Building RCP Apps with MPS

Markus Voelter

independent/itemis

Abstract.

1 Introduction

The term RCP is borrowed from the Eclipse ecosystem and stands for Rich Client Application. It refers to a stripped down version of the IDE that only contains those artifacts that are necessary for a given business purpose. In the MPS world it refers to a stripped down version of MPS that only contains those artifacts that are necessary to use a small domain-specific set of languages. Various aspects of MPS can be removed or customized to deliver a custom user experience for a given group of users. This article describes how. In particular, the following aspects of MPS can be customized:

- various icons, slogans, splash screens and images
- the help URL
- the set of languages available to the users
- the set of plugins available in MPS

2 Process Overview

To build a custom RCP version of MPS, you have to create a solution that contains a so-called *build script*. A build script is written in MPS' build language which is optimized for building RCP applications (as well as IntelliJ plugins and in the future, Eclipse plugins). When running the generator for this build script, MPS generates an `ant` file that creates the actual RCP distribution.

3 Building an example RCP build

In this document we describe the development of an RCP build script for the mbeddr project. mbeddr is a set of languages for embedded software development based on MPS. The project is available at `http://mbeddr.com`. It is Open Source, so all the code, including the RCP build script is available from the respository at `https://github.com/mbeddr/mbeddr.core/`. The project that contains the build script can be found in `/code/rcp-build/MPSIDE-build.mpr`.

3.1 Creating the Solution and the Build Script

In an arbitrary project create a new solution with a new model inside. In the model, configure as used languages

- `jetbrains.mps.build`
- `jetbrains.mps.build.mps`

Also import the `jetbrains.mps.ide.build` model. You can now create a new `build project` in the model (the name is confusing: it is not actually a new build *project*, just a build script). Fig. 1 shows the resulting, empty build script.

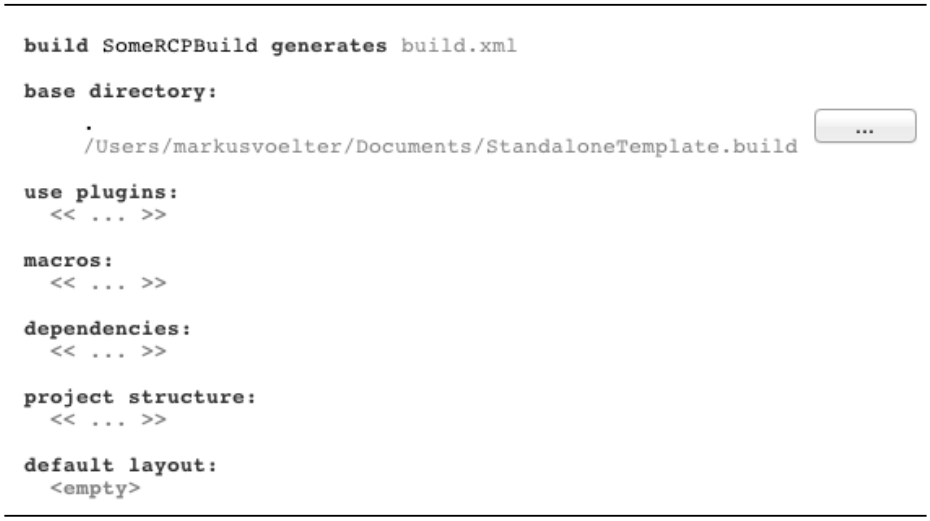

Fig. 1. An empty build script after directly after creation.

Let us look at the various sections of a build script:

base directory The base directory defines an absolute path relative to which all other paths are specified. By default, this is the directory in which the resulting `ant` file will be generated, and in which it will be executed.

use plugins The build language itself can be extended via plugins[1]. These plugins contribute additional build language syntax. Typically, the `java` and `mps` plugins are required. We will use syntax contributed by these plugins below.

[1] Note that these are *not* the plugins that make up MPS itself; those will be configured later.

macros Macros are essentially name-value pairs (similar to `${something}` in `ant`). In the Macros section, these names are defined and values are assigned. In the remainder of the build script these macro variables will be used. MPS supports two kinds of Macros: `var` macros are strings and can be assigned any value. `folder` represents paths, relative to the base directory defined above and/or based on other Macros. Note that MPS provides code completion for the path components in `folder` macros.

dependencies This section defines dependencies to other build scripts. MPS bundles a set of build scripts (e.g. `buildStandalone`, `buildWorkbench` or `buildMPS`). By establishing a dependency to any one of them, the structures defined in that references build script can be used in the referencing build script. For example, the macros defined in the referenced build scripts can be used.

project structure This section defines the actual contents of the to-be-built RCP application. Such contents may be MPS modules (languages, solutions, devkits), Java libraries, IDEA branding and plugins.

default layout This section defines the files that are output, it creates the directory, file and JAR structure of the new RCP distribution. It references and includes the artifacts defined in the project structure section and in other build scripts this one depends on.

3.2 Building the script for mbeddr

Macros We start by defining a couple of macros. The first two represent string variables we will use in the build, the set of folders represent the directory for MPS itself as well as the two root directories where mbeddr languages are stored. These languages will become part of the RCP distribution. Note how all of these folders are relative to the base directory.

```
macros:
  var date = date 20120601
  var build.number = 1.0
  folder mps_home = ./../../../tools/MPS2.5.app
  folder mbeddr.core.home = ./../languages/com.mbeddr.core
  folder mbeddr.mpsutil.home = ./../languages/com.mbeddr.mpsutil
```

Note that you can leave a macro undefined (i.e. just specify the name) and then define it when you call `ant` by using a property definition:

```
ant -Dsome.variable=a.value
```

Dependencies Next, we define the dependencies. We need dependencies to `buildStandalone` (it represents the minimal set of a standalone MPS installation), `buildDebuggerPlugin` (because mbeddr requires debugger integration) and `buildExecutionPlugin` (because one the mbeddr languages uses the execution framework to run an external executable). Most RCPs will likely just use `buildStandalone`.

```
dependencies:
  buildStandalone (artifacts location $mps_home)
  buildDebuggerPlugin (artifacts location $mps_home)
  buildExecutionPlugin (artifacts location $mps_home)
```

The `artifacts location` specifies from where the artifacts will be copied. The build scripts work by copying code from an existing MPS installation, so you don't have to check out the MPS sources. The `artifacts location` should hence be pointing to the MPS home directory.

Project Structure In the project structure we start with the branding. The `idea branding` section supports the definition of all kinds of icons, splash screens and the like. It is a good idea to take those from the MPS distribution and modify the pictures to suit your needs, while retaining the sizes and transparency values. You can also specify a help file and an update website.

TODO(*Explain how to hook in a help file; not clear what the three arguments mean*)

```
idea branding MBEDDR
  codename MBEDDR
  version 1.0, eap false
  full name mbeddr-IDE
  build number ${build.number}, date ${date}
  icons
    16x16 ./icons/MPS_16.png
    32x32 ./icons/MPS_32.png
    32x32 opaque ./icons/MPS_32.png
  splash screen ./icons/splash.png textcolor 002387
  about screen ./icons/mpsAbout.png
  dialog image ./icons/mpsNewProject.png
  welcome screen
    caption ./icons/mpsWelcomeCaption.png
    slogan ./icons/mpsSlogan.png
  <no updateWebsite>
  <no help>
```

In the mbeddr case we have copied the MPS icons into the `./icons` folder and changed them accordingly without changing the names — this is why most of them start with `mps`.

Next we define an `idea plugin` that contains the mbeddr languages. An `idea plugin` is a plugin to the IDEA platform on which MPS is built.

```
idea plugin MBEDDRStuff
  name mbeddr
  short (folder) name mbeddr
  version 1.0
  content:
    mbeddr.core
    mpsutil
  dependencies:
    jetbrains.mps.core
```

You can define a name and a version, dependencies to other plugins **TODO**(*why do we need the mps.core thing here?*) as well as the actual contents. Whenever your RCP contains languages, you will need `jetbrains.mps.core` because it provides things like `BaseConcept` or the `INamedConcept` interface. The `mbeddr.core` and `mpsutil` entries in the contents are references to `mps groups` defined below. An `mps groups` is a group of MPS artifacts. Since we have included the `mps` plugin (at the very top of the build script) we can use MPS-specific language constructs. `mps groups` are an example.

Let us look at the `mpsutil` group. It references a set of languages. A `language` reference points to a language defined in an MPS project. A language reference in a group consists of the name (which must be the same as the name of the actual language) and a pointer to the respective `mpl` file. The simplest way to enter it is to start with the path to the `mpl` file and then use the `load required information from file` intention to adjust the name of the reference.

```
mps group mpsutil
  language com.mbeddr.mpsutil.bldoc
    load from $mbeddr.mpsutil.home/languages/com.mbeddr.mpsutil.bldoc/bldoc.mpl

  language com.mbeddr.mpsutil.blutil
    load from $mbeddr.mpsutil.home/languages/com.mbeddr.mpsutil.blutil/blutil.mpl

  ...
```

Note that a group has to contain the transitive closure of all languages. For example, if you have references a language A which references another language B, then B must also be included in the group. This makes sense, because otherwise the resulting RCP application would not run because dependencies could not be resolved. If a language is missing, then an error will be reported in the build script (red squiggly line)[2].

In addition to languages, you can also reference solutions with the `solution` construct and devkits with the `devkit` construct.

[2] After adding the required languages you may have to rerun the `load required information from file` intention on the language with the error to make it "notice" that something has changed.

Default Layout The layout constructs the directory and file structure of the RCP distribution. It copies in files from various sources, and in particular from the project structure discussed in the previous section. It also also compiles, builds and packages these files if necessary. It can also refer to artifacts defined in other build scripts on which this script depends. We start out with the following code:

```
default layout:
  import buildStandalone::languages
  import buildStandalone::license
```

Those two imported are `folder` elements in the referenced `buildStandalone` build script. By importing them, the contents of these folders are imported (i.e. copied into) our RCP distribution.

Next up, we create a new folder `lib` into which we import all the stuff from `from buildStandalone::lib` except the MPS sources and the existing `branding.jar` (which contains the default MPS splash screen etc.). We then create a new jar file `branding.jar`[3] into which we put all the files defined in the `branding` section of the project structure defined above.

```
folder lib
  import files from buildStandalone::lib
    exclude MPS-src.zip
    exclude branding.jar
  jar branding.jar
    files of idea branding MBEDDR
```

We then create a folder `plugins`. The `mps-core` plugin is required in any RCP app, so it needs to be imported in any case. We then import various version control plugins (CVS, SVN anf git) defined in the `buildStandalone` build script. For obvious reasons, a standalone mbeddr IDE needs version control support, so it makes sense to import those plugins. Finally we include the debugger plugin, since mbeddr also integrates with the MPS debugger.

```
folder plugins
  import buildStandalone::plugins/mps-core
  import buildStandalone::plugins/cvsIntegration
  import buildStandalone::plugins/git4idea
  import buildStandalone::plugins/svn4idea
  import buildDebuggerPlugin::plugins/mps-debugger-api
  import buildDebuggerPlugin::plugins/mps-debugger-api/lib/debugger-api.jar
```

TODO(*Why do we have to have the last line? It is necessary to not run into an exception, but AFAIU, we shouldn't need it.***)**

[3] It has to have this name to work; the IDEA platform expects it this way.

We then integrate the MBEDDR plugin defined in the project structure earlier. Using the `plugin` construct we can import the complete definition of the plugin defined earlier.

```
plugin MBEDDRStuff
```

Inside the plugin we define further subdirectories that contain various library jars. These jar files are automatically made visible to the plugin classpath **TODO**(*How?*). Using the `file` construct we copy various files into the plugin directory, using the macros defined earlier.

```
plugin MBEDDRStuff
  folder runtime-libraries
    file $mbeddr.core.home/languages/com.mbeddr.core.modules.runtime/lib/commons-io-2.1.jar

  folder zvtm-libraries
    file $mbeddr.mpsutil.home/languages/com.mbeddr.mpsutil.graphview/lib/antlr-2.7.7.jar
    file $mbeddr.mpsutil.home/languages/com.mbeddr.mpsutil.graphview/lib/commons-io-2.1.jar
    ...

  folder debugger-libraries
    file $mbeddr.core.home/languages/com.mbeddr.core.debug/lib/mi/cdt.jar
    file $mbeddr.core.home/languages/com.mbeddr.core.debug/lib/mi/cdt2.jar
    ..
```

Note how this plugin does not just contain the jars we copied in, but also all the language artifacts necessary. That has been accomplished by defining these `mps groups` and referencing them from the plugin. The fact that we can use these `mps groups` is thanks to the `mps` build language plugin. It provides direct support for MPS artifacts, and all the necessary files and metadata are handled automatically.

Property Files Finally we create a property file called `build.number` that contains a bunch of meta data used in the running RCP application. It is required by the underlying IntelliJ platform.

```
properties file build.number
  build.number = ${build.number}
  date = ${date}
  version = 1.0
```

3.3 Generating and running the Build Script

Assuming your build script is free of errors, you can generate it (`Rebuild Model` or `Ctrl-F9`). This creates an `ant` file that has the name that is mentioned at the top of the build script:

```
build MBEDDR-IDE generates mbeddr.xml
```

This `mbeddr.xml` (in this case) resides directly in your project directory[4]. You can either run the file from the command line (using `ant -buildfile mbeddr.xml`) or directly from within MPS (via the context menu on the build script in the project explorer).

Note that the build script compiles and packages all your languages, but it does *not* regenerate them[5]! So you have to make sure by other means that your language definitions are fully generated and up to date.

3.4 Insepcting the Generated Directory Structure

Fig. 2 shows the project structure after running the build script. In particular, inside the `build/artifacts` directory, the structure defined in the `default layout` section of the build script has been created.

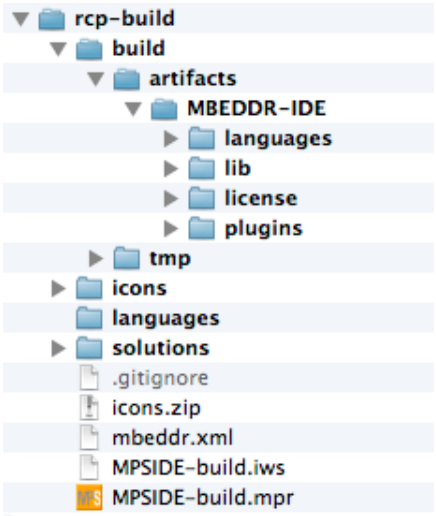

Fig. 2. The directory structure generated from running the genertated `ant` file.

We suggest you browse through the directory and file structure to connect what you see to the definitions in the build script.

[4] If you do not specify a file name it defaults to `build.xml` so it can be run by ant without using the `-buildfile` option.

[5] this will change in an upcoming release

3.5 Creating the Platform-Specific Distributions

At this point, the generated directory structure consitutes the platform indepenent core of a stripped down MPS distribution that contains all the artifacts you've configured into it, including your custom branding. However, you cannot run it, since the platform specific aspects are still missing.

TODO(*Here I am still waiting on the improved build scripts from Evgeny that handle the platform-specific part.*)

TODO(*also, the mps-make plugin still has to be manually copied - bug?*)