Packaging Language (obsolete)

Packaging Language

Introduction

When we think about building a project, we often do not think about a process of building, we usually imaging the result we want to have, for example, an archive with a program, documentation and libraries. We want a tool, which could take a description of the desired artifacts and generated a build script, for example an Ant file. That is one of the reasons why packaging language was made. The second reason for creating this language was lack of ability of simple deployment procedure for languages created in MPS.

Language Structure

This section describes the packaging language structure.

Defining Build Structure

Each packaging language script consists of two sections. The first one is a place for build properties, like base directory, and some build stuff, like variables or configurations. The second part is used for defining a build structure itself.
Let's talk more about the second part.

The build structure is written exactly like it should be in the resulting program distribution. There are several language elements you can use to describe it, for example, `Folder`, `Zip`, `Jar` and others (see all language elements). Some element can contain others, like in your distribution an archive could contain files in it.

```
project build script

basedir mps_home / << ... >>

generate compilation script false

configurations default

variables
  << ... >>

zip project.zip
  folder project from <no sourcePath>
    folder <no title>  from basedir / source
      file <no title>  from basedir / build.number
```

On the above screenshot you can see an example of packaging language script. In the build structure part a `Zip` element `project.zip` contains `Folder` element `project`, which contains `Folder` element copied from folder `sources`, and this element also contains `File` copied from `build.number`. This means that we want a build script to create a zip archive `project.zip` with a folder `project`, copy `sources` folder with a copy of `build.number` inside into `project` folder. As you can see, the definition is very literal.

Language elements, used in build structure description, are listed in language elements table. Next sections are considering some special features of packaging language.

Build Configurations

Sometimes build should be done in several slightly different ways. For example, we want to create two types of build: for other developers – with sources and some additional tools and for users. Packaging language allow to deal with this problem using build configurations. Let's see how it is done for already mentioned example.

```
Project layout

basedir $project.home$ set false

generate compilation script false

configurations dev, external

variables
  << ... >>

folder project from . / << ... >>
  folder <no title> from . / devtools include in dev
    <entries>
  folder <no title> from . / languages
    <entries>
  folder <no title> from . / src include in dev
    <entries>
```

On the above screenshot a build script example is shown. Configurations are defined in `configurations` sections. As it can be seen, the script defines two configuration: `dev` and `external` for two build types. Some build elements marked with them. Elements, which are not marked at all, are included in all types of build. Those elements are `Folder` "project" and `Folder` "lang uages". Other elements marked with configuration `dev`. They are included in build for developers and not included in build for users. Those elements are `Folder` "devtools" and `Folder` "src".

Variables

Variables allow to use properties, passed to the script through command line, in elements names. Variables are declared in `var iables` section of the build script. To declare a variable, you must define it's name and a name of ant property, which would be passed to the script by your build environment. On the below screenshot you can see an example of how variables declarations look like.

```
variables
  var build = ${ build.number }
  var revision = ${ build.vcs.number }
  var configuration = ${ teamcity.buildConfName }
```

The shown section defines three variables: `build`, `revision` and `configuration`.

Variables can be used in titles of all elements in script. For example, having variables, defined on a previous screenshot, one can write a script like shown on a screenshot.

```
zip MyProject-build.zip
  echo build.number=build\ndate=date\nrevision.number=revision\nconfiguration.name=configuratic
```

In this script a build number is used in a name of a zip archive with the project and build parameters are written into a file `bui ld.info`.

You can use not only variables, defined in the script, but also a number of predefined variables which are listed in the following table.

Variable Name	Ant Name	Description

basedir	basedir	Base directory of the script.
date	DSTAMP	Current date.
\n	line.separator	System line separator.
/	file.separator	System file separator.
:	path.separator	System path separator.

Macro

Some packaging language elements, such as `folder` or `copy` allow to enter a source path – a name of some file or directory. In many cases leaving those paths absolute would be a bad idea. That is why packaging language afford an opportunity to use macro in names of files and directories.

Two types of macro exists: path variables defined in `"Settings" -> "IDE Settings" -> "Path Variables"` and predefined macro: `basedir` – a base directory of build script (specified in the script beginning) and `mps_home` – an MPS installation directory. Build script base directory definition also allow usage of macros but with exception of `basedir`.

Macro used in a build script, will be generated to checked external properties of build language. This means, they will be checked in a build start and a build would fail, if their values were not specified.

Of course, packaging language does not force users using macro. To enter an absolute path you can select `no macro` item from macro menu.

Blocks

Blocks were introduced for structuring large builds. Essentially, a block is a part of build, a group of build elements. A block could be referenced from the main script. During generation, a contents of a block is copied to places if is referenced from. On the following screenshot there is a block defining a part of MPS build script.

```
block core.debug used in everywhere

  folder debug from <no sourcePath>
    module jetbrains.mps.debug.evaluation
    module jetbrains.mps.debug.customViewers
    module jetbrains.mps.debug.privateMembers
    module jetbrains.mps.debug.apiLang
```

Apart from a title and contents, a block can define which build script it is referenced from. This way variables from main build script could be referenced from the block.

Language Elements Table

This table gives a brief description of packaging language elements.

Element	Notation	Description
Antcall	`antcall [target declatation] from [project] <excludes patterns> <includes patterns>`	A call of an ant target. • target declaration – a link to declaration of target to call. • project – an build language concept in which to look for the specified target. • patterns – a coma- or space-separated patterns to exclude or include (can be omitted).
Block Reference	`-> [block name]`	A reference to a block. One could use blocks to split big scripts in parts.
Copy	`copy from [source] <excludes patterns> <includes patterns>`	A copy of folder contents. • source – a folder, which contents needs to be copied. • patterns – a coma- or space-separated patterns to exclude or include (can be omitted).

Echo	`echo [message] > <title>`	Echoes some message to the output stream or to the file. • title – file name (can be omitted). • message – a message to echo.
File	`file <title> from <source>`	A file. • title – a name of file. • source – a file to copy One of those parameters can be omitted.
Folder	`folder <title> from <source> <excludes patterns> <includes patterns>` `  <entries>`	A folder. • title – a folder name. • source – a folder to copy. One of those parameter can be omitted. • entries – a list of elements located in this folder (can be omitted). • patterns – a coma- or space-separated patterns to exclude or include (can be omitted).
Jar	`jar [title] <excludes patterns> <includes patterns>` `  <entries>`	A jar archive. • title – an archive name. • entries – a list of elements located in this jar archive (can be omitted). • patterns – a coma- or space-separated patterns to exclude or include (can be omitted).
Module	`module [name]`	A packaged language, solution or descriptor. • name – module name.
Replace	`replace <title> from [source]` `  <pairs token-value>`	Make replacements in file. • title – a new name of resulting file. • source – a source, from which file need to be taken. • pairs token-value – pairs to replace.
Plugin	`plugin <title> from [source]`	Idea plugin jar. • title – plugin title. If omitted, same as source folder name. • source – a folder where plugin files a located. Classes of the plugin should be located in `source/classes` and `plugin.xml` file in `soucre/META-INF` directory. Only classes and `plugin.xml` are packed in the jar.
Zip	`zip [title] <excludes patterns> <includes patterns>` `  <entries>`	A zip archive. • title – an archive name. • entries – a list on elements located in this zip archive (can be omitted). • patterns – a coma- or space-separated patterns to exclude or include (can be omitted).

Using Packaging Language

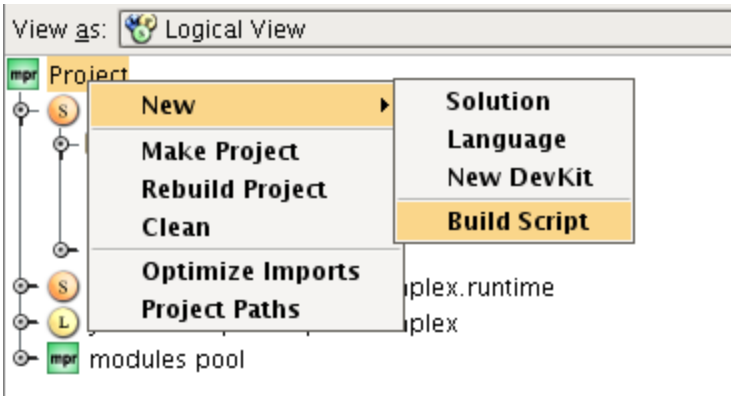
This section describes usage of packaging language.

Generating Build Scripts For Your Project

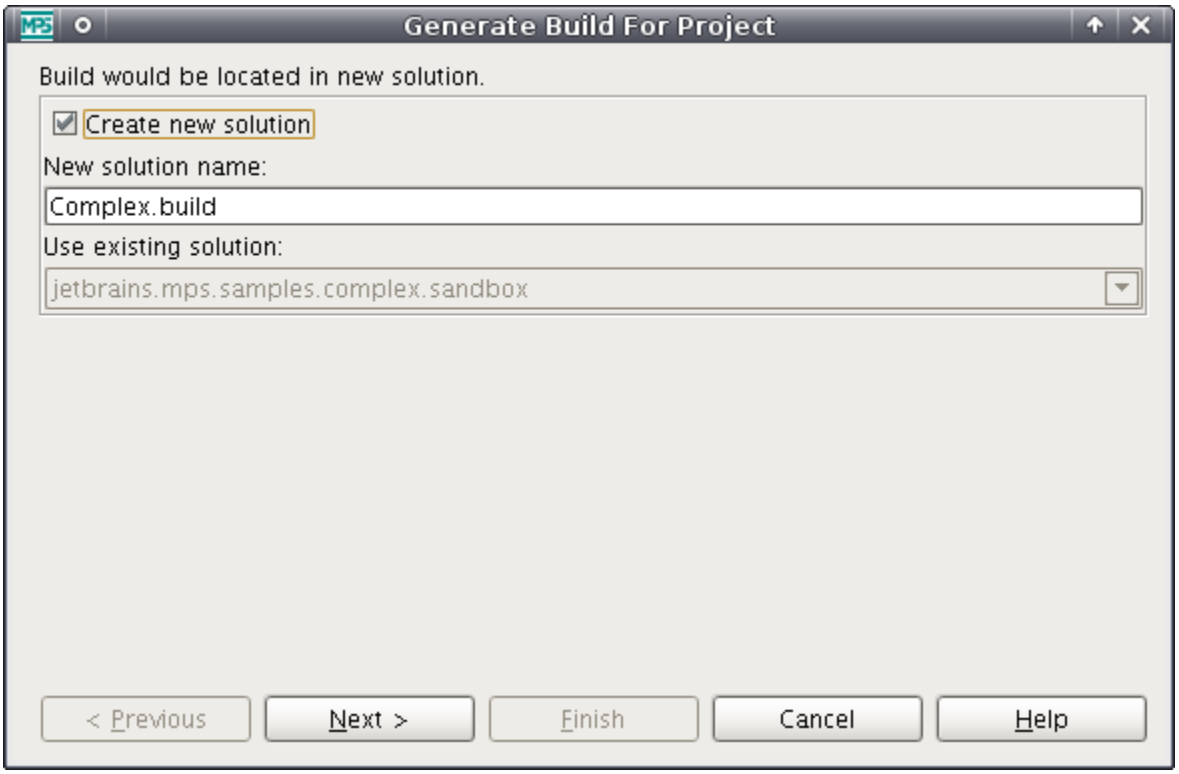
MPS allow to easily generate a build script on a packaging language for languages in your project. This functionality is very useful when you have a really big project and you do not want to enumerate every module name by hands.

Let's see, how it is done using complex language as an example.

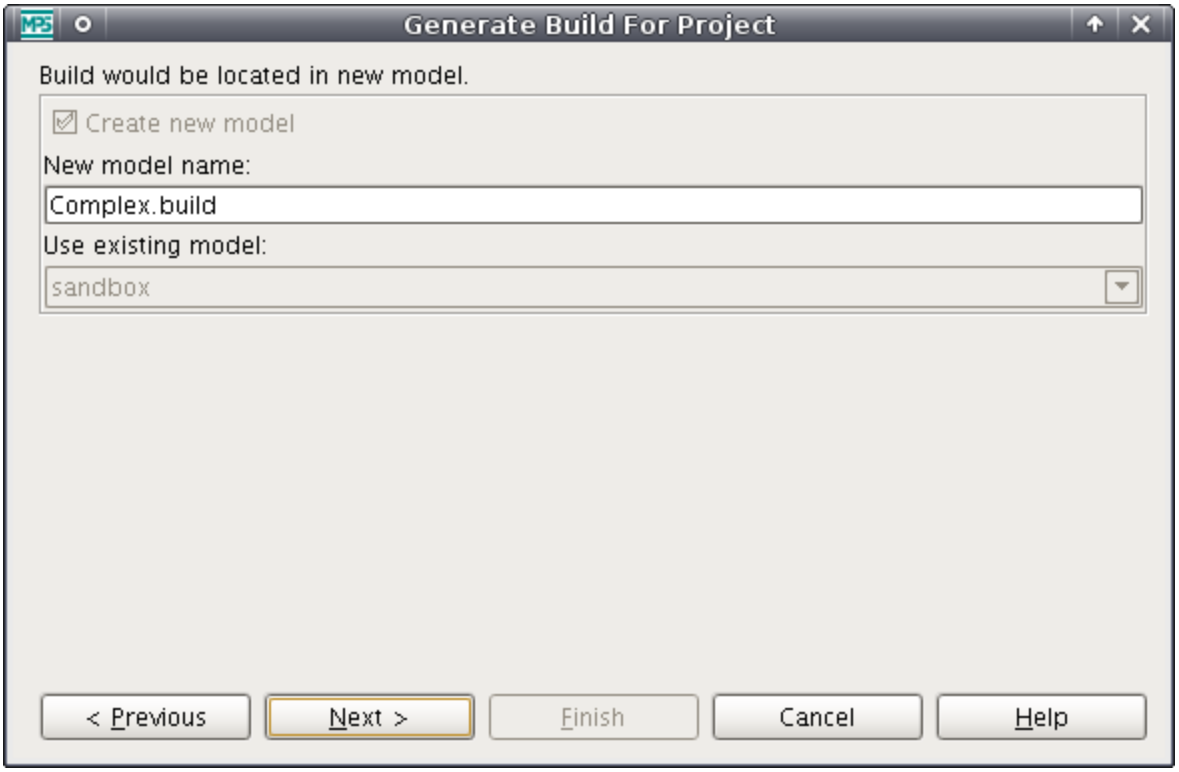
Let's open a complex language project which is located in file `%MPS_HOME%/samples/complexLanguage/Complex.mpr`. A first step in generating build process is calling a build generation wizard, by selecting `"New" -> "Build Script"` in project popup menu.



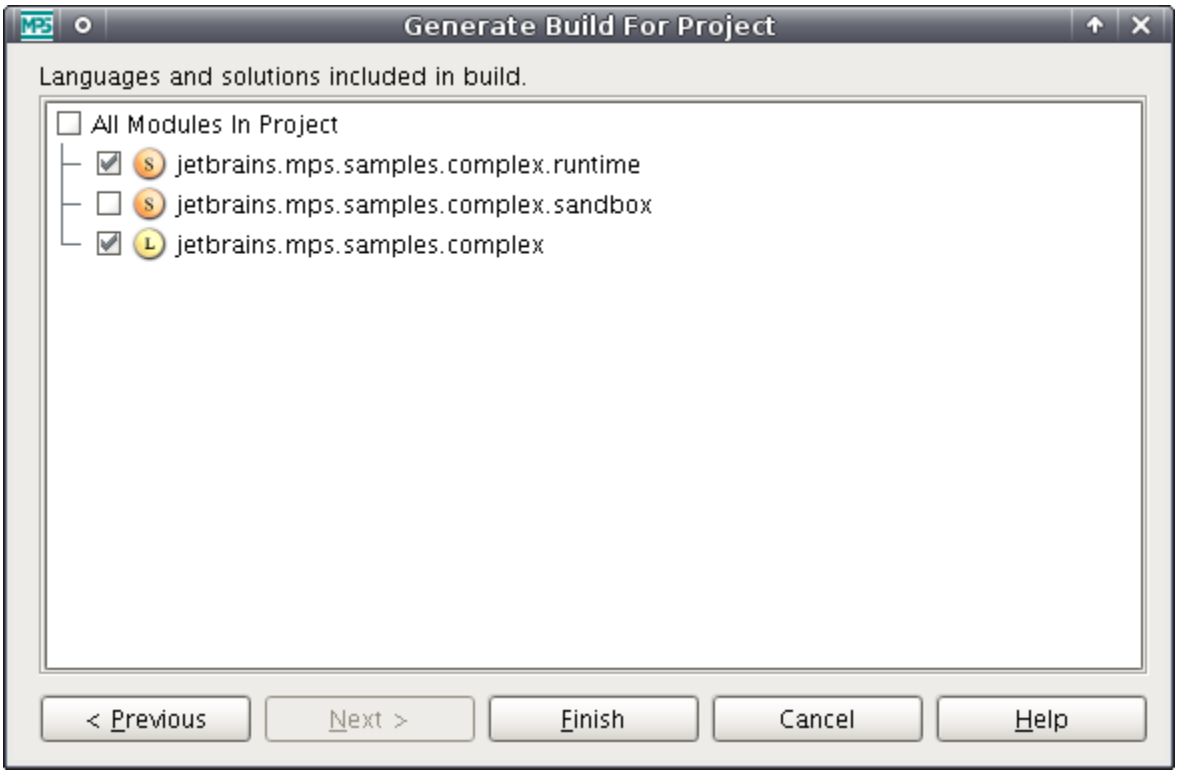
At first, the wizard asks to select, whether a new solution would be created for a build script, or an old one should be picked. For this example we select to create a solution named `Complex.build`.



The next step is to choose a model for a build script. If we selected an existing solution on previous step, we could use one of the models in that solution. But as we create a new solution for our build script, we have only one option: to create a new model. Let's call the new model `Complex.build`.



The last step is to select modules to include into build. We choose only two modules: the language `jetbrains.mps.samples.complex` and it's runtime solution `jetbrains.mps.samples.complex.runtime`.

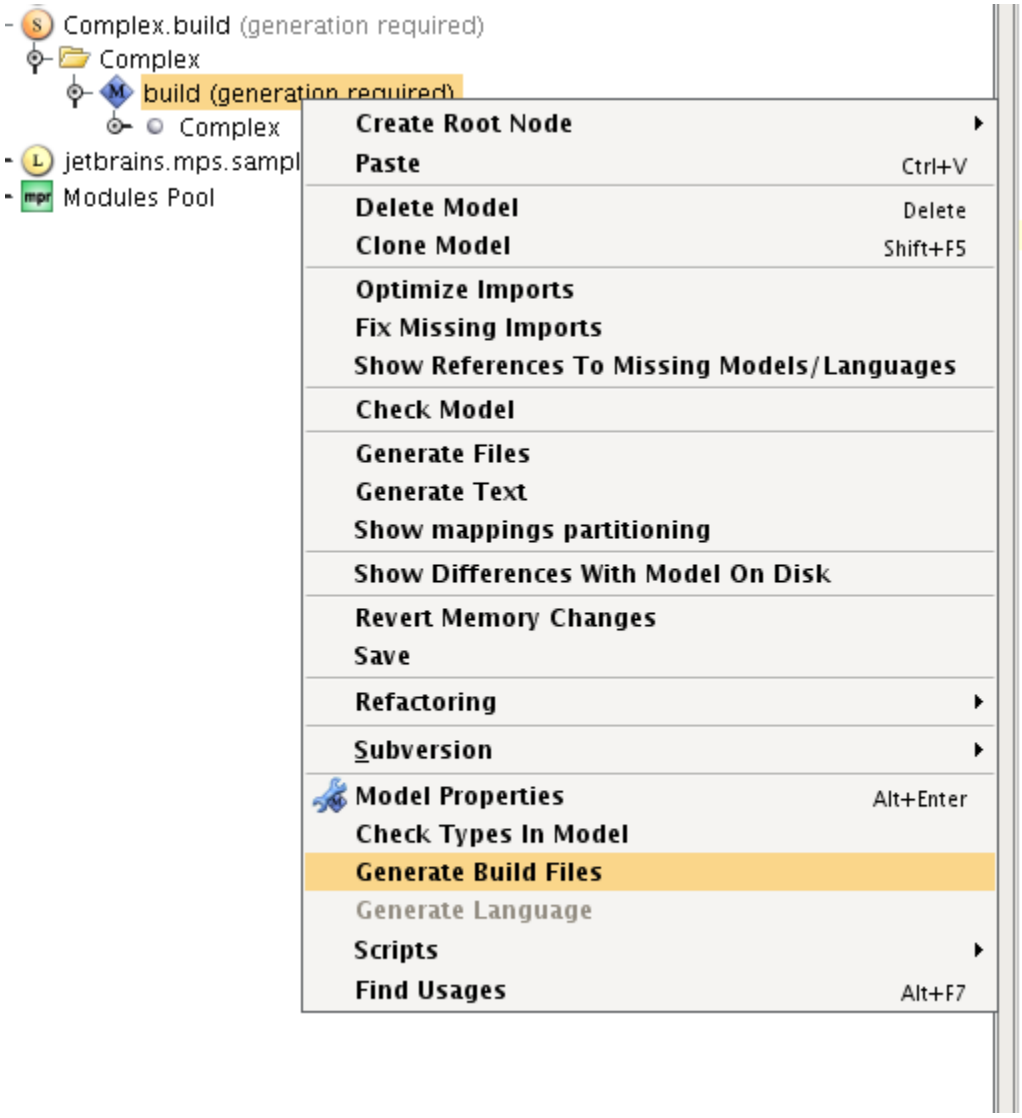


After pressing finish button, a resulting script is opened. Here is a build for complex language project.

```
Complex build script

basedir mps_home / samples / complexLanguage

generate compilation script true

configurations default

variables
  << ... >>

zip Complex.zip
  folder Complex from <no sourcePath>
    module jetbrains.mps.samples.complex
    module jetbrains.mps.samples.complex.runtime
```

Generating Files From Build Script

For generating files from a build script a "`Generate Build Files`" action can be used.

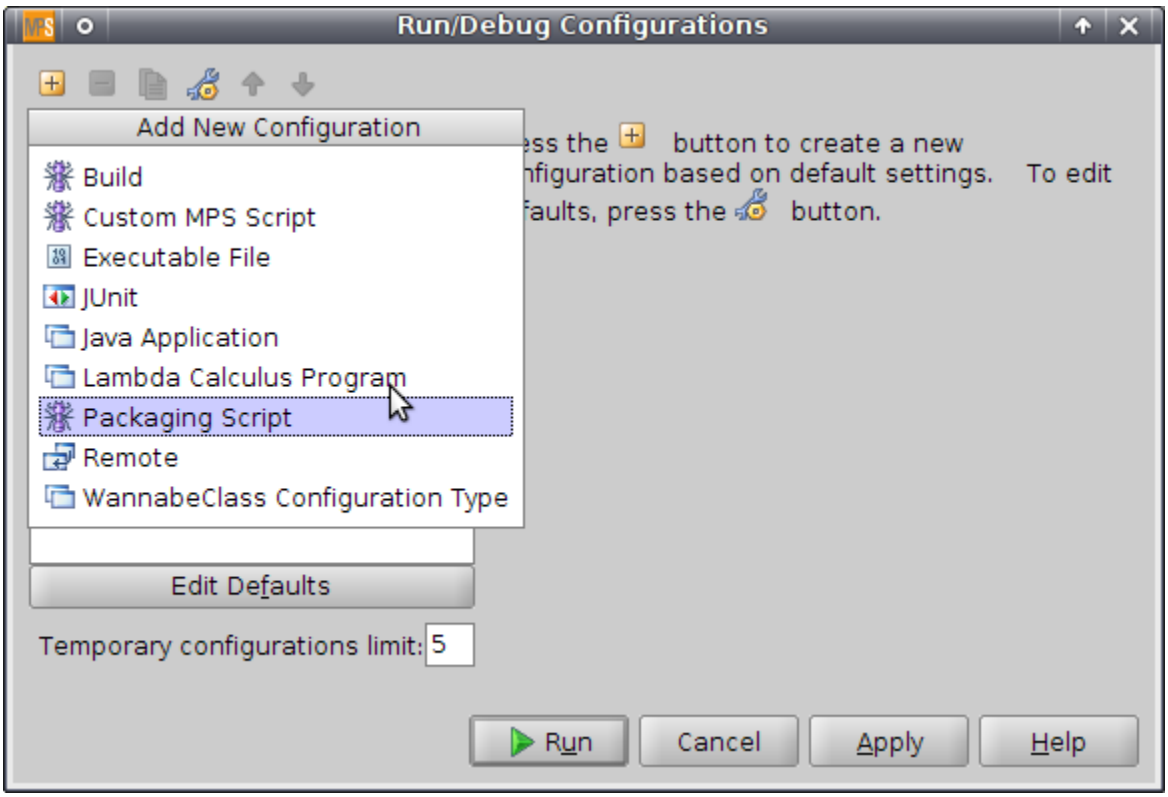


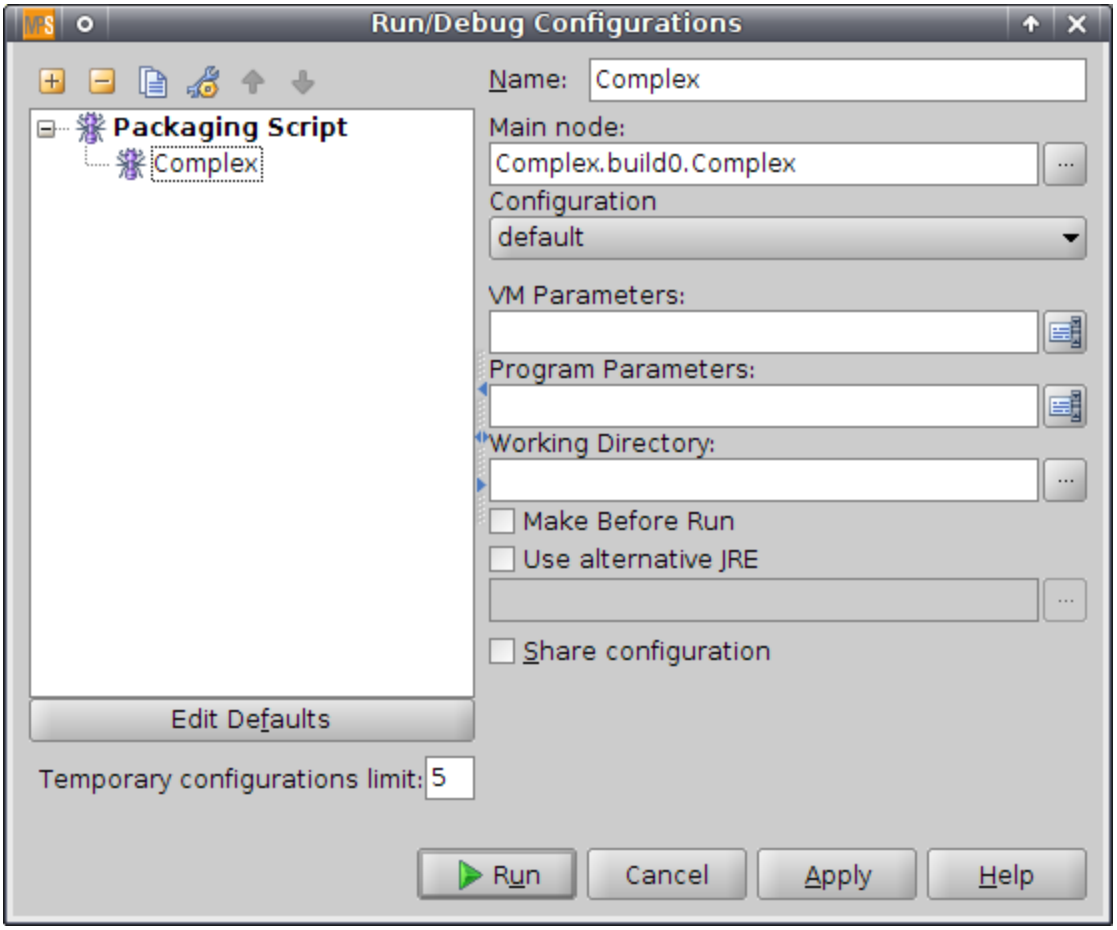
It generates the model with build and place output files to build script's base directory, so the files a ready to use. For complex language build script, created in previous section, those files would be:

File	Purpose
Complex-default.xml	The main script for building `default` configuration.
Complex-compile.xml	Compilation script.
Complex-languages.xml	Script for packaging modules into jars.
Complex.properties	Property file.

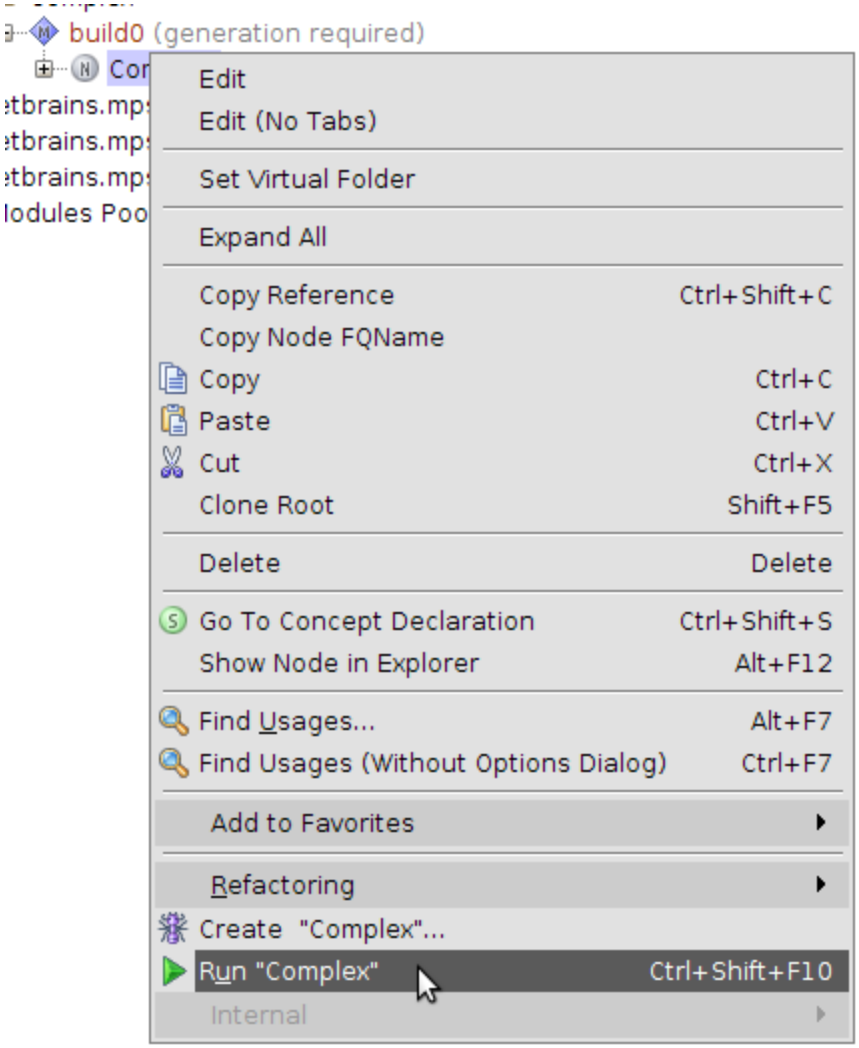
Running Build Scripts

To run build scripts from MPS it has a run configuration `"Packaging Script"`. It could be created via `"Run/Debug Configurations"` dialog:

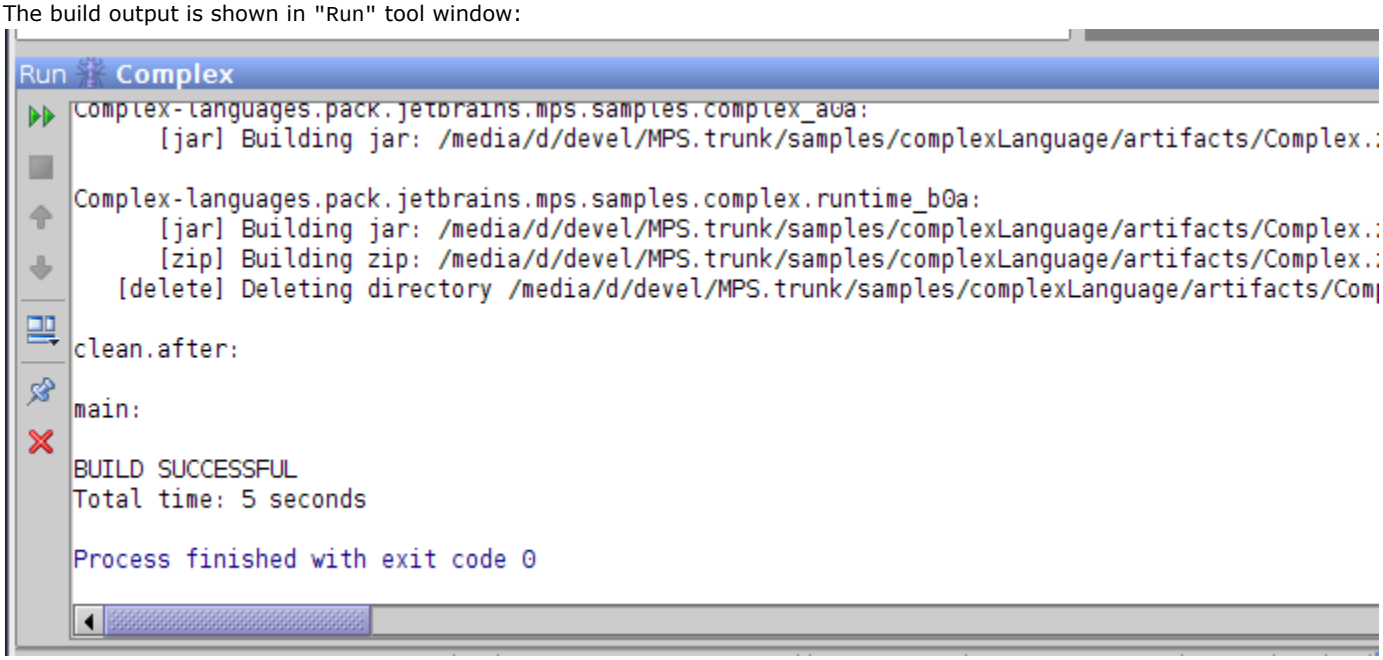




Build script could also be runned from context menu:



The build output is shown in "Run" tool window:



Previous Next