Migration to MPS 2.5

Introduction

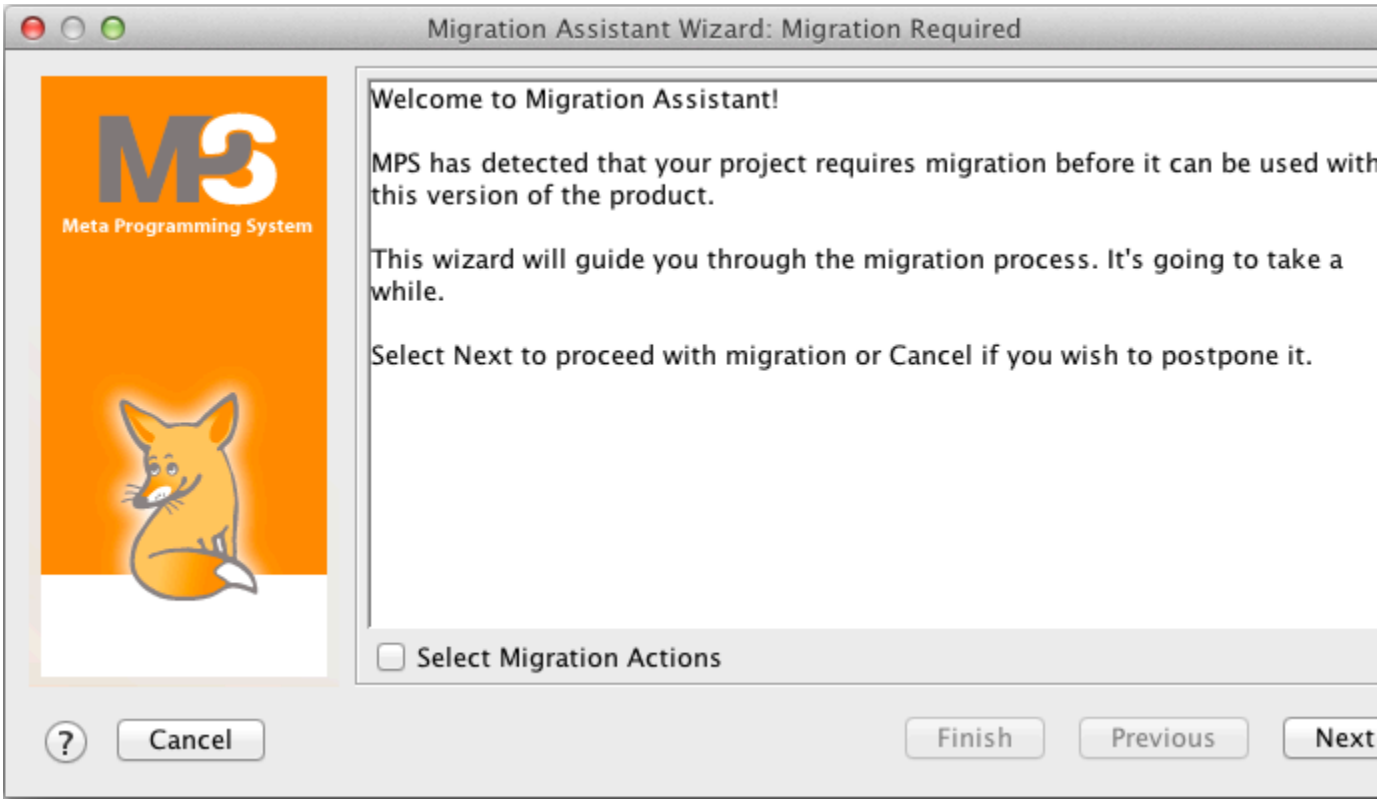
This document contains a brief explanation of the MPS 2.5 migration process. MPS 2.5 is being shipped with a comprehensive migration wizard, so most of the modifications applied to the project structure and to the models should be done automatically on first opening the old project in MPS 2.5. Nevertheless, since some important changes cannot be automated, there might be some work left for the developers to do manually. By reading this guide you will get an overview of the potential problems and the proposed solutions, which you can apply manually to your project to make it valid for MPS 2.5.

The migration assistant was designed to support migration from MPS 2.0.x versions only. If, by a chance, you have an older version of MPS you should migrate your project(s) to MPS 2.0 first, followed by the MPS 2.5 migration.

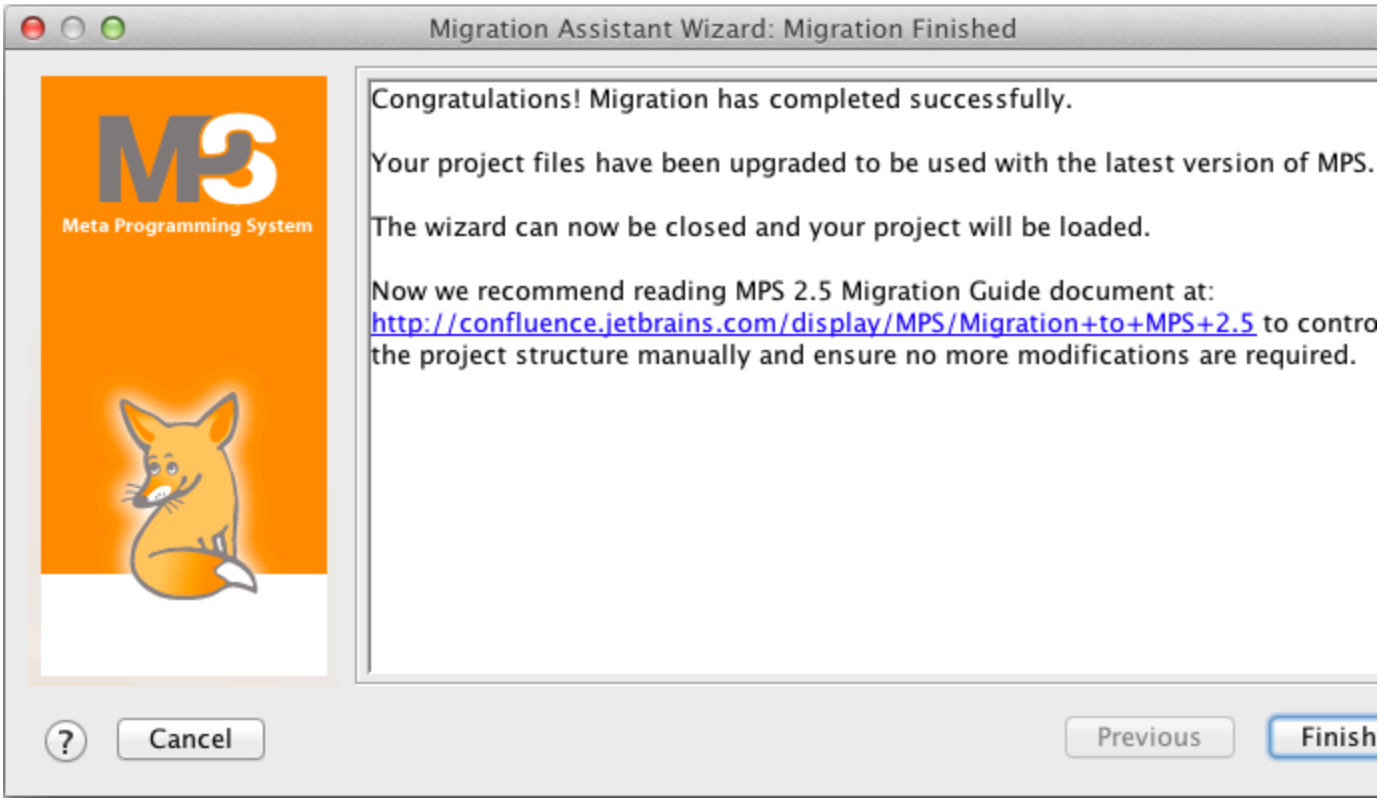
Note: Irrelevant errors can show up during migration process. To minimize this effect we suggest removing all .class files generated by previous version of MPS before you start the migration.

Running automatic migration process

To initiate the migration process you have to start MPS 2.5 and open your 2.0-based project. MPS should automatically detect the fact that this project has not been adopted to MPS 2.5 yet and will suggest starting the migration:



Just pass through all the wizard pages by clicking the "Next" button and wait until the migration process is done:



At this moment the project should be completely updated, so now it's time to run the ModelChecker to ensure that there are no unresolved references in the project. The next step is to fully re-generate the code.

Most of the necessary modifications to the project should be applied by migration wizard, but in some cases migration either cannot be performed automatically or the automatic migration process becomes very complex and unreliable. In such situations we propose applying manual modifications to the project. Below you can find useful information describing most of the possible errors you can face after the automatic migration phase. These errors can be either reported by the ModelChecker or show up during the code generation process.

Applying manual modifications to the project

The MPS API has changed considerably in the MPS 2.5 release, so it is possible that some code can be broken as a result of the MPS API changes. We do not provide a complete list of API modifications here, though. In case some Java (BaseLanguage) code is broken as a result of the migration process, the project developers will have to update the code so that it uses proper (new) MPS API. The code that deals with MPS API is typically located in the plugin aspects of user languages to call some methods of the Java API coming from the IntelliJ platform or MPS. We believe there should not be many places that will require these changes to be applied.

MPS.Classpath module has been removed

In MPS 2.0 there was a special module called MPS.Classpath. This module was used to expose all available Java API of the platform and MPS to the models. In MPS 2.5 this module was substituted by the following four modules:

- MPS.Core
- MPS.Editor
- MPS.Platform
- MPS.Workbench

Usually if some class was available through the MPS.Classpath module in the past, it is available through one of these four modules in MPS 2.5. In case the migration process could not update all existing references and some project modules/models still reference MPS.Classpath after the automatic migration, such places should be modified manually to eliminate usages of MPS.Classpath and replace them with one of the new MPS.* modules.

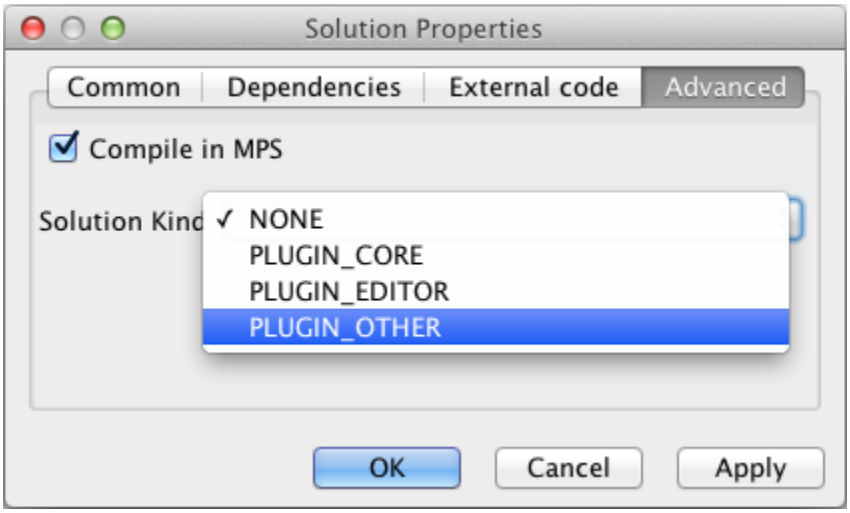
Introduction of the jetbrains.mps.openapi package

A new package jetbrains.mps.openapi was introduced in MPS 2.5. This is the first step towards MPS open API. This package (and it's sub-packages) contains a small set of interfaces with methods for core functionality. Only a minimal set of methods have been exposed in jetbrains.mps.openapi, so it is possible that some methods you were using earlier are not available through openapi.

To work around this situation you can temporarily cast the openapi interfaces to the implementation classes and call missing methods from there. Usually the implementation classes and the API interfaces share a common name, so you can locate them by name easily (pressing Ctrl+N). Please, do not forget to file improvement requests for MPS whenever you discover some functionality missing from the openapi, so as we may consider adding it.

Solution Kind property

To optimize the class loading process and to reduce the set of modules loaded by MPS from the current project, a new solution property "Solution Kind" has been introduced:



By setting this property you can control the process of loading a solution into the currently running instance of MPS and you also reduce the set of classes visible to this solution. This property has four possible values:

- NONE means the solution will be ignored by the class loading process so you will not be able to use classes of this solution from any other plugin modules
- PLUGIN_CORE means the solution will be loaded into the current instance of MPS, only classes from the MPS.Core modu le and nodes from other solutions marked as PLUGIN_CORE can be used there
- PLUGIN_EDITOR: the solution will be loaded, classes from MPS.Core, MPS.Editor, other PLUGIN_CORE and PLUGIN_EDI TOR solutions can be used
- PLUGIN_OTHER: the solution will be loaded, you have no restrictions on referencing MPS APIs or other solutions in the project.

By default, any newly created solution has the Solution Kind property set to NONE.

You need to manually specify the Solution Kind property to a value other then NONE for all solutions used in language models implementing IDE features for the language:

- constraints
- behavior
- editor
- actions
- type system
- generator macros/queries

You don't have to set the Solution Kind property for those solutions used in:

- generator templates (language runtimes)

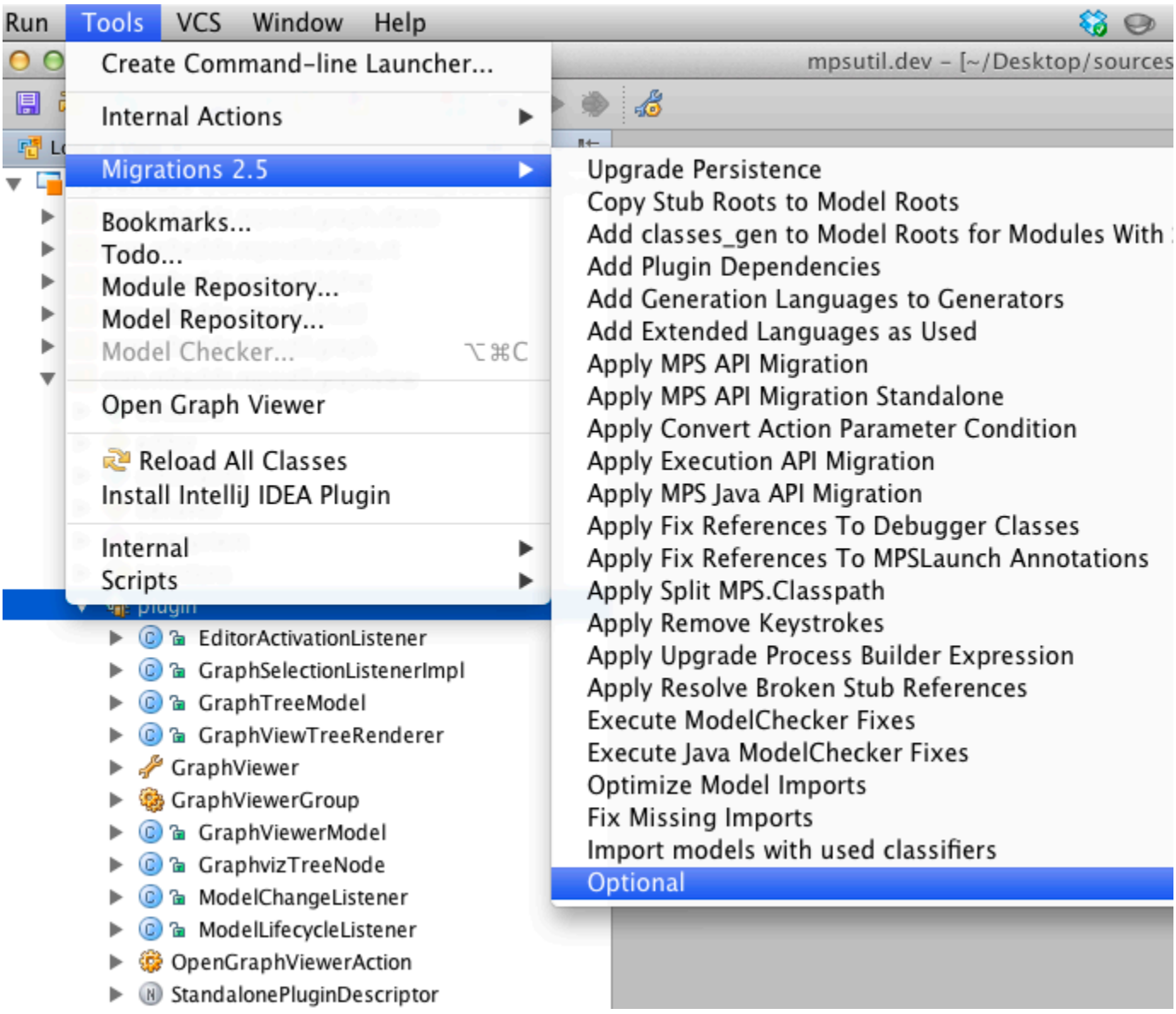
In other words, all languages from the project will be loaded to the current instance of MPS as "plugins", so if the solution code is used from the language code running in the current MPS instance, the solution should be marked with PLUGIN_... kind.

Use separate Plugin Solutions to hold all platform-specific elements instead of using the the plugin aspect in the language

One of the future goals for MPS is to be able to deploy the same DSLs to different target platforms (e.g. the Eclipse platform is planned in near future). To address this challenge we introduced one more restriction in MPS 2.5: it's not possible to use any platform- (i.e. UI-) specific code inside language definition. In MPS 2.5 all such code should be refactored out of the language definition into separate Plugin Solutions with Solution Kind set to PLUGIN_OTHER. In addition this refactoring should separate platform-specific as well as UI elements of the project from common (core) language definition and so clarify the cross-module dependencies. This is definitely a good thing to do, if you look at it from the project architecture perspective.

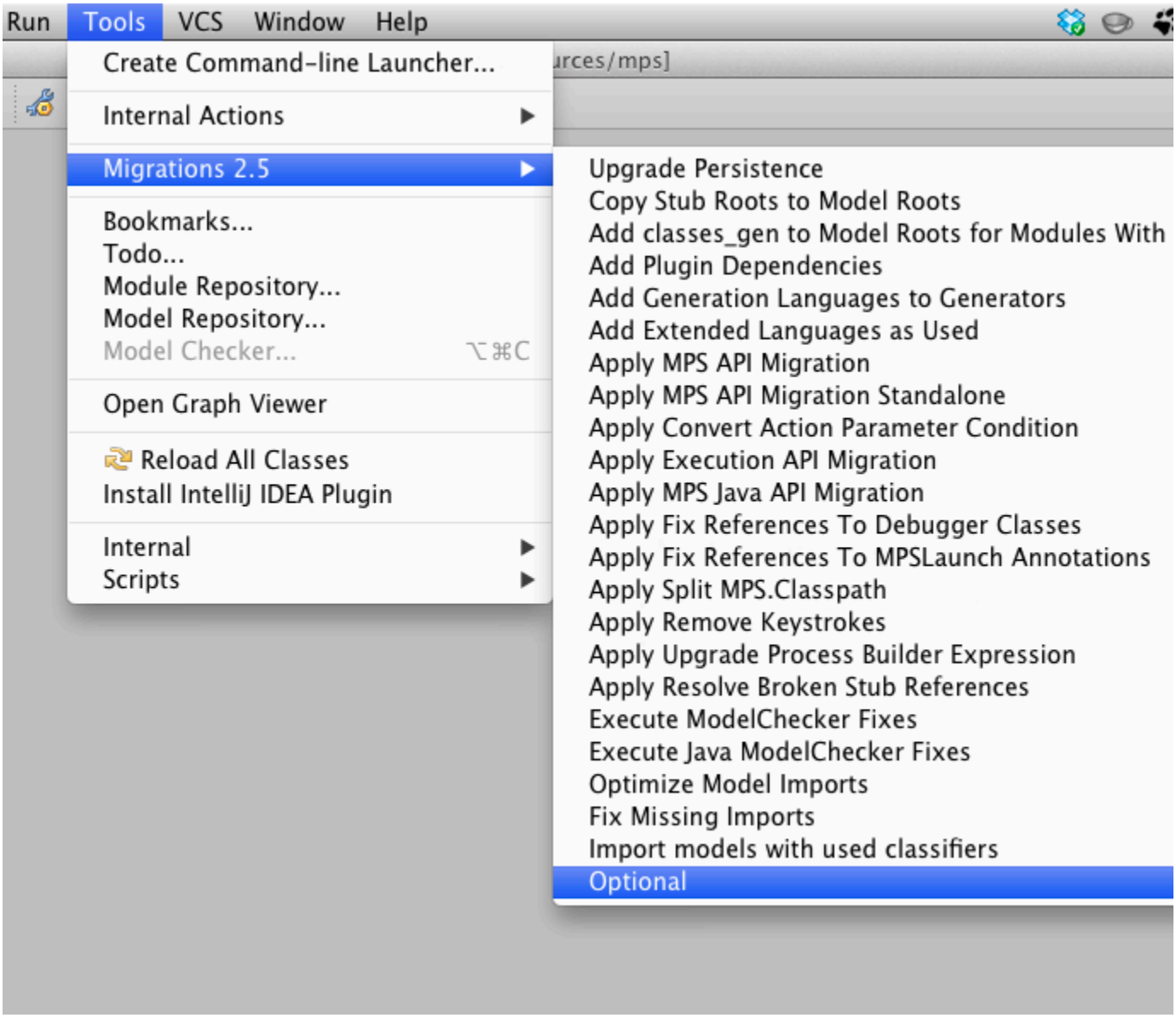
The most common place for UI-specific code in language definition used to be the plugin aspect, because this aspect was commonly used to declare Actions/ActionGroups/Tool definitions - pure UI and platform-specific elements. With MPS 2.5 you can execute an script for automated extraction of all the code from the plugin aspects of the language into a newly created separate plugin solutions. This script doesn't contain any artificial intelligence to differentiate between individual cases, so it will mechanically move all nodes from all plugin aspects to a separate solution. Sometimes this script can move non-UI aspects out of the language by mistake. In that case those nodes should be manually moved to the appropriate place (by default you can move them back to the plugin aspect of the language).
The main criteria here is: it's absolutely legal to add a dependency from the newly extracted plugin solution to the language it was extracted from, because the plugin solution literally adds platform-specific support for the language. At the same time it's absolutely impossible to add an opposite dependency, going from the language onto it's plugin solution, because the language definition code should not have any dependencies on the platform it's working on.

In other words, the following migration should be executed manually and the result of this execution should be verified. You can run the "Move Plugins Out of Language" migration from the main menu:



Correct Icons After Plugin move

Some model elements have been removed from languages to a separate plugin solution in the previous steps. If those elements were referencing icon files (via IconResource instances), most probably these references ended up broken. Icon resources are typically referenced from actions. At this stage it may be a good time to review broken icon references and correct them either by modifying these references or by moving/copying icon file to the corresponding plugin solution. You can use the automatic refactoring called "Correct Icons after Plugin Move", which will move all icons from the folder located below the language home directory to a folder located below the newly created plugin solution home directory:



Refactoring

Refactoring definition has changed significantly in MPS 2.5 (see Changes in the Refactoring language chapter of What's new in MPS 2.5 document for the details). Most probably you will need to review all existing refactorings in all the languages of your project and adopt them in a way that all UI is extracted from the refactorings and placed into the corresponding plugin solution.