Typesafe Calls

Chase Geigle (geigle1),
Sean Hurley (hurley5),
Smit Shah (shah73),
Dennis Lin (lin49),
Surgey Lupersolsky (luperso1),
Victor Zhagui (zhagui1)

December 20, 2012

Contents

1 Description

The **Typesafe Calls** project attempts to remedy a common issue in Fortran programs—lack of INTERFACE blocks for external subprograms (subroutines and functions).

1.1 The Problem

Why is this an issue? In Fortran, calls to external subprograms are not checked as part of the compiler's type system. The unfortunate side-effect is that calls to external subprograms could have incorrect argument types and the program will compile, but produce an unpredictable (and potentially difficult to debug) error at runtime.

Introducing an INTERFACE block remedies this issue. The INTERFACE blocks in Fortran 90 specify the number and types for all arguments (and return values for functions), allowing the compiler to check the calls using the types given in the INTERFACE block for the external function or subroutine.

1.2 The Solution

Our project extends the Photran architecture to provide warnings to users who make this mistake in their programs via IMarkers—this means that they will get an experience similar to that of using JDT. All calls to external subprograms that are not hidden behind INTERFACE blocks are highlighted with a marker in the editor (as well as by an orange line underneath them), alerting the user to their potential mistake *as they are writing it*. All of our functionality for detecting these mistakes happens in real-time.

We also provide a method for automatically generating correct INTERFACE blocks for the potentially unsafe calls—this fix can be invoked either through the standard Photran refactoring interface, or via a "Quick Fix" in Eclipse's "Problems View".

2 Architecture and Design

The **Typesafe Calls** project adds to main functionalities to Photran: *a*) a real-time "linting" functionality which is used to detect problems with source files currently being edited in Photran in a similar fashion as is done in Eclipse's JDT, and *b*) a refactoring that introduces explicit interfaces for external subroutines and functions to allow the unsafe calls to them to be checked by the type system of the compiler at compile time.

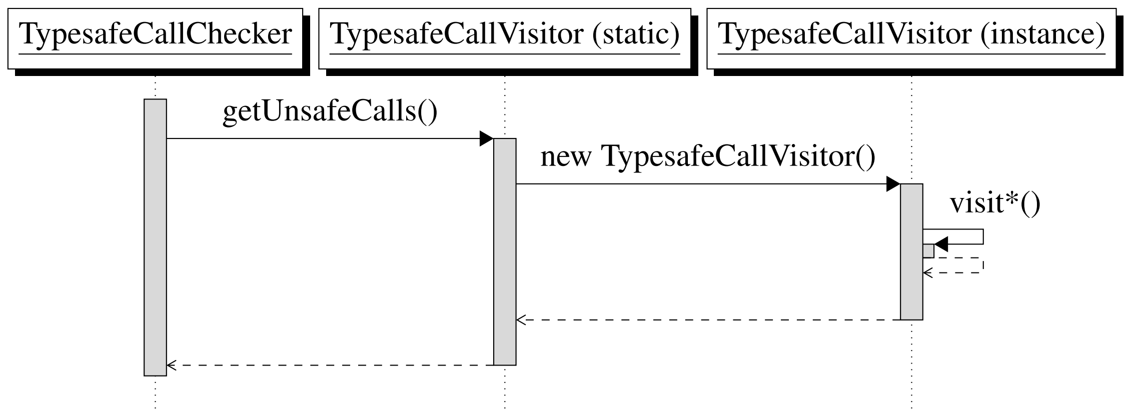

TypesafeCallChecker
TypesafeCallVisitor (static)
TypesafeCallVisitor (instance)
getUnsafeCalls()
new TypesafeCallVisitor()
visit*()

Figure 1: Interaction diagram for the linting capabilities.

2.1 Linting

The real-time linting capabilities of our project are added in the `org.eclipse.photran.internal.ui.editor_vpg` package. Within this is a `lint` package which contains our contributions.

Our linting process is implemented by using a `IFortranASTTask` called the `TypesafeCallChecker`. This task will, when both the AST and `DefinitionMap` are available to the task, traverse the AST given looking for calls to external subroutines and functions. In particular, it looks for `ASTCallStmtNodes`, `ASTVarOrFnRefNodes`, `ASTInterfaceStmtNodes`, `ASTInterfaceBodyNodes`, and `ASTUseStmtNodes` by using the `TypesafeCallVisitor`.

The visitor is used to determine *a*) where calls to external functions and/or subroutines are, *b*) whether or not there exist interfaces for them already (whether they be explicit or generic), and *c*) whether modules in use by the program contain interfaces for the called subroutines and/or functions already.

Once the above is determined, the `TypesafeCallVisitor` will determine which of the found calls to external subprograms are not hidden behind interfaces—the `TypesafeCallChecker` will then place `IMarkers` that highlight which calls are potentially unsafe. The severity of the `IMarkers` is “Warning”, since the calls are only *potentially* unsafe (we do not actually check whether the call is truly unsafe).

Additional real-time linting capabilities could be added using the same framework outlined above: create an implementation of the `IFortranASTTask` and use it to traverse the AST of the currently open file, looking for things that satisfy your particular linter’s criteria.

2.2 Quick Fix

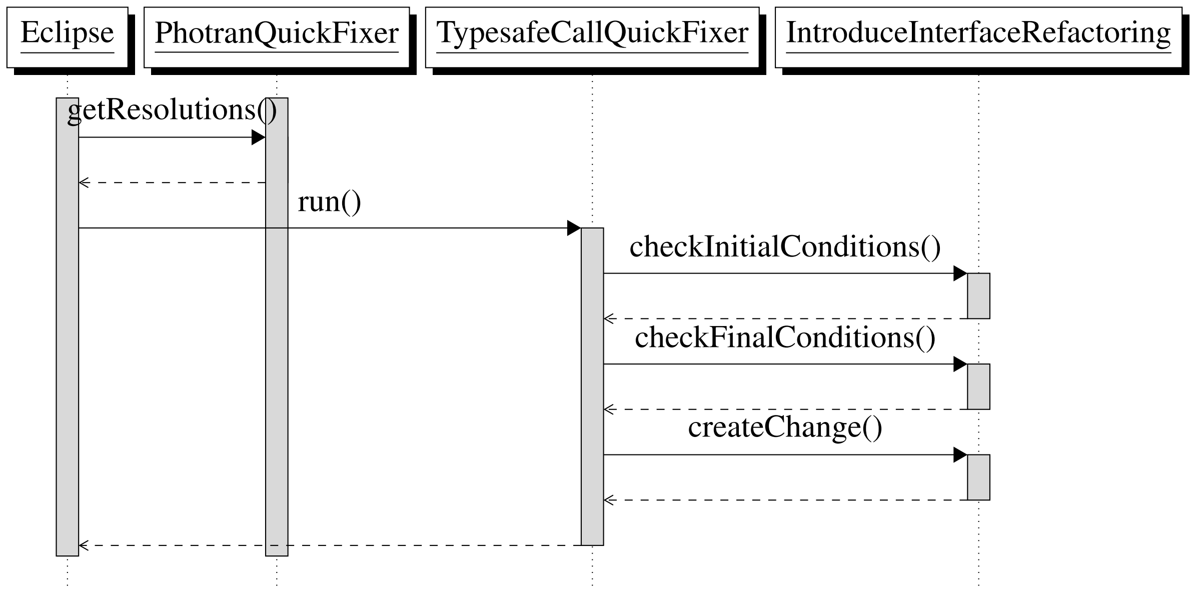


Figure 2: Interaction diagram for the quick fix capabilities.

Because of the decision to add `IMarkers` to the editor view via our linting framework, it makes sense to also add a "Quick Fix" for the problems we discover.[1] Our "Quick Fix" is added through the `TypesafeCallQuickFixer`, which is invoked by the `PhotranQuickFixer` when a quick fix operation is invoked on a marker in the problems view that *a*) is of the correct type (a lint marker) and *b*) was created by our `TypesafeCallChecker`. The former is done by a modification to the `plugin.xml`, and the latter is done at runtime by examining an attribute set on the marker itself.

To support future "Quick Fixes", we are using this attribute in the `PhotranQuickFixer` to invoke the correct quick fix action—future quick fixes could be added by creating markers with a different type and modifying the `PhotranQuickFixer` to invoke the new, correct quick fix action. This choice was made to avoid having to modify `plugin.xml` with many redundant lines whenever a new quick fix is added—it is easier to just modify the `PhotranQuickFixer` to call the new quick fix action than it would be to duplicate the `plugin.xml` modifications again and again for each new quick fix.

[1] It is worth distinguishing what a "Quick Fix" is compared to a "Quick Assist". We have implemented the former, but *not* the latter—more details on this are mentioned in Section 3.

Our `TypesafeCallQuickFixer` invokes a refactoring we have added to Photran to introduce an interface for a call to an external function or subroutine. This refactoring is discussed in Section 2.4.

2.3 Marker Annotation

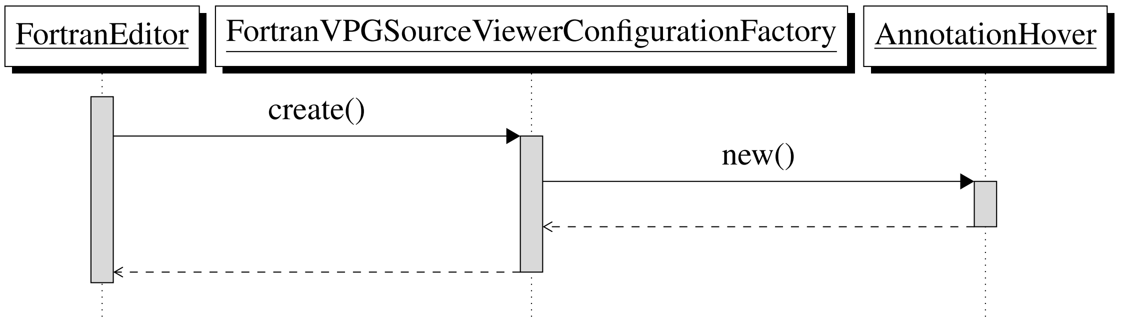


Figure 3: Interaction diagram for creation of the AnnotationHover.

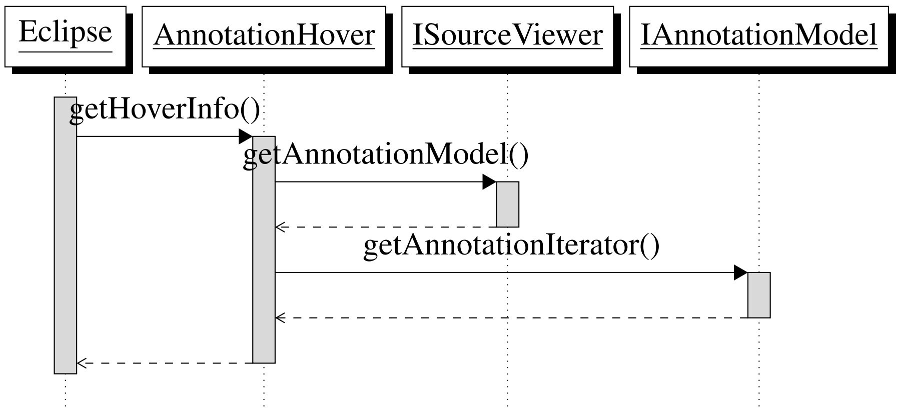


Figure 4: Interaction diagram for generation of hover messages.

In order to have a hover effect for our `IMarkers` that indicate what the required action is of our user (see Figure 11 for an example), we needed to add an implementation of `IAnnotationHover` to our tool.

The entry point for our annotation hover is in the `FortranVPGSourceViewerConfigurationFactory`—in this class there is a `getAnnotationHover()`, where our new annotation hover is instantiated.

The hover itself has a `getHoverInfo()`, which has been implemented to return the messages of all `IMarkers` on that line, allowing the user to see what action must be taken to rememdy the warning presented by our `IMarkers`.

2.4 Refactoring

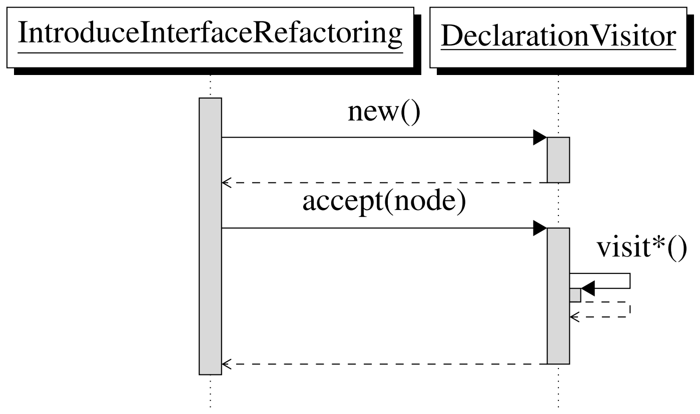


Figure 5: Interaction diagram for the refactoring.

Our refactoring, the `IntroduceInterfaceRefactoring`, attempts to introduce an interface to a program for the selected call to an external function or subroutine. It does this by *a*) examining the VPG of the project to find where the called external function or subroutine lives, *b*) traversing the body of the subprogram to determine what parameters it has and what their type declarations are, and finally *c*) introduces an interface body within the scope of the selected call that adheres to the parameters found and their types.

There is a limitation in our current implementation: we require the source code for the external function or subroutine be in the current project (though it may be in a separate file, this is fine) in order to determine the types of the parameters required for the subroutine or function being called. If the source is not available (for instance, if you're linking an external, pre-compiled library), introducing an interface for calls to that library may not work with our current implementation.

3 Future Plans

3.1 Submission

We plan on submitting a patch to the Photran developers list to get our refactoring and lint capabilities included into the main branch of Photran. Our real-time linting capability is a powerful one that could be incredibly useful to Fortran developers in many contexts beyond just checking for potentially type-unsafe calls. We hope that, in the future, more linting style capabilities will be added to Photran using the basic architecture we have created through the implementation of this project.

3.2 Quick Assist

Currently this refactoring supports the Quick Fix functionality of Eclipse. This means that by navigating to the Problems view, you can Right-click -> Quick Fix on our marker, and then the proper interface will be inserted into the program's body.

In the JDT (which we used as the standard for how lint-like functionality should work within Eclipse) a user can also right click on the marker that shows up just to the left of the source code and then select Quick Fix. This would then have to have what is called a Quick Assist menu pop up within the actual editor allowing the user to see possible fixes. The Quick Assist menu is generated in a completely different way than the problems view Quick Fix. Getting the Quick Assist functionality was out of scope to our project, but here is a general outline of how it could be added in the future. By setting up a clean architecture, future groups could add more lint-like capabilities easily in the future.

1. Create an `IQuickAssistProcessor`. The most important method here would be the `computeQuickAssistProposals()` method. This is where we would need to actually decide which possible solutions there could be for this particular problem. Note that there may be more than one possible solution, so the implementation would require an easy way to add different sources that could provide their own solutions. This way for one particular problem, the Photran architecture could easily provide many different solutions, and it would also be easy to add more possible fixes in the future.

2. Find a way to make it possible/easy to have the `computeQuickAssistProposals()` method of the `IQuickAssistProcessor()` be able to tie into the rest of the Photran architecture. By default the method only provides an `IQuickAssistInvocationContext` as a parameter, which doesn't give access to the rest of the nice functionality within Photran. Most of the Quick Assists

that would be added in the future would want to easily access the AST, VPG, as well as other points of interest.

3. Tie the `QuickAssistProcessor` created in step 1 into the actual editor. This is straightforward. Within the `FortranVPGSourceViewerConfigurationFactory`

   ```
   @Override
   public IQuickAssistAssistant getQuickAssistAssistant(
       ISourceViewer sourceViewer)
   {
     IQuickAssistAssistant quickAssist = new QuickAssistAssistant();

     quickAssist.setQuickAssistProcessor(
         new MyQuickAssistProcessor()
     );
     quickAssist.setInformationControlCreator(
         getInformationControlCreator(sourceViewer)
     );

     return quickAssist;
   }
   ```

   The above code snippet will setup the `QuickAssistProcessor` to actually be called when the user selects Quick Fix.

   The main difficulty in adding the Quick Assist menu is getting the `QuickAssistProcessor` to tie in nicely to all of the various aspects of Photran. Especially since in the future any other lint features will all use this same `QuickAssistProcessor` as their entry point, it will be important to have this in a nice, clean architecture.

3.3 Personal Reflections

3.3.1 Chase Geigle (geigle1)

Photran was by far the largest codebase I have had to work on, and there are a few things I've learned while doing so about making large codebases approachable.

First, good documentation is paramount—how this is done can take many forms serving many different purposes (Photran has two notable examples: its JavaDoc and its developers guides), but ensuring that a large project has a good source of documentation is a must.

Second, IDEs for large projects are incredibly useful. Prior to working intensively with Eclipse/Photran, I was skeptical of the benefits of having a full IDE for doing code editing, but now that I have worked with a larger codebase I see their value. I still have high hopes for the eclim project as it grows (so I can keep doing my editing in the almighty VIM), but Eclipse definitely brings unquestionable value when working on a large Java project.

Finally, a good use of Design Patterns in a large project can go a long way. Doing things with the ASTs in Photran could have been a huge pain in the neck had there not been a nice implementation of the visitor pattern in their implementation.

3.3.2 Sean Hurley (hurley5)

Working on the beast that was the Photran codebase was a bit intimidating. The combination of the Photran architecture tying into the CDT architecture with all of that sitting on top of the eclipse architecture made it difficult to comprehend exactly how everything tied together. After the initial scanning of the code it became apparent that this wasn't going to be quite as bad of a nightmare as we first imagined. It still took quite a bit of work to get everything to work together, but the entry points available in Photran made it a lot easier.

Overall, the project does a good job of teaching people how to learn and understand large codebases. However, I do wish that it could be taught using a more interesting tool. I can think of a ton of large open source codebases available online that would teach this just as well. If we would have been able to contribute to those instead, it would have been more interesting, and the class could have been helping a number of open source communities.

3.3.3 Dennis Lin (lin41)

The hardest part of this MP was developing a holistic understanding of the code. Once we figured out which parts fit together to get what we want, the task became much easier. However, figuring out the parts was quite difficult. This definitely helped improve my ability to find specific things in Eclipse and reverse engineering them.

At first, facing the enormous bulk of Photran seemed a very daunting task. However, once we focused on only the relevant packages, everything became a bit more simplified. This, along with learning and getting used to the various tools of Eclipse made this project change from its initial terror to merely an irritation.

3.3.4 Sergey Lupersolsky (luperso1)

This project was a good learning experience overall. It provided a useful experience of starting out with a massive, unknown codebase and then trying to make improvements to it. I feel like this could be very useful in the real world because you might have to take over a big project from a coworker at some point.

There are some things I don't like about the project though. I wish we worked with a more interesting open source project. Fortran and Photran seem very irrelevant to me in the future. Thus, I would have enjoyed and learned more if I was given the opportunity to pick a project that I found interesting and add something to it.

3.3.5 Smit Shah (shah73)

Overall, the specifications for this project were straightforward and working on this in a larger group was generally not difficult. Initially, most of the time would contribute to figuring out what package does what; due to large number of packages. Understanding what the ASTNodes are, what TypeSafe Calls are, and how they function was a bit challenging. Although, once the proper resources were found the project did not seem as frustrating.

Working in pairs and splitting the task into smaller groups was enjoyable and much more efficient. Tasks within our level of ability were progressed with proper management and with proper team coordination.

About the project itself, learning to implement various refactoring turned out to be a useful skill however working with Fortran was difficult many times and not as interesting at all.

3.3.6 Victor Zhagui (zhagui1)

Working with this codebase was very daunting in the beginning stages but understanding the context of the refactoring was relatively easy. It would have been more enjoyable if the project dealt with another more interesting codebase. I didn't like that the language was Fortran but there wasn't much of required to know other than semantics.

I gained a lot of experience with working with a real life application that uses a ton of modules and packages. Not only that but you gain a lot of knowledge on others coding styles. We also applied the visitor design pattern and practiced XP, a development pedagogy. It was interesting to see these methodologies being implemented to extend Photran. They saved us some time on figuring out how to structure the problem in an efficient manner, so that we could tackle the refactoring.

A Installation

A.1 Get the Code

Our modified copy of Photran can be obtained by checking the code out via github: `https://github.com/SeanHurley/photran`. You will, of course, need to have a working installation of `git`.

A.2 Fortran Compiler

You will need to install the correct Fortran compiler for your system.

For Linux based systems, this is typically `gfortran`, included with the `GCC` package (though you may need to install something extra from within your distro's package manager).

For OSX, it is `g95`, which may be installed using either MacPorts or Homebrew.

For Windows, you will need to download and install `gfortran`. Make sure you download the correct `gfortran` based on your system (if your system is 32-bit, make sure to download the 32-bit version. If you have a 64-bit system, make sure you download the 64-bit version).

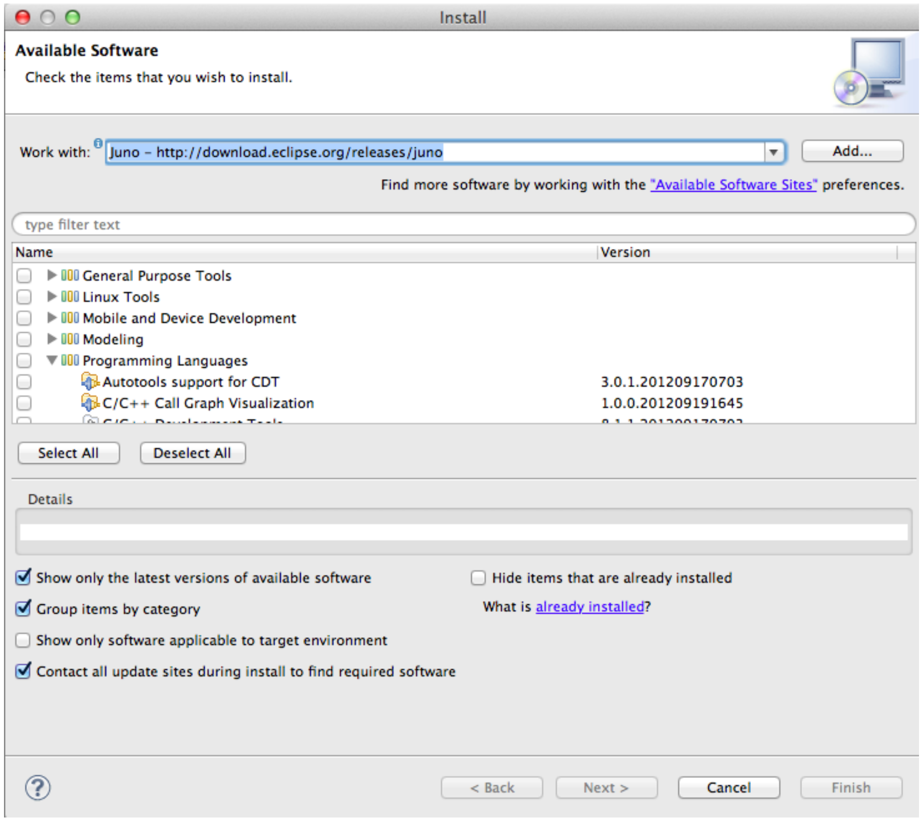
A.3 Eclipse Requirements

Photran requires that you have Eclipse Juno RCP (Rich Cliet Platform) installed. Ensure that you have this version, or things might not work exactly as intended. You will also need the Eclipse CDT (C/C++ Development Tools) SDK (Software Development Kit), as Photran is built on top of Eclipse's CDT.

You can do this by going to Help -> Install New Software. Select "Juno" from the "Work with" menu, then navigating to Programming Languages -> C/C++ Development Tools SDK. See Figure 6 for guidance.

Additionally, you will need to establish an API baseline. To do so, go to Preferences -> Plug-in Development -> API Baselines -> Add Baseline. Give it a name (the name does not matter), click "Reset", and then "Finish".

Once this is done, you will need to import all of the projects you checked out from our git repository. Go to File -> Import -> Existing Projects into Workspace. Select the root directory where you have checked out Photran, and ensure you have all of the projects checkmarked that existed in that directory (there should be lots). Then click "Finish".



Available Software

Check the items that you wish to install.

Work with: Juno - http://download.eclipse.org/releases/juno

Find more software by working with the "Available Software Sites" preferences.

type filter text

Name	Version
☐ ▶ General Purpose Tools	
☐ ▶ Linux Tools	
☐ ▶ Mobile and Device Development	
☐ ▶ Modeling	
☐ ▼ Programming Languages	
☐ Autotools support for CDT	3.0.1.201209170703
☐ C/C++ Call Graph Visualization	1.0.0.201209191645

Details

☑ Show only the latest versions of available software

☑ Group items by category

☐ Show only software applicable to target environment

☑ Contact all update sites during install to find required software

☐ Hide items that are already installed

What is already installed?

Figure 6: Finding and installing Eclipse CDT SDK.

B Running Photran

To run Photran, you can navigate to the `org.eclipse.photran.core` project, then right click -> Run As -> Eclipse Application. See Figure 7 for guidance.

B.1 Creating a Fortran Project

To create a Fortran project in Photran, right click on the Project Explorer -> New -> Fortran Project. See Figure 8 for guidance.

B.1.1 Enable Refactoring and Analysis Support

Without Refactoring and Analysis support, our plugin will not work. Ensure that these are enabled for your project by right clicking on the project itself -> Properties. Then,

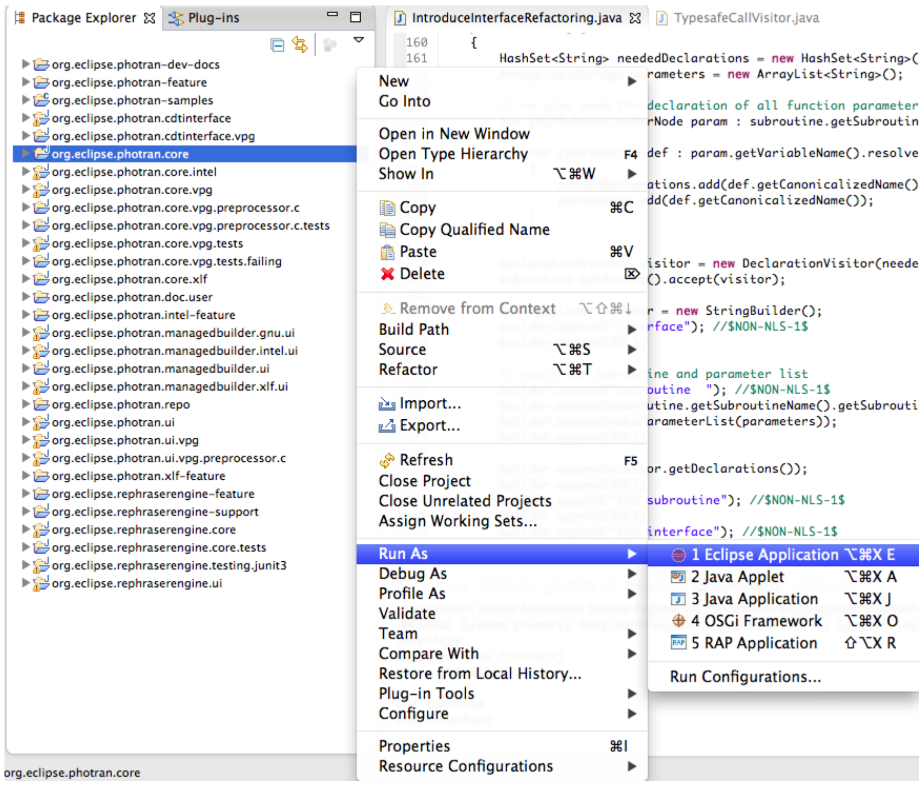

Package Explorer
Plug-ins
IntroduceInterfaceRefactoring.java
TypesafeCallVisitor.java
org.eclipse.photran-dev-docs
org.eclipse.photran-feature
org.eclipse.photran-samples
org.eclipse.photran.cdtinterface
org.eclipse.photran.cdtinterface.vpg
org.eclipse.photran.core
org.eclipse.photran.core.intel
org.eclipse.photran.core.vpg
org.eclipse.photran.core.vpg.preprocessor.c
org.eclipse.photran.core.vpg.preprocessor.c.tests
org.eclipse.photran.core.vpg.tests
org.eclipse.photran.core.vpg.tests.failing
org.eclipse.photran.core.xlf
org.eclipse.photran.doc.user
org.eclipse.photran.intel-feature
org.eclipse.photran.managedbuilder.gnu.ui
org.eclipse.photran.managedbuilder.intel.ui
org.eclipse.photran.managedbuilder.ui
org.eclipse.photran.managedbuilder.xlf.ui
org.eclipse.photran.repo
org.eclipse.photran.ui
org.eclipse.photran.ui.vpg
org.eclipse.photran.ui.vpg.preprocessor.c
org.eclipse.photran.xlf-feature
org.eclipse.rephraserengine-feature
org.eclipse.rephraserengine-support
org.eclipse.rephraserengine.core
org.eclipse.rephraserengine.core.tests
org.eclipse.rephraserengine.testing.junit3
org.eclipse.rephraserengine.ui
org.eclipse.photran.core
New
Go Into
Open in New Window
Open Type Hierarchy F4
Show In ⌥⌘W
Copy ⌘C
Copy Qualified Name
Paste ⌘V
Delete
Remove from Context ⌥⇧⌘↓
Build Path
Source ⌥⌘S
Refactor ⌥⌘T
Import...
Export...
Refresh F5
Close Project
Close Unrelated Projects
Assign Working Sets...
Run As
Debug As
Profile As
Validate
Team
Compare With
Restore from Local History...
Plug-in Tools
Configure
Properties ⌘I
Resource Configurations
1 Eclipse Application ⌥⌘X E
2 Java Applet ⌥⌘X A
3 Java Application ⌥⌘X J
4 OSGi Framework ⌥⌘X O
5 RAP Application ⇧⌥X R
Run Configurations...

Figure 7: Running Photran from inside Eclipse

search for "Fortran General" -> Analysis/Refactoring, and tick the "Enable Fortran analysis/refactoring" checkbox. See Figure 9 for guidance.

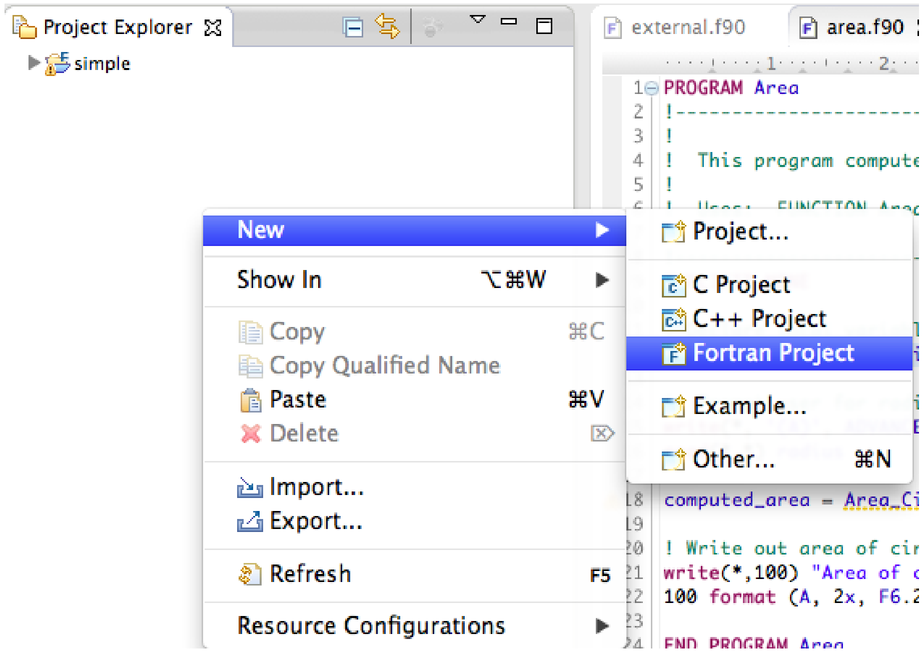
B.2 Using our Tool

Our `TypesafeCallChecker` should be running by default when Photran is being run through Eclipse. You should see error markers placed on potentially unsafe calls to external functions and/or subroutines automatically (see Figure 10 for an example).

If you hover over an existing `IMarker`, you should see the annotation as shown in Figure 11. Our Quick Fix can be triggered by right clicking on the warning created in the Problems View (see Figure 12 for an example).

To run the refactoring itself, outside of the Quick Fix, go to Refactor -> Coding Style -> Create Interface For External Subprogram (see Figure 13). Your cursor needs to be on the call to the external function or subroutine that is highlighted by one of our `IMarkers`.

An example of the output of the refactoring is given in Figures 14 and 15.



```
PROGRAM Area
!---------------------
!
!   This program compute
!
computed_area = Area_Ci

! Write out area of cir
write(*,100) "Area of c
100 format (A, 2x, F6.2

END PROGRAM Area
```

Figure 8: Creating a new Fortran Project in Photran

C Testing

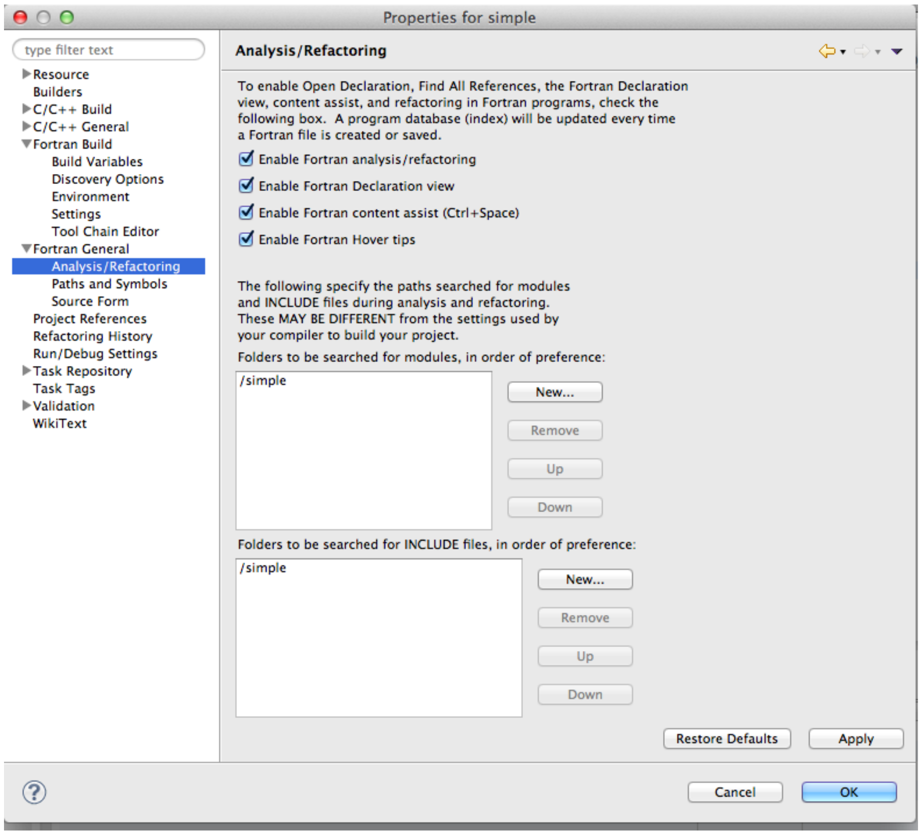
To run all of the tests for Photran, you can right click on the `org.eclipse.photran.vpg.tests` project -> Run As -> Run Configurations (see Figure 16). Then, right click on JUnit Plug-in Test -> New (see Figure 17). Give it a name, and ensure that the VM Arguments are set to those depicted in Figure 18. Also, ensure that the tests are being run as JUnit 3 tests, or you may have difficulty running the tests (see Figure 19).

C.1 Running Typesafe Call Tests

If you want to run *our* tests specifically, you can run the tests for the detector and the tests for the refactoring separately.

To run the tests for the detector, navigate to the `org.eclipse.photran.vpg.tests` package and find the `refactoring.detector` package. The file `DetectionTest.java` contains tests for the lint functionality (see Figure 20).

To run the tests for the refactoring, run `IntroduceInterfaceRefactoringTestSuite.java`, located in the `refactoring` package in the aforementioned project (see Figure 21).



Properties for simple

type filter text

- Resource
- Builders
- C/C++ Build
- C/C++ General
- Fortran Build
  - Build Variables
  - Discovery Options
  - Environment
  - Settings
  - Tool Chain Editor
- Fortran General
  - Analysis/Refactoring
  - Paths and Symbols
  - Source Form
- Project References
- Refactoring History
- 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.

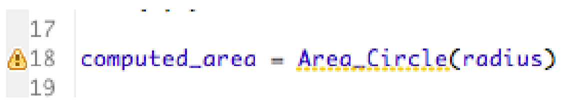
Folders to be searched for modules, in order of preference:

/simple

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

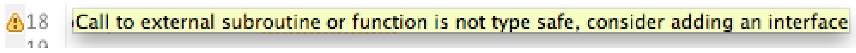
/simple

Figure 9: Enabling Fortran Analysis/Refactoring Support



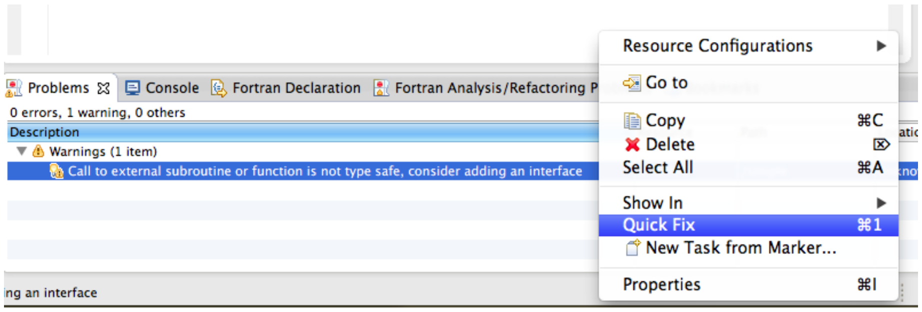
```
computed_area = Area_Circle(radius)
```

Figure 10: An example of an `IMarker` in Photran



Call to external subroutine or function is not type safe, consider adding an interface

Figure 11: An example of an `IMarker`'s Annotation in Photran



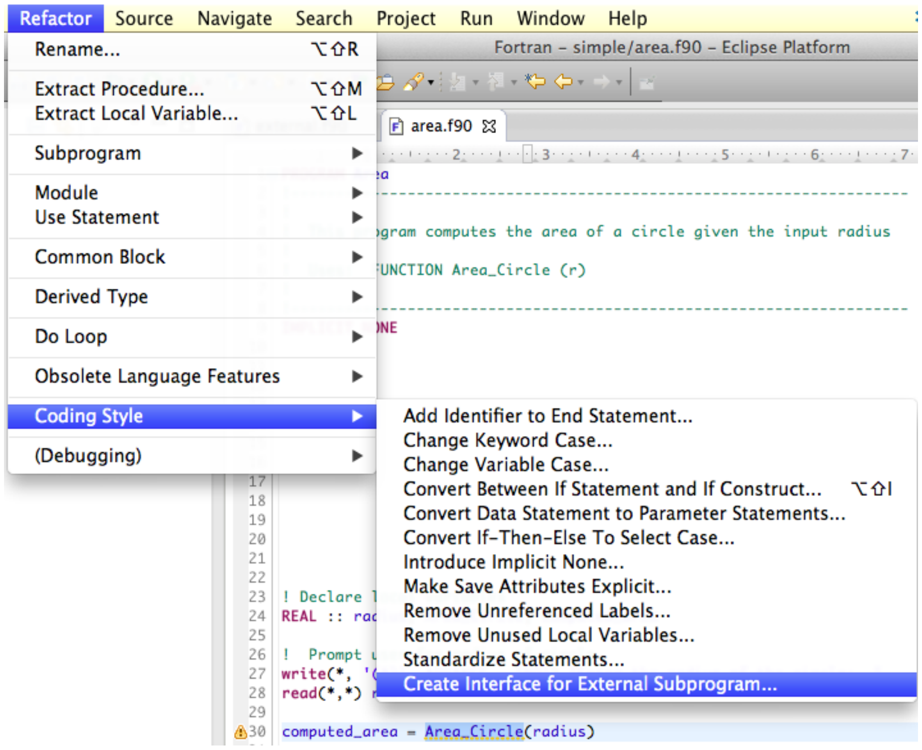
0 errors, 1 warning, 0 others

Description

Warnings (1 item)

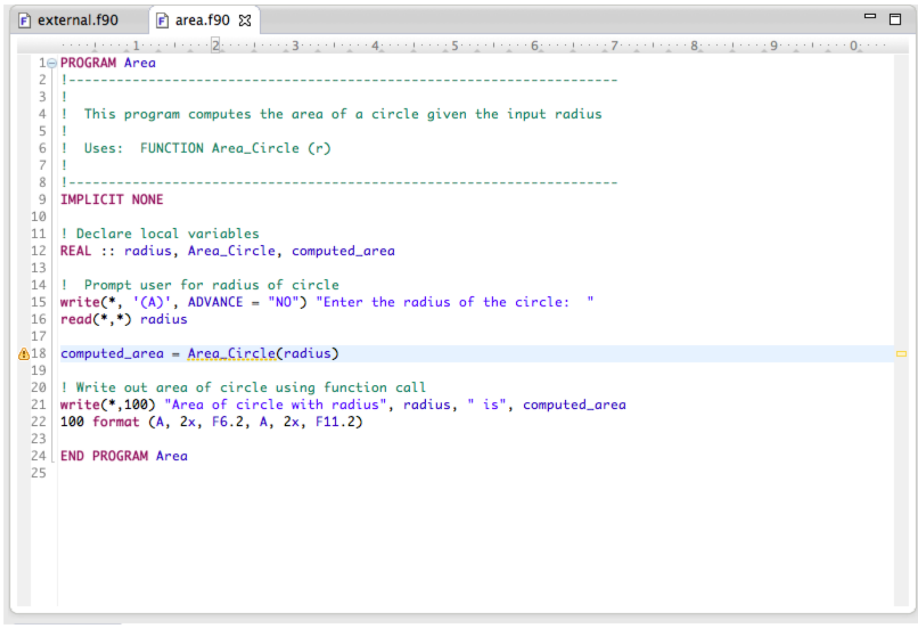
Call to external subroutine or function is not type safe, consider adding an interface

Figure 12: A warning in the problems view in Photran


Refactor
Source
Navigate
Search
Project
Run
Window
Help
Fortran – simple/area.f90 – Eclipse Platform
area.f90
Rename... ⌥⇧R
Extract Procedure... ⌥⇧M
Extract Local Variable... ⌥⇧L
Subprogram
Module
Use Statement
Common Block
Derived Type
Do Loop
Obsolete Language Features
Coding Style
(Debugging)
Add Identifier to End Statement...
Change Keyword Case...
Change Variable Case...
Convert Between If Statement and If Construct... ⌥⇧I
Convert Data Statement to Parameter Statements...
Convert If-Then-Else To Select Case...
Introduce Implicit None...
Make Save Attributes Explicit...
Remove Unreferenced Labels...
Remove Unused Local Variables...
Standardize Statements...
Create Interface for External Subprogram...

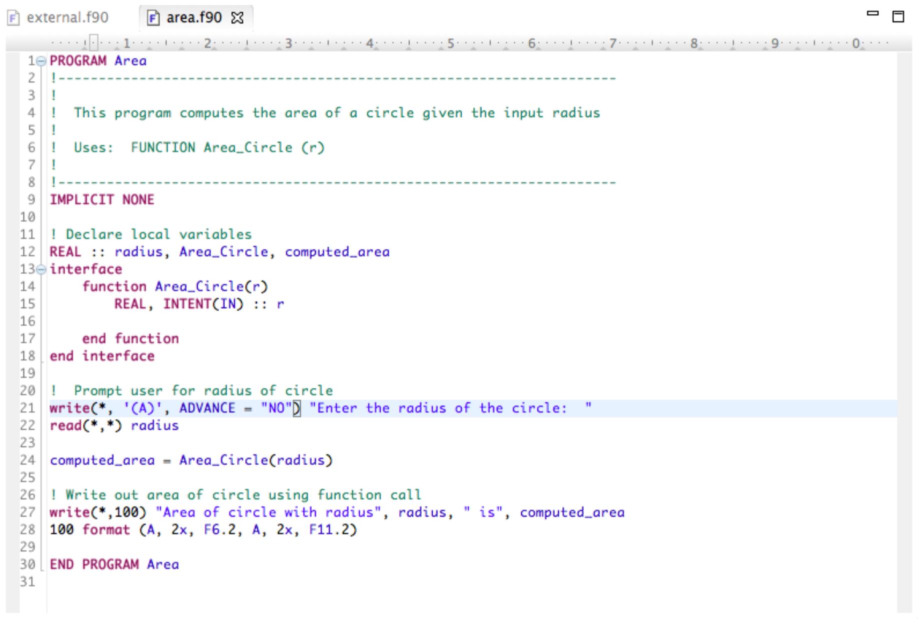
```
ogram computes the area of a circle given the input radius
UNCTION Area_Circle (r)
NE
! Declare 
REAL :: rad
!  Prompt 
write(*, '(
read(*,*) r
computed_area = Area_Circle(radius)
```

Figure 13: Running the refactoring via the refactoring menu


external.f90
area.f90

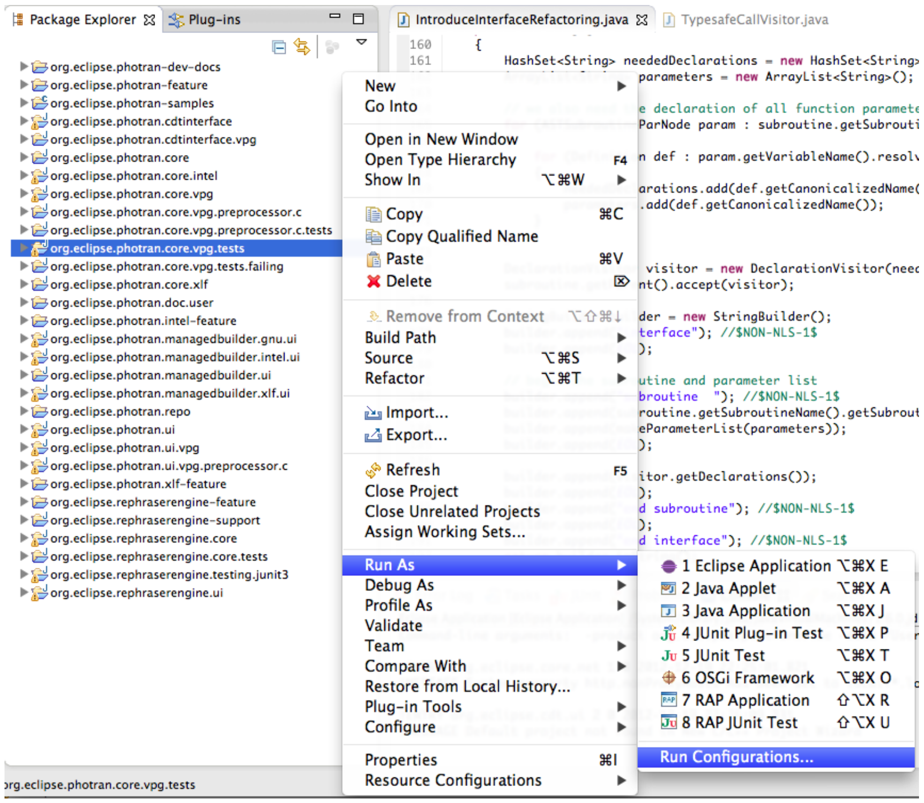
```
PROGRAM Area
!-------------------------------------------------------------------
!
!  This program computes the area of a circle given the input radius
!
!  Uses:  FUNCTION Area_Circle (r)
!
!-------------------------------------------------------------------
IMPLICIT NONE

! Declare local variables
REAL :: radius, Area_Circle, computed_area

!  Prompt user for radius of circle
write(*, '(A)', ADVANCE = "NO") "Enter the radius of the circle:  "
read(*,*) radius

computed_area = Area_Circle(radius)

! Write out area of circle using function call
write(*,100) "Area of circle with radius", radius, " is", computed_area
100 format (A, 2x, F6.2, A, 2x, F11.2)

END PROGRAM Area
```

Figure 14: An example of a program with an unsafe call


external.f90
area.f90

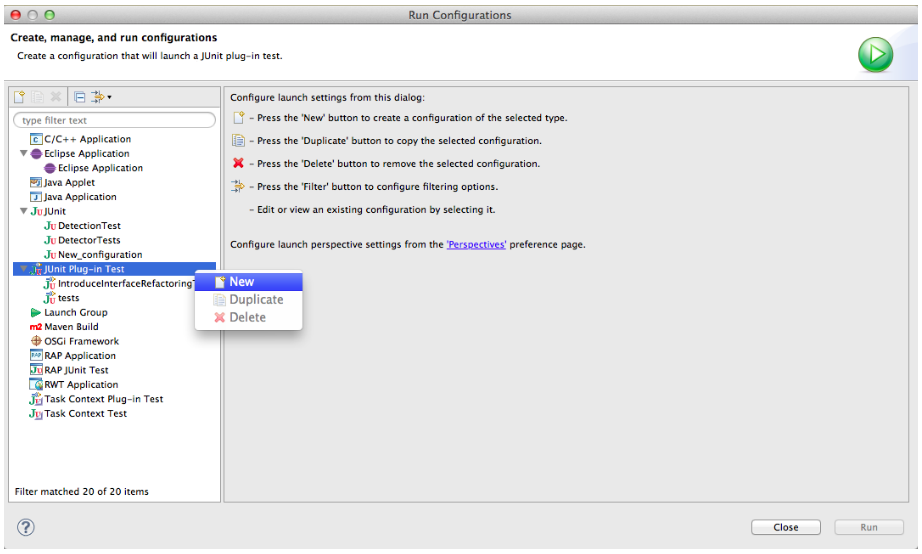
```
PROGRAM Area
!-------------------------------------------------------------------
!
!  This program computes the area of a circle given the input radius
!
!  Uses:  FUNCTION Area_Circle (r)
!
!-------------------------------------------------------------------
IMPLICIT NONE

! Declare local variables
REAL :: radius, Area_Circle, computed_area
interface
    function Area_Circle(r)
        REAL, INTENT(IN) :: r

    end function
end interface

!  Prompt user for radius of circle
write(*, '(A)', ADVANCE = "NO") "Enter the radius of the circle:  "
read(*,*) radius

computed_area = Area_Circle(radius)

! Write out area of circle using function call
write(*,100) "Area of circle with radius", radius, " is", computed_area
100 format (A, 2x, F6.2, A, 2x, F11.2)

END PROGRAM Area
```

Figure 15: An example of a program that has been refactored via our tool


Package Explorer
Plug-ins
org.eclipse.photran-dev-docs
org.eclipse.photran-feature
org.eclipse.photran-samples
org.eclipse.photran.cdtinterface
org.eclipse.photran.cdtinterface.vpg
org.eclipse.photran.core
org.eclipse.photran.core.intel
org.eclipse.photran.core.vpg
org.eclipse.photran.core.vpg.preprocessor.c
org.eclipse.photran.core.vpg.preprocessor.c.tests
org.eclipse.photran.core.vpg.tests
org.eclipse.photran.core.vpg.tests.failing
org.eclipse.photran.core.xlf
org.eclipse.photran.doc.user
org.eclipse.photran.intel-feature
org.eclipse.photran.managedbuilder.gnu.ui
org.eclipse.photran.managedbuilder.intel.ui
org.eclipse.photran.managedbuilder.ui
org.eclipse.photran.managedbuilder.xlf.ui
org.eclipse.photran.repo
org.eclipse.photran.ui
org.eclipse.photran.ui.vpg
org.eclipse.photran.ui.vpg.preprocessor.c
org.eclipse.photran.xlf-feature
org.eclipse.rephraserengine-feature
org.eclipse.rephraserengine-support
org.eclipse.rephraserengine.core
org.eclipse.rephraserengine.core.tests
org.eclipse.rephraserengine.testing.junit3
org.eclipse.rephraserengine.ui
org.eclipse.photran.core.vpg.tests
IntroduceInterfaceRefactoring.java
TypesafeCallVisitor.java
New
Go Into
Open in New Window
Open Type Hierarchy F4
Show In ⌥⌘W
Copy ⌘C
Copy Qualified Name
Paste ⌘V
Delete
Remove from Context ⌥⇧⌘↓
Build Path
Source ⌥⌘S
Refactor ⌥⌘T
Import...
Export...
Refresh F5
Close Project
Close Unrelated Projects
Assign Working Sets...
Run As
Debug As
Profile As
Validate
Team
Compare With
Restore from Local History...
Plug-in Tools
Configure
Properties ⌘I
Resource Configurations
1 Eclipse Application ⌥⌘X E
2 Java Applet ⌥⌘X A
3 Java Application ⌥⌘X J
4 JUnit Plug-in Test ⌥⌘X P
5 JUnit Test ⌥⌘X T
6 OSGi Framework ⌥⌘X O
7 RAP Application ⇧⌥X R
8 RAP JUnit Test ⇧⌥X U
Run Configurations...

```
    {
        HashSet<String> neededDeclarations = new HashSet<String>
                        parameters = new ArrayList<String>();
                        e declaration of all function paramete
                        ParNode param : subroutine.getSubrouti
                        n def : param.getVariableName().resolv
                        arations.add(def.getCanonicalizedName(
                        .add(def.getCanonicalizedName());
                         visitor = new DeclarationVisitor(neec
                        nt().accept(visitor);
                        der = new StringBuilder();
                        terface"); //$NON-NLS-1$
                        );
                        utine and parameter list
                        broutine  "); //$NON-NLS-1$
                        routine.getSubroutineName().getSubrout
                        eParameterList(parameters));
                        );
                        itor.getDeclarations());
                        );
                        d subroutine"); //$NON-NLS-1$
                        );
                        d interface"); //$NON-NLS-1$
```

Figure 16: Running the JUnit tests for Photran from Eclipse



Create, manage, and run configurations

Create a configuration that will launch a JUnit plug-in test.

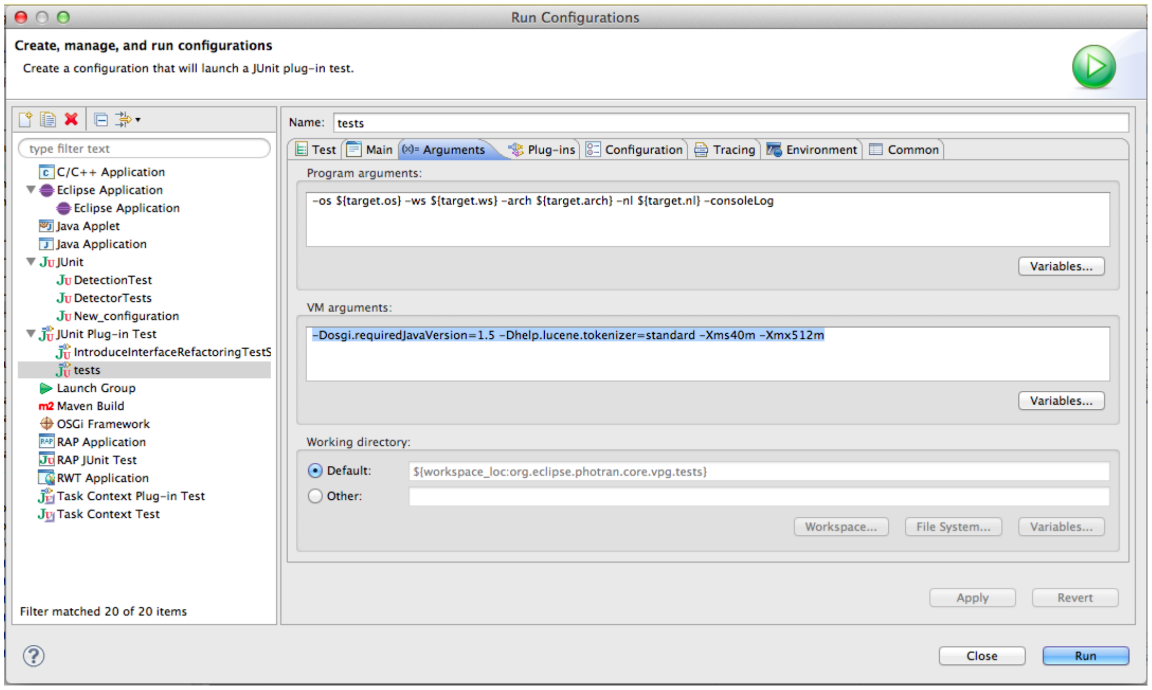
Configure launch settings from this dialog:

- Press the 'New' button to create a configuration of the selected type.
- Press the 'Duplicate' button to copy the selected configuration.
- Press the 'Delete' button to remove the selected configuration.
- Press the 'Filter' button to configure filtering options.
- Edit or view an existing configuration by selecting it.

Configure launch perspective settings from the 'Perspectives' preference page.

Filter matched 20 of 20 items

Figure 17: Creating a Run Configuration for the Photran JUnit tests



Create, manage, and run configurations

Create a configuration that will launch a JUnit plug-in test.

Name: tests

Program arguments:

```
-os ${target.os} -ws ${target.ws} -arch ${target.arch} -nl ${target.nl} -consoleLog
```

VM arguments:

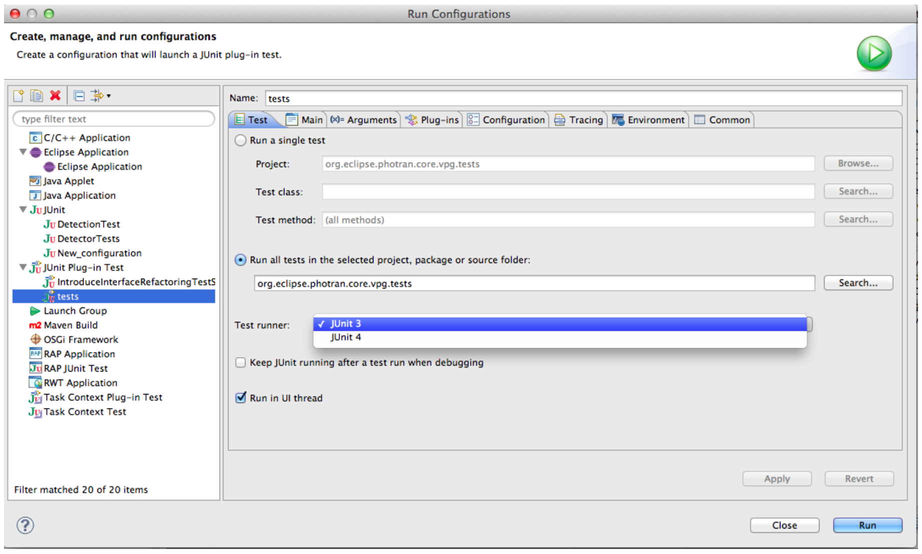
```
-Dosgi.requiredJavaVersion=1.5 -Dhelp.lucene.tokenizer=standard -Xms40m -Xmx512m
```

Working directory:

[✓ Default:] ${workspace_loc:org.eclipse.photran.core.vpg.tests}

Other:

Figure 18: Setting the VM Arguments for Photran's JUnit tests. They are "-Dosgi.requiredJavaVersion=1.5 -Dhelp.lucene.tokenizer=standard -Xms40m -Xmx512m".



Create, manage, and run configurations

Create a configuration that will launch a JUnit plug-in test.

Name: tests

( ) Run a single test

Project: org.eclipse.photran.core.vpg.tests

Test class:

Test method: (all methods)

(●) Run all tests in the selected project, package or source folder:

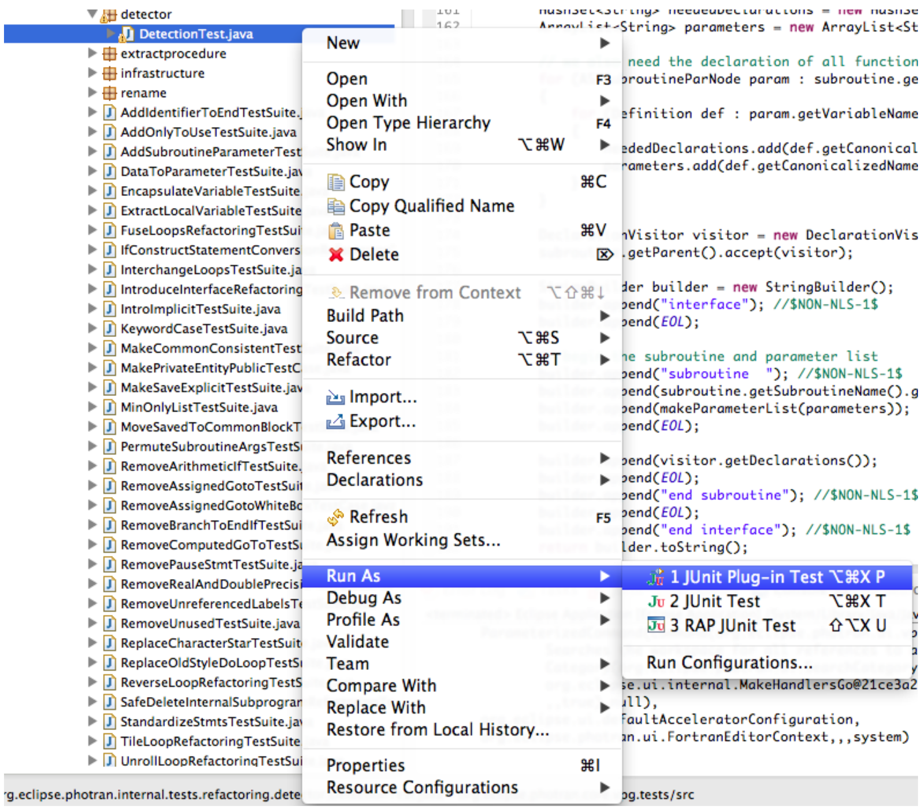
org.eclipse.photran.core.vpg.tests

Test runner: [✓ JUnit 3] JUnit 4

☐ Keep JUnit running after a test run when debugging

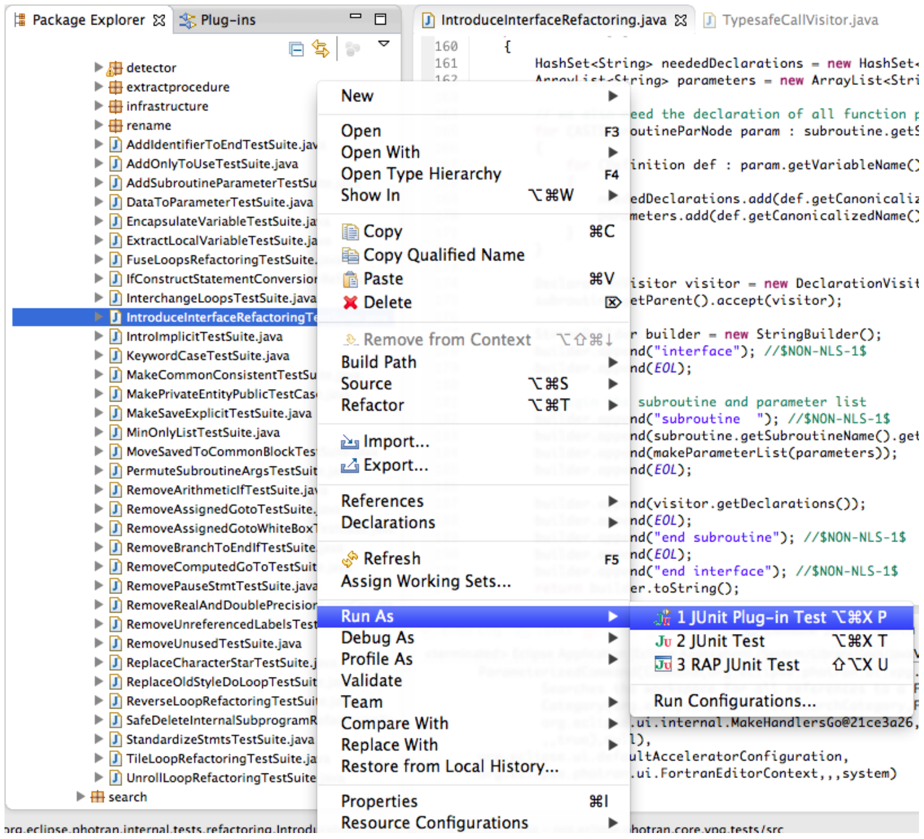
☑ Run in UI thread

Figure 19: Ensuring the Photran tests are run as JUnit 3 tests


detector
DetectionTest.java
extractprocedure
infrastructure
rename
AddIdentifierToEndTestSuite.
AddOnlyToUseTestSuite.java
AddSubroutineParameterTest
DataToParameterTestSuite.ja
EncapsulateVariableTestSuite.
ExtractLocalVariableTestSuite
FuseLoopsRefactoringTestSui
IfConstructStatementConvers
InterchangeLoopsTestSuite.ja
IntroduceInterfaceRefactoring
IntroImplicitTestSuite.java
KeywordCaseTestSuite.java
MakeCommonConsistentTest
MakePrivateEntityPublicTestC
MakeSaveExplicitTestSuite.jav
MinOnlyListTestSuite.java
MoveSavedToCommonBlockT
PermuteSubroutineArgsTestS
RemoveArithmeticIfTestSuite.
RemoveAssignedGotoTestSui
RemoveAssignedGotoWhiteBo
RemoveBranchToEndIfTestSui
RemoveComputedGoToTestS
RemovePauseStmtTestSuite.ja
RemoveRealAndDoublePrecis
RemoveUnreferencedLabelsT
RemoveUnusedTestSuite.java
ReplaceCharacterStarTestSuit
ReplaceOldStyleDoLoopTestS
ReverseLoopRefactoringTestS
SafeDeleteInternalSubprogram
StandardizeStmtsTestSuite.ja
TileLoopRefactoringTestSuite
UnrollLoopRefactoringTestSui
New
Open F3
Open With
Open Type Hierarchy F4
Show In ⌥⌘W
Copy ⌘C
Copy Qualified Name
Paste ⌘V
Delete
Remove from Context ⌥⇧⌘↓
Build Path
Source ⌥⌘S
Refactor ⌥⌘T
Import...
Export...
References
Declarations
Refresh F5
Assign Working Sets...
Run As
Debug As
Profile As
Validate
Team
Compare With
Replace With
Restore from Local History...
Properties ⌘I
Resource Configurations
1 JUnit Plug-in Test ⌥⌘X P
2 JUnit Test ⌥⌘X T
3 RAP JUnit Test ⇧⌥X U
Run Configurations...
rg.eclipse.photran.internal.tests.refactoring.dete
og.tests/src

```
HashSet<String> neededDeclarations = new HashSe
ArrayList<String> parameters = new ArrayList<St
need the declaration of all function
broutineParNode param : subroutine.ge
efinition def : param.getVariableName
ededDeclarations.add(def.getCanonical
rameters.add(def.getCanonicalizedName
nVisitor visitor = new DeclarationVis
.getParent().accept(visitor);
der builder = new StringBuilder();
pend("interface"); //$NON-NLS-1$
pend(EOL);
he subroutine and parameter list
pend("subroutine  "); //$NON-NLS-1$
pend(subroutine.getSubroutineName().g
pend(makeParameterList(parameters));
pend(EOL);
pend(visitor.getDeclarations());
pend(EOL);
pend("end subroutine"); //$NON-NLS-1$
pend(EOL);
pend("end interface"); //$NON-NLS-1$
lder.toString();
se.ui.internal.MakeHandlersGo@21ce3a2
ull),
faultAcceleratorConfiguration,
an.ui.FortranEditorContext,,,system)
```

Figure 20: Running the `TypesafeCallChecker` tests


Package Explorer
Plug-ins
IntroduceInterfaceRefactoring.java
TypesafeCallVisitor.java
detector
extractprocedure
infrastructure
rename
AddIdentifierToEndTestSuite.jav
AddOnlyToUseTestSuite.java
AddSubroutineParameterTestSu
DataToParameterTestSuite.java
EncapsulateVariableTestSuite.ja
ExtractLocalVariableTestSuite.ja
FuseLoopsRefactoringTestSuite.
IfConstructStatementConversior
InterchangeLoopsTestSuite.java
IntroduceInterfaceRefactoringTe
IntroImplicitTestSuite.java
KeywordCaseTestSuite.java
MakeCommonConsistentTestSu
MakePrivateEntityPublicTestCas
MakeSaveExplicitTestSuite.java
MinOnlyListTestSuite.java
MoveSavedToCommonBlockTes
PermuteSubroutineArgsTestSuit
RemoveArithmeticIfTestSuite.ja
RemoveAssignedGotoTestSuite.
RemoveAssignedGotoWhiteBox
RemoveBranchToEndIfTestSuite
RemoveComputedGoToTestSuit
RemovePauseStmtTestSuite.java
RemoveRealAndDoublePrecisior
RemoveUnreferencedLabelsTest
RemoveUnusedTestSuite.java
ReplaceCharacterStarTestSuite.j
ReplaceOldStyleDoLoopTestSuit
ReverseLoopRefactoringTestSui
SafeDeleteInternalSubprogramR
StandardizeStmtsTestSuite.java
TileLoopRefactoringTestSuite.ja
UnrollLoopRefactoringTestSuite
search
New
Open F3
Open With
Open Type Hierarchy F4
Show In ⌥⌘W
Copy ⌘C
Copy Qualified Name
Paste ⌘V
Delete
Remove from Context ⌥⇧⌘↓
Build Path
Source ⌥⌘S
Refactor ⌥⌘T
Import...
Export...
References
Declarations
Refresh F5
Assign Working Sets...
Run As
Debug As
Profile As
Validate
Team
Compare With
Replace With
Restore from Local History...
Properties ⌘I
Resource Configurations
1 JUnit Plug-in Test ⌥⌘X P
2 JUnit Test ⌥⌘X T
3 RAP JUnit Test ⇧⌥X U
Run Configurations...
org.eclipse.photran.internal.tests.refactoring.Introd
hotran.core.vpg.tests/src

```
    {
        HashSet<String> neededDeclarations = new HashSet<
        ArrayList<String> parameters = new ArrayList<Stri

eed the declaration of all function p
outineParNode param : subroutine.getS

inition def : param.getVariableName(

edDeclarations.add(def.getCanonicaliz
meters.add(def.getCanonicalizedName(

isitor visitor = new DeclarationVisit
etParent().accept(visitor);

r builder = new StringBuilder();
nd("interface"); //$NON-NLS-1$
nd(EOL);

 subroutine and parameter list
nd("subroutine  "); //$NON-NLS-1$
nd(subroutine.getSubroutineName().get
nd(makeParameterList(parameters));
nd(EOL);

nd(visitor.getDeclarations());
nd(EOL);
nd("end subroutine"); //$NON-NLS-1$
nd(EOL);
nd("end interface"); //$NON-NLS-1$
er.toString();

.ui.internal.MakeHandlersGo@21ce3a26,
1),
ultAcceleratorConfiguration,
.ui.FortranEditorContext,,,system)
```

Figure 21: Running the `IntroduceInterfaceRefactoring` tests