CS 427 Fall 2008 : MP5 Summary

This page last changed on Oct 18, 2008 by nchen.

Some minor inconveniences that could be avoided:

- If would be nice if all of you explicitly state (or name your test inputs) which test cases are meant to pass and which are meant to fail. We had to scrutinize some tests to actually decide which ones are currently passing.
- It would also be nice if all of you stuck to the current convention on where to put your code and what to name your classes (the monkey see, monkey do principle).

Some interesting observations for Metrics:

- Number of Parameters > *Threshold*. This could be fixed by using the *Introduce Parameter Object* refactoring.
- The Metrics plug-in found some problems with the lexer but because it is auto-generated from JFlex there isn't much refactoring that you could do to fix it unless you change the way code generation works in JFlex.

Some interesting observations for Code Coverage:

- Some of you did not run the coverage tool on the org.eclipse.photran.core.vpg.tests package. That will explain why you got really low numbers for FortranRefactoring and its subclasses.
- Don't expect code coverage to be 100% accurate/similar on each run or on different machines. We took this into account when we graded.

Setting up the automated tests:

Inside MoveCommonToModuleTestCase.java (or whatever you called your class), the doRefactoring() method needs to do **three** things:

1. call doCheckInitialCondition
2. call setNewModuleName (either explicitly or implicitly via getSuggestedNewModuleName)
3. call doCheckFinalCondition

If you forget Step 2, your tests (both those that are supposed to pass and fail) will likely error out with a NullPointerException.

The failure cases that we were primarily looking for:

1) For common blocks (named/unnamed) that have variables of different names

For instance

```
subroutine s2
  implicit none
  integer :: one
  integer :: two
  common one, two

  one = 3
  two = 4
end subroutine s2

subroutine s3
  implicit none
  integer :: whateverIWantToCallThis
  integer :: iDontCare
  common  iDontCare, whateverIWantToCallThis

  iDontCare = 5
  whateverIWantToCallThis = 6
```

```
end subroutine s3
```

It creates a module for those common block but doesn't **standardize** the name.

So it becomes

```
module common
        implicit none
        integer :: one
  integer :: two
end module common

subroutine s2
   use common
   implicit none

   one = 3
   two = 4
end subroutine s2

subroutine s3
   use common
   implicit none

   iDontCare = 5
   whateverIWantToCallThis = 6

end subroutine s3
```

where the variables iDontCare and whateverIWantToCallThis cannot be resolved.

2) It clobbers any module with the same name that your specify as the new module name for the extracted common block (the proper behavior might be to warn the user or to merge the modules).

3) It cannot handle the case where the variables in the block are declared on a single line.

For instance

```
subroutine s1
   implicit none
   integer     :: a_2, b_2 ! variables declared on a single line.
   common /block1/ a_2, b_2

   a_2 = 1
   b_2 = 2

end subroutine s1
```

It will error out with a "Child node not found" exception because it tries to remove the line "integer :: a_2, b_2" twice. Once for each value in the common block.

4) It can change the semantics of programs when multiple variables are declared on one line:

For instance

```
subroutine s1
   implicit none
   integer     :: a_2, b_2
   common /block1/ a_2

   a_2 = 1
   b_2 = 2

end subroutine s1
```

becomes

```
module block1
```

```
        implicit none
        integer     :: a_2, b_2
end module block1

subroutine s1
         use block1
   implicit none

   a_2 = 1
   b_2 = 2

end subroutine s1
```

b_2 should be local variable in this case but it has been moved to the module after the refactoring.

Cases 3 and 4 actually belong to the same class of problems. But the only way to figure this out was by reading the source code. Since we decided that this MP was about writing black-box tests, we don't think we should require you to read the source code for MoveCommonToModuleRefactoring. So we accepted both cases in the end.

Some *bugs* that that students found that are being reported to Jeff

1) The current parser does not support this declaration of variables

```
integer * 8 var
```

2) There is a bug with the testing setup/ VPG where if multiple test cases have a common block with the same name (e.g. block1) the VPG might erroneously hold a reference to the old block on the next test. This will cause a NullPointerException since even though the VPG thinks there is references, all test are supposed to be run in isolation. So the reference to the old common block is stale.

3) The issue with the inconsistent \r\n and \n on Windows.

4) Comments that are "attached" to the common block are not moved with the refactoring (this is actually a tricky issue that even the Java tools in Eclipse get wrong sometimes because there isn't a proper semantics for it: are the comments still *valid* after the refactoring? After all, the comment could be for the comment block and now that it has been moved to a module, that comment could be obsolete.)

```
INTEGER :: I, J, ISTP
!Some Comment that should not be deleted
COMMON /TST/ PI
```

Currently, performing the refactoring doesn't move the comments to the module.