JSON RPC service.

Class Summary	
RPCDispatcher	A class to dispatch calls via a map of method names to handlers
RPCHandler	An interface to handle one particular RPC method
RPCRequest	A JSON RPC request
RPCResponse	A JSON RPC response
RPCMessageList	A list of RPC messages
RPCErrors	An enum with constants for standard RPC error numbers

An *RPCRequest* extends *JSONObject* and adds methods to conveniently construct a request from raw parameters or a *JSONObject*. Similarly, *RPCResponse* extends *JSONObject* with methods to construct a response.

A handler is a class that implements the *RPCHandler* interface and provides a *request* or *response* method, depending on whether it is designed to handle RPC requests, responses or both. A *request* method is passed an *RPCRequest* returns an *RPCMessageList* of *RPCResponses* – a list because often a request provokes a set of notification messages in addition to its own response. A *response* handler is passed an *RPCResponse* message, but returns void because handling a response is not expected to generate more requests.

Handlers bridge the gap between JSON and a native Java representation of data. So for example, most handlers unpack/convert their arguments from the *RPCRequest* passed and then call methods on plugins (below) to process them, rather than processing them directly. The plugin methods are therefore only minimally aware of JSON (they build *RPCResponses*, but using helper methods).

An *RPCDispatcher* defines a map of String method names to *RPCHandlers*, plus *register* methods to register a new handler for a particular RPC method or list of methods. It also defines a *dispatch* method which is passed an RPCRequest, looks up the handler and calls its *request* method, passing back any response messages. If a handler is registered without any method(s), this is used to handle unknown method dispatches. This is used to allow dynamic analysis plugins to be called.

So an RPC server can be constructed by extending *JSONServer* to define the communications, creating an *RPCDispatcher* to handle requests, and registering a set of method handlers with that dispatcher. This is what the *LSPServer* does (below).

2.2.1. Comments

There are some public static factory methods on *RPCRequest* and *RPCResponse*, to allow objects to be created with relatively sensible method calls rather than having to guess the right parameters to pass to a universal (or overloaded) constructor. This works relatively well, but the method names are

still overloaded. Perhaps these ought to be clarified even more.

The idea that the Handlers separate JSON from Java data representations is almost true, though in practice it seemed absurd to have a Java version of the *RPCResponses* simply to keep to this rule. So the workspace methods called from the handlers return an *RPCMessageList* (ie. a list of *JSONObjects*), even though the parameters to the methods have been unpacked and converted from the *RPCRequest*.

2.3. lsp

The *lsp* package defines RPC handlers for each of the supported LSP methods.

Class Summary	
LSPHandler	Abstract root of all LSP method handlers
InitializeHandler	Handler for the initialize method
LSPInitializeResponse	An RCResponse to the initialize method
ShutdownHandler	Handler for the shutdown method
ExitHandler	Handler for the exit method
UnknownHandler	Handler for unknown methods
workspace.xxxHandler	Workspace method event handler(s)
textdocument.xxxHandler	Text document method event handler(s)
lspx.xxxHandler	LSP extension method event handler(s)
CancelHandler	Handler for the cancel method
CancellableThread	A thread to run background tasks that can be cancelled
LSPServer	The abstract root of LSP servers
LSPServerSocket	A concrete LSP Server that reads/writes from sockets
LSPServerStdio	A concrete LSP Server that reads/writes from stdio
LSPException	An exception from LSP
LSPMessageUtils	Various utilities to create messages
Utils	Low level utilities

The *lsp* package mainly comprises classes that implement the *LSPHandler* abstract class, which in turn extends the *RPCHandler* class (above).

The first handler invoked in the LSP protocol is the *InitializeHandler*, which handles the “initialize” and “initialized” methods from the LSP client. Both methods are simply forwarded to equivalent methods on the *LSPPlugin* instance. The initialize method returns an *InitializeResponse*, which defines the capabilities of the LSP Server.

The *UnknownHandler* is registered to handle any LSP methods that are unknown. These are passed to the *LSPPlugin* which uses the *PluginRegistry* to look for plugins that can handle the method dynamically.

At the other end of the LSP protocol, the *ShutdownHandler* and *ExitHandlers* handle “shutdown” and “exit” method calls from the Client. The shutdown method simply marks the LSP Server as uninitialized (so that it expects a new initialize call), but otherwise does not affect the running server. The exit method kills the LSP server, and is used when the Client is terminating.

Classes in the *workspace*, *textdocument* and *lspx* subpackages handle LSP methods in those three groups (ie. their method names start with “workspace/” etc.). These all unpack the JSON arguments from the *RPCRequest* passed and then call a method on the *LSPPlugin* or other plugins. The latter handle LSP extensions that are not part of the base protocol, which are used to implement specialized methods for the VDM Client (eg. for proof obligations and combinatorial testing).

Long running operations in LSP can be cancelled by the Client, which should interrupt them. To implement this, a *CancelHandler* makes method calls on an abstract *CancellableThread*, which is used to asynchronously execute long running methods. Concrete asynchronous threads extend *CancellableThread* and implement the *body* method; the registering of active and completed threads is managed by the abstract parent. A static *cancel* method is provided to cancel a thread by "id", which sets a flag that the running operation is responsible for monitoring. A *setCancelled* method can be overridden to perform bespoke cancellation actions when the cancel is sent.

The *LSPServer* class extends *JSONServer* and is a base implementation of a server that can use different communication methods. The constructor is passed a Java *InputStream* and *OutputStream,* then creates an *RPCDispatcher,* registering all of the LSP handlers (above) with the method names that they support. It also creates a map of response handlers, which are used to handle *RPCResponse* messages that are received from the (very few) requests that are sent from the server to the Client. The main *run* method of the server loops, reading JSON messages and either calling the dispatcher to handle requests, or dispatching responses via the response handlers. Responses generated by the handling of requests are sent back to the Client before listening for more requests. The loop runs until the *JSONServer readMessage* method indicates that the communication channel has closed.

There are two uses of *LSPServer*, though more can be added. *LSPServerSocket* is used in environments like VSCode, where a TCP socket is used to communicate; and *LSPServerStdio* is used to support Clients that use stdio links, such as the Eclipse *lsp4e* [10] and the *neovim* [9] tools. Both of these classes contains a *main* method and construct an *LSPServer* after they have established their communication links.

The *LSPMessageUtils* and *Utils* classes include methods to perform basic common tasks that are used by many other classes. *LSPMessageUtils* is concerned with assisting the creation of *RPCResponse* message bodies, whereas *Utils* has even lower level methods that deal with such things as the translation of VDMJ *LexLocation* information into LSP/JSON P*osition* and *Range* types, and the conversion of protocol URIs into canonical Java *Files*.

2.3.1. Comments

Note that LSP Positions and Ranges are zero-based, whereas VDMJ *LexLocations* are 1-based.

All file references in the LSP and DAP protocols use URIs with absolute names, so the Utils package has methods to convert URIs to and from absolute Java Files.

2.4. dap

The *dap* package defines messages and handlers for each of the supported DAP methods.

Class Summary	
DAPDispatcher	A class to register and dispatch DAP requests to DAPHandlers
DAPHandler	A class to handle one or more DAP commands
handlers.xxxHandler	A set of DAPHandlers for the supported DAP commands
DAPInitializeResponse	A DAPResponse that contains the supported DAP features
DAPRequest	A DAP Client request
DAPResponse	A DAP Server response
DAPMessageList	A list of DAP server responses
DAPServer	A JSONServer implementation that uses a Socket
DAPSocketServer	A thread that listens on a port, and creates a DAPServer
AsyncExecutor	An abstract base for cancellable executions
ExpressionExecutor	An interruptible evaluation of one VDM expression

InitExecutor	An interruptible initialization of the specification loaded
ScriptExecutor	An interruptible evaluation of a script file of commands
DAPOutConsoleWriter	A VDMJ ConsoleWriter that writes to a DAP stdout link
DAPErrConsoleWriter	A VDMJ ConsoleWriter that writes to a DAP stderr link

The *DAPDispatcher* class performs a similar role to *RPCDispatcher,* except it deals with *DAPRequests* and *DAPResponses*, which are subtly different from JSON RPC messages. But in a similar way, *DAPRequests* are dispatched to one of a number of *DAPHandlers* that are registered for various methods (known as “commands” in the DAP protocol), and these return a *DAPMessageList.*

There are a number of *xxxxHandler* implementations of the *DAPHandler* abstract base within a “dap.handlers” subpackage. Similar to the LSP handlers, these unpack the DAP arguments and call on the *DAPPlugin* to implement the method.

A *DAPServer* is similar to an *LSPServer* (both implement *JSONServer*), and the *DAPServerSocket* class is a thread that listens for incoming connections and creates a *DAPServer* to handle each one.

The *AsyncExecutor* class is an abstract parent for a number of classes that perform cancellable and interruptible evaluations in VDMJ. The abstract class extends *CancellableThread* and defines a lifecycle, executing methods before and after the evaluation, as well as methods if an exception is caught. Concrete subclasses then implement this for evaluating individual expressions, evaluating a script of commands and initializing a specification.

Lastly, the two DAPxxx*ConsoleWriter* classes implement the VDMJ *ConsoleWriter* interface, and contain a *DAPServer* instance, which can be used to directly send event messages to the Client for stdio,. The *DAPServer* uses these classes to tell VDMJ where to send arbitrary output, for example from the standard IO library or from @Printf annotations.

2.4.1. Comments

The *DAPRequest* and *DAPResponse* classes ought to have friendly static constructor helpers, like their RPC equivalents.

Some of the DAP handlers make direct calls on the *DAPDebugReader*, if there is one. For consistency, perhaps these cases ought to be handled via to the *DAPPlugin*.

2.5. vdmj

The *vdmj* package contains support classes that link the LSP and DAP protocol handlers to the VDMJ core functionality. A “vdmj.commands” subpackage defines methods that implement the commands that can be entered into an interactive debugging session (like “print <expression>” and “quit”).

Class Summary	
DAPDebugReader	A class to interact with multiple threads, being debugged.
DAPDebugLink	The DAP implementation of a VDMJ DebugLink
DAPDebugExecutor	A class to perform commands in one debugged thread.
AnalysisCommand	Abstract root of all debug session commands
xxxCommand	Console commands for a debugger session
LSPDefinitionFinder	Find symbols in a file at a give line/position or LexLocation
LSPDefinitionLocationFinder	Visitor to support LSPDefinitionFinder
LSPExpressionLocationFinder	Visitor to support LSPDefinitionFinder
LSPPatternLocationFinder	Visitor to support LSPDefinitionFinder

LSPStatementLocationFinder	Visitor to support LSPDefinitionFinder
LSPBindLocationFinder	Visitor to support LSPDefinitionFinder
LSPMultipleBindLocationFinder	Visitor to support LSPDefinitionFinder
LSPImportExportLocationFinder	Visitor to support LSPDefinitionFinder
LSPModuleDefinition	A dummy TCDefinition to represent a module.

The VDMJ *DebugLink* design is described in [8] section 2.12, and is illustrated in outline above in section 1.2. The principle is that a “debug reader” thread is created to wait for VDMJ threads that reach breakpoints or throw exceptions that should trap into the debugger. A single *DebugLink* instance bridges the gap between the debug reader and all the threads being debugged. A *DebugExecutor* then executes debugging commands in the context of a particular stopped thread (ie. a VDMJ *Context* stack). The “DAP” implementations of these concepts interpret the VDMJ interfaces in a DAP protocol context. VDMJ itself is not aware that it is interacting with a DAP Client.

The *DAPDebugReader* instance is created, started and killed by *AsyncExecutor*, whenever a VDMJ expression is being evaluated. Note that this is the case even if debugging is not enabled, because any evaluation may throw an exception or be paused by the user if it takes too long.

Each *DAPDebugReader* instance creates a new *DAPDebugLink*.

When the *DAPDebugReader* thread is running, it waits for all threads to stop, by calling the *waitForStop* method of the *DebugLink.* When this method returns, all of the threads in the specification will be in an interrupted state, and can be interacted with via the link. The reader then enters a loop, waiting for commands via the DAP protocol, converting these to *DebugLink* calls, and interpreting the responses in DAP protocol terms to send back to the Client. For example, the Client may set or clear breakpoints, or ask for information about a thread’s stack frame.

Note that the *DAPDebugLink* extends the *ConsoleDebugLink* that VDMJ uses for command line debugging. This is because the core of the functionality required is the same, but the “console” is with an LSP Client rather than being a stdio terminal.

Most of the sophisticated DAP message creation is handled by the *DAPDebugExecutor* instances, one of which is created when each thread stops. In particular, these classes keep note of the stack frames and variable references that have been issued to the Client to refer to each variable. They also unpick the (quite complicated) VDMJ *Context* stack and interpret that in DAP message terms. For comparison, the *ConsoleDebugExecutor* does the same for a command line VDMJ debug session, but the messages returned are just text output to be displayed directly to the user.

When a DAP debug/evaluation session is open, (typically) the Client has some sort of console available to the user, from which they can evaluate expressions. In the VDMJ case, we generalise this and allow arbitrary commands to be entered, one of which (called “print”) will evaluate expressions. But for example, the “default” command can be used to set the default class or module name before an evaluation. Every recognised command is implemented as a subclass of an abstract *AnalysisCommand* class, which defines a *run* method that accepts a *DAPRequest* and returns a *DAPMessageList* response. Requests from the Client are sent as DAP “evaluate” commands, which are handled by the DAP *EvaluateHandler* and sent to the *DAPPlugin*, which parses the command line and produces an *AnalysisCommand* object, which is then run. So for example, typing “default MyModule” on the console sends a DAP evaluate request, which is parsed by the *DAPPlugin* to produce a *DefaultCommand* object that contains the argument “MyModule”. The run method of that object finds the VDMJ interpreter instance and calls its *setDefaultName* method, before returning an “OK” *DAPResponse,* or some error message if the MyModule name cannot be found.

Commands that evaluate expressions are more complicated, and start an *ExpressionExecutor,* which is a *CancellableThread*, discussed above. The *ScriptCommand* that runs a sequence of commands from a file is even more complicated and requires a *ScriptExecutor* that is also a *CancellableThread*.

A *HelpCommand* is included, and lists all of the other commands that are available on stdout. Only one command is hidden from this list: the *RuntraceCommand* is used by combinatorial testing to evaluate a single test, and is not advertised to the user (though it ought to work, if called manually in the right context).

Note that a “print” command can be added to the DAP “launch” request, passing it as the “command”

field. This allows one-shot evaluations to be set up, since the debug session evaluates that one command and then terminates.

When stopped at a breakpoint, the *EvaluateHandler* sends requests to the *DAPDebugReader* rather than the *DAPPlugin*. This is used when the Client sets “watches” of a variable or expression. These evaluations are not parsed as commands and cause the expression sent to be evaluated directly. The results populate the “Watch” window. If you type directly into the console at a breakpoint, the expression you type will be evaluated this way, and not parsed as a command.

The LSP functionality to “goto definition” (eg. pressing F12 on a symbol in VSCode) is implemented using the *LSPDefinitionFinder* class and a set of VDMJ visitors to search the AST for a symbol at a particular line and position. It is relatively simple to locate an AST node at a particular location, but tracing that back to the corresponding definition can be tricky. The detail of the process here is in the *lookupNodeDefinition* method of the *LSPDefinitionFinder.* The same visitors are used to implement the “find references” functionality (eg. pressing Shift F12). In this case, the definition finder is used to locate the definition of the current location as above, but then all occurrences of the (text) name within the specification are checked to see which also resolve to the same LexLocation as the current position. This is used to build the response for the references of the current symbol.

2.5.1. Comments

It would probably be better to have an abstract common root with *DebugLink* methods, rather than having the DAP link extend the Console link.

2.6. workspace

The *workspace* package defines classes that implement the functionality of the LSP, DAP and extension methods. A “plugins*”* subpackage defines classes that implement functions this. A “lenses” subpackage defines classes that generate code lenses. An “events” subpackage defines events sent from the plugins.

Class Summary	
EventHub	The workspace event dispatcher
EventListener	An interface implemented by event processors
event.*Event	Events sent via the EventHub
MessageHub	The workspace error/warning message hub
plugins.AnalysisPlugin	The abstract base of all analysis plugins
plugins.XXPluginDD	A plugin for the XX analysis of the DD dialect
PluginRegistry	A registry for all available plugins
lenses.CodeLens	The abstract base of all code lenses
lenses.LaunchDebugLens	A code lens to add “launch” and “debug” for function/operations
Diag, DiagUtil	A utility for diagnostic logging

The *LSPPlugin* and *DAPPlugin* classes are singletons that handle individual protocol requests from the Client. Other plugins perform particular analyses, or implement protocol extensions.

The *LSPPlugin* handles LSP requests. The *getInstance* method constructs a plugin and registers the analysis plugins that are required (based on the dialect).

The *LSPPlugin* is responsible for initializing the state of the VDMJ language service and loading specification files into memory. The process is initiated via the *lspInitialize* method, which identifies the *rootUri* of the specification as well as the Client’s capabilities. The method uses this information to load all of the project files, which makes a recursive search for files from the root that match the dialect extension, but which are not within “dot” folders, like .vscode or .generated. Note that the

LSPPlugin keeps files in memory, via a map of *Files* to *StringBuilders*; if files are edited, that is performed by the Client and the Server is informed via “didChange” editing messages.

When this loading process is complete and the response sent, the Client may send breakpoints and when ready it sends an “initialized” message to the Server, which is handled by the *lspInitialized* method. This sends back any dynamic registrations that the Server needs, as well as performing a parse and typecheck of the loaded files, which may send back diagnostics. The only dynamic registration used currently is to “watch changed files”. This enables the Server to see when new files and folders are created or moved in the specification.

Subsequent file edits are sent to the *LSPPlugin* via “didChange” requests. These are used to edit the map of *Files* to *StringBuilders*, and the specification is re-parsed to see whether there are syntax errors. When a file is saved to disk, the Client sends “didSave” messages, which causes the server to perform a type check and return any new errors.

One problem with VDMJ is that, depending on the dialect used, a specification is either a list of classes or a list of modules. This means that workspace processing that is relevant for all dialects has to make dialect-specific choices. To avoid “if/else” tests throughout the workspace code, methods that are dialect-specific are delegated to plugins which are specialised for each dialect. Plugins are further separated by VDMJ analysis type (eg. syntax, type checking, interpreter, POG…). Plugins extend an abstract class for their analysis, like *ASTPlugin*, and these in turn extend an *AnalysisPlugin* base class. The workspace registers each plugin with a *PluginRegistry* on construction. Later, plugins can be requested via the registry, using the analysis name.

So when the *LSPPlugin* wants to (say) get the outline symbols for the specification, it calls the registry to locate the “TC” plugin, which will be specialised for the current dialect. The *documentSymbols* method on the TC plugin will then process the classes or modules that it contains, returning a list of symbols independent of the dialect.

Some LSP and DAP events are of interest to more than one plugin, and user provided plugins may be added (see below). So the *LSPPlugin* cannot know which plugin(s) to invoke. To solve this, plugins can implement the *EventListener* interface and register interest in events via an *EventHub.* When events occur, the *LSPPlugin* publishes them via the *EventHub,* which then invokes *handleEvent* methods of the *EventListeners*. This allows arbitrary plugins to (for example) be informed when the specification has changed or been typechecked, etc.

One consequence of having multiple plugins is that there has to be a central way of knowing the complete list of errors or warnings that should be displayed on the GUI. This is handled by the *MessageHub* singleton. Plugins raise errors here by calling the *addPluginMessages* and *clearPluginMessages* methods. Subsequently, the *LSPPlugin* uses the *MessageHub* to get all of the diagnostics for each source file in the project.

The *DAPPlugin* is similar in principle to the *LSPPlugin*, except that it is concerned with the processing of DAP requests. For example, the *DAPPlugin* includes the current VDMJ *Interpreter* in its state, since several DAP operations interact with the interpreter in some way.

The *LSPPlugin* additionally loads *AnalysisPlugin* instances identified from a resource list, optionally overridden by the “lspx.plugins” property. This allows external analyses to be loaded. Analyses can register their own dispatch handlers. For example, an external analysis could require a new “dischargeObligation” command and create a handler to direct messages to it.

Code lenses allow the specification to add Client-side commands that are launched when a label is clicked in the editor, associated with a particular position in the source. In principle there can be many code lenses, but only one is defined currently. This adds “Launch” and “Debug” labels to definitions that can be executed. This lens passes enough information to the client for it to be able to create a launch configuration entry.

All code lenses extend an abstract *CodeLens* class and implement methods to return the lenses for a given file and definition – either an ASTDefinition or a TCDefinition, depending on whether the editor buffer is dirty.

2.6.1. Comments

The idea of the plugins was partly to allow processing to be dialect independent, and partly to allow

arbitrary analysis functionality to be added to the LSP server via pluggable services that could be discovered. The dialect independence works well, but the *LSPPlugin* and *DAPPlugin* are still partly aware of which plugins are available and which ones they depend on. For example, the *DAPPlugin* depends on the *INPlugin* to set breakpoints. To be completely pluggable, this would have to generalise to an event available to all plugins to achieve this action. While not impossible, this is an enhancement that is yet to be done.

3. Scenarios

This section describes the sequence of actions that occur during common interaction scenarios with the LSP Server. The intention is to provide a more tangible description of how the classes described in section 2 work together in practice.

3.1. Server Creation and Lifecycle

The following sequence of events occurs when an LSP Server instance is started.

- The *LSPServerSocket* or *LSPServerStdio* **main** method is invoked. The command line arguments are parsed to provide the VDM dialect and LSP/DAP port numbers (stdio only requires the DAP port).
- An *LSPServerSocket* or *LSPServerStdio* instance is created, passing the dialect and ports, and its run method is called. The method waits on the LSP port (or stdin).
- If a DAP port is provided, a *DAPServerSocket* thread is created and started, which waits on the DAP port.

Then nothing happens until an LSP Client connects to the LSP port (or sends a connect request on stdin, for the stdio server). When that happens, the first message will be an LSP “initialize”:

- The listener accepts the connection and creates an *LSPServer* object, passing the input and output streams from the connection (or stdin/stdout) and calling its run method. When the run method is complete, the listener waits for another connection.
- The *LSPServer* constructor creates an *LSPPlugin* which registers the list of *RPCHandlers* defined for each supported LSP method. Various global VDMJ settings are made and the built-in analysis plugins are created.
- The *LSPServer* run method loops, reading a JSON messages from the input stream (via *JSONServer* methods).
    - When a JSON message is received, an *RPCRequest* is made from it and the dispatcher is used to send the request to the right handler, returning a list of *RPCResponse* messages.
    - The responses are sent back to the LSP Client via the output stream as JSON messages.

Note that the DAP thread will be idle while LSP messages are being processed, but when the LSP Client enters an execute/debug mode, a connection will be opened to the DAP port. The first message sent will be a DAP “initialize”:

- The DAP listener accepts the connection and creates an *DAPServer* object, passing the input and output streams from the connection and calling its run method. When the run method is complete, the listener waits for another connection.
- The *DAPServer* constructor creates a *DAPPlugin* which registers the list of *DAPHandlers* defined for each supported DAP method.
- The *DAPServer* run method initializes VDMJ’s *Console* stdio to use DAP support classes, then loops reading JSON messages from the input stream (via *JSONServer* methods).
    - When a JSON message is received, an *DAPRequest* is made from it and the dispatcher is used to send the request to the right handler, returning a list of response messages.
    - The *DAPResponses* are sent back to the Client via the output stream as JSON messages.

Both the LSP and DAP server loops terminate when a “running” flag is cleared. This action is performed by the “terminate” message handler, and causes the LSP Server process to end cleanly.

3.2. Initialization and Termination of an LSP session

When an LSP Server is started, the first message it expects to receive is an “initialize” request:

- The initialize request is received by the *LSPServer* and dispatched, via the *RPCDispatcher*, to the *InitializeHandler.*
- The handler delegates the call to the *LSPPlugin’s* lspInitialize method.
- lspInitialize takes the “rootUri” argument, and uses it to load the files matching the dialect’s extension below that root into memory. The “capabilities” argument is also stored, allowing it to be queried later.
- Assuming there are no problems reading the files, a success message is sent back to the Client, which includes the (fixed) server capabilities.
- The Client responds with an “initialized” message, which is also handled by the *InitializeHandler* and delegated to the lspInitialised method on the *LSPPlugin*.
- The lspInitialized method sends a dynamic registration for “didChangeWatchedFiles” along with any syntax or type checking errors that are present in the specification.

Nothing then happens until the user starts to interact with the specification in some way. When the Client is finally terminated, messages are sent to the LSP Server to indicate that it should shut down:

- Firstly a “shutdown” message is sent, which is handled by the *ShutdownHandler.* This just sets a flag to say that the server is no longer considered initialized.
- Secondly an “exit” message is sent, which causes the server process to exit completely.

3.3. Initialization and Termination of a DAP session

Several DAP sessions can be started and stopped under one LSP session. Such a session is initialized when the user starts an execution or debugging session, and terminated when that session is completed. The initialization is handled as follows:

- The “initialize” message is read from the DAP port and dispatched to the (DAP) *InitializeHandler*, which delegates the call to the *DAPPlugin’s* dapInitialize method.
- The response to the initialize contains the server’s DAP capabilities and an “initialized” event.
- The Client can respond with several messages, typically including a “launch” message but possibly also including “setBreakpoint” messages. This information is handled by the *LaunchHandler* and *SetBreakpointHander* classes, delegating to the *DAPPlugin*.
- The Client eventually responds with a “configurationDone” message which is handled by the *InitializeHandler* and delegated to the configurationDone method on *DAPPlugin.*
- The configurationDone method creates an *InitExecutor*, which runs the VDMJ initialization of the specification and optionally also evaluates the “launchCommand” that was passed with the launch request.
- The *InitExecutor* runs in a *CancellableThread* in the background, and the configurationDone method returns success to the Client immediately.
- The *InitExecutor* sends a startup message to the Client stdout using DAP events.
- If the initialization runs for a long time, the Client may click the pause button, which sends a request that is handled by the LSP *PauseHandler.* This uses VDMJ to effectively set a transient breakpoint at the current location, which then trips into the debugger.
- But normally the initialization of the specification completes and a message is sent to stdout to give the result value and the time taken.

The Client console then sends a series of “evaluate” messages to the server, which parses and runs the commands concerned. Eventually, the execution session will be ended, usually by the user clicking some sort of close icon:

- The Client sends a “terminate” message to the DAP server, which is handled by the *TerminateHandler*.
- The handler requests that any cancellable threads are stopped, in case the user is trying to close a session while an expression is being evaluated. An “exit” event is then sent back to the Client, which includes an exit code (usually zero).
- The *DAPServer* then returns from its run method and allows the outer *DAPServerSocket* to loop, waiting for the next connection from the Client.

3.4. Evaluation of Expressions and Commands

Once a DAP session has been initialized, the user can enter multiple commands before closing the session with a “quit” command (or clicking the “stop” icon). There are several commands available, including a “help” command that lists them. All produce some output on the console. They are all handled in a similar manner:

- The user enters a command, like “print f(123)” on the console and presses enter. This sends an “evaluate” DAP command to the server, including the text of the line.
- The message is handled by the *EvaluateHandler,* which delegates to the *DAPPlugin’s* evaluate method.
- The *DAPPlugin* parses the command line passed and turns it into a *AnalysisCommand* object, subclasses of which implement each command supported. This is done by asking each of the configured plugins whether it supports the command typed.
- All commands can have a property called “notWhenRunning” set, which means that you cannot execute them when VDMJ is evaluating something else. Otherwise you can.
- If a *PrintCommand* is executed, the *DAPPlugin* further checks that the specification has no TC errors and has not changed in memory since the last save. If either are the case, the print gives an error response.
- Otherwise the command’s run method is called. This typically interacts with the VDMJ interpreter to set or list something, or it starts a new evaluation using an *ExpressionExecutor*. The *DAPMessageList* returned from the run method is returned to the Client, though often it only comprises stdout messages or errors.
- If the *ScriptCommand* is invoked, it uses a *ScriptExecutor* to read through a file of commands and execute each one in turn.

3.4.1. Comments

The user console is typically available from a Client, even when stopped at a breakpoint. In this case, expressions typed have to be very limited, and only support the *PrintCommand.*

3.5. Stopping at a Breakpoint or Exception

When a running specification encounters a breakpoint, or has an error or exception during an evaluation, it drops into the “debug mode” of DAP, which allows a Client to look at the state of the specification and its running threads.

- Every running specification is accompanied by a *DAPDebugReader* thread, which waits for the specification to hit a break or error, and then takes over the conversation with the Client.
- When a breakpoint is encountered, VDMJ uses a *DebugLink* object to communicate this change to the debug reader, which is waiting via the link’s waitForStop method.
- As each thread of the specification stops, *DAPDebugLink* sends a “stopped” event to the Client to say that it is at a breakpoint or stepping or being paused because another thread is at a breakpoint.

- When all threads have stopped, control is handed to the debug reader, which enters a loop waiting for Client messages. The client typically asks for threads and stack information for each thread, though it can also change breakpoints etc. These JSON messages are converted to *DebugLink* calls, and the responses turned into JSON responses for the Client.
- Note that in this mode, DAP message handlers have to detect that a debug reader is active, and pass messages to it rather than to the *DAPPlugin*. The debug reader has an “isListening” method that indicates that it is waiting for client commands, rather than waiting for the specification to trap into the debugger.
- If a DAP command resumes execution (including a step into/over/out), all of the VDMJ threads are resumed via the *DebugLink* and the debug reader goes back to waiting on the link for threads to stop again – in the case of a step command, this will be almost immediately.
- The Client can request that a debug session is terminated or even disconnected, and these are also enacted via the *DebugLink.*
- The handling of an exception is almost identical to that of a breakpoint, except the specification cannot be resumed.
- The “pause” and “terminate” buttons on the GUI (in VSCode) are treated very similar to breakpoints and exceptions.
- The handling of “Log message” breakpoints is different. The *DAPDebugReader* implements the VDMJ *TraceCallback* interface, which is called in-line when such a log-breakpoint is encountered. The callback just prints a line to stdout, evaluating the expression defined in the breakpoint text. But the execution is not paused and there is no other exchange with the Client.

3.5.1. Comments

VDMJ cannot mix multiple breakpoint types, although it can handle “hits” as well as “conditions”. So you cannot have a single breakpoint that stops (say) after 100 hits if x > 0. Similarly, VDMJ cannot mix trace messages with evaluations, so a “Log message” is always interpreted as an expression rather than a message string with possibly embedded expressions.

