Surround Statement Refactoring

LemonMango
CS427 Fall 2010
henke2, nramesh2, rudwick2, rsulliv7, sjohns67

Table of Contents

I. Refactoring Descriptions

A. Surround If Refactoring

1. The `SurroundIfRefactoring` class surrounds the selected code statements and surrounds them with an IF condition. The IF condition is specified by the user on refactor, and the validity of the argument passed is checked (variable exists, function exists, etc). The goal of this refactoring is to allow the user to quickly and simply specify IF conditions in Fortran.

Example:

```
program suround_if_4
    implicit none
    integer :: i = 0
    print *, "Using PRINT: Iterator = ", i
    write (*,*) "Using WRITE: Iterator = ",i
end program
```

Highlighting and Refactoring the second and third lines using i < 0 results in...

```
program suround_if_4
    implicit none
    integer :: i = 0

    IF (i < 0) THEN
      print *, "Using PRINT: Iterator = ", i
      write (*,*) "Using WRITE: Iterator = ",i
    END IF
end program
```

Refactoring must preserve the original intention/functionality of the code. Surrounding statements with IF will change the code's functionality if used in certain ways. The initial precondition check makes sure that no statement was highlighted in the middle, or any improper statements were selected for the IF body (`implicit none`, etc). The final condition check makes sure the condition specified by the user is valid. If the program specifies `implicit none,` the variable inside the statement must be declared before the statements highlighted for refactoring. This also applies to subroutines being called in the statement and the variables being passed to those subroutines.

Example:

```
program suround_if_4
    implicit none
    print *, "Using PRINT: Iterator = ", i
    write (*,*) "Using WRITE: Iterator = ",i !<<<<< 3, 0, 4, 45, i > 0, fail-final
end program
```

Refactoring with the condition `i<0` will result in a failure because `implicit none` requires a declaration. Similarly, if the `implicit none` statement were not included in the program, the refactoring would be valid and allowed.

2. The `SurroundIfRefactoring` class only checks that refactoring is enabled in `doCheckInitialConditions` because the user doesn't specify the IF condition until the `doCheckFinalConditions` method. The `doCheckFinalConditions` is responsible for checking if `implicit none` is declared, as well as checking the for the declaration of the variable specified. The method finds all the declarations in the program before the highlighted statement using `declaredLoop.` If the condition specified is a function call, the method calls `isFunctionNode,` finds the variables passed to the function and searches for declarations and the function definition.

`doCreateChange` performs the refactoring. It calls `insertBlock` after it has created the proper IF block node. In order to deal with nested statements, `insertBlock` finds the body that contains the highlighted statements by using `rootNotNullBranch` and recursing to find the proper body node. Once the node is found, the correct body will get the new `ASTIfConstructNode` inserted into it and reindent and update the program.

B. Surround Block Refactoring

1. In our class `SurroundBlockRefactoring`, we implemented the functionality to surround sequences of statements with a BLOCK construct. The BLOCK constructs function is to perform local computations that do not have consequences outside of the BLOCK. For instance, if you declared A = 1 before a BLOCK, and then performed A = A + 1 inside of the block, A would equal 2 inside the block. Once the program reached the end of the block, however, A would revert back to its value before the BLOCK was started. In this case, A would equal 1 after the block.

Example:

```
program extract_local
    implicit none
    A = Transpose (A)
    LU(A)
end program
```

after refactoring:

```
program extract_local
    implicit none

    BLOCK
      A = Transpose (A)
      LU(A)
    END BLOCK
end program
```

For more examples, please visit https://agora.cs.illinois.edu/display/cs427fa10/Surround+statements.

It is important to check that refactorings conserve functionality of written code, and either a warning or error is thrown when there is a possibility that a refactoring could change how a program runs. Therefore, we do both initial and final precondition checks on our surround block refactoring operation.

The purpose of initial precondition checks is to make sure there are not any problems with the code highlighted to be refactored. For instance, we check that the variables assigned in the highlighted selection of code are not used for computation after the selection; adding a block to this code would change it's functionality and therefore be an unsound refactoring.

Example:

```
program extract_local
    implicit none
    A = Transpose (A)
    LU(A)
end program
```

Cannot be refactored to:

```
program extract_local
    implicit none

    BLOCK
      A = Transpose (A)
    END BLOCK
    LU(A)
end program
```

Final precondition checks are done to check for any errors that may result from user input. In the case of Surround Block Refactoring, the user has the option to enter a temporary variable that they would like removed during the refactoring. If this temporary variable is entered, we check to make sure it exists before the highlighted selection, has been initialized to value, and is not used after the BLOCK statement in a way that would alter the functionality of the program.

2. There are a few functions beyond the standard overridden methods – `doCheckInitialConditions`, `doCheckFinalConditions`, `doCreateChange` – that contribute to surround block's functionality and are worth mentioning here.

First, the function `getNameNodesFromExpression()` is used to recursively retrieve `ASTNameNodes` from an expression passed in through the node variable. The function requires a helper variable, called `nameNodes`, which keeps track of `ASTNameNodes` that have already been found. There are two base cases in this function, one when an `ASTNameNode` is found, and the other when an `IASTNode` has no children. In the case where the `IASTNode` does have children, the function recurses into each of them.

`GetNameNodesFromExpression()` is used in multiple places where the name nodes are needed to get `ASTNameNodes` of temporary variables to replace, the variables that replace temporary variables, and the variables that are used on the right hand side of an expression during and after a surround block refactoring is performed.

Because the node representing a function in Photran is also used to represent a variable, we needed to create the function `isFunctionNode()`, which distinguishes between a function and variable encapsulated by `ASTFunOrVarNode`.

The function `getUsedVars()` is a helper function for `doCheckInitialConditions` that finds the variables used in a selection so that we can make sure those variables are not used inappropriately later.

In order to replace the temporary variable, we created two functions called `replaceTempVariable()` and `replaceVar()`. The former function iterates over the statements that were selected for refactoring and calls the latter function to replace the variables on each line. `replaceVar()` is recursive, with a base case that returns when a `ASTNameNode` is found with the temporary variables name or an `IASTNode` has no more children.

`insertBlock()` wraps the selected statements with a block statement, and it does this by creating a new body surrounded by block and inserting each statement into that body.

C. Surround Critical Refactoring

1. The purpose of the `SurroundCriticalRefactoring` class is to surround a block of statements with `CRITICAL` and `END CRITICAL`. The purpose of the critical statement is to provide a solution for code that must be executed atomically in a multi-threaded Fortran program.

Note: `CRITICAL` functionality is only available in the newest version of Fortran (2008). The critical refactoring does not provide any checks other than the initial check to make sure refactoring is enabled. This is due to virtually any lines of code being surroundable. The user interface warns the user of potential functionality change when performing the refactoring.

Example:

```
program extract_local
    implicit none
    print *, "This is a test"
    print *, 3+4*5+6 !<<<<< 4, 0, 4, 21, pass
end program
```

Surrounding the statements with critical results in:

```
program extract_local
    implicit none

    CRITICAL
    print *, "This is a test"
    print *, 3+4*5+6
    END CRITICAL
end program
```

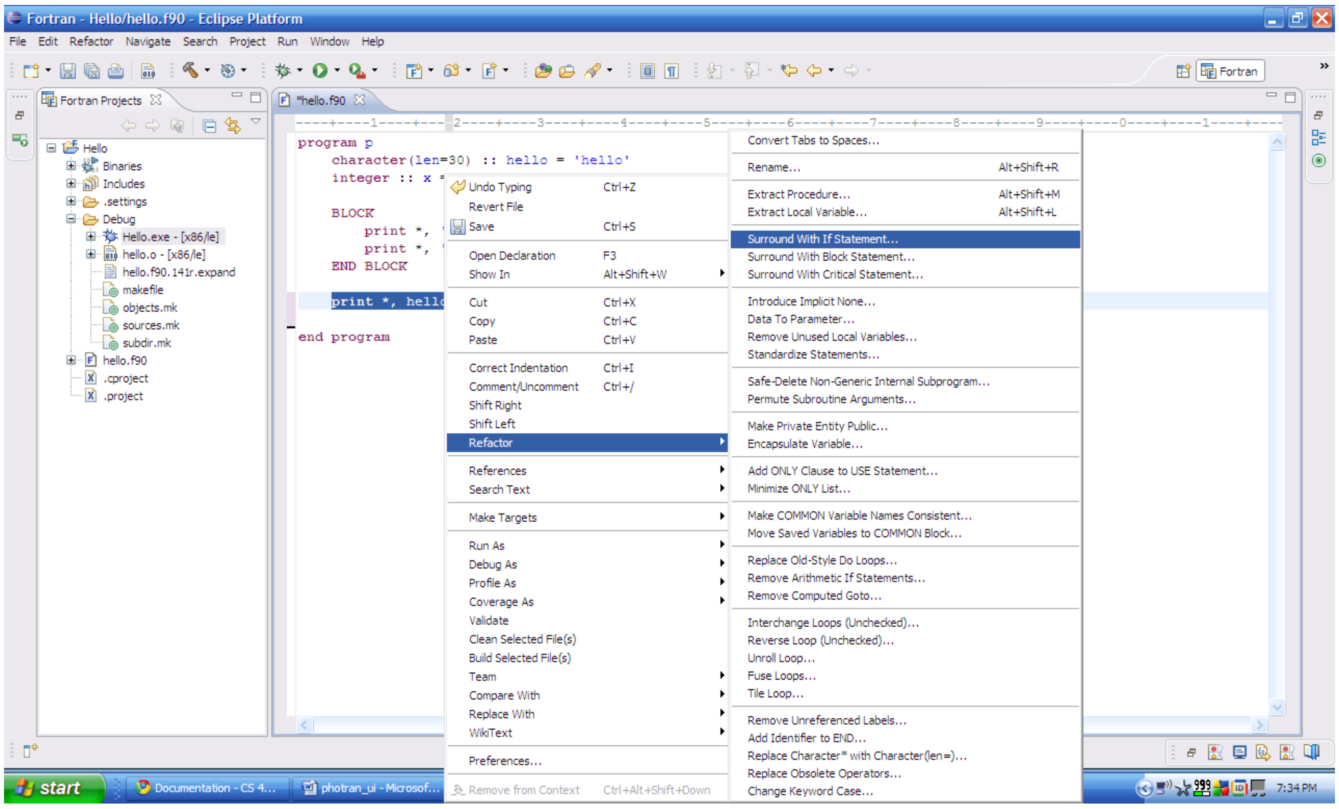
2. The methods used to implement the `SurroundCriticalRefactoring` class were less complicated than the previous two refactorings due to the nature of the `CRITICAL` statement. The reindenter didn't properly implement indentation for `CRITICAL`, so the refactoring doesn't properly

indent the block of code. The only important method is `insertBlock` which is called by `doCreateChange`. This function will check for nesting structure and find the correct body node to insert the new statements into. This is very similar to the way the code works in surround if.

II. User Interface Description and Implementation

Designing the UI component of the system was fairly straightforward. Our main project goal was to design three new refactorings in eclipse for Fortran. These included surrounding the highlighted block of code with IF, BLOCK and CRITICAL statements as requested by the user. First we had to introduce these new refactorings in the Eclipse view. This was achieved by modifying the plugin.xml file in the org.eclipse.photran.core.ui.vpg package where the Refactor menu is defined.

Since all three refactorings are similar editor refactorings, we put them in their own group and issued editor group refactoring commands for them. We then needed to define these commands by giving each refactoring a name, id and category. We decided to skip the option of setting a shotcut/accelerator key for each one. We defined new actions for the three refactorings and added the commands to photran's refactoring action set. Associating the action set with Fortran's perspective and editor ensured that the refactoring options actually showed up in the Fortran editor under the Refactor drop-down menu and also on right-clicking and selecting Refactor (Figure 2.a).

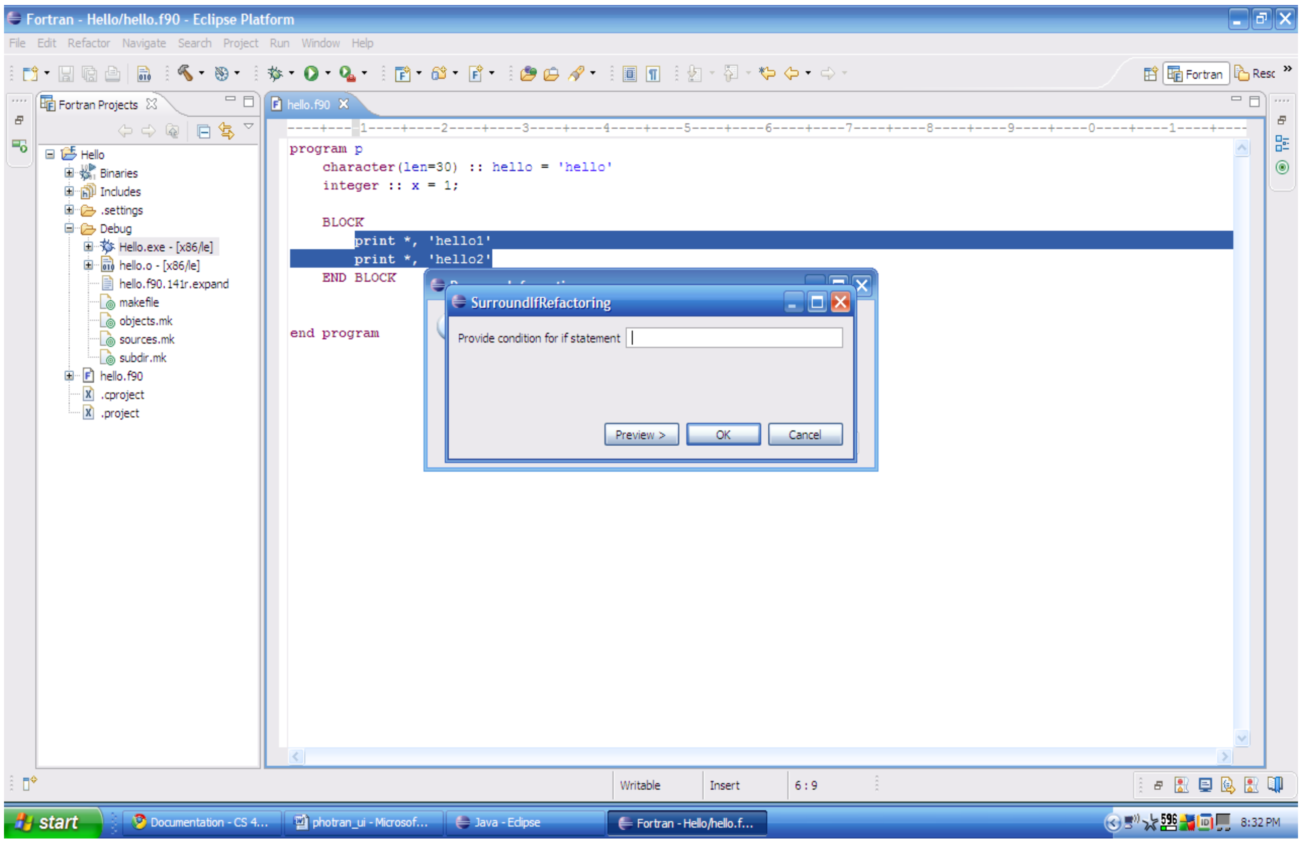

Fortran - Hello/hello.f90 - Eclipse Platform
File Edit Refactor Navigate Search Project Run Window Help
Fortran Projects
Hello
Binaries
Includes
.settings
Debug
Hello.exe - [x86/le]
hello.o - [x86/le]
hello.f90.141r.expand
makefile
objects.mk
sources.mk
subdir.mk
hello.f90
.cproject
.project
*hello.f90
Undo Typing Ctrl+Z
Revert File
Save Ctrl+S
Open Declaration F3
Show In Alt+Shift+W
Cut Ctrl+X
Copy Ctrl+C
Paste Ctrl+V
Correct Indentation Ctrl+I
Comment/Uncomment Ctrl+/
Shift Right
Shift Left
Refactor
References
Search Text
Make Targets
Run As
Debug As
Profile As
Coverage As
Validate
Clean Selected File(s)
Build Selected File(s)
Team
Compare With
Replace With
WikiText
Preferences...
Remove from Context Ctrl+Alt+Shift+Down
Convert Tabs to Spaces...
Rename... Alt+Shift+R
Extract Procedure... Alt+Shift+M
Extract Local Variable... Alt+Shift+L
Surround With If Statement...
Surround With Block Statement...
Surround With Critical Statement...
Introduce Implicit None...
Data To Parameter...
Remove Unused Local Variables...
Standardize Statements...
Safe-Delete Non-Generic Internal Subprogram...
Permute Subroutine Arguments...
Make Private Entity Public...
Encapsulate Variable...
Add ONLY Clause to USE Statement...
Minimize ONLY List...
Make COMMON Variable Names Consistent...
Move Saved Variables to COMMON Block...
Replace Old-Style Do Loops...
Remove Arithmetic If Statements...
Remove Computed Goto...
Interchange Loops (Unchecked)...
Reverse Loop (Unchecked)...
Unroll Loop...
Fuse Loops...
Tile Loop...
Remove Unreferenced Labels...
Add Identifier to END...
Replace Character* with Character(len=)...
Replace Obsolete Operators...
Change Keyword Case...
Fortran
start
Documentation - CS 4...
photran_ui - Microsof...
7:34 PM

```
program p
    character(len=30) :: hello = 'hello'
    integer :: x =
    
    BLOCK
        print *,
        print *,
    END BLOCK
    
    print *, hello

end program
```

(Figure 2.a: Fortran refactoring menu)

When the user attempts to incorrectly perform one of our refactorings by selecting an invalid block of code for example, then the UI outputs an error message since the refactoring cannot be performed. However when the user attempts to perform a valid surround if refactoring then the interface creates a user input dialog box where the user can enter the condition for the IF block. The user can also look at a preview of what the refactoring will do and then decide whether or not he wants to proceed.

For SurroundBlock refactoring, the process is similar with the only difference being that instead of entering an entry condition, the user is prompted for any temporary variable that he may have been using earlier that he no longer wishes to be included inside the block (Figure 2.b). When performing SurroundCritical refactoring, all the user interface does is warn the user that the refactoring to be performed may change the functionality of the program. The user can once again preview the refactoring and then make an informed decision.



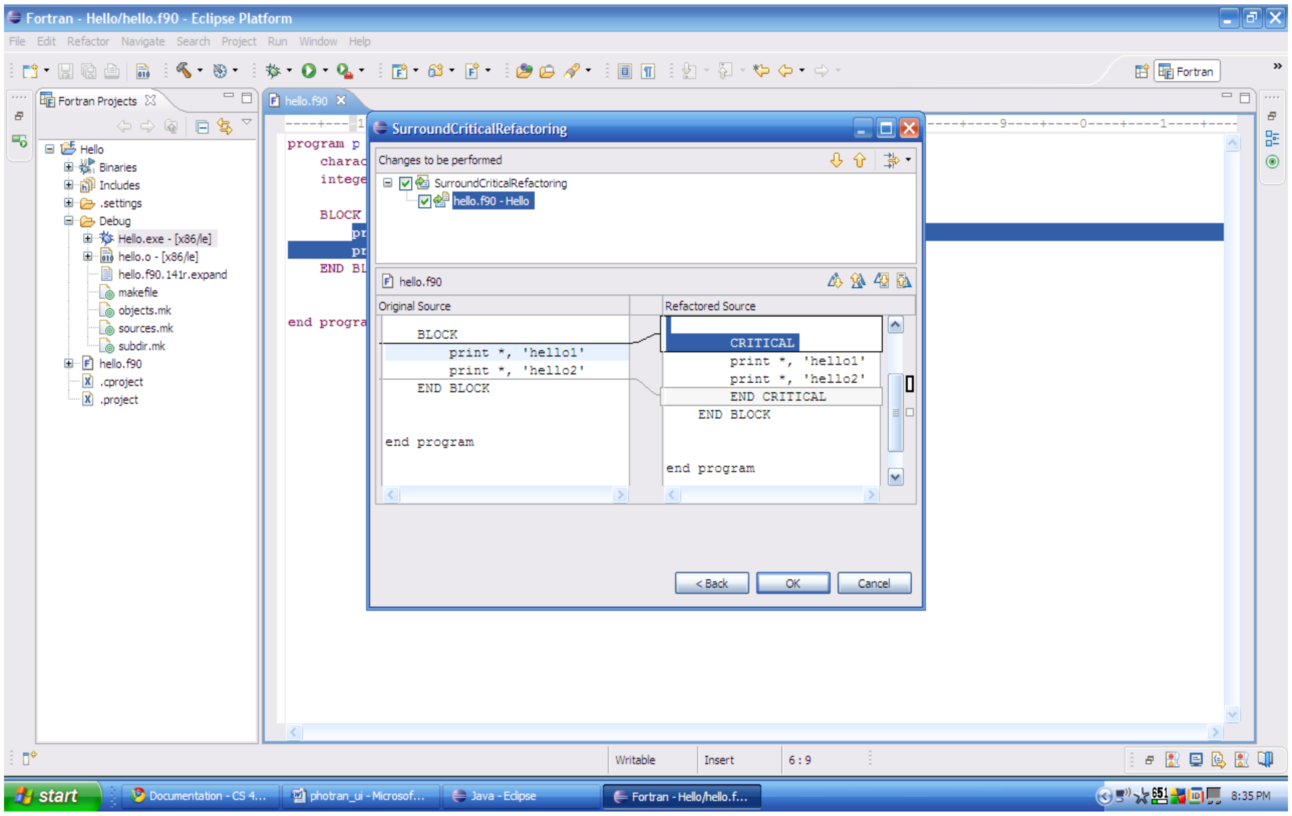
```
program p
    character(len=30) :: hello = 'hello'
    integer :: x = 1;

    BLOCK
        print *, 'hello1'
        print *, 'hello2'
    END BLOCK


end program
```

Provide condition for if statement

(Figure 2.b: Surround Block Prompt)

To actually perform the refactorings described above we needed to create three new files in the org.eclipse.photran.internal.ui.refactoring package. We named these classes *SurroundIfAction*,

`SurrounBlockAction` and `SurroundCriticalAction`. In these files we defined specific classes that exted the `AbstractFortranRefactoringWizard` class to perform our refactorings. We add a UserInputWizardPage since we require some form of input from the user in all three cases. Once the user input is retrieved, the corresponding refactoring class in the **org.eclipse.photran.internal.core.refactoring** package is called.
These classes are named `SurroundIfRefactoring`, `SurroundBlockrefactoring` and `SurroundCriticalRefactoring` respectively and are responsible for modifying the program abstract syntax trees in order to output the new, refactored piece of code. While performing the refactoring if we discover that any of the initial or final condition checks is not satisfied then once again we present the user with an error message (Figure 2.d).



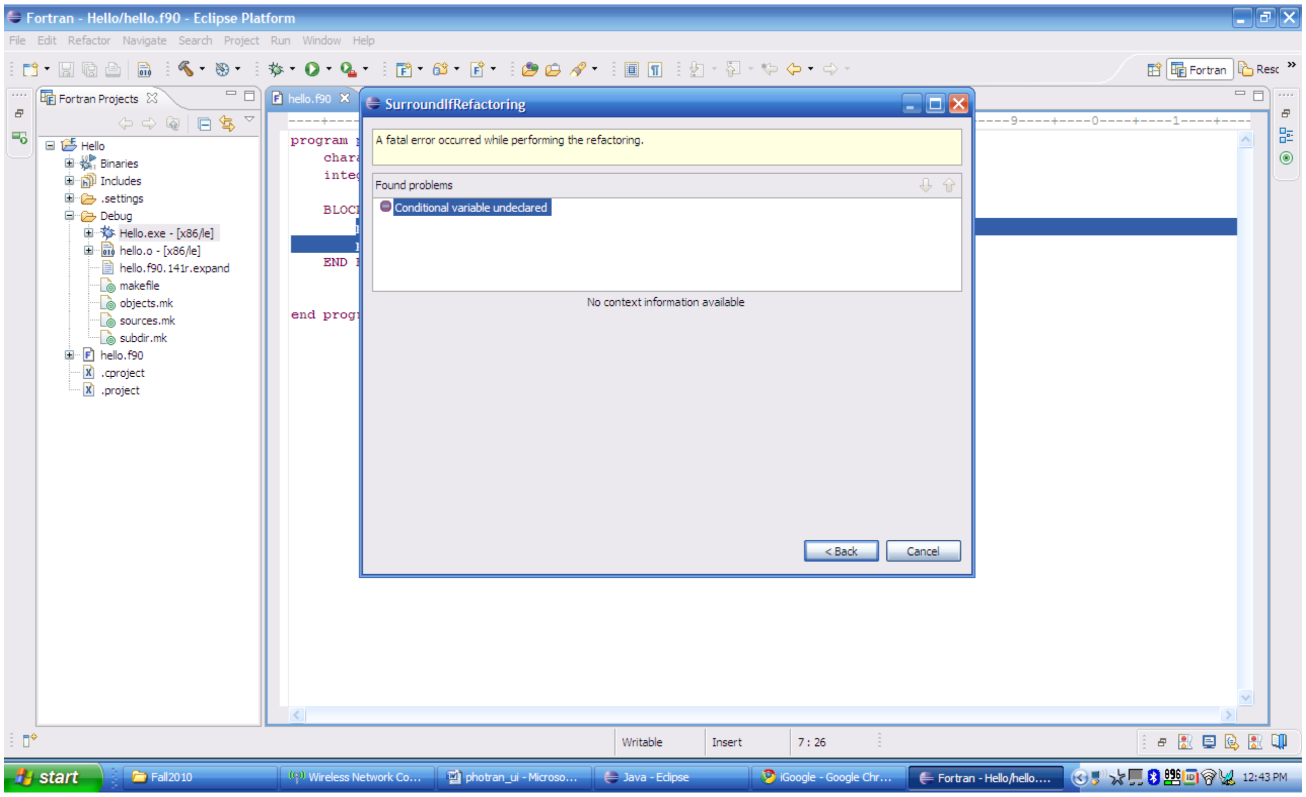
Original Source:

```
    BLOCK
        print *, 'hello1'
        print *, 'hello2'
    END BLOCK

end program
```

Refactored Source:

```
        CRITICAL
        print *, 'hello1'
        print *, 'hello2'
        END CRITICAL
    END BLOCK

end program
```

(Figure 2.c: Surround Critical Preview)



A fatal error occurred while performing the refactoring.

Found problems

Conditional variable undeclared

No context information available

(Figure 2.d: Surround If Error)

III. Future Plans

A. Future Project Outlook

Despite the project being an interesting experience, our group feels we will have no intention of continuing our work or visiting the code ever again. The code will be on the CS 427 subversion for student reference if desired, but there are no plans to submit to the source project or improve/continue the project.

B. Individual Reflections

henke2

This project awarded many opportunities as well as headaches. Though I appreciated the experience of learning an established system, the task proved daunting. It was difficult to start off and learn the system's implementation as well as how one might add to it. Once our group became more familiar with the Photran, the process still felt very much like "hacking" because we were never certain if our assertions about the system were correct. There not being a definitive expert on such a large system, and a lack of adequate documentation/commenting, getting help for our project was difficult as well. The course staff did a good job at *supplementing and assisting where possible. As stated, there were benefits to the project. Working in a large group offered many different insights on the same issue, and partner programming helped problem solving immensly. Overall, I hope the Photran project experience does not reflect the workplace experience in regards to an already implemented system.*

sjohns67

Working on the Photran refactoring Surround Block Statement gave me an opportunity to learn and develop on a system much larger and more complex than I ever have before. While I did not understand all of the system's design, I appreciated the ease with which we were able to extend the system. There was not much supporting documentation, but as much can be expected from such a large project that is not widely used. I'm glad that - while we paired programmed - we had people who were more knowledgeable about certain areas of the system. As a result, we were more effective in carrying out our tasks. One thing our group could have done better is goal setting. We accomplished our objectives, but at first we weren't sure what our objectives were. I believe future groups could benefit from clearer expectations at the onset of the Final Projects.

nramesh2

I must admit that when I first started working on this projetct, I was not a huge fan. This was mainly because just getting the system up and running on my computer was painful and annoying. Also, the fact that I had a Windows machine didn't help. I encountered quite a few problems while installing Cygwin, the Fortran compiler and the Photran system itself. I had to delete and re-install some of these programs a couple of times before I got everything to work correctly and the Photran system to compile and run without errors. Programming in Fortran was kind of a drag too mainly because of the tabs issue with the Photran compiler. The Photran system was not very easy to understand but I guess that can be said for any large system in general. Following the XP and reverse engineering processes that we learnt in class helped me get a grip on the system and once I began to understand its core components, trying to extend it to include the new refactorings became less frustrating.

rsulliv7

This project was an interesting experience. This is the biggest project that I have worked on, and it was quite intimidating at first. We began by doing our initial code walkthrough by reading through all the code in one hour and reading all function headers, but I felt lost and overwhelmed at first. It wasn't until we began writing our code that I began to understand what was going on. By the end of the project, I felt like I had a much better understanding of what was going on in the system. I also feel like I have learned Fortran even though I haven't written a single line of code using it. I'm not particularly fond of the project that we just completed, but I do feel like it was a valuable learning experience.

rudwick2

I thought this was a fairly silly project. The whole idea of it was to write refactorings for a language that is barely used for writing new code. Obviously, most sane people would find this useless. Refactoring code is only helpful if you are actually trying to fix your code as you write it. First of all, one of our sections, CRITICAL, can only be tested with the newest compiler. That means that a third of our work can only be used for brand new fortran code, which basically no one is writing. That being said, it could have been worse. I certainly improved my debugging skills with Eclipse, and we got some experience with dealing with huge coding projects, which is obviously something we will do more of in the coding world after school. All in all, it was not an especially helpful project as far as coding, but more experience with group work and dealing with customers were both helpful.

Appendix

A. Installation Instructions

To install the refactoring source code, four packages must be downloaded from the LemonMango subversion. First, be sure to download the latest version of Eclipse and Photran source code from the Photran CVS, as well as the Fortran compiler and linker. For detailed instructions, see the Photran developer documentation.

1. To get the surround statement refactoring, remove the following packages from your CVS download:

org.eclipse.photran.core.vpg

org.eclipse.photran.core.vpg.tests

org.eclipse.photran.core.vpg.tests.failing

org.eclipse.photran.ui.vpg

2. Now, the same 4 packages should now be checked out from https://subversion.ews.illinois.edu/svn/fa10-cs427/LemonMango

3. Select Project - > Clean... and clean up all associated project files.

4. Select Project - > Build All to rebuild the Photran project files.

5. To run the modified Photran IDE, select the package org.eclipse.photran.core.vpg, right click, Run As → Eclipse Application. Now another instance of eclipse should open. This is the modified copy and the surround statement refactorings should be enabled. For use, please see appendix section B.

6.

B. How to Use

This guide assumes you have the refactoring, Eclipse, and Photran source installed (see appendix A).

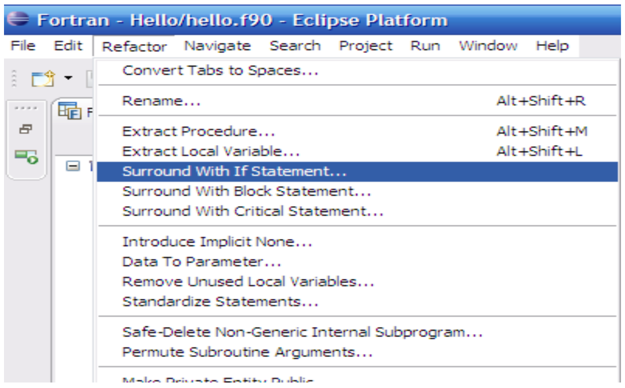
1. Select the **org.eclipse.photran.core.vpg** package and right click -> Run As... -> Eclipse Application.

2. Open or Create a new Forran project.

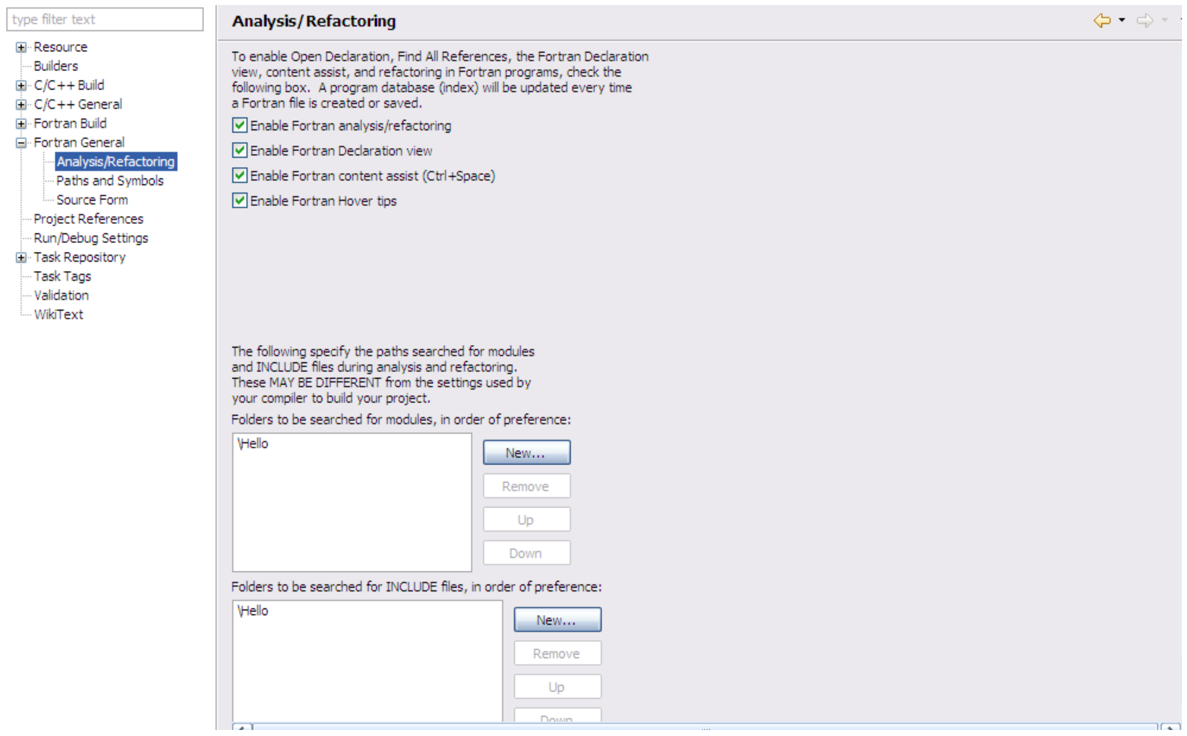
3. Make sure your Fortran project has refactoring enabled. To do this, select the apporpriate package, choose properties. In the properties menu, expand Fortran General, and select Analysis/Refactoring. Check to enable refactoring (Figure A.2).

4. Once refactoring is enabled, highlight the appropriate lines of code and either right click or use the refactor menu (Figure A.1). The options are Surround If, Surround Block, and Surround Critical.

5. Follow the on-screen instructions, warnings, and previews.


Fortran - Hello/hello.f90 - Eclipse Platform
File Edit Refactor Navigate Search Project Run Window Help
Convert Tabs to Spaces...
Rename... Alt+Shift+R
Extract Procedure... Alt+Shift+M
Extract Local Variable... Alt+Shift+L
Surround With If Statement...
Surround With Block Statement...
Surround With Critical Statement...
Introduce Implicit None...
Data To Parameter...
Remove Unused Local Variables...
Standardize Statements...
Safe-Delete Non-Generic Internal Subprogram...
Permute Subroutine Arguments...

(Figure A.1: Refactor Menu)



type filter text

- Resource
- Builders
- C/C++ Build
- C/C++ General
- Fortran Build
- Fortran General
  - Analysis/Refactoring
  - Paths and Symbols
  - Source Form
- Project References
- Run/Debug Settings
- Task Repository
- Task Tags
- Validation
- WikiText

Analysis/Refactoring

To enable Open Declaration, Find All References, the Fortran Declaration view, content assist, and refactoring in Fortran programs, check the following box. A program database (index) will be updated every time a Fortran file is created or saved.

[✓] Enable Fortran analysis/refactoring

[✓] Enable Fortran Declaration view

[✓] Enable Fortran content assist (Ctrl+Space)

[✓] Enable Fortran Hover tips

The following specify the paths searched for modules and INCLUDE files during analysis and refactoring. These MAY BE DIFFERENT from the settings used by your compiler to build your project.

Folders to be searched for modules, in order of preference:

\Hello

Folders to be searched for INCLUDE files, in order of preference:

\Hello

(Figure A.2: Enable Refactoring)

C. Testing

The `SurroundBlockRefactoring`, `SurroundIfRefactoring`, and `SurrounCriticalRefactoring` classes all have associated test suites for further test driven development.

Running tests:

1. Be sure you have the refactoring, Eclipse, and Photran source installed (see appendix A).
2. In the package **org.eclipse.photran.core.vpg.tests,** expand to the src folder, and then **org.eclipse.photran.tests.refactoring.**
3. The test suites are contained in the classes `SurroundBlockTestSuite`, `SurrounIfTestSuite`, and `SurroundCriticalTestSuite`. In order to run the tests, right click the test suite file that corresponds to the refactoring to be tested, select Run As...->JUnit Plugin Test. This will bring up a new Eclipse window and then immediately close it while running the tests. The test results will show up in the JUnit results pane.

Editing tests:

1. In order to edit/add tests, add your new tests into the **org.eclipse.photran.core.vpg.tests -> refactoring-test-code -> surround-X-test-code** folder, where X is which refactor you'd like to test (if,

block, critical).

2. Once your new test files are in subfolders in the proper place, running the tests should include your new test results.