Using Perfection

Introduction

Perfection is based on Eclipse and comes with content-assist, syntax highlighting, validation and hyperlinking.

Perfection can check many aspects of P4 code for errors without needing to actually run the code. The Perfection builder applies to all files in all P4 projects in the workspace, and will make all errors and warnings visible in the Project Explorer and Problems views. All the errors are marked in the P4 editor when it is opened so the developer can easily see the problems and fix them straight away, which is a great time saver.

The builder runs in the background when a file is saved, so it does not interfere with the responsiveness of the application. And, if a module is changed, then the builder is smart enough to re-compose all dependent files and update error markers accordingly.

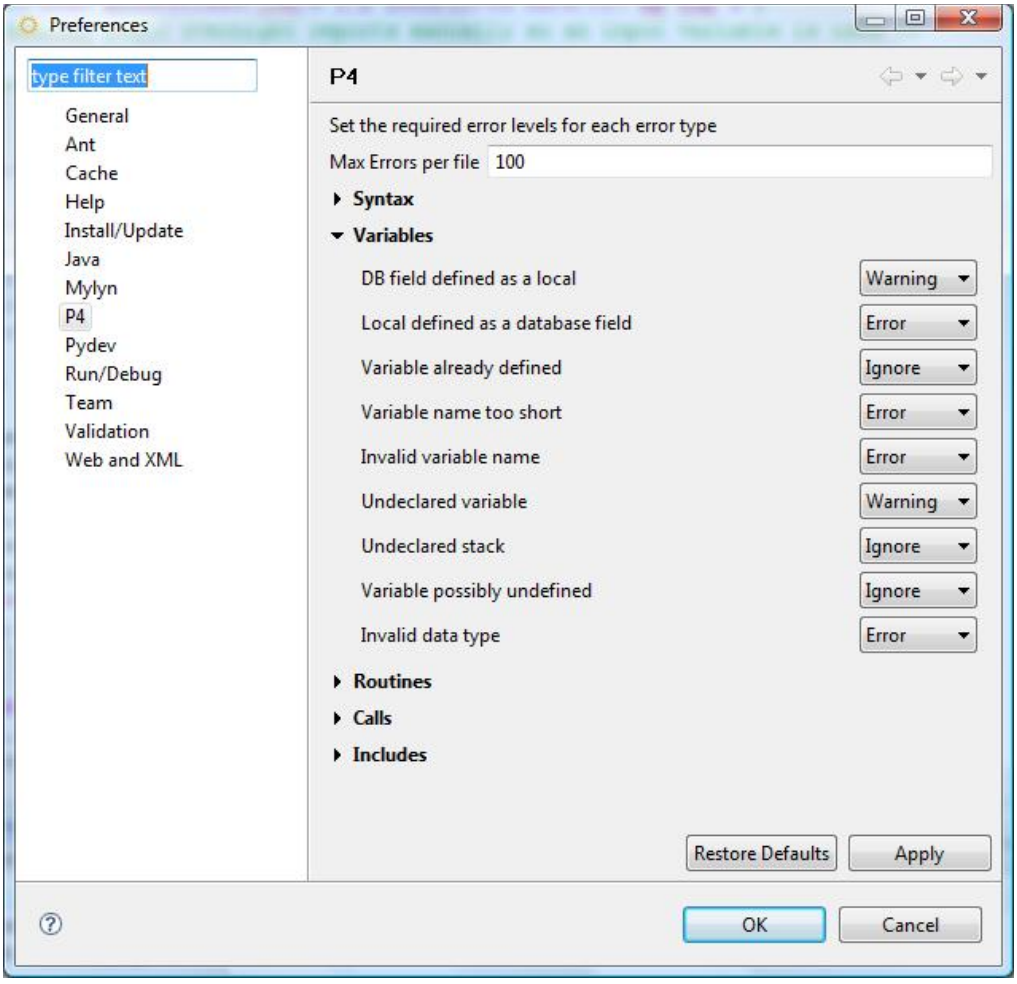
The rest of this document outlines some of the more advanced capabilities of the Perfection platform.

Errors, Warnings and Preferences

Perfection is capable of detecting the following types of errors and warnings:

- Many kinds of syntax errors e.g. missing brackets or curly brackets
- Database fields that are declared as locals
- Local fields that are declared as database fields
- Duplicate routines or functions
- Modules that are not suitable for inclusion at a particular code location
- Included files that are not modules
- Undeclared variables
- Undeclared stacks
- Routines that don't exist or haven't been included
- Variable names that are too short
- gosub's to a function
- Routines without exit or return statements
- Scripts without an endscript
- Unreachable code statements
- Invalid data types

Each type of error raised by the builder can have its setting changed to Error or Warning, or turned off completely. The errors levels are maintained as user preferences in the P4 Preference page:



The Builder in Action

Figure 1 below is an example of editing a P4 view in the current version of Perfection without the P4 builder in place. See how many errors you can spot.

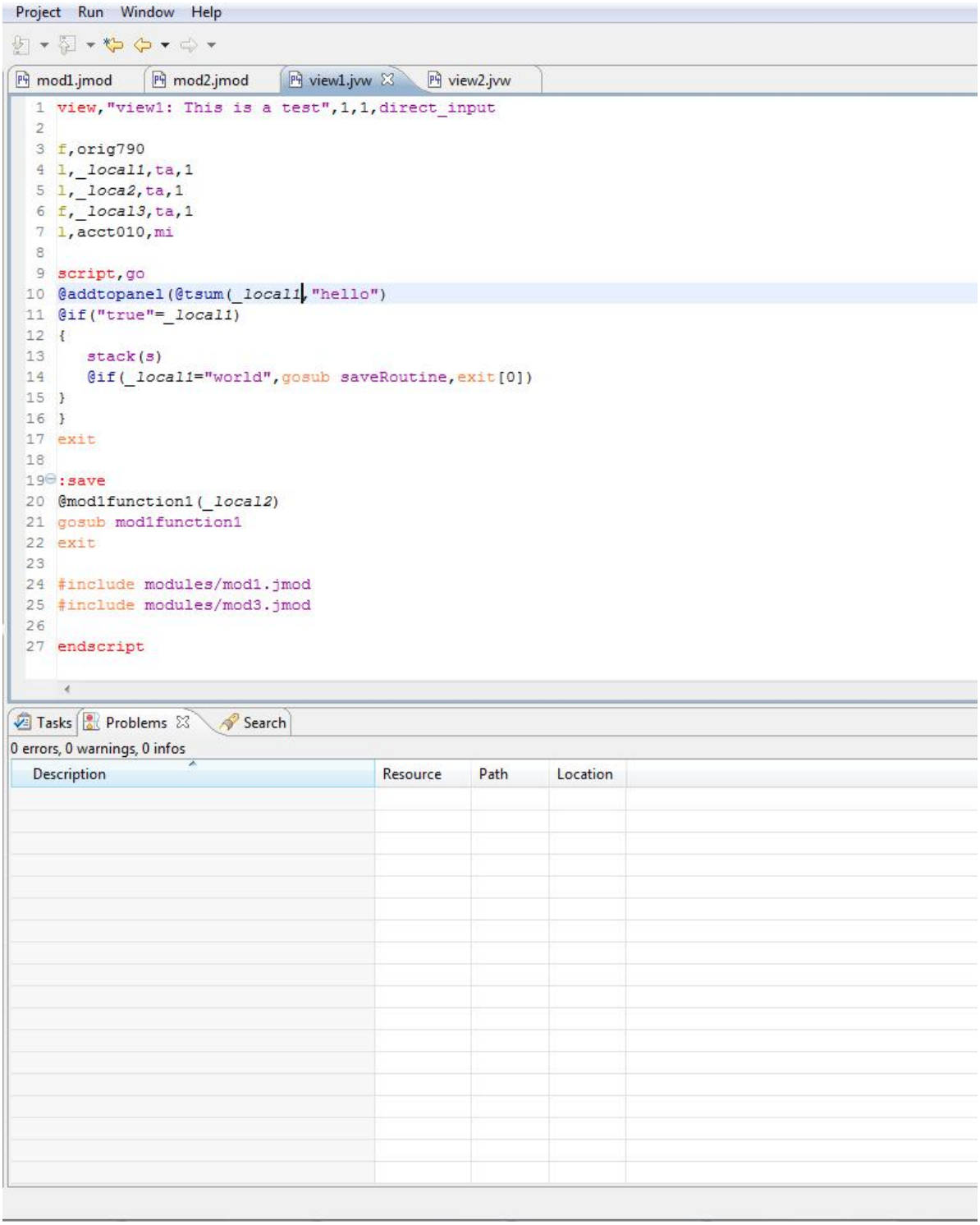


Figure 1 – Editing a P4 file without the builder

Now compare Figure 1 with Figure 2 below, where the builder has been turned on. In both figures it is the same file being edited, except that the builder has picked out many errors for you:

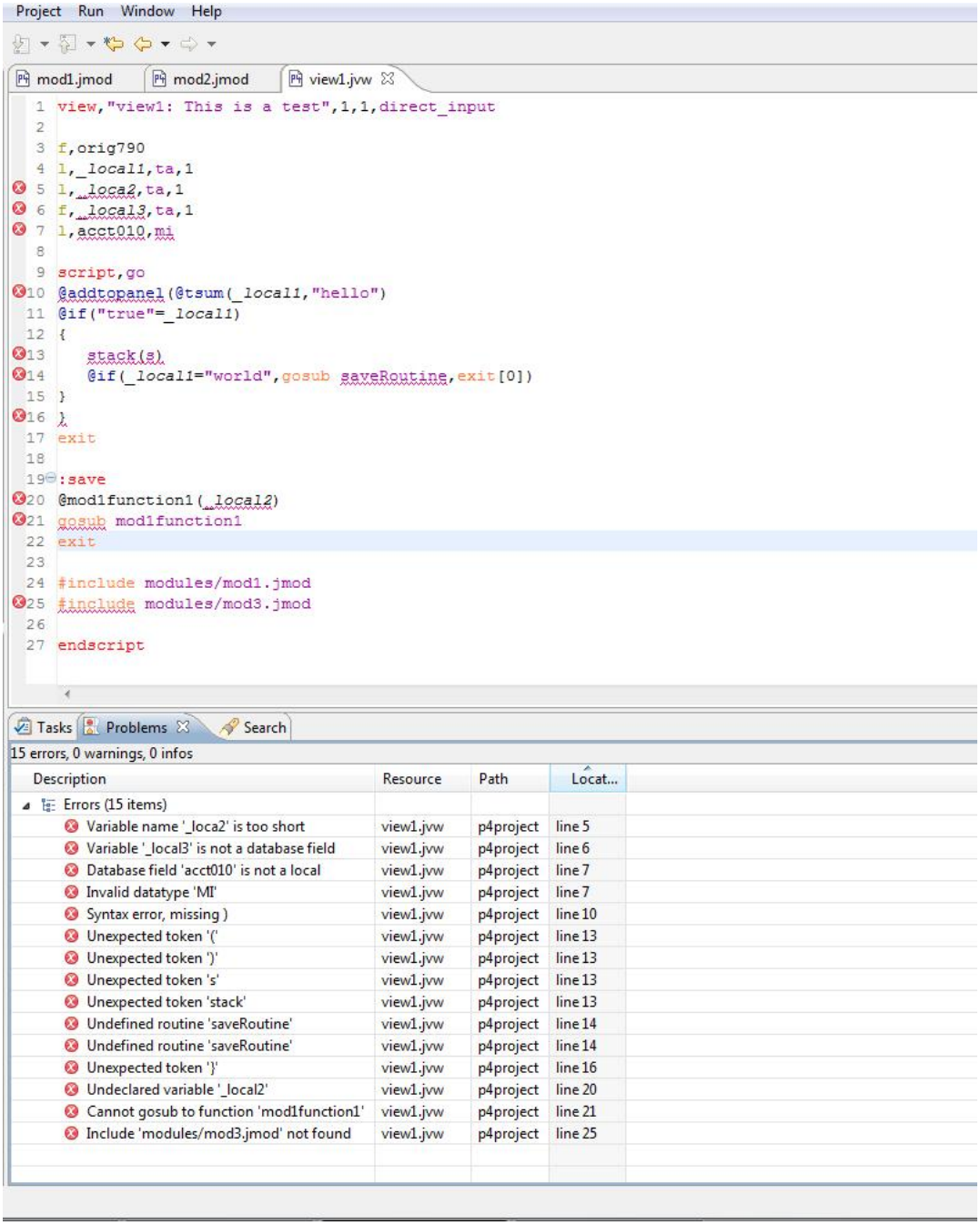


Figure 2 - Editing the same P4 file with the builder

In the past, these errors would have to have been found by the developer executing the code and laboriously fixing all the issues one by one. Or, worse still, Phoebus would have failed silently and maybe not thrown an exception, but not done what the programmer expected either.

Note: please ensure you are looking at the Project Explorer and not the Navigator, as this latter view will not display error markers on views. To open the Project Explorer, either go to the P4 perspective and reset it, or just open the Project Explorer view from the view menu.

Enhancements to the P4 Editor

Another big advantage of the builder is that it can be used for other important aspects of P4 code editing.

Variables

The P4 editor has now been enhanced so that it now provides context-sensitive content assistance, including local variables as in Figure 3 below. Just press Ctrl+Space to get such proposals within a function call.

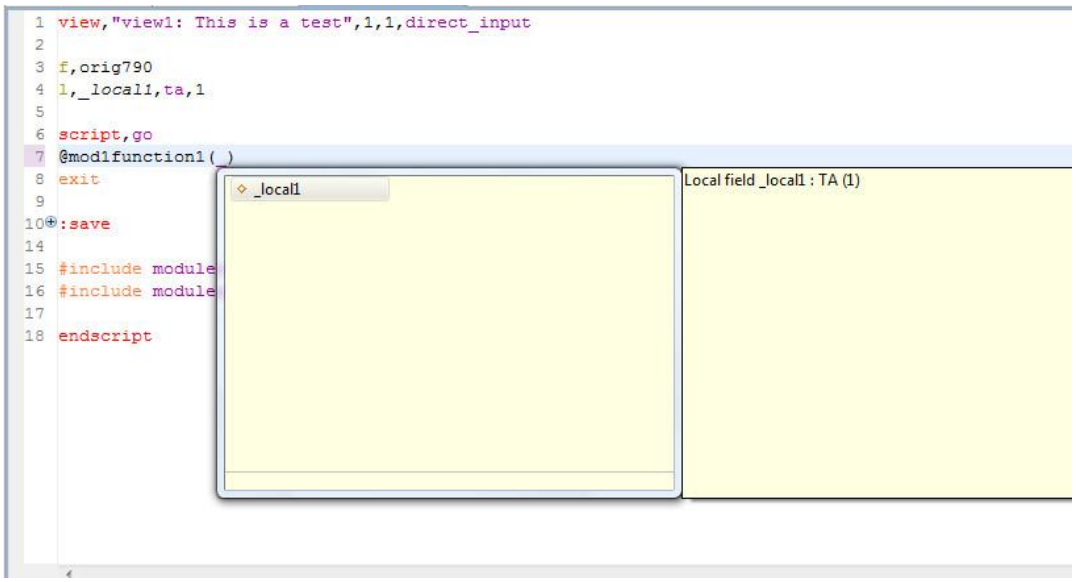


Figure 3 - Local variable proposals

If a variable is defined then at lines of code before that definition the variable will not appear as a proposal within the content assist. But for lines of code after the declaration, the variable is proposed (but only as an argument within a P4 function call).

You can hyperlink to the variable definition by pressing the Shift key while hovering over the name and clicking the mouse, as in Figure 4:

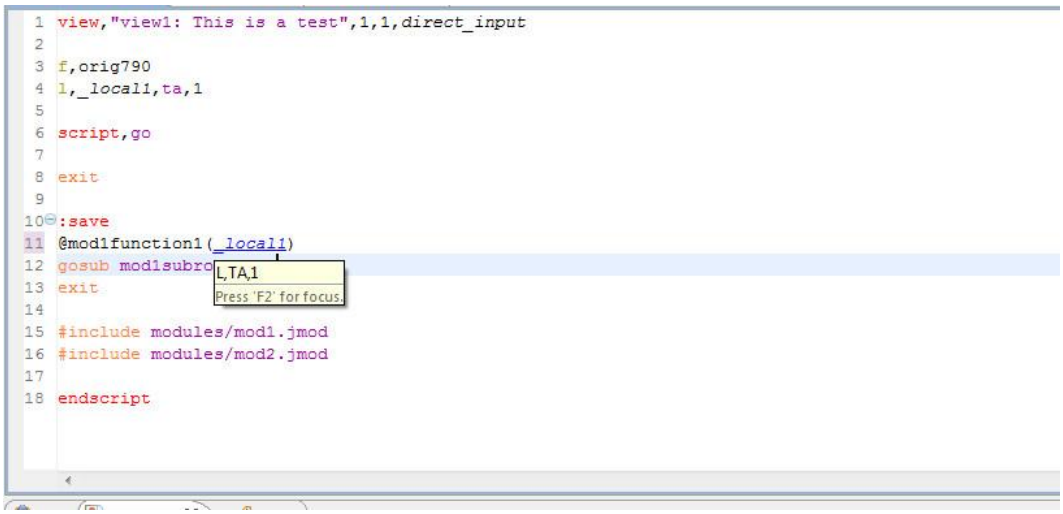


Figure 4 - Local variable hyperlinks and hover

Functions and Subroutines

Similarly to variables, the builder is able to suggest functions and subroutines as proposals for function calls, gosub's and goto's – see Figures 5 and 6:

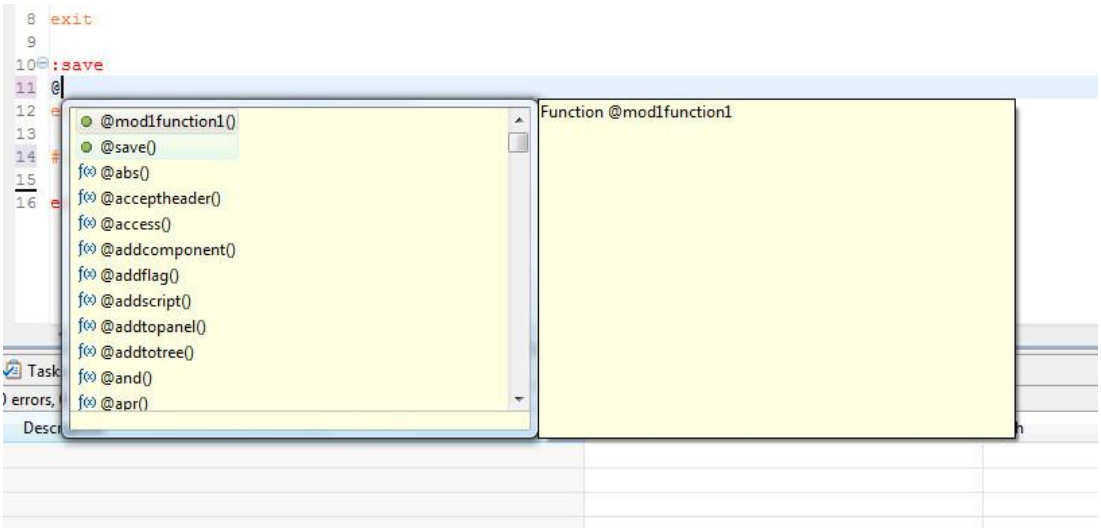


Figure 5 - Function proposals

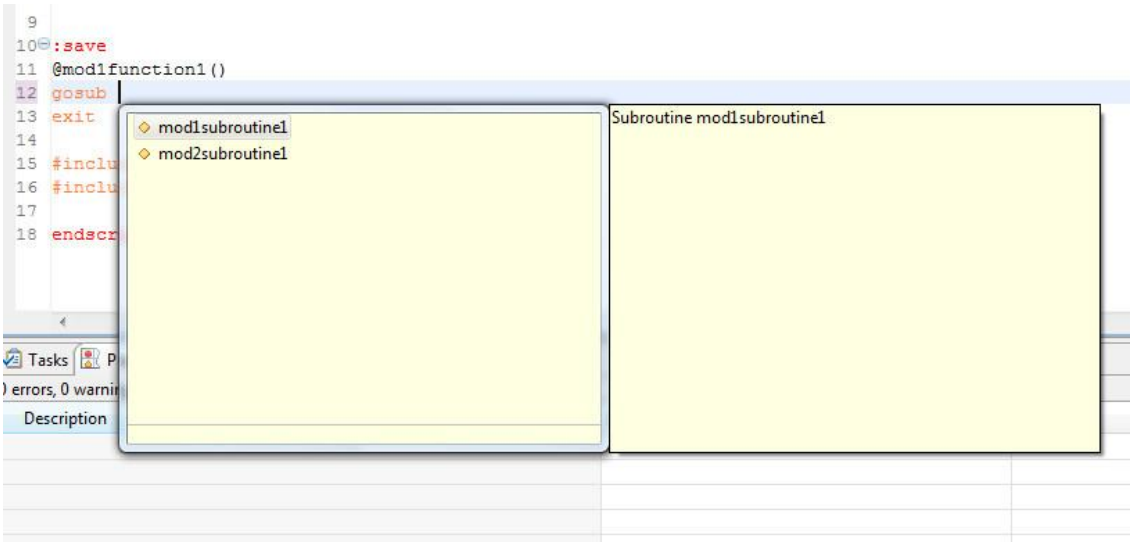


Figure 6 - Subroutine proposals

Also, since the builder knows where exactly a function has been defined (and in which file), the user is now able to move to the definition of the function simply by hovering over the function name with the Shift key down. Clicking the mouse on the hyperlink takes you to the function definition, as in Figures 7 and 8.

Note: To get back to your previous position, you can use the forward and back navigation buttons on the toolbar.

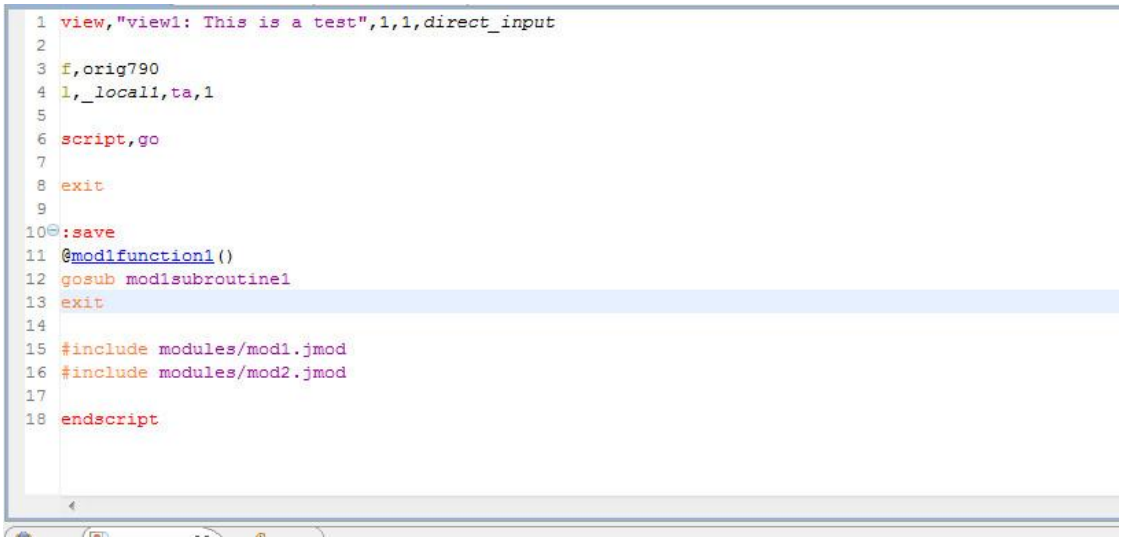


Figure 7 - Hyperlink to functions

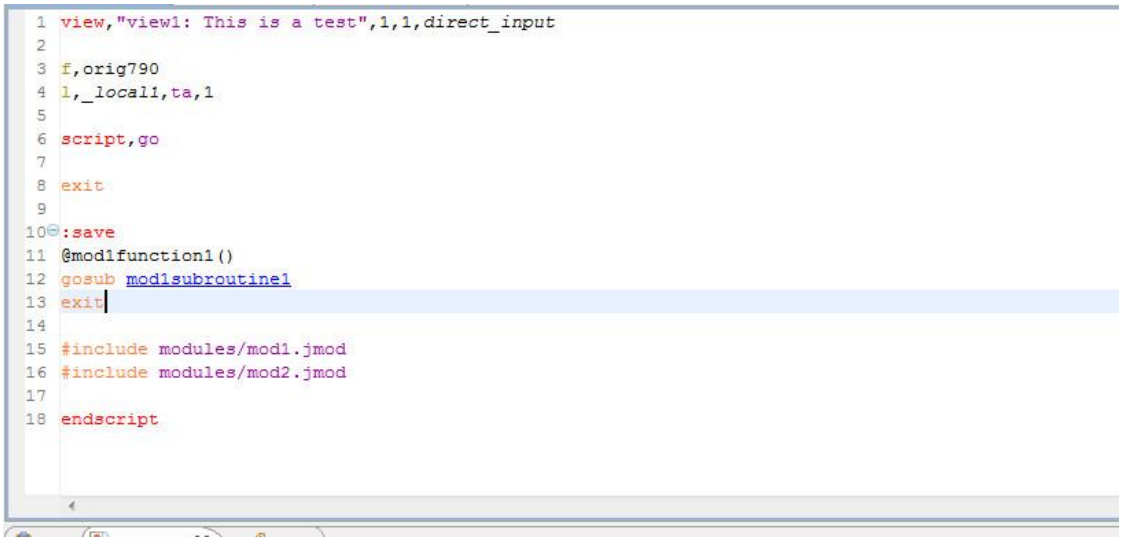


Figure 8 - Hyperlink to subroutines

Includes

The P4 editor now suggests modules that can be added to the view when pressing Ctrl + Space after typing #include:

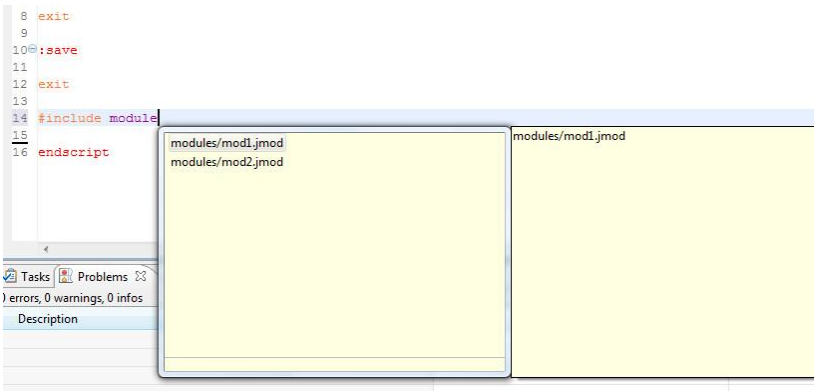


Figure 9 - Include proposals

Project Setup

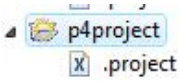
If you create a new P4 project from scratch, you need not do anything at all because the builder will automatically be added to the project.

For existing projects, you must add the builder by editing the .project file. Open the Navigator view and edit the file in the root of the project so that it looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<projectDescription>
	<name>NAME_OF_YOUR_PROJECT</name>
	<comment></comment>
	<projects>
	</projects>
	<buildSpec>
		<buildCommand>

	<name>uk.co.chasetechnology.perfection.p4.builder.builder</name>
			<arguments>
			</arguments>
		</buildCommand>
	</buildSpec>
	<natures>
	<nature>uk.co.chasetechnology.perfection.p4.core.p4nature</nature>
	</natures>
</projectDescription>
```

If things are set up correctly, your project should now build and show a sun overlayed on the usual folder icon like this:



Working with the P4 Builder

String or Variable?

In P4 strings do not need to be quoted, which means it is very difficult for the builder (and the programmer!) to determine what is a string and what is a variable. For example, if you have a statement such as @tsum(counter, ,times), which of the arguments are strings and which are variables? The answer depends on whether a variable called "counter" or "times" has been declared or not. But this is not always straightforward to determine, particular if the statement is within multiple nested function calls.

The builder takes a pragmatic approach to the problem:

- If an argument is a word containing a at least one letter and one of _, $ or a digit, then it is a variable
- Otherwise, it is a string

So, the builder considers all arguments in the above code statement to be strings. This is obviously not quite accurate since either argument could in reality be a variable, but it works in most cases since the majority of variables do follow the pattern.

However, to be sure that the builder detects as many problems as possible though, you should follow these two simple rules when developing your code:

- **Make sure all variable names contain a $, _ or a digit**
- **Surround all string literals with quotes**

Scope

The other issue that the builder must cope with is scope. In P4 more or less everything has global scope, which is fine until it comes to using routines.

A routine can freely make use of variables that have not been declared anywhere inside that routine (or the view or script that contains it). It is assumed that the caller has declared such variables before calling the routine, but this will not always be true and it is easy for the programmer to forget to do that.

In situations like this, the builder will place a warning next to such possibly undeclared variables. The recommended practice is to use a parameter instead of assuming a variable has been declared before the routine is called.

Another common situation is where you have code similar to below, where a variable is defined within the scope of an 'if' statement. Since the builder cannot verify whether _something really is true, then it must assume that $$myvar may not have been declared and raise a warning.

```
@if(_something=true)
{
   @def($$myvar,ta,1)
}
@ass($$myvar,"X") // raises an error
```