Removing bootstrapping dependency problems

Definition:

Whenever you have

- a language that uses itself or
- a solution that uses a language and that language requires (i.e. indirectly or directly depends on) the same solution in order to work

you have bootstrapping dependency problem.

Why is this a problem

- you cannot rebuild the whole project from the ground up using build scripts
- you are forced to keep generated artifacts in the VCS repository

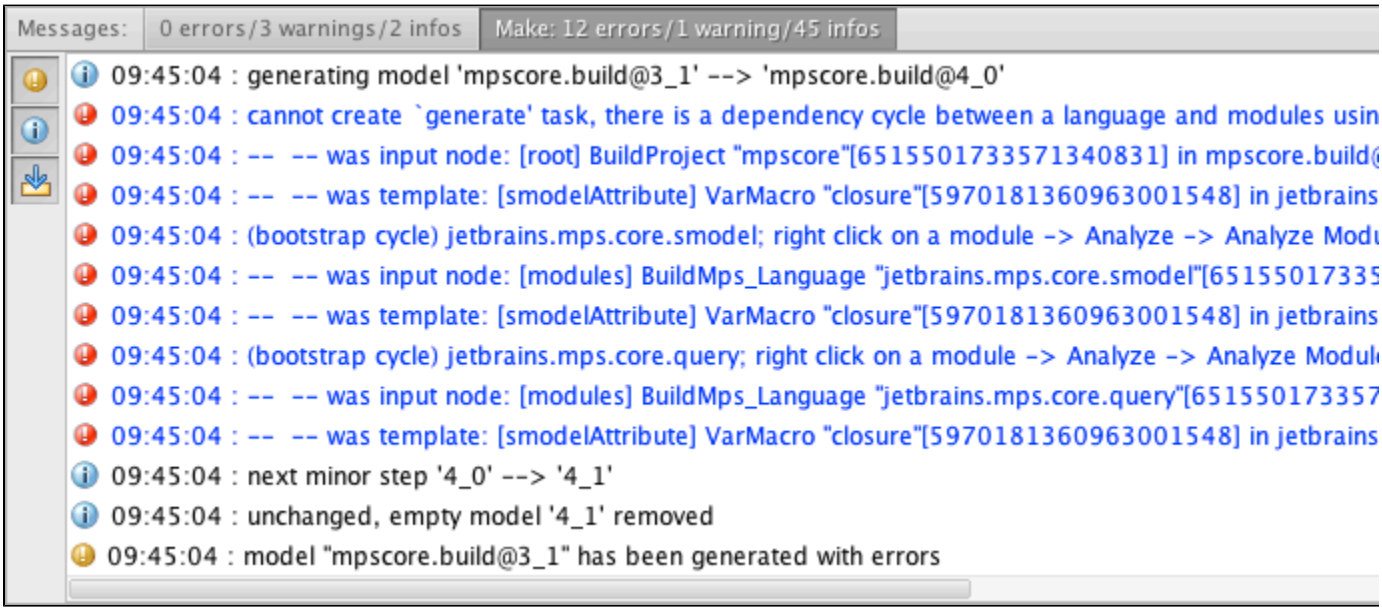
This bootstrapping dependency circle prevents your project from ever being rebuilt from the ground up completely. Instead, building your language requires the previous version of your solution build artifacts to have been generated. Similarly, building your solution requires the language to have been built first. Thus you cannot rebuild your whole project with a single command, you need to keep generated artifacts in VCS and the dependency structure of your project contains loops.

The build scripts in MPS 2.5.4 and beyond can generate MPS code and so you no longer depend on the MPS workbench for code generation. Bootstrapping dependencies in the project structure, however, stand in the way.

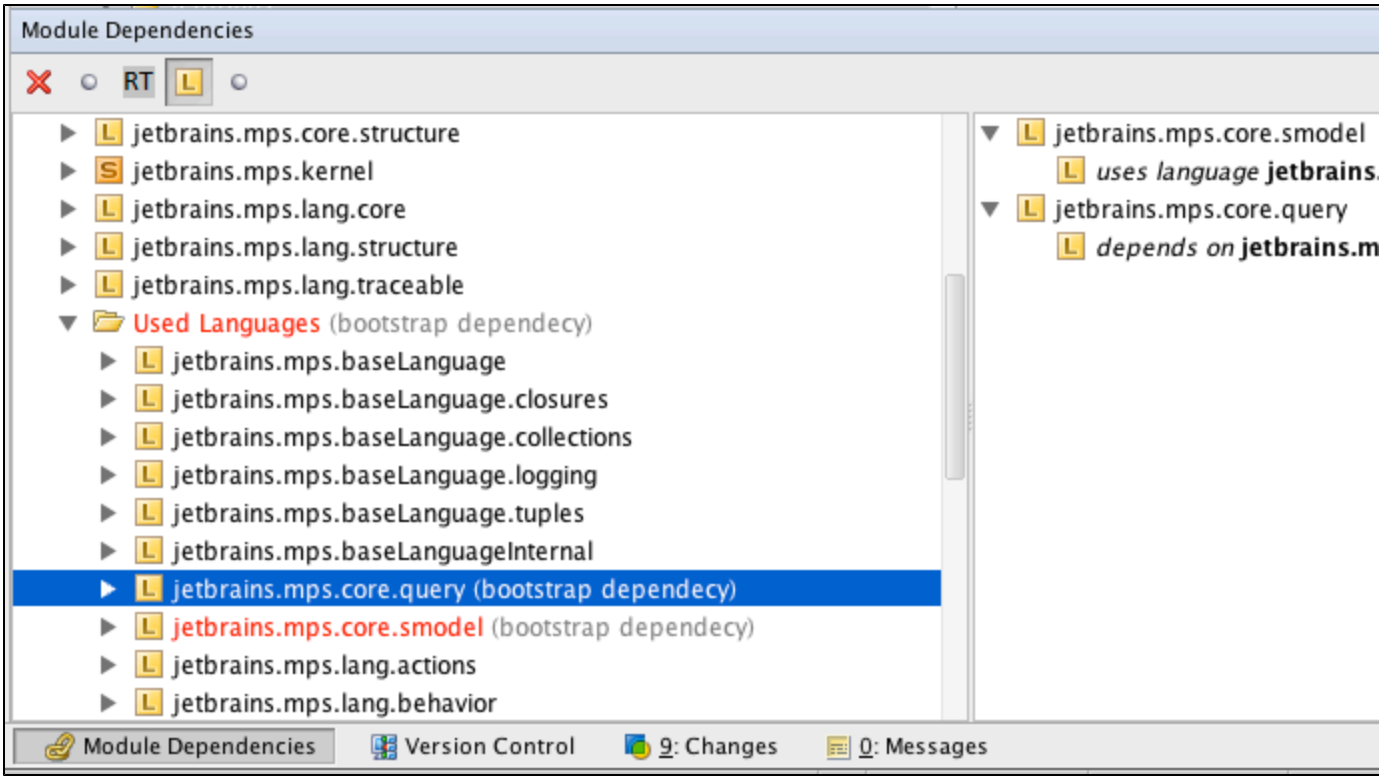
The detection and elimination toolchain in MPS

We consider the inability to fully regenerate projects to be a serious issue. Serious enough so that we decided to provide tools that detect and help you eliminate all such dependencies in MPS starting with version 2.5.4. Nothing changes for rebuilding your projects in MPS. However, when rebuilding build scripts, MPS will detect and report circular dependencies between languages and modules that use them as errors.

When rebuilding a build script, you may get error reports like:



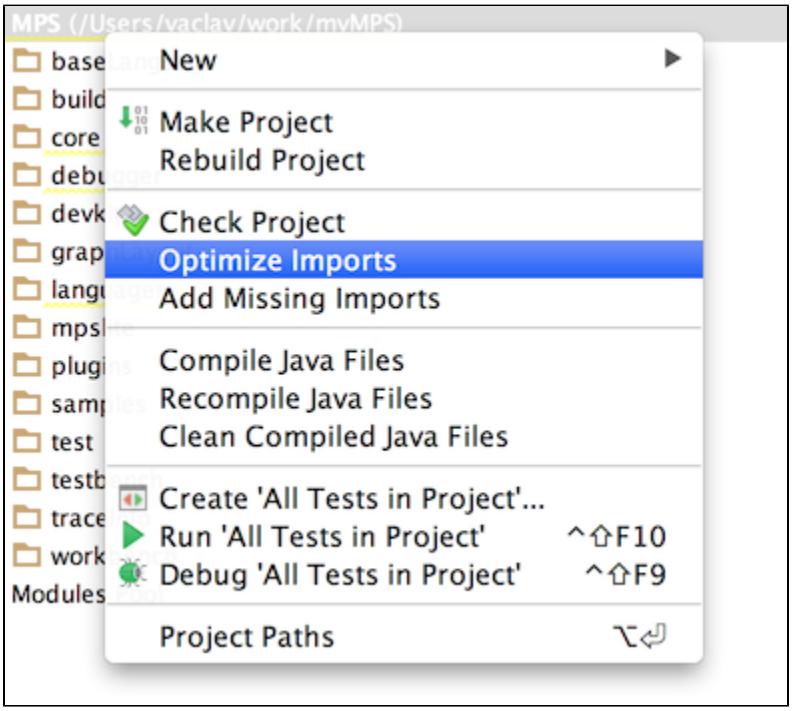
The output text contains a hint for solving the issue: right click on a module -> Analyze -> Analyze Module Dependencies



You will get the analyzer report panel displaying module dependencies. The gray (bootstrap dependency) indicator will highlight all problematic bootstrapping dependencies so you can quickly spot them. After selecting one in the left panel, the right panel will further detail the dependency circle. You see the dependency of the module on the language is listed, followed by the dependency of the language on the module. Now you can choose, which of the two dependency directions you want to remove in order to fix the problem.

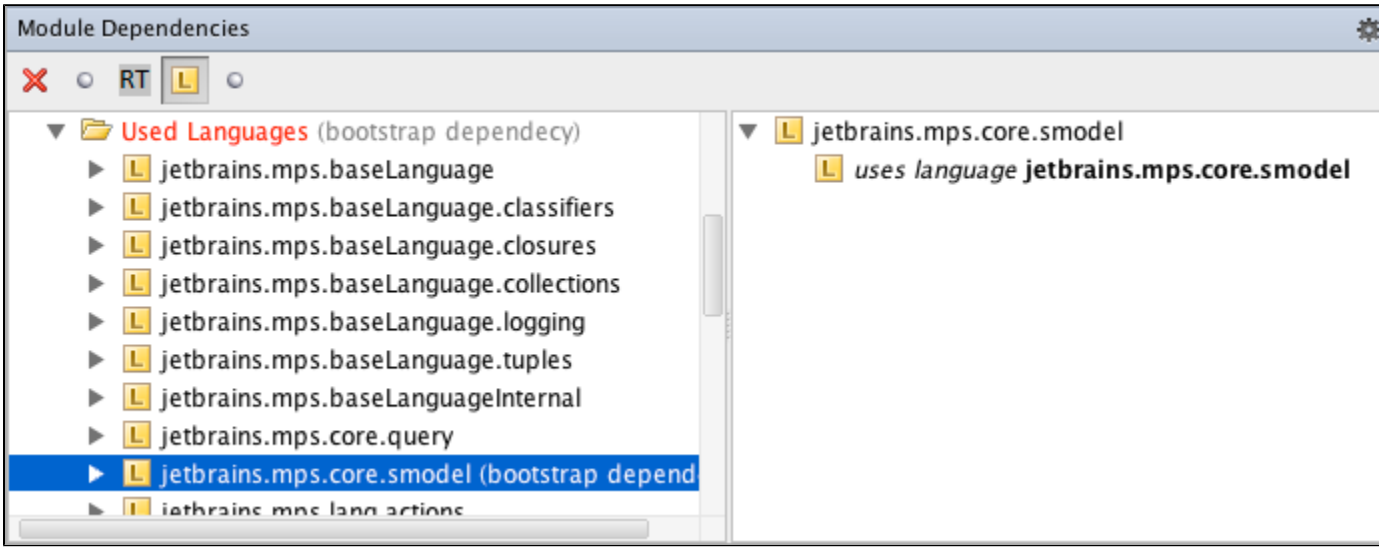
Fixing the problem

Essentially, you need to break the dependency cycles. The first attempt you should make is to invoke Optimize Imports. If the dependencies between modules were only declared but not really used in code, this will remove them and potentially solve the bootstrapping problem. If the problem remains, you'll have to play around with the analyzer a bit more.

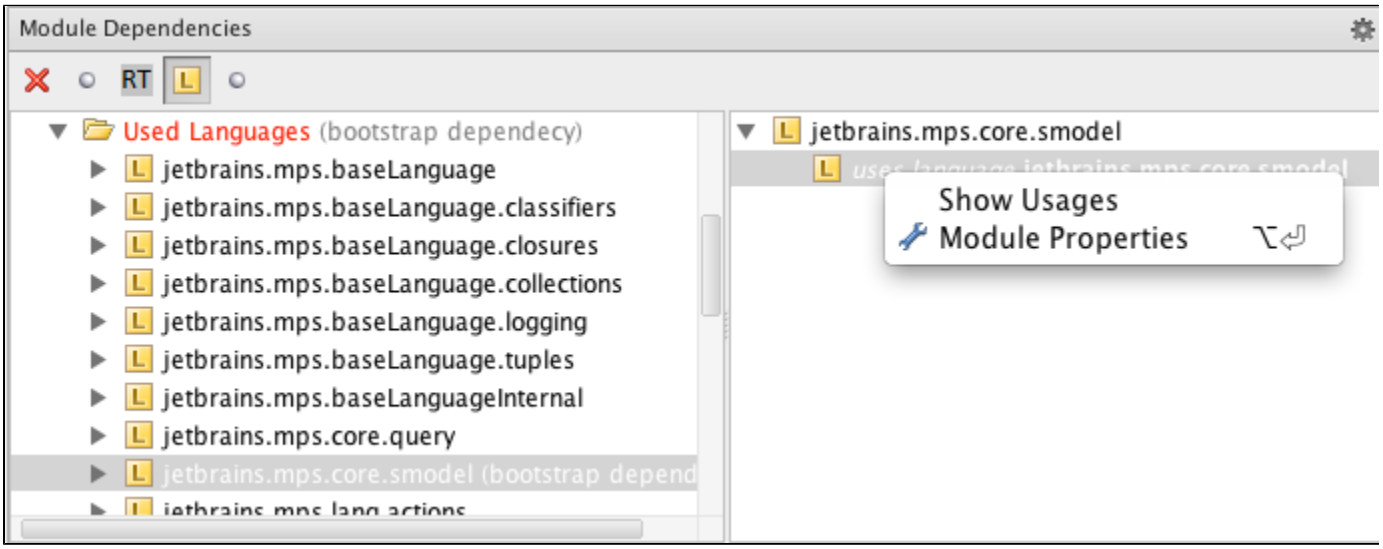


Analyzing the bootstrapping dependencies

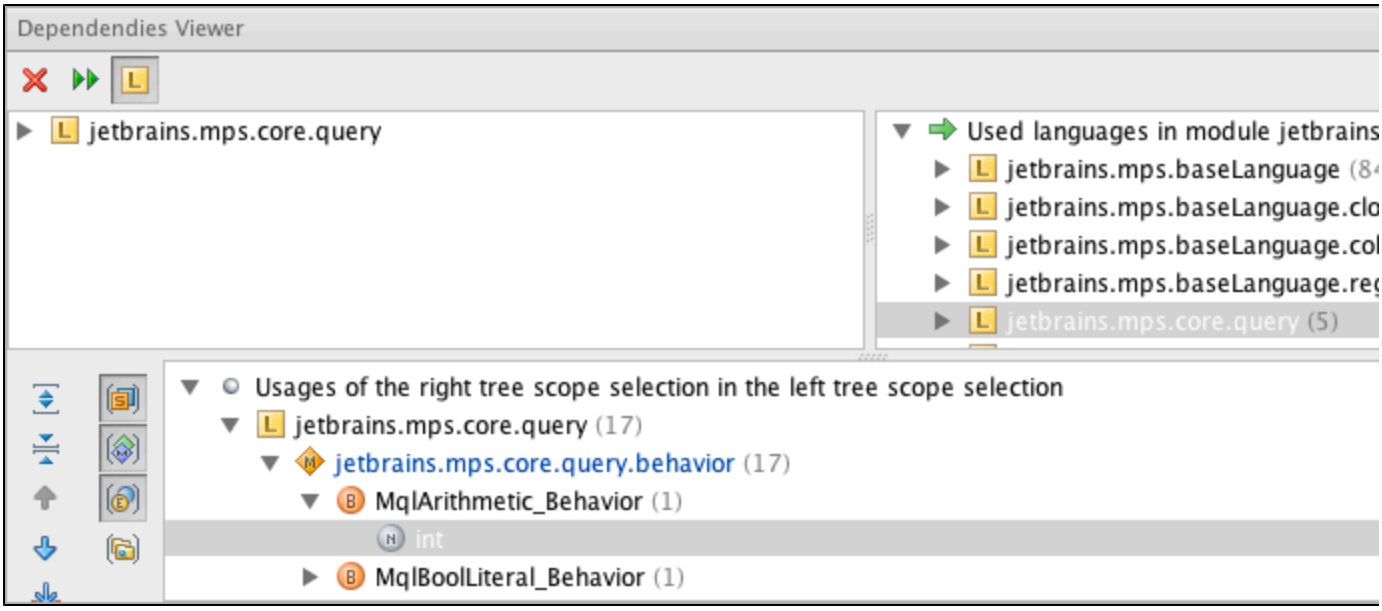
Imagine, for example, that we are fixing a frequent problem of a language uses itself for its own definition - a bootstrapping language.



We are only seeing a single problematic dependency direction in the right panel.



With the Show Usages pop-up menu command we get a third panel, which lists all the concrete occurrences in code, where the dependency is needed.

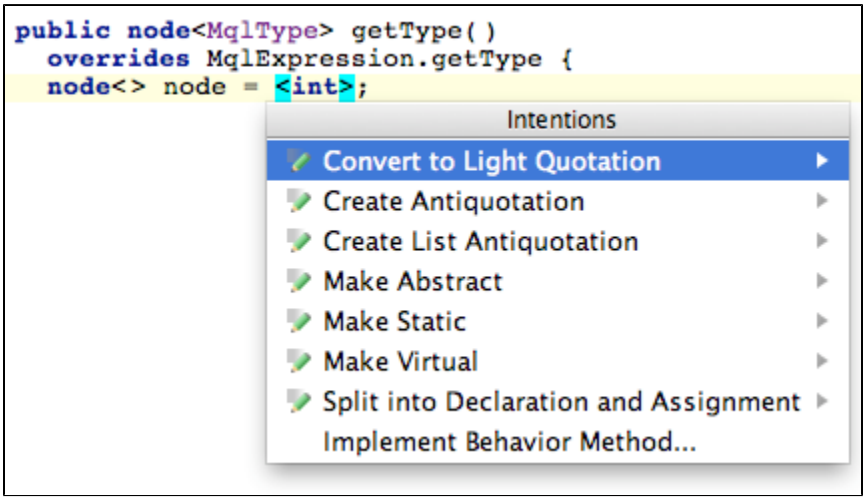


Unfortunately, you'll have to decide yourself, which ones to remove and how. This will be a manual process.

Typical cases of bootstrapping dependencies

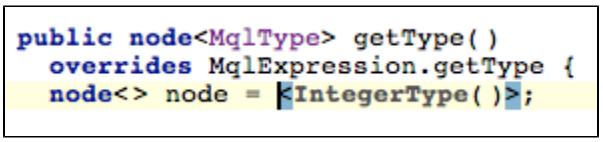
Case 1: Self-refering quotations

A very frequent example of a mistake that leads to a self-reference in a language is using the language in quotations, for example to define its own type-system rules. You declare your own type and then instantiate this type in the type-system aspect through quotation. Now the language needs itself to have been built in order to build.



Light quotations should be used in such situations instead of quotations. They provide lightweight quotation-like functionality without the need to use the editors of the nodes in quotation. For Light quotations to create appropriate nodes you need to supply the concept name and the required features.

A handy intention for quick conversion from quotation to a Light quotation is also available.



Light quotations compared to quotations are slightly less convenient, but they avoids the bootstrapping problem. You may also favor them to quotations to better express your intent, when you need to combine quotations and anti-quotations heavily.

```
// Quotation
<if not wall ahead do
  pick
  step
end else do
  turnLeft
end>;

// NodeBuilder
<IfStatement(
    condition: Not(original: IsWall( )),
    trueBranch: CommandList(
                  commands: Pick( ),
                  commands: Step( )),
    falseBranch: CommandList(commands: LeftTurn( )))>;
```

Case 2: A language uses itself in an accessory model

Languages can contain accessory models. If these accessory models use the containing language and they have not selected the "do not generate flag", we get a circular dependency between the accessory model and the owning language. Such a language cannot be rebuilt from scratch, since the accessory model needs to be generated together with the language.

The solution to the problem is to move the accessory model to a separate solution. It is possible in MPS to have an accessory model reside outside of a language.

Case 3: A language uses itself inside patterns created by jetbrains.mps.lang.pattern

This problem is similar to the one with quotations. It does not prohibit rebuilding a language from scratch, but the cyclic dependency stays anyway. Currently, there is no straightforward solutions to that other than avoiding using the language inside patterns or allowing for bootstrapping dependencies in build scripts.

Case 4: A runtime solution of a language uses the very same language

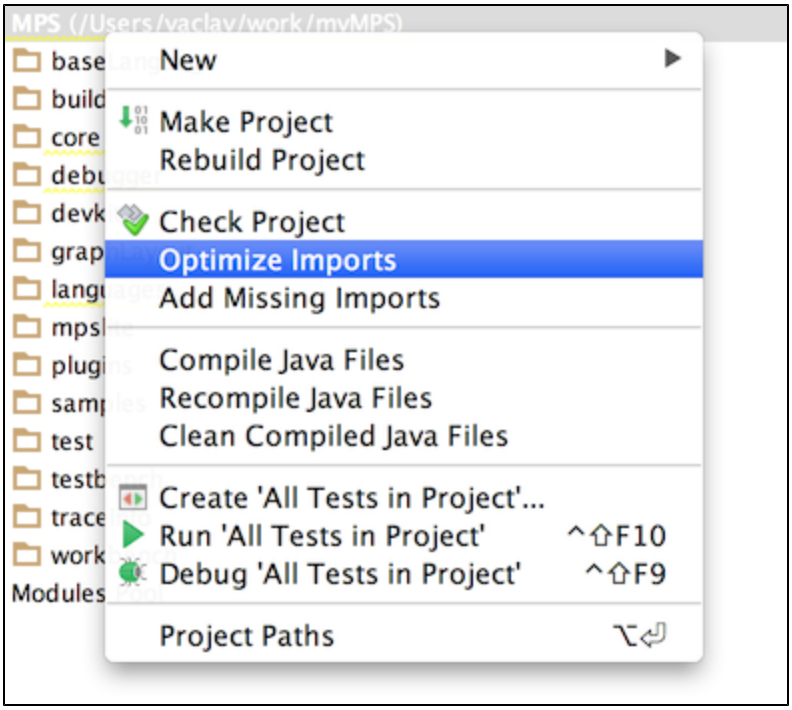
> This one is not really a bootstrapping problem and will not prevent your build scripts from being build, however, it should still be considered to be a problematic and discouraged project dependency structure.

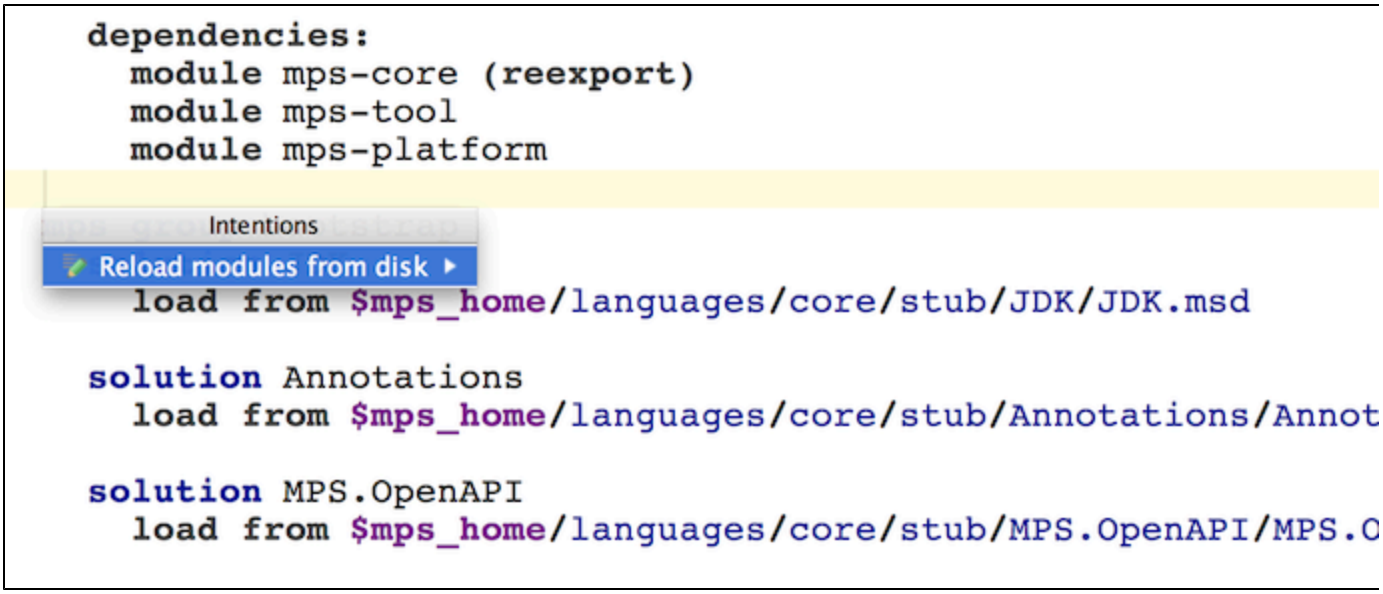
A runtime solution for a language A typically contains the code executed by whatever gets generated from the models that use the language A. So the runtime solution code logically belongs to the same abstraction level as the generated code, that is one level below the language A level. It is a good practice to separate the lower-level code from the higher-level code and organize your project hierarchically.

The solution to the problem is to move the usages of language A away from the runtime solution into another, separate module.

What else needs to be done

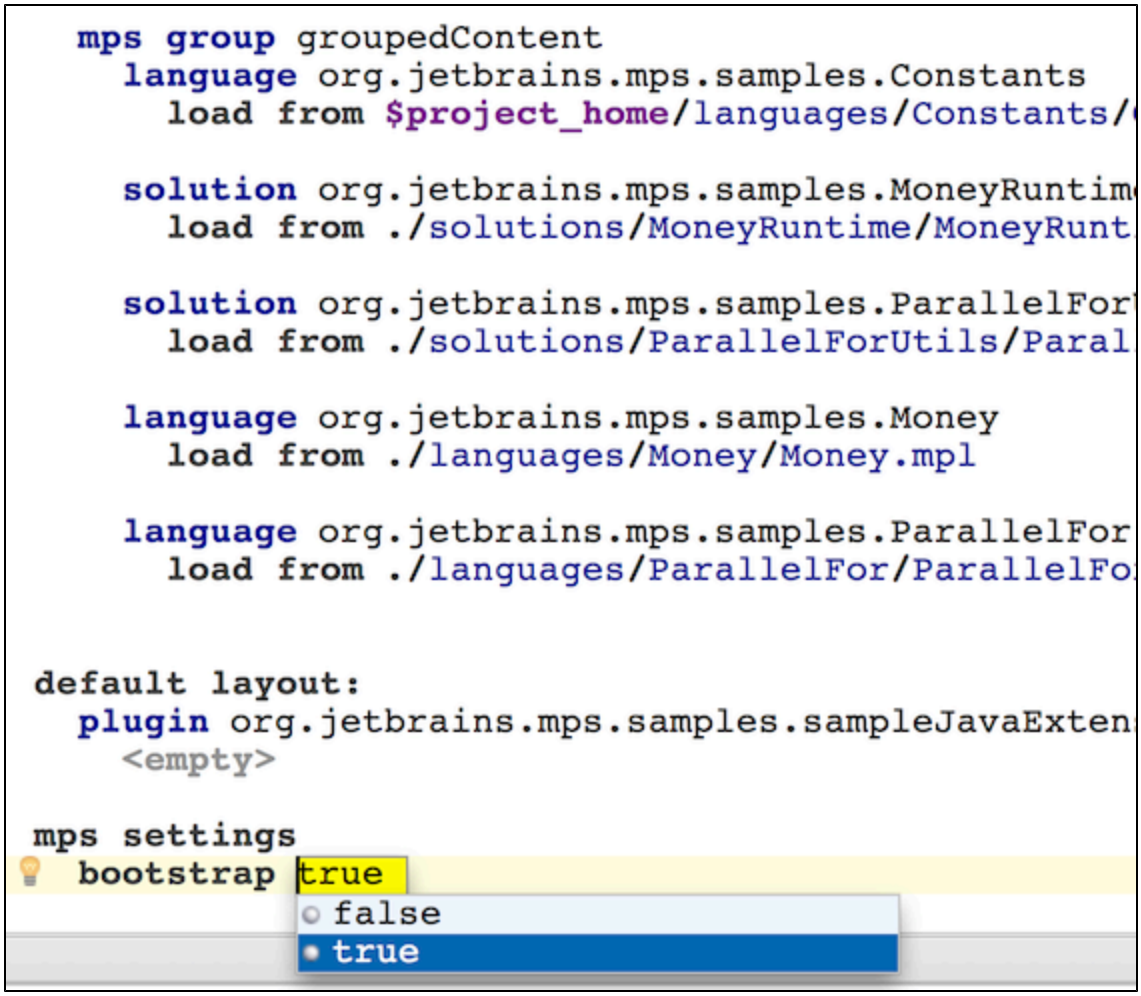
After attacking all problems individually, you need to optimize imports in your project and invoke the Reload modules from disk intention in the build script. This will clean up the dependency structure and allow your project to be fully rebuilt again.





A quick way to suppress the problem

A quick and dirty trick to suppress the error is to set the bootstrap property in the MPS settings section of the build script to tru
e.



Consider this to be a last resort way to temporarily enable building your project. Proper solutions with eliminating all offending bootstrapping dependencies should be preferred.