1 Preparation of the working environment

The Magda IDE is implemented as Eclipse plug-in[1], and it is released in the form of two *jar* archives, one for the *domain* and the other for the *ui*, both contained into the folder *mtj/Eclipse plugin/*[2]. After installing the plug-in by copying the two *jar*s under the folder *plugins/* of Eclipse and by restarting Eclipse, the content of the folder *mtj/* must be imported into a new Eclipse project. To do that, you have to choose from menu *File* → *New* → *Project...*, then, after selecting *General* → *Project* from the window that will open, on the click on *Next* the window in Fig. 1 will be shown; after inserting the name for the project (e.g., *MagdaProject*) and selecting the destination folder, it is possible to create the new empty project.

To fill this project, you can choose *File* → *Import...* from the menu, and then select, in the dialog window that will open, *General* → *File System*, by clicking on *Next*. In the window that will follow (Fig. 2) you must choose to import from the directory *mtj/*, by selecting therefore the following folders:

- *Magda*, which contains the prototypical implementation of the language, which is based on Java classes and organized into subfolders[3]; in particular, the *src/*

[1]You can download Eclipse at `https://www.eclipse.org/downloads/`.

[2]Notice that this folder also contains other two *jar*s with the corresponding source code of the *domain* and *ui* packages.

[3]The prototypical compiler of Magda, which is actually a preprocessor, given a Magda source code in input, returns some Java classes with the same behaviour in output. As a consequence, this Java classes obviously must be compiled then with the standard Java compiler `javac` before they can be executed. You can download Java at `http://www.java.com/download`.

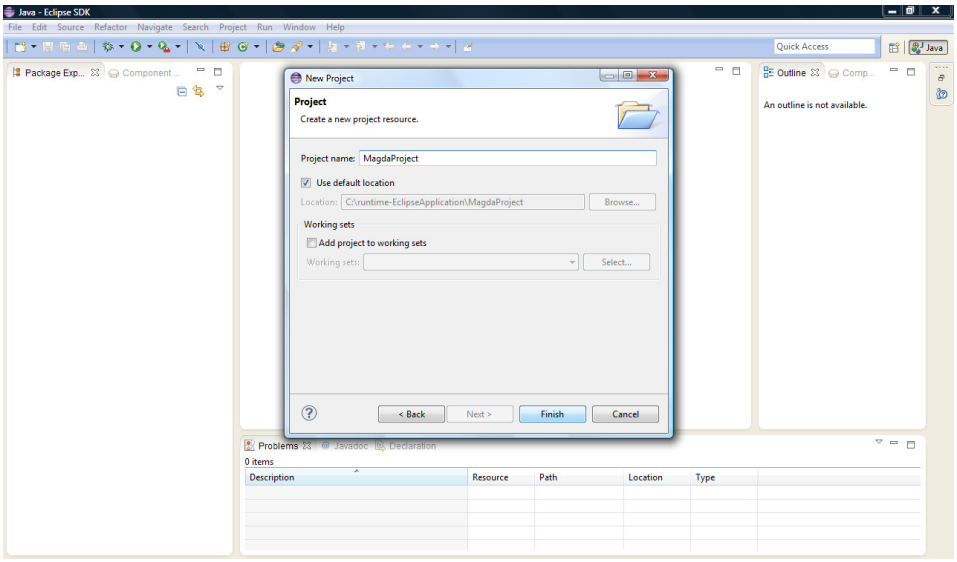

Figura 1: Creating an Eclipse project for Magda

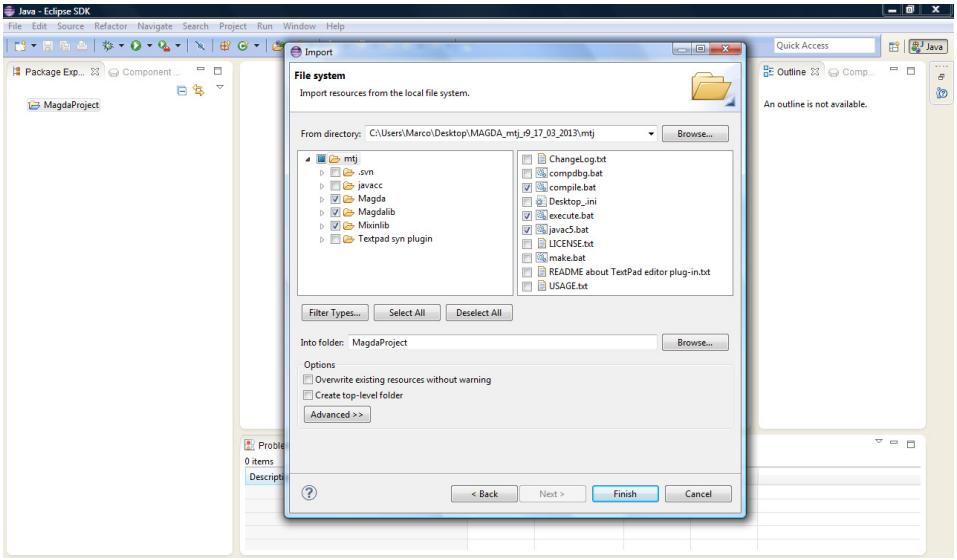

Figura 2: Importing from the *mtj/* folder

subfolder contains the user Magda programs;

- *Magdalib*, which contains the *built-in types* of Magda (e.g., `Integer`, `Float`, and so on), defined as mixins, as well as a file *library.magda* that includes all of them;
- *Mixinlib*, which already contains some example mixins, but where the user can also define new mixins for his/her own applications, to include by the `include` directive.

It is indeed necessary to import in the project the following batch files, essential for compiling and executing of the Magda source code into the IDE:

- *compile.bat*, which allows to compile a Magda program;
- *execute.bat*, which allows to execute a Magda program;
- *javac5.bat*, which simply calls the Java compiler `javac`, and it is used by *compile.bat*.

Specific attention should be paid to the *Magda/src* folder, which contains all the programs written by the user in the Magda language. The constraints of the current prototypical implementation of the language are the following:

- you must create a dedicated folder under *Magda/src* for each new program; this folder must have necessarily the same name of the source file which contains the program (without extension);

- within this folder, you must create only one file with extension *.magda*, which will contain the source code of the program; therefore, you currently can have just a single source file for each program, but you can include other files into it (from *Magdalib* and *Mixinlib*).

The Magda prototype also requires that into the folder created for a user program it is created an empty subfolder *obj/*[4]; however, creation of this folder is automatically handled by the IDE, so it must not be done manually by the programmer.

2 Compile and run

Once implemented a Magda program in a source file *.magda* placed in the folder used for the program within *Magda/src/*, it is possible to compile and run it into the IDE by selecting from the menu *Run* → *Run As* and then, respectively, *Magda Compiler* or *Magda Runner*; the execution result will appear in the *Console View*, on the bottom right into the IDE (Fig. 3).

[4]Into this folder, at compile time it will be generated the Java source MagdaProgram.java which represents the translation in Java of the Magda progra. The *Magda/obj/* folder, therefore, will contain also, after the compilation, some files *.class* that are nothing but the result of the compilation of such intermediate Java source.

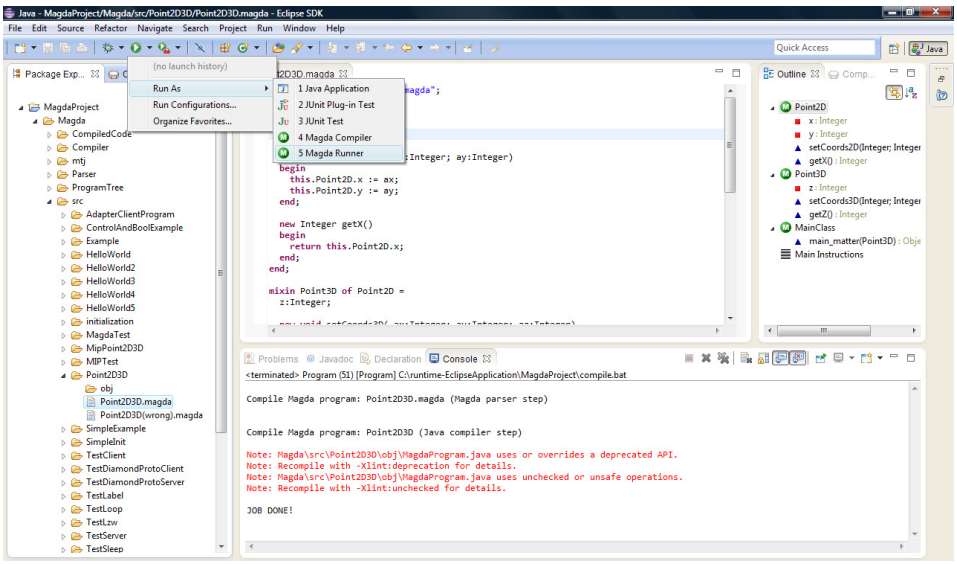

Figura 3: Compile and run

3 Extended and shortened views

By opening a Magda source file in the editor, for example by double-clicking it on the left in the *Resource Navigator*, it will be displayed by default in extended view (Fig. 4): the visible and editable syntax is the complete one of Magda, which uses hygienic identifiers, so each method call or field selection is prefixed by the name of the mixin into which such a method/field was introduced. By selecting from the context menu the item *Switch Long/Short View* (or by pressing the key combination *CTRL + 1*), it is possible to switch to the shortened view (Fig. 5), in which identifiers become non-hygienic (the mixin names are hidden here); the editor provides visual feedback of the transition to one view to the other by changing the background colour, that from white becomes coloured.

In the shortened view, the mixin names are hidden and not directly typable by the user. So they are managed by the IDE in this way: if the identifier is not ambiguous, the IDE adds automatically the only mixin name to the non-hygienic method call or field selection; otherwise, if the non-hygienic identifier is ambiguous (there are different versions of the method/field in more than one mixin), then the IDE signals an error to the programmer and it suggests, by a *quick fix*, the different versions of the method/field, thus allowing him/her to easily resolve the ambiguity (Fig. 6).

Anyway, during typing of code in the editor, by pressing "." after an object (or by pressing the key combination *CTRL+ space bar*) it is displayed the autocomplete drop-down (Fig. 7), which proposes to the user the callable methods and the selectable fields on the current object; on selection of a method or a field, the IDE prefixs automatically the mixin name to it.

In the case of ambiguity, also after the user has resolved it by selecting a mixin name, the editor will show an info (represented by an icon with a light bulb and an "i"), which indicates the existence of at least another version, in some other mixin from which the object was created, of the called method/selected field; the user can therefore even change the name of the chosen mixin through the same quick fix seen previously (Fig. 8).

4 Mixin sequences in the creation of an object

The IDE supports the programmer to respect the constraints on the mixin sequence in an object creation as follows:

- If you try to assign to a variable the result of an object creation, the created object must be coherent with previously declared formal type; otherwise, the IDE points

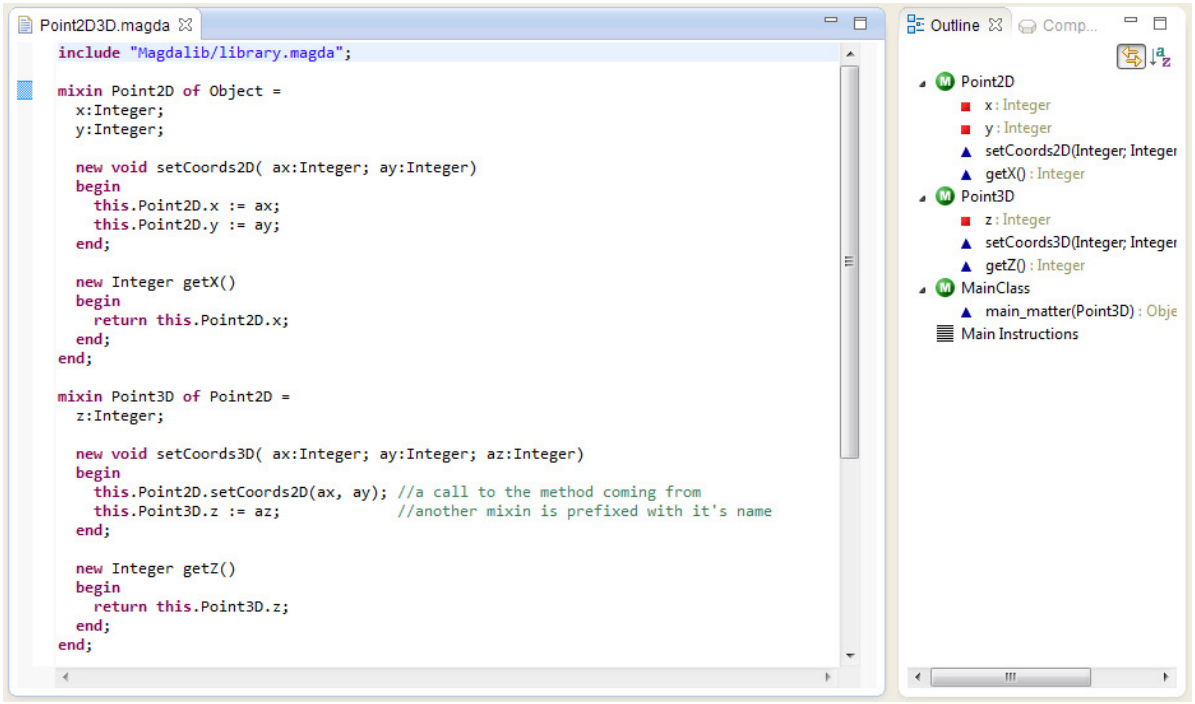

Figura 4: The extended view

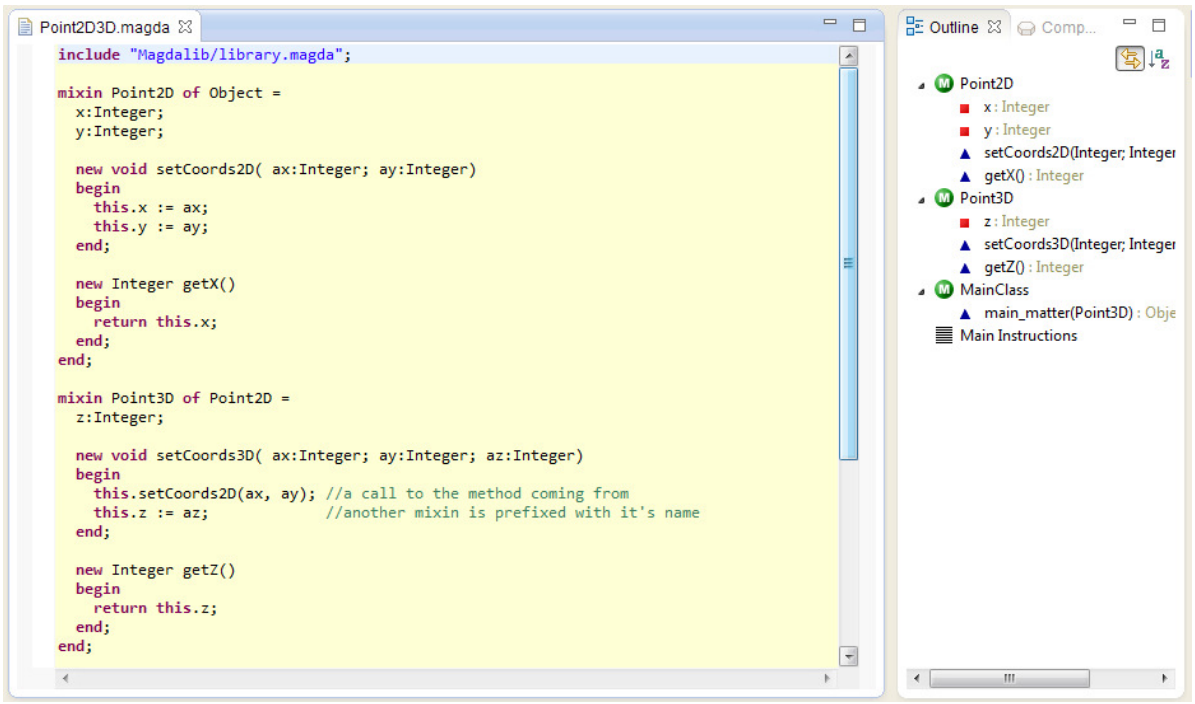

Figura 5: The shortened view

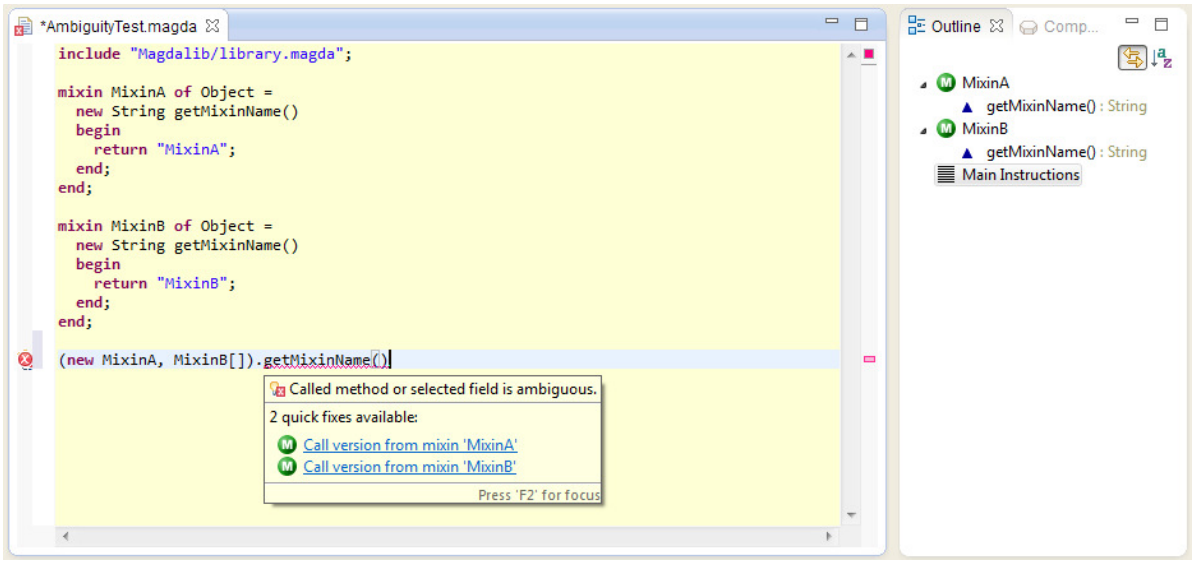

Figura 6: Quick fix for an ambiguous non-hygienic identifier

out the incoherence with an error and proposes two different quick fixes (Fig. 9):

  - the first one adds the missing mixins from declared formal type;
  - the second one conversely removes such mixins from declared formal type.

- If the programmer omits one or more base mixins, the IDE points out this fact with an error and proposes a quick fix which allows to add automatically, and by

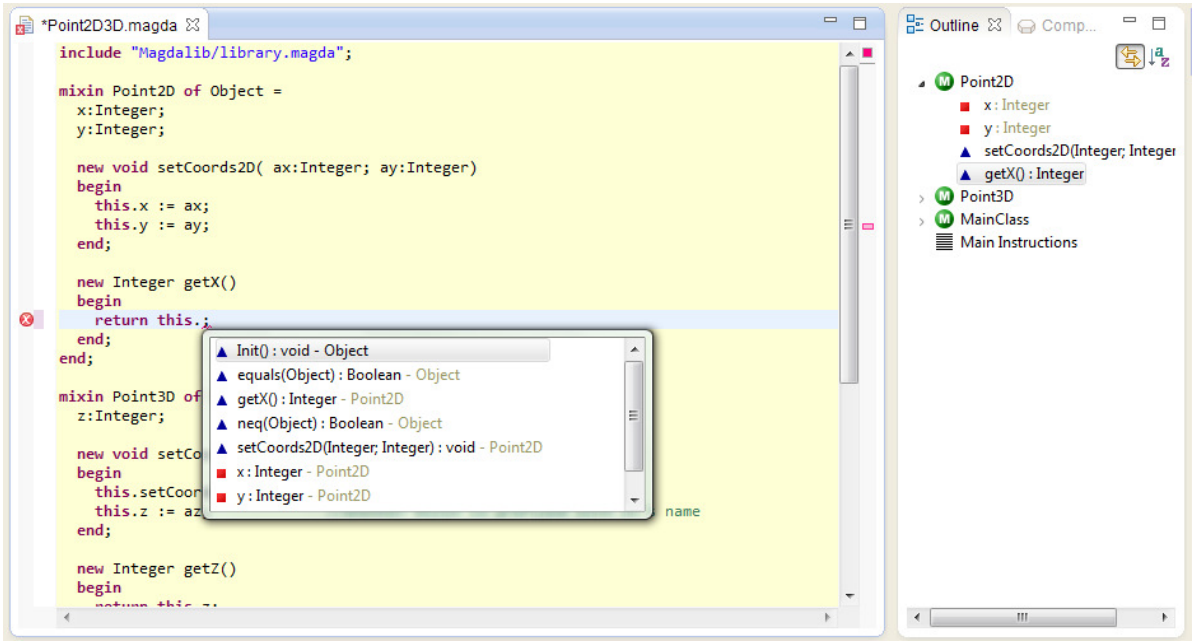

Figura 7: The content assist functionality

Figura 8: An info points out other versions of the method/field

respecting the ordering constraint, the missing mixins to the creation sequence of the object (Fig. 10).

- If a base mixin is specified in an object creation sequence, but *after* one or more mixins that extend it, then the IDE points out the error and allows to automatically solve it (Fig. 11).

- The IDE shows an info every time the object creation sequence contains at least a

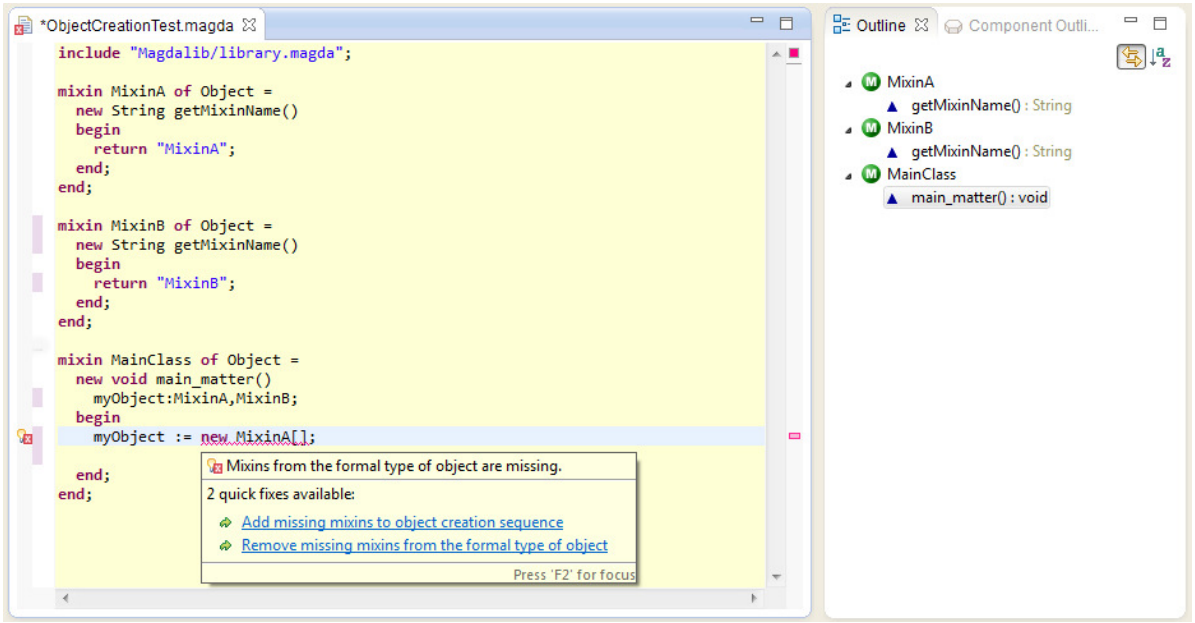

Figura 9: Missing mixins from declared formal type

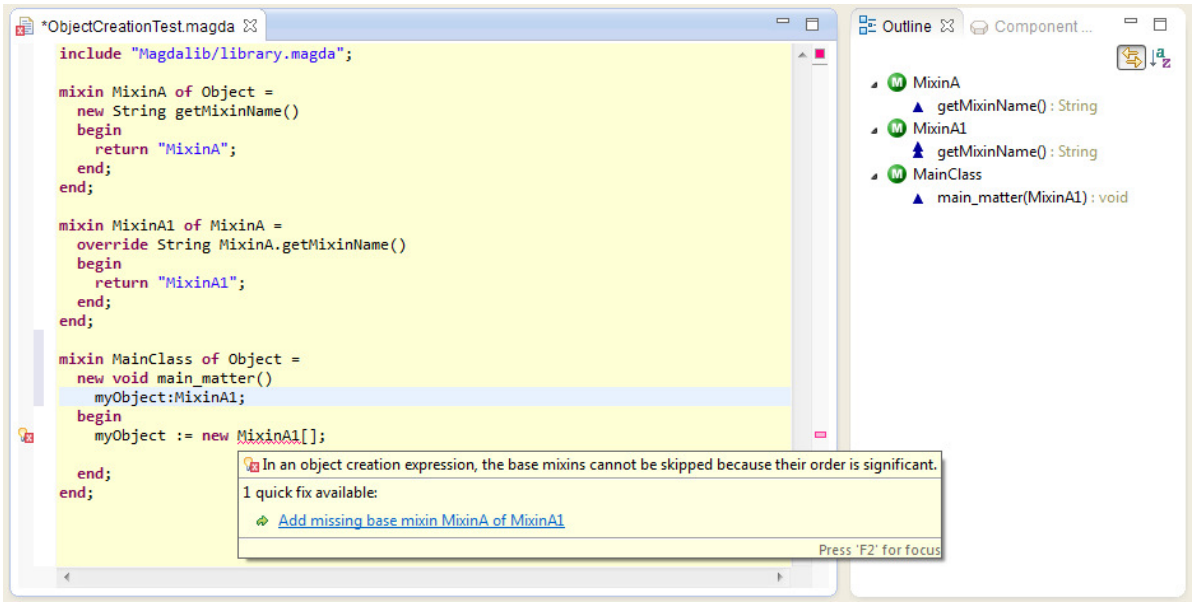

Figura 10: Missing base mixins in a `new`

pair of mixins such that (1) both do not appear in the fully expanded form of the other and (2) have at least a common ancestor: in this case, the two mixin can be swapped and their different ordering in the creation sequence could also alter the behaviour of the program (Fig. 12).

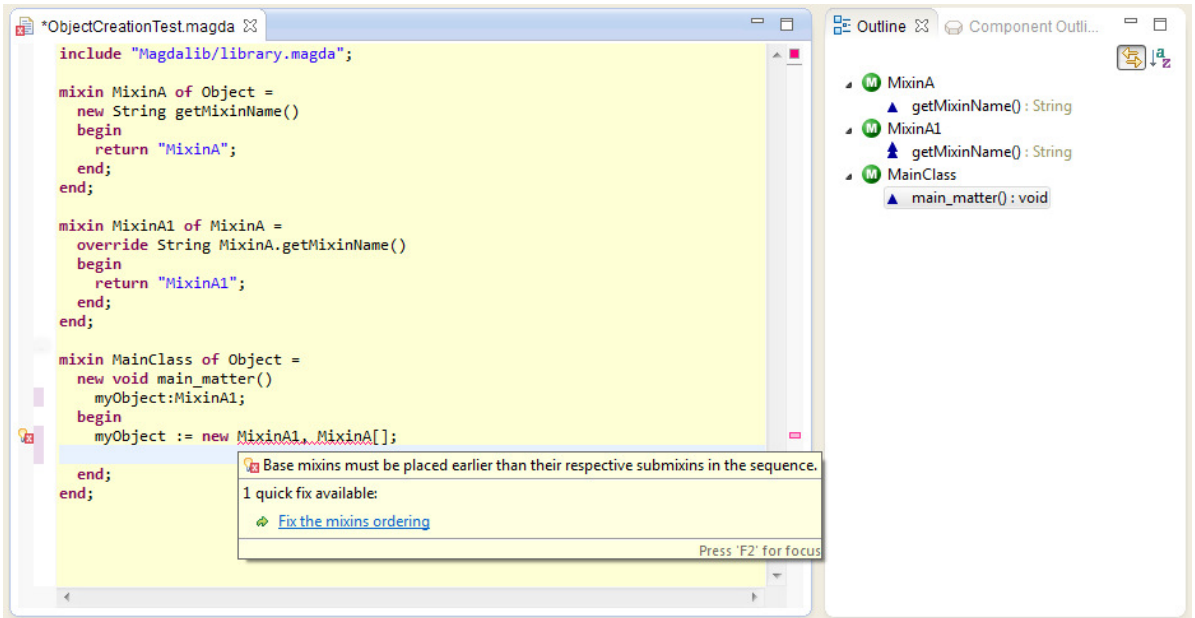

Figura 11: Wrong ordering of mixins in a `new`

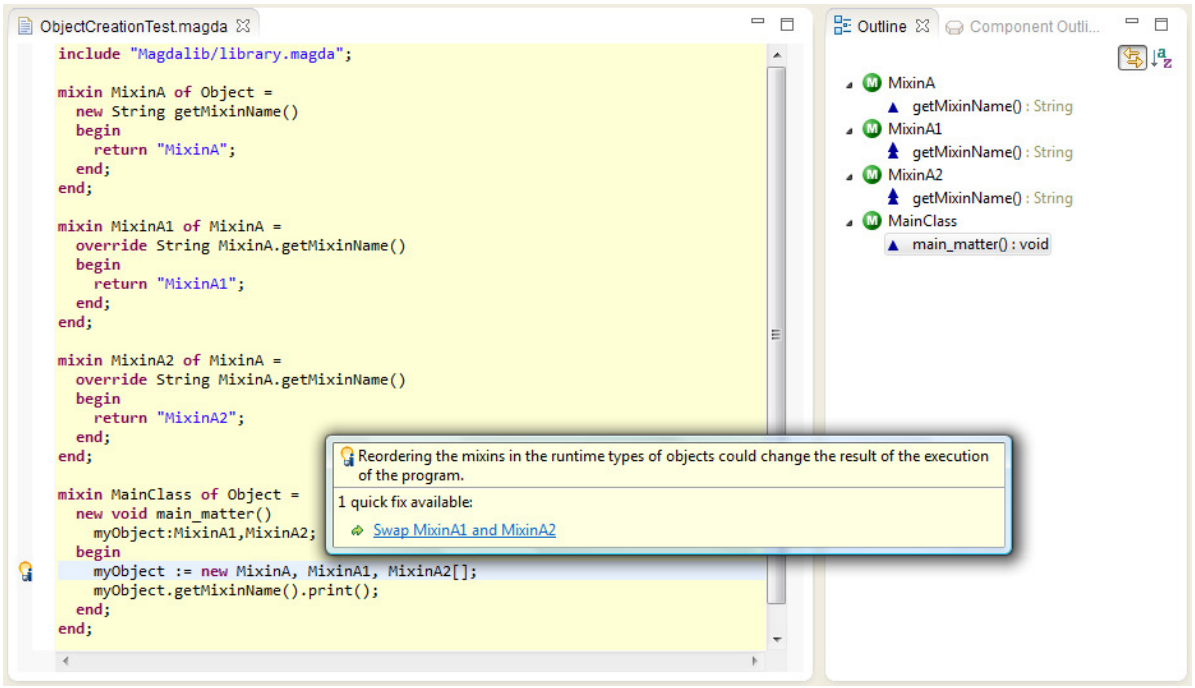

Figura 12: Interchangeable mixins in a `new`