Debugger API

Debugger

MPS provides a java api for creating a debugger engine to work with MPS.

Where to start

A good way to start writing a debugger is to study the Java Debugger plugin implementation in MPS. The source code is located in solutions jetbrains.mps.debugger.java.api and jetbrains.mps.debugger.java.runtime. The Debugger API itself can be found in jetbrains.mps.debugger.api.api. Solution jetbrains.mps.debugger.api.runtime can be studied to understand how api classes are used in MPS, but classes from this solution should not be referenced from a custom debugger code. Both Debugger API and Java Debugger are packaged as IDEA plugins for MPS, and so should be every implementation of Debugger API. Implementing a debugger for MPS requires some knowledge of Intellij IDEA, so Intellij IDEA Plugin Development page should also be studied. And there is of course a Build Language Guide to help with plugin packaging.

Key classes

Interface jetbrains.mps.debug.api.IDebugger represent a debugger as a whole. Should be registered in a jetbrains.mps.debug.api.Debuggers application component. There is a default implementation, AbstractDebugger. IDebugger implementation is required to provide 2 things: a way to start an application under debugger and a way to create breakpoints. For that MPS Debugger API has 2 interfaces:

1. jetbrains.mps.debug.api.AbstractDebugSessionCreator – its main job is to start an application under debug and create an instance of AbstractDebugSession which corresponds to a single application, executed under debug. A state of debugger (either running or paused on breakpoint or other) is represented by jetbrains.mps.debug.api.AbstractUiState. When debug session transfers from one state to another, a new instance of AbstractUiState should be created and set to AbstractDebugSession. AbstractUiState holds the information about current threads and stack frames, visible variables etc.
2. jetbrains.mps.debug.api.breakpoints.IBreakpointsProvider is responsible for creating breakpoints, providing editing ui and persisting breakpoints state to and from xml. A breakpoint class should implement IBreakpoint. AbstractBreakpoint is a base class to extend. For breakpoints that can be set on some location (which basically means "one a node", like a breakpoint on line of code) exists an interface ILocationBreakpoint. Each breakpoint is of some kind (IBreakpointKind), for example there can be kinds for line breakpoints, field breakpoints or exception breakpoints.

Values and watchables

jetbrains.mps.debug.api.programState package contains a number of interfaces to reflect a state of the program. When paused on the breakpoint AbstractUiState has a list of threads that are running in an application (interface IThread). Each thread has a list of stack frames (IStackFrame), and each stack frame has a list of watchables (IWatchable) visible there. A watchable can be a local variable, or "this" object or a "static context". Each watchable has one value (IValue). In turn each value can have a number of watchables. For example, java local variable is a watchable, an object assigned to this variable is a value, and fields that this object has are watchables etc. This tree like structure is what is shown in "Variables" section of "Debug" tool window.

SourcePosition and IPositionProvider

jetbrains.mps.debug.api.source.IPositionProvider is an interface to customize the way source code (text) is mapped into a location inside of MPS (a text file or a node). It calculates an instance of jetbrains.mps.debug.api.source.SourcePosition for a location either given by an instance of a ILocation interface or by name of a unit name, file name and line number. There are two predefined providers: jetbrains.mps.debug.api.source.NodePositionProvider and jetbrains.mps.debug.api.source.TextPositionProvider. They are responsible for finding a node or a text position for a location.

SourcePosition is a class that contains a position inside a MPS. This class has two implementations: jetbrains.mps.debug.api.source.NodeSourcePosition (contains a pointer to a node) and jetbrains.mps.debug.api.source.TextSourcePosition (contains a text file). Default line highlighting in MPS supports only these two kinds of positions.

Providers are registered in jetbrains.mps.debug.api.source.PositionProvider class. Each provider has a key, associated with it, which tell whether this provider returns a NodeSourcePosition or a TextSourcePosition. Also, a provider accepts a specific DebugSession, and only providers that are shipped with MPS debugger by default can accept all sessions.

Here is an example of position provider:

```
/*
NodePositionProvider is extended here in order to provide NodeSourcePosition.
It is not necessary to extend one of the default providers.
*/
public class MyCustomNodePositionProvider extends NodePositionProvider {
  private final Project myProject;

  public MyCustomNodePositionProvider(Project project) {
    myProject = project;
  }

  @Nullable()
  @Override
  public node<> getNode(@NonNls string unitName, @NonNls string fileName, int position) {
    node<> node = ... (some code that calculates a node);
    return node;
  }

  public void init() {
    // call this method on your debugger initialization
    PositionProvider.getInstance(myProject).addProvider(this, NodeSourcePosition.class.getName());
  }

  public void dispose() {
    // call this method on your debugger disposal
    PositionProvider.getInstance(myProject).removeProvider(this);
  }

  @Override
  public boolean accepts(AbstractDebugSession session) {
    // it is necessary to override this method and to accept only AbstractDebugSession instances that
are created by your debugger
    return session instanceof MyCustomDebugSession;
  }
}
```

TraceInfoResourceProvider

jetbrains.mps.generator.traceInfo.TraceInfoCache.TraceInfoResourceProvider is an interface that allows to customse how paths to trace.info files are calculated for a module. As an example, modules that are written on baseLanguage have their trace.info files in classpath. The instance of an interface is given a module and a name of resource to find (for example, "jetbrains/mps/core/util/trace.info" if a caller needs a location of trace.info file for a model jetbrains.mps.core.util) and should return an instance of java.net.URL. TraceInfoResourceProviders are registered/unregistered in TraceInfoCache instance using addResourceProvider/removeResourceProvider methods.

Changes in MPS 3.0

1. Variables tree in debugger tool window now uses a background thread for its updates. In order to support that, several methods were added to the debugger api:
    - IValue.initSubvalues() method which is called in background thread and should perform all the calculations necessary for acquiring subvalues of this value. The old IValue.getSubvalues() now only supposed to return the data which were calculated in the initSubvalues() method.
    - AbstractUiState.invokeEvaluation() method which is used by debugger api to invoke all evaluation code (i.e. the code which does some calculations inside the debugged program; specifically this method is used to invoke IVal ue.initSubvalues()). The default this method just schedules the command to one of the threads in the thread pool, but one can override it for some custom behavior.