Build Language

What is MPS build language?

Build Language is an extensible build automation DSL for defining builds in a declarative way. Generated into Ant, it leverages Ant execution power while keeping your sources clean and free from clutter and irrelevant details. Organized as a stack of MPS languages with ANT at the bottom, it allows each part of your build procedure to be expressed at a different abstraction level. Building a complex artifact (like an MPS plug-in) could be specified in just one line of code, if you follow the language conventions, but, at the same time, nothing prevents you from diving deeper and customize the details like file management or manifest properties.

As with many build automation tools, project definition is the core of the script. Additionally, and unlike most of the other tools, Build Language gives you full control over the output directory layout. The expected build result is defined separately in the build script and not as a part of some (third-party) plugin.
Every build script is made up of three parts. The first is dependencies, something required that comes already built. Think of libraries or third-party languages, for example. Next is the project structure. It contains declarations of everything you have in your repository and what is going to be built, as well as the required build parameters. Note that declaring an item here does not trigger its build unless it is needed, i.e. referred to from the last part of the script - the output layout. The output could be as straightforward as a set of plain folders and copied files, or much more complex with zipped artifacts such as packaged plug-ins or MPS languages. For example, to build a jar file out of Java sources you need to declare a Java module in the project structure and the respective jar file with a reference to the module in the output layout.

Thanks to MPS, Build Language comes with concise textual notation and an excellent editing experience, including completion and on-the-fly validation. Extension languages (or plugins if we stick to the terminology of the other build tools) add additional abstractions on top of the language. In our experience, it is quite an easy process to create a new one compared to developing Maven or Gradle plugins.

Build script structure

See an example below of a build script, which builds a plugin for Intellij IDEA:

```
build Complex generates build.xml

base directory: ../../                                                    ...
                /home/julia.beliaeva/MPSSamples.3.0/complexLanguage

use plugins:
  java
  mps

macros:
  folder idea_home = ./../../soft/devel/idea-IC-123.178
  folder plugins_home = ./../../devel/mps/mps.build/IdeaPlugin/build/artifacts/mpsPl

dependencies:
  IDEA (artifacts location $idea_home)
  mpsPlugin (artifacts location $plugins_home)

project structure:
  idea plugin Complex
    name Complex
    short (folder) name Complex
    description <no description>
    version 1.0
    << no vendor >>
    content:
      Complex
    dependencies:
      jetbrains.mps.core

  mps group Complex
    solution jetbrains.mps.samples.complex.runtime
      load from ./solutions/jetbrains.mps.complex.runtime/jetbrains.mps.samples.comp

    solution jetbrains.mps.samples.complex.library
      load from ./solutions/jetbrains.mps.samples.library/jetbrains.mps.samples.libr

    language jetbrains.mps.samples.complex
      load from ./languages/complex/jetbrains.mps.samples.complex.mpl


default layout:
  zip Complex.zip
    plugin Complex
      <empty>

<<additional aspects>>
```

Let's look at it closely. The header of the script consists of general script information: the name of the script (Complex in the screenshot), the file it is generated into (build.xml) and the base directory of the script (in the screenshot one can see it is relative to the script location ../../ as well as full path).

The body of the script consists of the following sections:

- use plugins contains a list of plugins used in the script. Plugins in Build Language are similar to those in Gradle: they are extensions to the language that provide a number of tasks to do useful things, like compiling java code, running unit tests, packaging modules, etc. In the screenshot two plugins are used: java and mps, which means that the script

can build java and mps code.
- macros section defines path macros and variables used in the project (idea_home and plugins_home) together with their default values, which could be overridden during execution of the script.
- dependencies defines script dependencies on other build scripts. If a script references to something defined in the other build script it must specify this script in the dependencies section. The example script on the screenshot depends on two other scripts IDEA and mpsPlugin. These are provided by MPS, so in order to use them one has to specify the artifacts location for them, i.e. place where ant can find the result of their work (in the example, idea_home should point to the location of Intellij IDEA jars and plugins_home should point to the location of MPS plugins for IDEA). One can as well depend on some build scripts in the same MPS project. In that case, artifacts location is not required and it is assumed that the required script would be built just prior to the current script (there is a target buildDependents to do so).
- project structure section contains the description of the project, i.e. which modules does it have, where the source code is located, what the modules classpath is etc. The example project in the screenshot consists of a single idea plugin na med Complex and of a group of MPS modules.
- default layout defines how to package the project into the distribution. The example project on the screenshot is packaged into a zip file named Complex.zip.
- additional aspects defines some other things related to the project, for example, various settings, integration tests to run, and so on.

Macros

There are two types of macros:

- Folder - represents a physical folder in the file-system, if left empty (default), the value is attempted to retrieve from a path variable defined in MPS with the same name
- Var - custom variable representing a value

The Vars can be initialized in one of several ways:

- reference to an earlier macro
- date - a date value, a pattern parameter that follows the Java's date format rules should be specified, e.g. yyyy-MM-dd, the date value of the time when the build script is run will be inserted into the macro
- load from file - reads a specified property value from a specified property file
- text - a plain text value

Built-in plugins

Build Language provides several built-in plugins.

Java plugin

The Java plugin adds capability to compile and package java code. Source code is represented as java modules and java libraries.

Java module defines its content (source folders locations) and dependencies on other modules, libraries and jars. In content section java module can have:

- folder – a path to source folder on disk;
- resources – a fileset of resources. Consists of a path to resources folder and a list of selectors (include, exclude or inclu des).
- content root – a root with several content folders.

In dependencies section java module can have:

- classpath – an arbitrary xml with classpath;
- external jar – a jar file from other build script layout;
- external jar in folder – a jar file referenced by name in a folder from some other build script layout;
- jar – a path to local jar;
- library – a reference to a java library;
- module – a reference to a java module.

Each java module is generated into its own ant target that depends on other targets according to the source module dependencies. For compiling cyclic module dependencies, a two-step compilation is performed:

1. A "cycle" target compiles all modules in the cycle together.
2. Each module in the cycle is compiled with the result of compilation of "cycle" target in classpath.

Java library consists of jars (either specified by path or as references to the other project layout) and class folders. The available elements are:

- classes folder – a folder with classes;

- external jar – a jar file from other build script layout;
- external jars from – a collection of jars from a folder from some other buils script layout;
- jar – a path to local jar;
- jars – a path to local folder with jars and a list of selectors (include, exclude or includes).

Compilation settings for java modules are specified in java options. There can be several java options in the build script, only one of them can be default. Each module can specify its own java options to be used for compilation.

Java Targets

Java plugin adds the following targets:

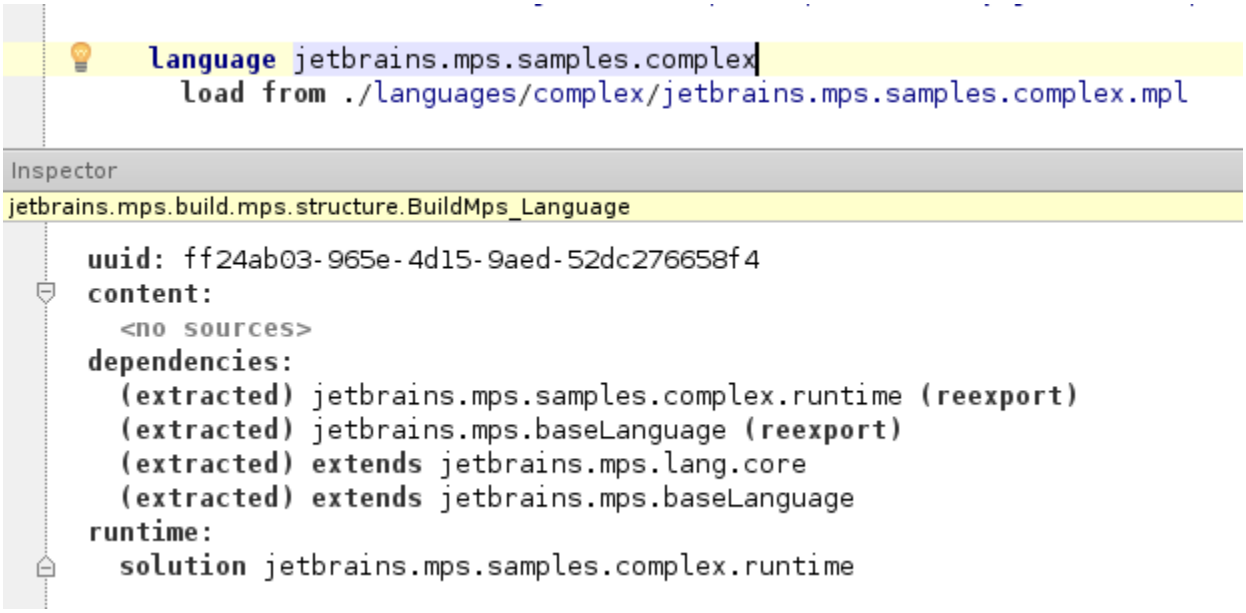
- compileJava compiles all java modules in the project.
- processResources extension point for additional resource processing.
- classes does all compilation and resource processing in the project. It depends on targets compileJava, processResources.
- test extension point target for running unit tests.
- check does all testing and checking of project correctness. It depends on target test.

MPS plugin

The MPS plugin enables the build language in scripts to build mps modules. In order to use the MPS plugin one must add jetbrains.mps.build.mps language into used languages.

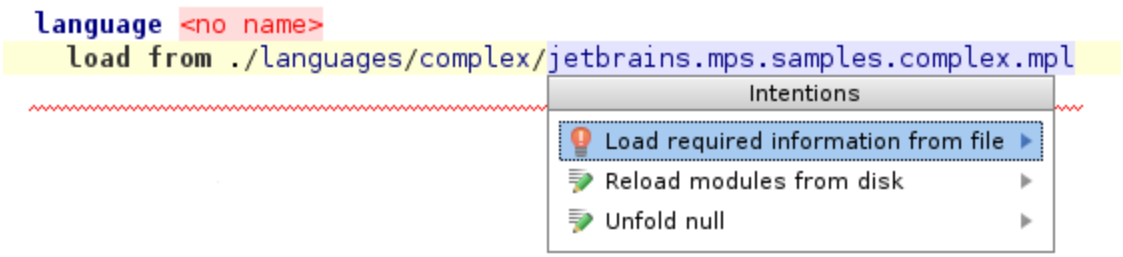
MPS modules and groups

The MPS plugin enables adding modules into project structure. On the screenshot there is an example of a language, declared in a build script.

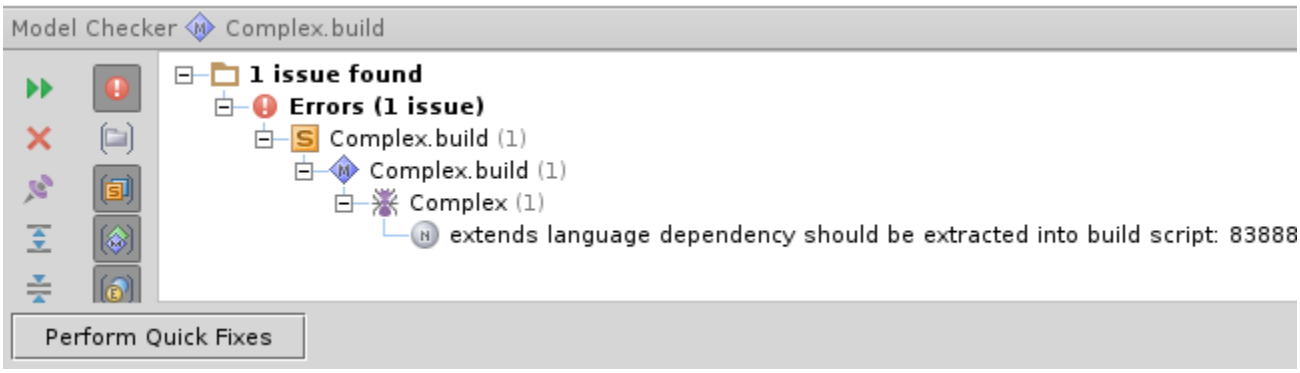


Note that there is a lot of information about the module specified in the build script, most of which is displayed in the Inspector tool window: uuid and fully qualified name, full path to descriptor file, dependencies, runtime (for a language) etc. This information is required for packaging the module. So, every time something changes for this module, for example a dependency is added, the build script has to be changed as well. There is of course a number of tools to do it easily. The typical process of writing and managing mps modules in the script looks as following:

1. Adding a module to the script. One specifies, which type of module to add (a solution, a language or a devkit) and the path to the module descriptor file. Then the intention "Load required information from file" can be used to read that file and fill the rest of the module specification automatically.



2. Reflecting the changes made in the module. One can check a model with build scripts using the Model Checker to find whether it is consistent with the module files. Model checker will show all problems in the script and allow you to fix them using "Perform Quick Fixes" button. Instead of Model checker one can use the same "Load required information from file" intention to fix each module individually.



Another thing to remember about MPS module declarations in a build scripts is that they do not rely on modules being loaded in MPS. All the information is taken from a module descriptor file on disk, while module itself could be unavailable from the build script.

MPS modules can be added into an mps group in order to structure the build script. An MPS Group is just a named set of modules, which can be referenced from the outside, for example one can add a module group into an IDEA plugin as one unit.

Module resources

Resources required by a module (images, icons, etc.) should be specified using the resources content root:

```
mps group notesOrganizer
  language jetbrains.mps.samples.notesOrganizer (4b0f115a-8868-4d72-8d61-97071eaaa5f1)
    load from
        ./languages/jetbrains.mps.samples.notesOrganizer/jetbrains.mps.samples.notesOrganizer.mpl
    content:
      resources files from ./languages/jetbrains.mps.samples.notesOrganizer
        includes icons/**, resources/**
    dependencies:
      (extracted) MPS.Platform
      (extracted) jetbrains.mps.kernel
      (extracted) MPS.Editor
      (extracted) jetbrains.mps.samples.notesOrganizer
      (extracted) MPS.OpenAPI
      (extracted) JDK
      (extracted) MPS.Core
      (extracted) jetbrains.mps.baseLanguage
      (extracted) jetbrains.mps.lang.editor.forms.runtime
    runtime:
      <no runtime>
```

How generating and compiling MPS modules works internally

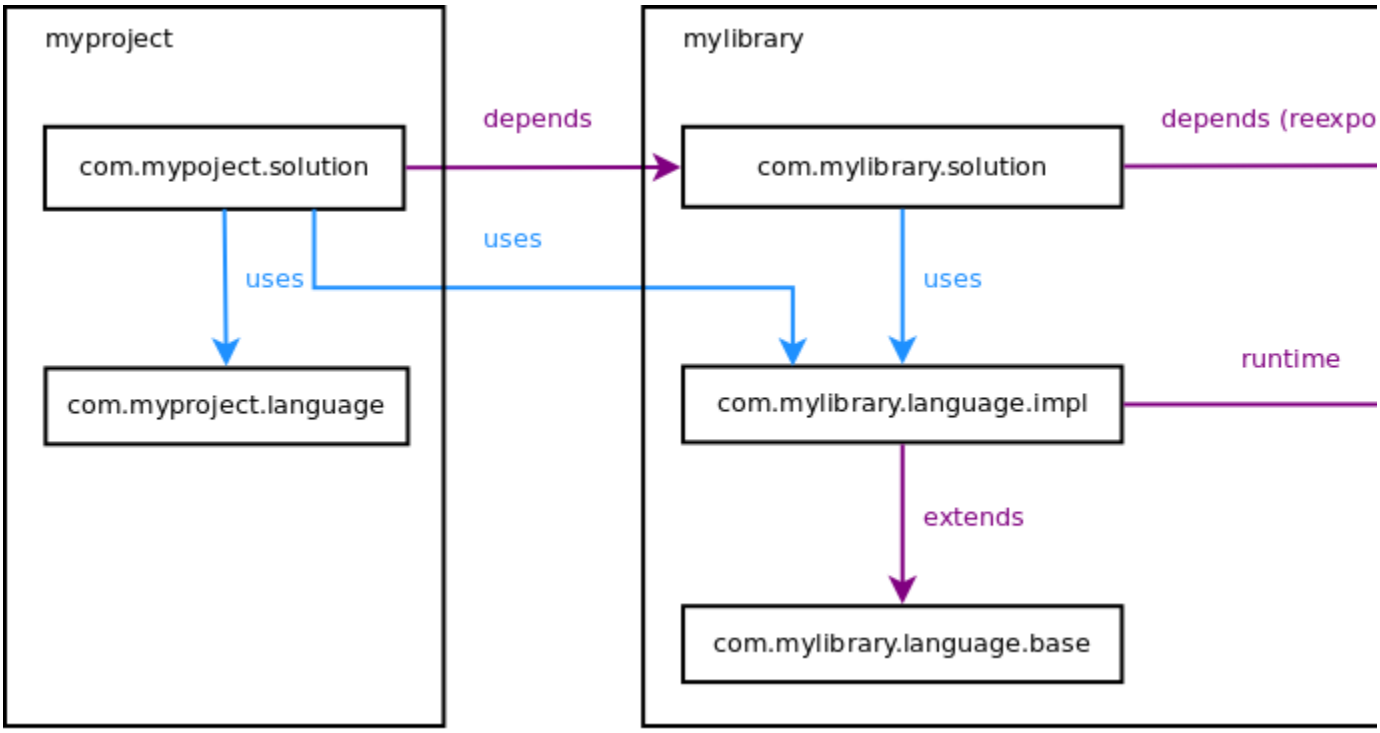
As it was written above, a lot of information about a module is extracted into the build script and stored there. This mandates the user to properly update the script whenever module dependencies change. For a Solution, it's both the reexported and non-reexported dependencies that are extracted to the script. For a Language, apart from the dependencies, runtime solutions and extended languages are also extracted.

"Building" a module with build script consists of two parts: generating this module and compiling module sources. Generating is an optional step for projects that have their source code stored in version control system and keep it in sync with their models. For generating, one target is created that generates all modules in the build script. Modules are separated into "chunks" – groups of modules that can be generated together – and the generate task generates the chunks one by one. For example, a language and a solution written with the language cannot be generated together, therefore they go into separate chunks. Apart from the list of chunks to generate, the generate task is provided with a list of idea plugins to load and a list of modules from

other build scripts that are required for the generation. This lists of plugins and modules is calculated from the dependencies and therefore their correctness is crucially important for successful generation. This is a major difference between generating a module from MPS and from a build script: while when generating a module from MPS, the generator has all modules in the project loaded and available; when generating a module from a build script, the generator only has whatever was explicitly specified in the module dependencies. So a build script can be used as some kind of a verifier of correctness of modules dependencies.

Compilation of a module is performed a bit differently: for every MPS module a java module is generated, so in the end each MPS module is compiled by an ordinary ant javac task (or similar other task, if it was selected in Java Options).

So in order to generate and compile, dependencies of a module are to be collected and embedded into the generated build xml file. Used languages and devkits are collected during generation of a build script from the module descriptor files. The other information is stored inside the build script node. In the picture below a module structure is shown for a project called "myproject", which uses some third-party MPS library called "mylibrary".

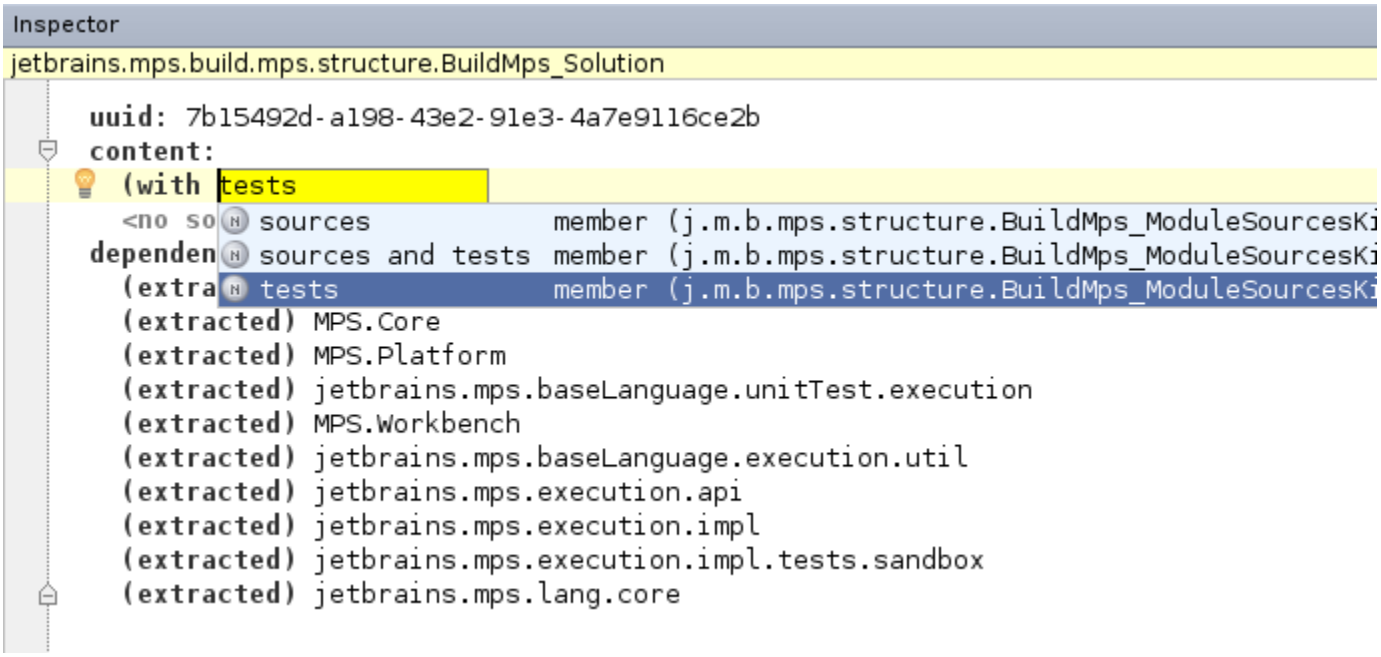


The arrows illustrate the dependency system between modules. The purple arrows denote dependencies that are extracted to the build script, the blue arrows indicate, which dependencies are not extracted. It can be easily observed that in order to compile and generate the modules from my project a knowledge of the "blue arrows" inside of mylibrary is not required. Which means that the actual module files from my library may not even be present during myproject build script generation. Every information that the generator needs is contained in the build script. Which is really very convenient: there is no need to download the whole library and specify its full location during build generation and so the generation process saves time and memory by not loading all module descriptors from project dependencies.

Sources and tests

When an MPS solution contains test models, i.e models with the stereotype "@tests", they are generated into a folder "tests_gen" which is not compiled by default. To compile tests, one needs to specify in the build script that a solution has test

models. This is done manually in the inspector. There are three options available for a solution: "with sources" (the default), "with tests" and "with sources and tests".



MPS Settings

mps settings allow to change the MPS-specific parameters for a build script. No more than one instance of mps settings can exist in the build script in the "additional aspects" section. Parameters that can be changed:

- bootstrap – setting this flag to "true" indicates that there are some bootstrapping dependencies between modules in the script. Normally the flag is set to false. See Removing bootstrapping dependency problems for details.
- test generation – if set to true, the build script tests modules generation and difference between generated files and files on disk. Files can be excluded from diff in excludes section.
- generation max heap size in mb – maximum heap size for generation and generation testing.

Testing Modules Generation

Projects that keep their generated source files in version control can check that these generated files are up-to-date using build script. After setting test generration in mps settings to true a call of gentest task appears in test target of generated build script. Similarly to generate task, gentest loads modules in the script, their dependencies from other build scripts and idea plugins that are required. For each module gentest task invokes two tests: "%MODULE_NAME%.Test.Generating" and "%MODULE_NAME%.Test.Diffing". Test.Generating fails when module has errors during generation and Test.Diffing fails when generated files are different from the ones on disk (checked out from version control). Test results and statistic are formatted into an xml file supported by the TeamCity build server.

IDEA plugins

idea plugin construction defines a plugin for IntelliJ IDEA or MPS with MPS modules in it. In the screenshot you can see an example of such plugin.

```
idea plugin jetbrains.mps.samples.complex
  name MPS Complex Language
  short (folder) name mps-samples-complex
  description Enables working with complex numbers in base language
  version 1.0
  vendor JetBrains
    url http://www.jetbrains.com/mps/
    icon16 <no icon16>
  content:
    Complex (custom packaging for jetbrains.mps.samples.complex.library)
  dependencies:
    jetbrains.mps.core
```

The first section of the plugin declaration consists of various information describing the plugin: its name and description, the name of the folder, the plugin vendor, etc.. The important string here is plugin id, which goes after the keywords idea plugin. This is the unique identifier of the plugin among all the others (in the example the plugin id is jetbrains.mps.samples.complex).

The next section is the actual plugin content – a set of modules or module groups included into the plugin. If some module included in the plugin needs to be packaged in some special way other than the default, this should also be specified here (see the line "custom packaging for jetbrains.mps.samples.complex.library").

The last section is dedicated to the plugin dependencies on other plugins. The rule is that if we have a "moduleA" located in plugin "pluginA", which depends on "moduleB" located in "pluginB", then there should be a dependency of "pluginA" on "pluginB". A typesystem check exists that will identify and report such problems.

The layout of the plugin is specified last:

```
plugin jetbrains.mps.samples.complex
  folder languages
    module jetbrains.mps.samples.complex.library
```

In the screenshot, module jetbrains.mps.samples.complex.library is packaged into the plugin manually as it is specified in idea plugin construction not to package it automatically.

MPS Targets

The MPS plugin provides the following targets:

- generate - generates the mps modules that are included in the project structure.
- cleanSources - cleans the generated code (only for modules without bootstrapping dependencies). See more about bootstrapping dependencies in article Removing bootstrapping dependency problems.
- declare-mps-tasks - a utility target that declares mps tasks such as generate or copyModels.
- makeDependents - invokes the generate target for a transient closure of this script's dependencies (if there is one) and then invokes assemble to put them together. It is guaranteed that each script is executed only after all its dependencies have been built.

Module Testing plugin

The Module testing plugin, provided by jetbrains.mps.build.mps.tests language, adds to build scripts the capability to execute NodeTestCases and EditorTestCases in the MPS solutions. Tests are executed after all modules are compiled and packaged into a distribution, i.e. against the packaged code, so they are invoked in an environment that closely mimics the real use of the code.

Test modules configurations

Solutions/module groups with tests are grouped into test modules configurations, which is a group of solutions with tests to be executed together in the same environment. All required dependencies (i.e. modules and plugins) are loaded into that environment.

In the screenshot, you can see a test modules configuration, named execution, which contains a solution jetbrains.mps.execution.impl.tests and a module group debugger-tests.

```
  mps group execution-tests
    solution jetbrains.mps.execution.impl.tests.sandbox
      load from ./plugins/execution-languages/languages/tests.data/jetbrains.mps.exe

    solution jetbrains.mps.execution.impl.tests
      load from ./plugins/execution-languages/languages/tests/jetbrains.mps.executic

  mps group debugger-tests
    solution jetbrains.mps.debugger.java.runtime.tests
      load from
          ./plugins/debugger-java/solutions/jetbrains.mps.debugger.java.runtime.test


default layout:
  folder execution
    module jetbrains.mps.execution.impl.tests
    module jetbrains.mps.execution.impl.tests.sandbox

  folder debugger
    module jetbrains.mps.debugger.java.runtime.tests

test modules configuration execution
  jetbrains.mps.execution.impl.tests
  debugger-tests
```

There is a precondition for solutions to be included into a test modules configuration. A solution should be specified as containing tests (by selecting "with tests" or "with sources and tests" in inspector). A module group should contain at least one module with tests.

Test results and statistic are formatted into an xml file (which is supported by TeamCity).

MPS-runner plugin

The MPS-runner plugin, provided by jetbrains.mps.build.mps.runner, enables a new build script entry - run code from solution. By pointing it to a solution that holds your Java code the build script will be able to run it as part of the build process.

```
run code from solution org.jetbrains.mps.samples.ParallelForUtils
                       org.jetbrains.mps.samples.ParallelForUtils.MainClass.mpsMain()
```

A minimalistic build script that invokes a Java class located in a "sandbox" solution of a project could lose somewhat like this:

```
build Project2 generates build.xml

base directory: ../../
                /Users/vaclav/MPSProjects/Project2

use plugins:
  java
  mps
  mps-runner

macros:
  folder mps_home = <no defaultPath>

dependencies:
  mps (artifacts location $mps_home)

project structure:
  idea plugin Project2
    name Project2
    short (folder) name Project2
    description <no description>
    version 1.0
    << no vendor >>
    content:
      Project2
    dependencies:
      jetbrains.mps.core
    << ... >>

  mps group Project2
    solution NewLanguage.sandbox
      load from ./languages/NewLanguage/sandbox/NewLanguage.sandbox.msd


default layout:
  module NewLanguage.sandbox

run code from solution NewLanguage.sandbox
                       NewLanguage.sandbox.MainClass.mpsMain()
```

Control over the repository

The MPS ant task provides full control over the repository contents with several new tags - module, modules and allmpsmodule s.

```
<migrate project="${project}">
  <repository>
    <allmpsmodules/>
    <modules dir="${moduleLibFolder}"/>
    <module file="${uniqueModule.msd}"/>
  </repository>
</migrate>
```

How-to's

The following articles explain how to build a language plugin:

- Building IntelliJ IDEA language plugins
- Building MPS language plugins
- Building standalone IDEs

Articles on the topic of building with MPS:

- Removing bootstrapping dependency problems