Refactoring Extenstion for the Yeti Eclipse Plugin

Accelerating NesC Code Development

Distributed Systems Lab (Spring Semester 2010)

Authors:
Noah Heusser
Max Urech

Supervisor:
Prof. Dr. Roger Wattenhofer
Benjamin Sigg

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Abstract

In today's software development processes agile development is the state of the art. Agile processes lead to frequent code reorganizations, or in one word, to "Refactoring". Refactoring can be a lot of boring and error-prone work, if a developer has to do it by hand. One can easy see that a computer is much faster and far more reliable then a human, if the task is to find all occurrences of a variable and give them a new name.

A refactoring plug-in like the one we wrote, allows the software developer to concentrate on the design of the software and delegates the busy, cumbersome work to the computer. This document describes the used frameworks and explains the solution we developed.

The solution is the so called "refactoring" plug-in for Eclipse. The plug-in extends the existing Yeti plug-in for Eclipse. The implementation we developed, allows to rename most elements of the NesC language. We have also forged some other refactorings like "Introduce Alias" or "Extract Function". But we can consider a lot more of possible refactorings.

We hope you enjoy using our software and also the reading of this document.

Contents

Part I

What is refactoring

Chapter 1

Definition of refactoring

Wikipedia gives us the following definition for code refactoring: "Code refactoring is the process of changing a computer program's source code without modifying its external functional behavior in order to improve some of the nonfunctional attributes of the software. Advantages include improved code readability and reduced complexity to improve the maintainability of the source code, as well as a more expressive internal architecture or object model to improve extensibility." Since this document is about a refactoring facility for NesC, we are of course actually always referring to code refactoring, when we talk about refactoring.

Chapter 2

Why do we need refactoring

Reusability and maintainability are two terms, which are known to everybody, who ever heard something about software engineering. A refactoring facility is a means for supporting those properties. I.e. if you want to reuse an existing function in a program in a second place, you might suddenly realise, that the name of the function actually not really matches its purpose and you consequently want to rename it to a more appropriate name. If you have a function rename refactoring at hand at this point, you simply have to select the refactoring, type in the name, press enter and you are done. Without such a tool you had to replace the function name by hand at every position in the program, where it appears, possibly in different files. This example is also good for the cast, that refactoring supports maintainability, because a readable program is also a maintainable program. And if the names of the different entities in a program match their purpose, the program is for sure more readable, then if they do not.

Part II

Eclipse plug-ins

Chapter 3

General info about writing plug-ins

Eclipse is famous for its plug-in architecture. Everything in Eclipse is a plug-in. An Eclipse plug-in is a JAR file or a folder in the plug-ins directory of the Eclipse program folder. For a plug-in it takes at least three files.

- META-INF/MANIFEST.MF
- plugin.xml
- plugin.class

3.1 META-INF/MANIFEST.MF

The Manifest file is the first file that is read by Eclipse while loading the plug-in. It contains all the information about what requirements are needed to load the plug-in and how it can be loaded. We will now explain the most important entries in the Manifest file.

Listing 3.1: MANIFEST

```
1  Manifest-Version: 1.0
2  Bundle-ManifestVersion: 2
3  Bundle-Name: TinyOS_Refactoring
4  Bundle-SymbolicName: tinyos.yeti.refactoring;singleton:=true
5  Bundle-Version: 1.0.0
6  Bundle-Activator: tinyos.yeti.refactoring.RefactoringPlugin
7  Require-Bundle: org.eclipse.ui,
8   org.eclipse.core.runtime,
9   tinyos.yeti.core;bundle-version="2.2.17",
10  tinyos.yeti.parser.nesc12;bundle-version="1.2.17",
11  org.eclipse.ui.workbench.texteditor;bundle-version="3.5.1",
12  org.eclipse.ltk.core.refactoring;bundle-version="3.5.0",
13  org.eclipse.ltk.ui.refactoring;bundle-version="3.4.101",
14  org.eclipse.core.resources;bundle-version="3.5.2",
15  org.eclipse.jface.text;bundle-version="3.5.2",
16  org.eclipse.ui.ide;bundle-version="3.5.2",
17  org.eclipse.ui.editors;bundle-version="3.5.0",
18  tinyos.yeti.preprocessor.nesc12;bundle-version="1.2.17",
19  org.eclipse.core.expressions;bundle-version="3.4.101"
20 Bundle-ActivationPolicy: lazy
```

```
21 Bundle-RequiredExecutionEnvironment: JavaSE-1.6
```

Manifest-Version This line shows, that the manifest entries have the form of "header: value" pairs. The name of a header is separated from its value by a colon. [1]

Bundle-ManifestVersion Manifest header identifying the bundle manifest version. A bundle manifest may express the version of the syntax in which it is written by specifying a bundle manifest version. Bundles exploiting OSGi Release 4, or later, syntax must specify a bundle manifest version. The bundle manifest version defined by OSGi release 4 or, more specifically, by version 1.3 of the OSGi core specification is "2". (from Eclipse Help Version: 3.5.2)

Bundle-Name A human readable, meaningful name for the plug-in you write.

Bundle-SymbolicName A unique string identifier for you plug-in. Usually you use the path of your package structure. The *singelton:=true* makes the OSGi Framework only load your plug-in once. It has to be set in every Eclipse plug-in. It is only necessary because Eclipse uses OSGi, which is not only used for Eclipse.

Bundle-Version Defines the Version of the plug-in. When the plug-in is used by an other plug-in, it can specify the version in the *Required-Bundle* parameter. A version is composed of 3 positive natural numbers and a string separated by "." sign. Two versions are equal, if the numbers are equal. The string in the end might be used to specify different versions with the absolute same interface (for example different compilers). The numbers from left to right are called: Major.Minor.Mirco.

Bundle-Activator This is the path of the class, which is used to start the whole plug-in. The class must extend *org.eclipse.core.runtime.Plugin*.

Require-Bundle When you write you plug-in, you will use the functionality of other plug-ins. To do that you have to specify here, which plug-ins you use. Eclipse will refuse to load your plug-in if the plug-ins you require are not available. In addition you can specify the version of the plug-in you need to be available. If you don't specify anything else Eclipse will assume to have a compatible version as long as there is one having the same Major-Number. In case you have to be pickier, you can also add a *match* parameter with *perfect*, in which case Major, Minor and Micro number have to match, or *equivalent*, then Major and Minor Number have to match.

Bundle-ActivationPolicy The only parameter you can add here is *lazy*, which tells OSGi to wait with loading the plug-in, until it is used.

Bundle-RequiredExecutionEnvironment Defines a lower bound for the version of the JVM, that is allowed to be used.

3.2 plugin.xml

The *plugin.xml* file was once responsible for all the things, that are now configured in the *MANIFEST.MF* file. If you browse the Internet, you will have problems to find, what you are looking for, because of that. Today the *plugin.xml* is responsible for offering and using extension points. We will not talk about how to offer an extension point, because we did not need this feature in this project.

[1] from http://java.sun.com/developer/Books/javaprogramming/JAR/basics/manifest.html access 2.9.2010

3.2.1 Using extension points (making extensions)

When you want to use an extension point, you have to add the plug-in, which defines the extension point to your *Require-Bundle* list in the *MANIFEST.MF*. Mostly you will have found the extension point you want to use by Google and often you do not know what plug-in is offering that extension point. The only advice we can give you, is to use Google again or grep through the plug-ins directory of Eclipse. We did not find a way, which allowed us to easily find the right plug-in.

When the right plug-ins are included, one can use the extension points by adding a <*extension*> tag.

Listing 3.2: Use extension point (*plugin.xml*)

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <?eclipse version="3.4"?>
3 <plugin>
4         <extension
5             point="org.eclipse.core.expressions.propertyTesters">
6                 <propertyTester ... />
7         </extension>
8 </plugin>
```

The <*extension*> tag has only one important attribute. It is the extension point you want to use. In this case we use the extension point *propertyTesters* offered by the plug-in *org.eclipse.core.expressions*.

The tags between <*extension*> and </*extension*> are information for the extension point. Which tags need to be there, can be read in the XML-Schema file in the *org.eclipse.core.expressions* plug-in Jar. We do not want to go into detail here, but the *plugin.xml* of the plug-in, which provides the extension point, defines where in the Jar the Schema file is. Eclipse can show the Schema file by clicking on *Show extension point description* in the *Extention*-Tab of the *plugin.xml* editor.

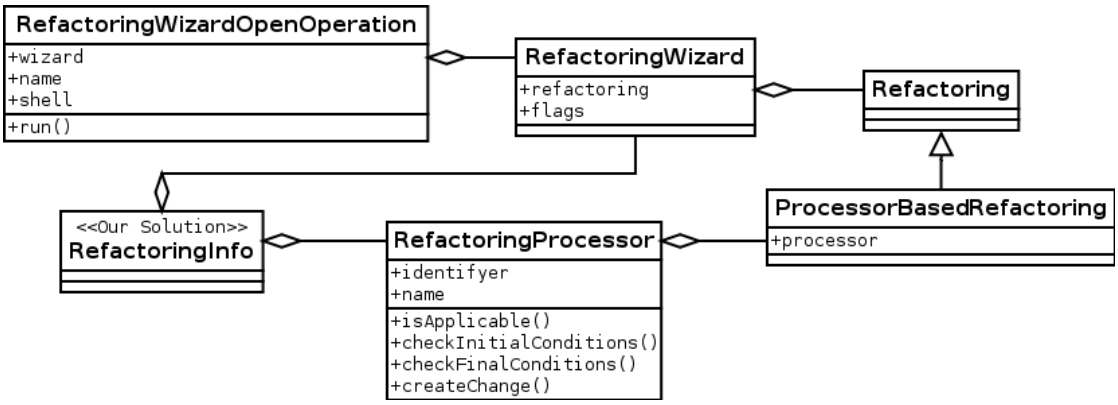
Important is to understand how Eclipse calls your code. First you find the extension point you want to extend. Then you write your XML code to extend it. For some extension points a class implementing an interface has to be passed as attribute. If so, the extension point loads your class and executes a method of that interface.

Chapter 4

Language Toolkit for processor based refactoring

For our refactoring plug-in we used the processor based refactoring, offered by the Language Toolkit of Eclipse. We will explain now, how such a refactoring works.

4.1 Important classes



In the class diagram the most important classes are shown. While writing refactorings you do not have to care about the classes *RefactoringWizardOpenOperation*, *Refactoring* and *ProcessorBasedRefactoring*. This classes can be used as they are. All the other classes need to be extended.

RefactoringWizard When the user starts a refactoring, he will be asked for additional information. The required form for this, is painted by the *RefactoringWizard* class. The connection to the Info class is necessary to store the information, which the user typed into the form.

RefactoringInfo The *RefactoringInfo* class is not a class of the LTK framework, we introduced it to allow communication between the *RefactoringProcessor* and the *RefactoringWizard*.

The idea to do it that way is from the article http://www.eclipse.org/articles/Article-LTK/ltk.html. When writing refactorings for our project, you can extend the *RefactoringInfo* class. It supports already some utility functionality.

RefactoringProcessor This class does the actual Refactoring.

isApplicable() This method checks if the refactoring can be used. It can be used to enable or disable menu entry's. During our work with the refactoring processor, this method got never called by the Framework. It can easily be implemented by calling *checkInitialConditions()*.

Listing 4.1: Generic isApplicable Implementation

```
1 public boolean isApplicable() throws CoreException {
2   return checkInitialConditions(new NullProgressMonitor()).
3            hasFatalError();
4 }
```

identifier A unique identifier for the processor.

name A human readable name for what the refactoring does.

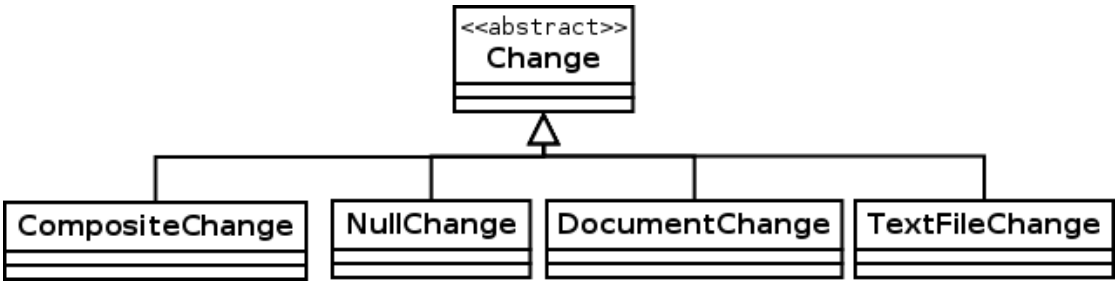
checkInitialConditions Checks whether the refactoring is applicable before the user gives any information, about how the refactoring should happen. Mostly we checked here, if the user did a valid selection. For the conditional visibility we implemented, it was used to decide whether a refactoring is offered or not. The return value is a status. If the refactoring can not be executed, the status must contain fatal errors. Otherwise the refactoring is meant to be executable.

checkFinalConditions Checks whether the information, the user gave to configure the refactoring, are valid.

createChange This is the method where the magic happens. All the changes of your refactoring are made within this method. The return value is a *Change*. A *Change* describes what you want to change in your refactoring. You do not execute the change yourself, this is done by the framework. LTK offers a lot of subtypes of the *Change* class.

4.2 The *Change* classes

The LTK offers the *Change* class. By creating *Change* classes in stead of doing the changes yourself, you allow LTK to provide "change preview functionality". LTK offers a lot of changes for moving, renaming and so forth. We just talk about the most important ones.



CompositeChange Most refactorings change more than one thing. That's way you will use this *Change* in almost every refactoring.

NullChange A *Change* that does nothing. It allows you to avoid null-values within your Code.

DocumentChange Changes a text file, which is opened in the editor. It allows you to change code that has not yet been saved. We intended to use this, but we got exceptions, because threads tried to access data, which are not allowed to be accessed by them. In the Internet we found people having the same problem, but they also had no solution. The problem seems to occur only in the newest version of Eclipse. Maybe it is already solved when you read this. If so, use this Change for files that are opened instead of *TextFileChange*.

TextFileChange We used the *TextFileChage* to do all our text changes. It works, but the User is forced to save all files, before he can start doing refactoring.

4.3 Refactoring operation sequence

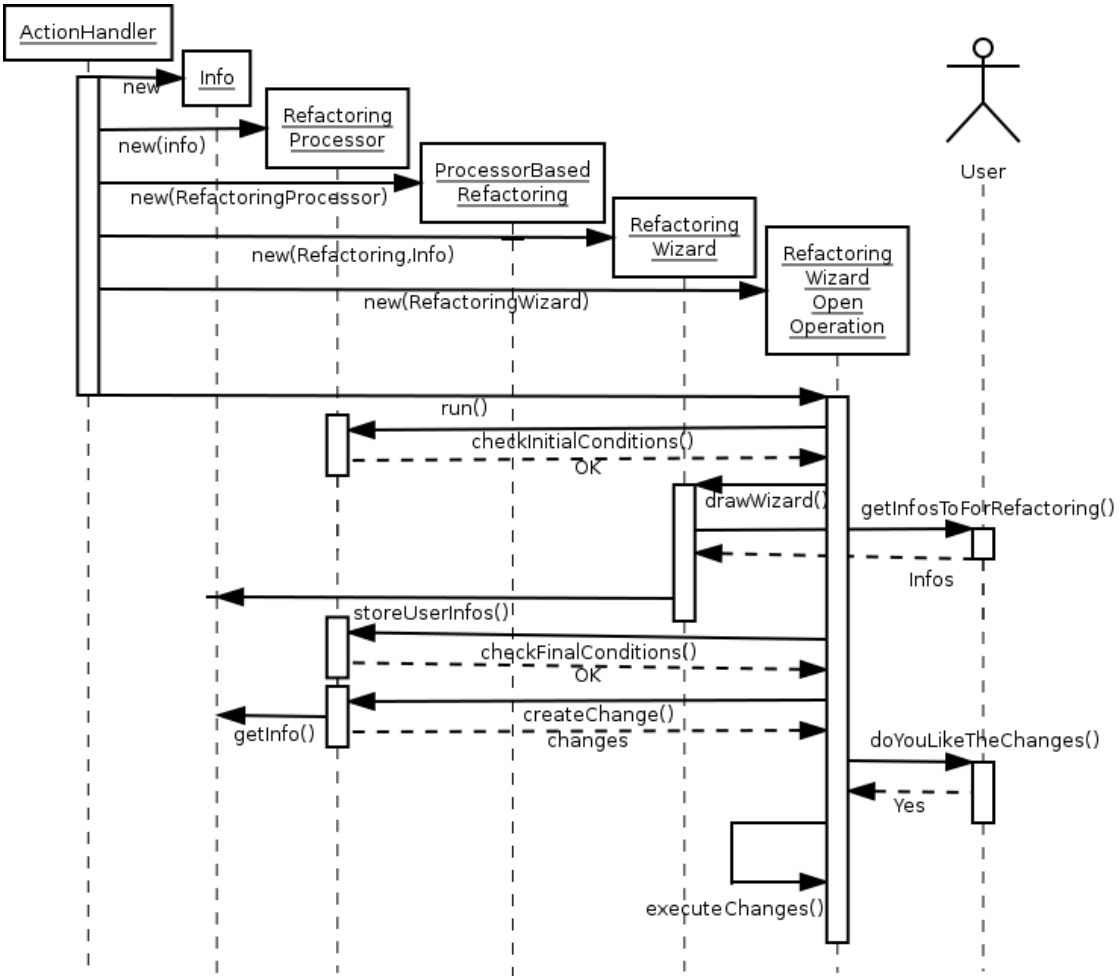
Now you know all the classes. We will not outline what happens, when a refactoring takes place. First, (not in picture 4.3) the user starts a refactoring by clicking on a menu entry. The *Action-Handler* gets called and initializes all necessary classes. You can see that the *RefactoringWizard* and the *RefactoringProcessor* both get a link to the *Info* Object. If you are working with the selection, save the selection to the *Info* Object in the ActionHandler, later it is hard, because you are in the wrong thread.

Then the *ActionHandler* calls the *run()* method of the *RefactoringWizardOpenOperation*. We have to say that the calls, which are done by the *RefactoringWizardOpenOperation*, are not really done by the *RefactoringWizardOpenOperation* class. But it would become very complex to explain how it really works. The effect you see, when your methods are called, is as shown in the picture.

The *RefactoringWizardOpenOperation* checks if the initial conditions are OK. If not, it would show the status error message to the user and terminate. If there are no fatal errors, the *RefactoringWizard* is called to draw the form and save the necessary information of the user to the *Info* object.

Now, the final conditions can be checked by the *RefactoringProcessor*. If they are not OK, the status errors will be shown to the user and the *RefactoringWizard* will be shown again. If they are OK, the *createChange()* method of the *RefactoringProcesser* gets called. A preview of the changes is shown to the user.

If the user accepts the changes, they get applied and the Refactoring is finished. During the whole Process, LTK makes sure, that the user has *back* and *cancel* buttons.

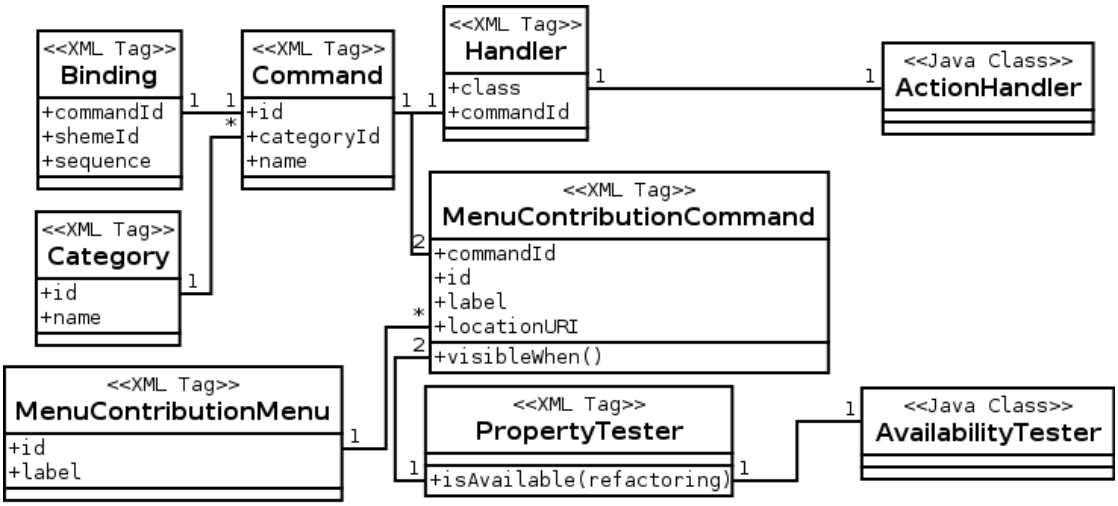

ActionHandler
Info
Refactoring Processor
ProcessorBased Refactoring
Refactoring Wizard
Refactoring Wizard Open Operation
User
new
new(info)
new(RefactoringProcessor)
new(Refactoring,Info)
new(RefactoringWizard)
run()
checkInitialConditions()
OK
drawWizard()
getInfosToForRefactoring()
Infos
storeUserInfos()
checkFinalConditions()
OK
createChange()
changes
getInfo()
doYouLikeTheChanges()
Yes
executeChanges()

Chapter 5

Menu's with conditional visibility

In our plug-in the use case is always the same. The user selects some code, the user selects a refactoring, the user executes the refactoring. To achieve that we need to have menus, where the user can select the refactoring he wants to execute. But this is not enough, our plug-in supports a lot of refactorings. If the User has to find the refactoring he wants to use in a list of all possible refactorings, it will take him a lot of time. To make it easy for the user, we want to list only those refactorings, which are actually possible for his current selection. This means we need conditional visibility.

In the picture 5 the necessary components for a menu with conditional visibility are shown. During the following sections we will explain every component in the picture 5.



5.1 Create a menu entry

Eclipse knows about a hundred ways to create a new menu entry. Almost none of them make it necessary to write Java code, but all of them have different possibilities. During our project we were not able to discover what ways are deprecated or what the state of the art is. After testing

three ways which did not fit our needs, we chose the one, which we will explain now.

5.1.1 Add a new submenu

We want our refactoring functionality to be available in two menus. In the main menu of the Eclipse window and the context menu, which appears when one right clicks on a selection. To do that, we used the *org.eclipse.ui.menus* extension point. In this extension point every menu can be identified by a URI. The URI of the main menu is: *menu:org.eclipse.ui.main.menu* and to define it's position, we add *?after=additions* to the URI. This adds the menu just after the menus of the additions group, which is the core functionality of the Eclipse IDE. In the UML this *<menu>* tag is the *MenuContributionMenu* class.

In XML code it looks like this:

Listing 5.1: Create menu folder (*plugin.xml*)

```
9  <extension id="add.item" point="org.eclipse.ui.menus">
10   <-- Entrys in the Top Menu -->
11   <menuContribution
12       locationURI= "menu:org.eclipse.ui.main.menu?after=additions">
13     <menu id="tinyos.yeti.RefactoringMenu"
14           label="Refactoring" />
15   </menuContribution>
16 </extenstion>
```

Important is to see that we use the tag *<menu>*, which means that we add a submenu and that the id of the new submenu *tinyos.yeti.RefactoringMenu* is. Later when we add commands to that submenu, we have to refer to that id.

To add the submenu to the context menu too, we add the same XML block again, but use the URI *popup:org.eclipse.ui.popup.any* and give it the id *tinyos.yeti.RefactoringPopup*.

If you would now start Eclipse with that new menu, you wouldn't see anything. The submenus are only displayed if the submenu are not empty.

5.1.2 Add a command to the submenu

Adding a command (an entry which triggers an action) is also done by the tag *<menuContribution>*, but this time not by the child tag *<menu>* but *<command>*. In the UML diagram we called it *MenuContributionCommand*. For now we do not look at the *visibleWhen*, it will be explained later.

The *<command>* tag has three attributes. *lable* is the string the user will see in the menu. *id* is just a unique identifier and *commandId* is the id of the command, that shall be executed, when the user clicks on this entry. Creating that command will be explained in the next section.

In XML it looks like this:

Listing 5.2: Adding a command to a menu (*plugin.xml*)

```
17 <extension id="add.item" point="org.eclipse.ui.menus">
18   <menuContribution
19         locationURI= "menu:tinyos.yeti.RefactoringMenu?after=additions">
20     <command commandId="tinyos.yeti.refactoring.rename.local.variable"
21              id= "RenameLocalVariable"
22              label= "Rename Local Variable" />
23   </menuContribution>
24 </extension>
```

You can see, that the *menuContribution* goes this time to *tinyos.yeti.RefactoringMenu*, which we defined in the last section.

5.1.3 Createing a command

This is now the command, that is called command in the UML diagram. It is not in the extenstion point *org.eclipse.ui.menus* but in the extenstion point *org.eclipse.ui.commands*. The <*command*> tag has three attributes. An *id* which is a unique identifier. The *categoryId* which groups the the <*command*>'s to categories. We never used the Categories, so we just introduced one and gave all our commands this category. We hoped, that it would be possible to fill a submenu with a category of commands. But we did not find a way to achieve that. The third parameter is a *name*. We never used it either, we usually took the label in the menu also as name of the command.

In XML it looks like this:

Listing 5.3: Creating a command (*plugin.xml*)

```
25 <extension point="org.eclipse.ui.commands">
26   <category id="tinyos.yeti.refactoring"
27             name="Refactoring" />
28   <command id="tinyos.yeti.refactoring.rename.local.variable"
29            categoryId="tinyos.yeti.refactoring"
30            name="Rename Local Variable"/>
31 </extension>
```

One can ask now, where does Eclipse know from what method it has to invoke to call that command. Well it does not. To know that, we have to define a *handel*.

5.1.4 Introducing a handler for a command

A handler connects a command in the XML file to a Java class, which implements the interface *org.eclipse.core.commands.IHandler*. The extension point *org.eclipse.ui.handlers* helps doing that job. It is very simple. Just the *class* and the *commandId*.

Listing 5.4: Connecting a command to a handler (*plugin.xml*)

```
32 <extension point="org.eclipse.ui.handlers">
33    <handler class="tinyos.yeti.refactoring.entities.variable.rename.local.
          RenameLocalVariableActionHandler"
34             commandId="tinyos.yeti.refactoring.rename.local.variable" />
35 </extention>
```

5.1.5 Bindings

We did not yet speak about bindings. They allow you to introduce shortcuts for your commands. We did it first, but removed it later again. We think the following code explains itself:

Listing 5.5: Introduce shortcuts (*plugin.xml*)

```
36 <extension point="org.eclipse.ui.bindings">
37                 <!--
38                         One could also add a Shortcut, by adding the following
                               Attribute
39                         to the <key>-Tag,
40                         for example: sequence="M1+M2+r"
41
42                         The M Keys in the sequence are Platform independent keys.
43                                 On PCs they are mapped to:
44                 M1 = Ctrl
45                 M2 = Shift
46                 M3 = Alt
```

```
47
48                  The upper example would the the shortcut: Ctrl+Shift+r
49                  -->
50                  <key commandId="tinyos.yeti.refactoring.rename.local.variable"
51                       schemeId="org.eclipse.ui.defaultAcceleratorConfiguration"
52                       sequence="M1+M2+r" />
53 </extenstion>
```

5.2 Add conditional visibility

Up to now all our menu entries will always be available no matter whether the user did a valid selection or not. But in the beginning of this chapter we decided to show only those refactorings, which are executable for the current selection.

5.2.1 Property tester

To do that we use the extension point *org.eclipse.core.expressions.propertyTesters*. It allows the definition of a *propertyTester* with a list of properties. In our case we defined one *propertyTester* called *tinyos.yeti.refactoring.AvailabilityTester*, which has one property for each Refactoring.

In XML:

Listing 5.6: Property tester (*plugin.xml*)

```
54 <extension point="org.eclipse.core.expressions.propertyTesters">
55   <propertyTester id="tinyos.yeti.refactoring.AvailabilityTester"
56                   type="org.eclipse.jface.text.ITextSelection"
57                   namespace="tinyos.yeti.refactoring.isAvailable"
58                   properties="renameLocalVariable,..."
59                   class="tinyos.yeti.refactoring.RefactoringsAvailabilityTester" /
                         >
60 </extension>
```

One can see that is has a *class* parameter. This is a Java class that extends *org.eclipse.core.expressions.PropertyTester*. It has only one abstract method:

Listing 5.7: extends PropertyTester

```
5 public class AvailabilityTester extends PropertyTester {
6
7   public boolean test(Object receiver, String property, Object[] args,Object
        expectedValue) {
8
9   }
10 }
```

One can see, that there is a string *property*. It is exactly the String, that is defined in the *property* attribute of the *propertyTester* tag. This way we wrote for each refactoring a test, whether it is available or not.

5.2.2 Visibility condition

Listing 5.8: Visibility condition (*plugin.xml*)

```
61 <extension id="add.item"
62              point="org.eclipse.ui.menus">
63   <menuContribution locationURI= "menu:tinyos.yeti.RefactoringMenu?after=additions
         ">
```

```
64      <command commandId="tinyos.yeti.refactoring.rename.local.variable"
65               id= "RenameLocalVariable"
66               label= "Rename Local Variable">
67        <visibleWhen checkEnabled="false">
68          <iterate ifEmpty="false">
69            <!-- The forcePluginActivation-Parameter is absolutly necessary.
70                 Otherwise the Property gets never checkt -->
71            <test property="tinyos.yeti.refactoring.isAvailable.renameLocalVariable"
72                  forcePluginActivation="true" />
73          </iterate>
74        </visibleWhen>
75      </command>
76    </menuContribution>
77  </extenstion>
```

In the *MenuContributionCommand* we add a *visibleWhen* tag. The *checkEnabled="false"* disables the check whether the Command is available or not. It is a functionality provided by the handler class. Technically it would work with just writing *checkEnabled="true"* and implementing the functionality in the *ActionHandler*. But as the name *isEnabled* says, it is not the idea of this function. That's way we preferred doing it with the property.

The *visibleWhen* is necessary cause it seems to be a collection that we get there. If the tag is not there it won't work.

Part III

The Refactoring Plug-in

Chapter 6

About our plug-in

The refactoring plug-in is an extension for the Eclipse Yeti plug-in. Its target is to accelerate the development of NesC code and make it more convenient. This is achieved by automating low level, time consuming tasks.

6.1 The general refactoring life cycle

From a user perspective the general refactoring life cycle looks like this:

1. The user selects a new text range in his NesC editor.
2. Eclipse instantiates a *RefactoringAvailabilityTester* class, which is a subtype of the *PropertyTester* class, which is defined by Eclipse.
3. Eclipse checks all refactorings defined in the *plugin.xml* against the property tester and memorises, which of them are available.
4. Eclipse shows the available refactorings in the top menu bar, as well as in the popup menu.
5. The user selects a refactoring, which he wants to execute.
6. Eclipse will execute the appropriate action handler.
7. The user probably has to do some input.
8. Eclipse shows the changes to be done in a preview window.
9. The user confirms or denies the changes.
10. Eclipse finishes the refactoring.

This life cycle can be divided in two main parts. The first part is the evaluation of the available refactorings and the second part is the actual execution of a specific refactoring. This partitioning has two major advantages. First it allows a better user experience, since he sees only the refactorings, which are really executable for the current selection. Second, the execution of a refactoring does not have to care, if the refactoring is applicable.
In the graphic on page 21 you can see the steps the user has to take in order to execute a refactoring.

6.2 Package structure

The root package of the plug-in is *tinyos.yeti.refactoring*. It contains the *RefactoringPlugin* class, which is required for an Eclipse plug-in, as well as the classes we will talk about in chapter 7 on page 22.
The root package contains four subpackages:

1. The *abstractrefactorings* package
2. The *ast* package
3. The *utilities* package
4. The *entities* package

The *abstractrefactorings* package contains infrastructure classes, used to implement concrete refactorings. We take a closer look at this classes in chapter 8 on page 25.
The *ast* and *utilities* packages include classes, which are used to interact with the AST and the project, in which we are doing refactoring. The contents of this packages are explained in the chapter 9 on page 28.
The *entities* package is actually the root package for all concrete refactoring implementations. By entity we mean an object, which can be modified by a refactoring. I.e. the entity of a rename or extract function refactoring is a function. The entity of a rename or introduce interface alias is an interface alias. Therefore the next level of packages in the *entities* package will designate a entity, and the package in such a specific package will designate a concrete refactoring. This means that you will find the concrete implementation of the rename interface alias refactoring implementation in the subpackage *entities.interfaces.alias.rename*. About concrete refactorings we will talk in the chapter 10 on page 31.

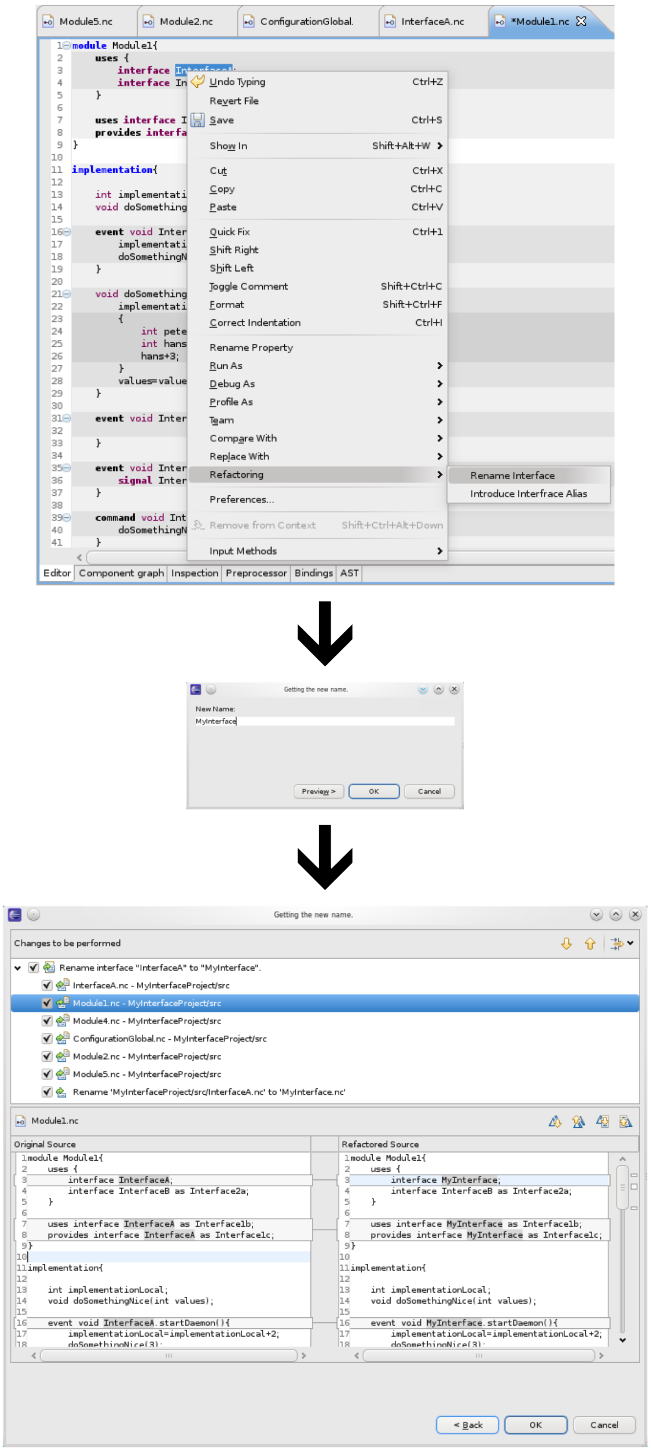


Figure 6.1: A standard user interaction with the refactoring plug-in.

Chapter 7

From plug-in XML to Java code

7.1 How to decide the availability of a refactoring

After we have introduced the appropriate lines of XML in the *plugin.xml* for a specific refactoring, the plug-in is now able to ask the question: "Is the current selection appropriate for this refactoring?" But the software is not yet capable to answer this question. That is where the *Refactoring* enum and the *RefactoringAvailabilityTester* come in. An overview of the relations described in this chapter is given in the graphic on page 23.

7.1.1 The *Refactoring* enum

The *Refactoring* enum defines for each refactoring, which is defined in the *plugin.xml*, its corresponding counterpart in java code as an enum constant. Every such enum constant has three fields:

1. *propertyName* of type *String*
2. *entityName* of type *String*
3. *tester* of type *IRefactoringAvailabilityTester*

The *propertyName* string must match exactly the string, which is given in the *plugin.xml* as property name for the refactoring. It is later on used, to direct the question, if a selection is appropriate for a specific refactoring, to the right answer.
The *entityName* string is used only to output information to the user. It is intended to designate the entity, which is modified by the refactoring. I.e. the entity name could be "function", if the refactoring is about renaming a function, or "alias", if the refactoring introduces a new alias.
The *tester* field contains for each refactoring an instance of type *IRefactoringAvailabilityTester*. This instance is the one, which will finally answer the question, if the refactoring is available for the current selection.

7.1.2 The *RefactoringAvailabilityTester* class

The *RefactoringAvailabilityTester* is the class, which is designated in the *plugin.xml* as the property tester. This means that for every refactoring, which is defined in the *plugin.xml*, eclipse will ask an instance of this class, if the refactoring is available for the current selection. Eclipse does so by calling the function *test* of *RefactoringAvailabilityTester*. Everytime the user changes the

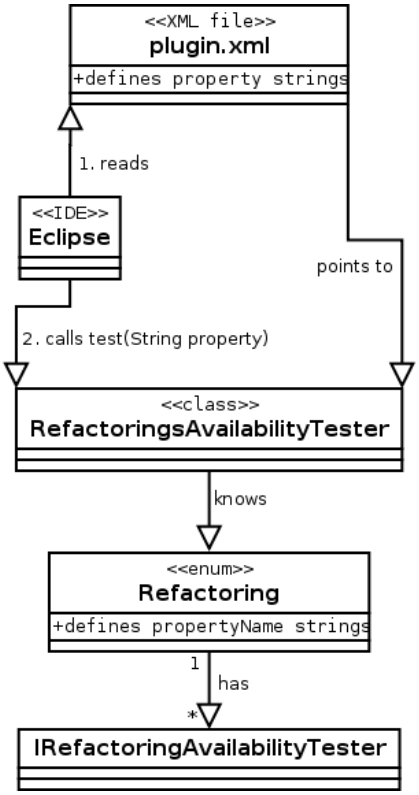


Figure 7.1: The *RefactoringAvailabilityTester* class and its context

selection in the editor, one such call is executed for every defined refactoring. The declaration of this function is actually inherited from the abstract supertype *PropertyTester*, which is a class defined by Eclipse.
The *test* function has a return value of type *boolean* and four parameters:

1. *receiver* of type *Object*
2. *property* of type *String*
3. *args* of type *Object[]*, we do not make use of this parameter.
4. *expectedValue* of type *Object*, we do not make use of this parameter.

The *receiver* parameter must be an instance of type *ITextSelection*, since the property tester is configured like that in the *plugin.xml*. This instance contains information about the range of the current selection.
The second parameter of the *test* function is of type *String* and is named *property*. During a call this will be one of the strings, which we have defined in the *plugin.xml*, for a specific refactoring.
The *test* function now first checks, if the plug-in is even fully loaded at this point. If we would omit this check, it could be that not all information are available in a refactoring processor, which are assumed to be always available. This would lead to nondeterministic behaviour. If this check fails the function just returns false, which means that there will be no refactoring available. Furthermore there is a check, if all source files are saved. If there are modified ones,

the function returns true only for the *NotSaved* property. If this is the case, the user can save all modified files with a click on a button under the refactoring menu.
If the plug-in is fully loaded, we come now to the point, where the real matching from XML to java code happens. The *test* function looks for the Refactoring enum constant, which's *propertyName* string matches the *property* string passed as an argument. If there is no such enum constant, then either the programmer forgot to define the corresponding enum constant for a refactoring defined in the *plugin.xml*, or the *propertyName* and the *property* strings are not equal, i.e. if there is a typo in one of them.
If we now have the *Refactoring* enum constant for a given *property* string, then we can read out its *tester* field. This gives us an instance of type *IRefactoringAvailabilityTester* and we can execute a call to the function *test* of this instance. This function will return true, if this refactoring is available and false otherwise. This return value is also the appropriate return value for our *test* function of the *RefactoringAvailabilityTester*.

7.1.3 The *IRefactoringAvailabilityTester* interface

Each refactoring has an instance of type *IRefactoringAvailabilityTester* assigned to it. This assignment is done in the *Refactoring* enum. This interface has a single function *test*, with a return value of type *boolean*, and a single parameter *selection* of type *ITextSelection*. The purpose of the function is to decide, if the current selection is a selection, which is appropriate for the specific refactoring to be executed. If it is, the function will return true and false otherwise.
To find this decision, the refactoring can make use of the class *ActionHandlerUtil* to gain access to further information, i.e. to get the selected editor, or the selected file.

7.2 How to execute a specific refactoring

At this point Eclipse knows, which refactorings are available for the current selection. Now the refactoring has to be executed, if the user clicks on the corresponding button in the refactoring top menu, or the popup menu of Eclipse. Behind this buttons sits an implementation of the *AbstractHandler* abstract class. This class is defined by Eclipse. For each refactoring such an implementation has to be defined in the *plugin.xml*. Thats how a refactoring is mapped to its execution.

7.2.1 The *AbstractHandler* abstract class

This class is defined by eclipse as an abstract class. For our implementations there is only the function *execute* of interest. This function is declared abstract in the *AbstractHandler* class and therefore has to be implemented in its subtypes.
This function is actually the place, where the Language Toolkit for Processor Based Refactoring goes into action. Here will the wizard be initialized, which leads the user through a specific refactoring. This includes also initializing the appropriate subtype implementation of the *RefactoringProcessor* class, which is defined as part of the Eclipse ltk library. This implementation will finally execute the real work, which is the actual surplus of the refactoring.

Chapter 8

Abstract refactorings

8.1 What we mean by abstract refactoring

Refactorings have the peculiarity, that they can be grouped into classes, which appear to the user to be similar. In our case the only such group we really have implemented is the renaming of program identifiers. All but the *extract function* refactoring are actually implemented as subtypes of the elements in the *abstractrefactoring.rename* package. But we could also imagine other groups, i.e. in Java there exist the group of generators, which generate some code for you, or there are refactorings, which allows you to push up code in a supertype, or pull it down in a subtype.
The similarities of the elements of such a group have a direct impact on the needed infrastructure, which is needed to execute such a refactoring. I.e. the user has the same input to do, and the same steps to follow.
The subpackages of the package *abstractrefactorings* right in the root package of the plug-in, are intended to hold classes, which define the infrastructure for a specific group . This classes gather the code, which is reused for every element of a specific group. The direct consequence is writing less code, and especially introducing less errors, when creating new refactorings. Also in the sense of extensibility this package structure makes for sure sense.
Because nearly all of our refactorings are about renaming, we take a closer look on the associated classes in the *abstractrefactoring.rename* package.

8.2 The *abstractrefactoring.rename* classes

8.2.1 The *RenameAvailabilityTester* abstract class

Before a user is allowed to execute a rename refactoring, eclipse has to force him, to select a program identifier. Namely the identifier which has to be renamed. The *RenameAvailabilityTester* implements the *IRefactoringAvailabilityTester* interface, which we already talked about in the last chapter. Therefore it overrides the interface's *test* function. In this function it tries to get an identifier out of the given user selection. If this is not possible, the function will just return false, which means that the associated refactoring is not available for the selection. Otherwise it calls its own abstract function *isSelectionAppropriate* which will solve the question, if the refactoring is available, in a subtype. This function has an argument of type *Identifier*, which is actually an AST element, which will be the currently selected identifier, during a call. This takes the burden

of the subclasses to find the identifier by themselves. Such an implementation then normally solves the question in about four lines of code by means of an instance of type *SelectionIdentifier*.

8.2.2 The *SelectionIdentifier* class

Identifiers of an entity, which is to be renamed, appear mostly in different places for different purposes. I.e. a function identifier can represent a function definition, a function declaration or a function call. A subtype of *SelectionIdentifier* is intended to identify what purpose an identifier represents. This enables a *SelectionIdentifier* to identify the entity of a selected identifier and therefore, if a refactoring is available for the given selection. The information about the kind of entity of an identifier can be found in the AST. This is the reason why the *SelectionIdentifier* class facilitates the *AstAnalyzerFactory* and its associated classes. About these classes will we talk in a later chapter.

8.2.3 The *RenameActionHandler* class

The *RenameActionHandler* class plugs together the classes, which are needed to set up the refactoring wizard. These are the *RenameInfo*, *RenameInputPage* and the *RenameProcessor* classes. It takes all the work from its subclasses, such that they just have to provide their specific *RenameProcessor* instance.

8.2.4 The *RenameInputPage* class

This class builds the representation of the window, which the user will use to enter a new name for a given entity. It is especially interesting, since it uses the *InputValidation* class to avoid renaming of identifiers to non C names. If the user enters an unappropriate name, proceed buttons will be disabled, until he corrects his input.

8.2.5 The *RenameProcessor* class

The *RenameProcessor* class extends the class with the same name from the Eclipse ltk library. It provides a lot of functions, which are reused in many subtypes. Besides this it enforces it subtypes to follow a little framework by means of abstract functions. Each subtype has to implement at least four functions:

1. *getProcessorName* with return type *String*
2. *initializeRefactoring* with return type *RefactoringStatus*
3. *checkConditionsAfterNameSetting* with return type *RefactoringStatus*
4. *createChange* with return type *Change*

The *getProcessorName* function is expected to return the name of the entity, which is renamed by this refactoring. This string is used only for user output.
The *initializeRefactoring* function is the first function of a sublcass which will be called. Here a subclass can gather all its information, to be sure, the refacoring is even possible or even has an effect. Experience shows, that this is actually the function which gathers all *Identifier* AST nodes, which are affected by the renaming. Errors in this function normally lead to adding an *FatalError* message to the returned *RefactoringStatus*, since the refactoring will not be able to do any reasonable thing. A *FatalError* message is shown to the user and leads to an abort of the

refactoring.
The *checkConditionsAfterNameSetting* function is the second function of a sublclass which will be called. It is called after the user entered a new name for the entity to be renamed. In this function a sub class can check if the new name is a reasonable choice. Which means that this is the place were you should check, if renaming would lead to name collisions in the program. Errors in this place are often not reported back as *FatalError* but just as *Error* messge instead. If we report just an *Error* message, then the user still has the choice to proceed. An *Error* message informs the user, that proceeding will change the semantics of the source, or will even lead to compile errors. But since the user possibly wants such a change in semantics by intention, it was false to not allow proceeding.
The *createChange* function is the last function, which will be called by the *RenameProcessor* framework. It is the place, where a subtype finally can create a subtype of *Change* class from the Eclipse ltk library. This object then should include all changes, which are necessary for the refactoring to fullfil its task.

Chapter 9

Harnessing the AST

The Yeti NesC parser generates an AST. This AST includes all information about the NesC source code on a per file base. Big parts of our program build upon analyzing these ASTs, in order to find information about the program, which is to be refactored. In early phases of the development, we used a lot of static code, to gather information out of the AST. This lead to a more imperative then object oriented design, with high coupling. Since this is an unpopular attribute for good software, we decided to refactor our refactoring plug-in. The result was the birth of the so called *AstAnalyzer* classes.

9.1 The *AstAnalyzer* classes

This classes are intended to wrap AST's of a whole file, or at least parts of an AST. The interaction with an *AstAnalyzer* is more of the kind:"Give me all objects which have the property...'. I.e. you can ask a *CAstAnalyzer* to give you all global C functions, which appear in its AST. Without the analyzers the interaction is more like:"I have an AST node, is this AST node of type A, and if so, is it a child of type B...". This means that in the whole program appeared code fragments, which included actually knowledge dedicated to the AST. This is pretty much the opposite of coherence and encapsulation.

9.1.1 The *AstAnalyzerFactory* class

Basically we can distinguish the types of AST's included in a source file. I.e. has a AST of a NesC interface a different structure, then the one of a NesC module. The *AstAnalyzerFactory* class takes an *ASTNode* or a source file and generates the appropriate *AstAnalyzer* type.
We than can ask the *AstAnalyzerFactory*, which type it has created and can then read the associated type out of the factory.

9.1.2 *AstAnalyzer* types

For the *AstAnalyzers*, which represent a AST of a whole file, we have defined an hierarchy of types. It is shown in the graphic 9.1 on page 29. Depending on the analyzer type we can get specific information. I.e. a *ModuleAstAnalyzer* will provide information about the Nesc code, which its module implementaion contains, while a *ConfigurationAstAnalyzer* holds data about the NesC wirings, which its configuration implementation includes.

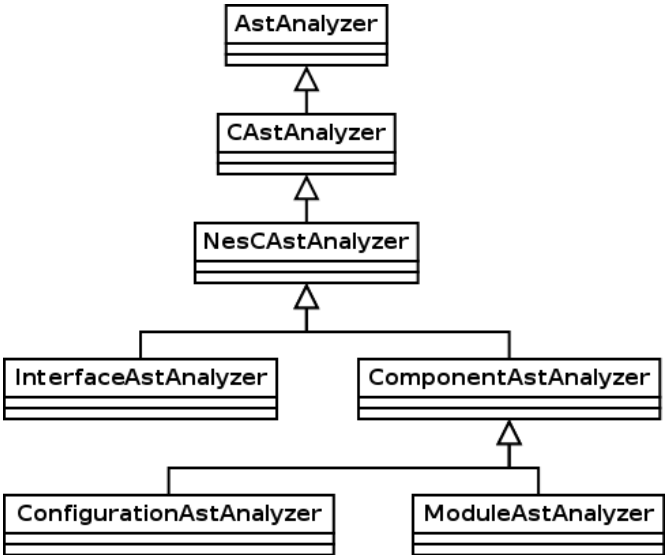


Figure 9.1: ASTs, which represent a whole NesC source file, are abstracted by the *AstAnalyzer* type hierarchy.

9.2 Utility classes

9.2.1 Auxiliary AST classes

As we already mentioned, our first approach was more kind of an imperative one. Because of time issues we were not able to totally get rid of the old design. Thats the reason why there exist still two classes, which are in the oldschool style.

1. *ASTUtil4Functions* class includes a lot of functions related to the AST structure of functions.

2. *ASTUtil4Variables* class includes a lot of functions related to the AST structure of variables.

If we had have some more time, we probably had designed something like a *FunctionAstAnalyzer* class. This class than had replaced the other two, which had been more likely a good object oriented design.
A very useful class dedicated to the AST, is the *ASTUtil* class. It includes a lot of convenience functions, which are useful everywhere, where we interact with the AST.
Last but everything else then least we have to mention the *AstPositioning* class. This class's main task is to find the corresponding AST element for a given character position in a source file. This task is complicated by the fact, that the position may come from a preprocessed file. The correct functioning of the class is especially mission critical for deciding the user selection.

9.2.2 Project wide classes

There is a number of further utility classes int the *utility* package.
The most important one is the *ProjectUtil* class. Refactorings which not only affect the file from which they were triggered, have to get access to other affected files. Therefore the *ProjectUtil* class provides exactly the right functions. I.e. you can find a NesC Module by specifying its

name, or you can search for references to a specific IASTModelPath in a certain file. A file is always the first step, in order to get to its AST.
The ProjectUtil also includes functions for logging messages to the project log and some other stuff, which is related to the whole project.
Another interesting class is the *ParserCache* class. As its name says, it tries to cache Parsers, so that not every file has to be parsed again and again, even if it was not modified, since it was parsed the last time. In our implementation the parser cache is only in use, if you obtain your parser from the *ProjectUtil* class.

Chapter 10

Concrete refactorings

10.1 Implemented refactorings

One of the main parts of our work was of course the implementation of concrete refactorings. We have implemented the ones, which we thought were the most useful:

1. Renaming of local variables
2. Renaming of function parameters
3. Renaming of variables in the implementation scope of a NesC Module
4. Renaming of global variables
5. Renaming of C functions in the implementation scope of a NesC Module
6. Renaming of global C functions
7. Renaming of NesC functions, like events and commands
8. Renaming of NesC interfaces
9. Renaming of NesC Components, like modules and configurations
10. Renaming of NesC component aliases
11. Renaming of NesC interface aliases
12. Introducing of NesC component aliases
13. Introducing of NesC interface aliases
14. Extracting of code parts to new C functions

As we can see, most of the refactorings are related to renaming something. In fact even the introducing of aliases is realised as a rename refactoring. This is the reason why we took a closer look at the classes in the *abstractrefactoring.rename* package. The implementation of refactorings, which use the infrastructure given in this package, is reduced to not more then four classes:

1. A *ActionHandler* class as subtype of *RenameActionHanlder*

2. A *AvailabilityTester* class as subtype of *RenameAvailabilityTester*
3. A *SelectionIdentifier* class as subtype of *SelectionIdentifier*
4. A *Processor* class as subtype of *RenameProcessor*

It is even simpler, since the *AvailabilityTester* and the *ActionHandler* classes are kind of connector classes to the *SelectionIdentifier* and *RenameProcessor*, respectively. This means that they have about three lines of code and do not much more than instantiating the other class.
The only refactoring, which we have implemented, which does not rely on the rename infrastructure, is the refactoring for extracting code parts to new C functions. The so called *Extract Function* refactoring. This is besides the complexity of it a second reason, why it has much more code.

10.2 What else could be done

If we had more time, there were of course a lot, which we could have done too. Everybody who can write code knows, that there is always something, which could be done nicer or more efficient. In short, there is always something, which you can refactor. I.e. as we already mentioned, it was nice to get rid of the *AstUtil4Functions* and *AstUtil4Variables* classes and instead implement something like a *FunctionAstAnalyzer* class, to reach a nicer, more object oriented design style. With more time we also could have implemented some more refactorings, i.e.:

1. Renaming of global C typedefs
2. Renaming of C typedefs in the implementation scope of a NesC Module
3. Renaming of C enums, as well in the global as in the implementation scope
4. Renaming of C enum constants
5. Renaming of C structs, as well in global as in the implementation scope
6. Renaming of NesC tasks
7. Renaming of C preprocessor macros
8. More sophisticated extract function refactoring, which allows user to select in and out parameters
9. Magic number refactoring, which converts magic numbers to constants

And we are sure, there are a lot more of possibilities.

10.3 How to implement a new refactoring

If you intend to implement a new refactoring, this section should give you an overview, which steps you have to follow, and where in this document you can find more information to a specific topic.

10.3.1 Enable your refactoring

First you have to add new, appropriate elements in the *plugin.xml* for your refactoring. The fastest way to do this, is to copy and paste existing elements, and adapt their content. You have to add the following elements:

1. A *command* element in the *commands extension* element, important is the *id* attribute, since it is referenced in all other elements
2. A *handler* element in the *handlers extension* element, important is the *class* attribute, which references the handler implementation
3. A *command* element in the *menu menuContribution* element under the menus *extension* element, important is the *label* attribute, which is the name shown to the user and the *property* attribute, since it identifies the refactoring in the property tester
4. A *command* element in the *popup menuContribution* element under the menus *extension* element, important is the *label* attribute, which is the name shown to the user and the *property* attribute, since it identifies the refactoring in the property tester

Finally you have to add the *property* string, which you have specified in the *command* elements, which you added to the *menuContribution* elements, to the *properties* attribute in the *propertyTester extensinon* element.
This is all you have to do with the *plugin.xml*, in order to introduce your new refactoring. If this information is not enough for you, maybe you find some answers in chapter 5 on page 13.

Now you have to create a subtype of *IRefactoringAvailabilityTester*. It has to test, if the user selection is appropriate for your refactoring. The last step in order to enable your new refactoring, is to define a new enum constant in the *Refactoring* enum. Make sure that its *propertyName* field matches the *property* name you chose in the *plugin.xml* and set your new *IRefactoringAvailabilityTester* as its *tester* field.
If you have done all these steps, you should now see your refactoring in the Eclipse user interface, if the user selection matches your definitions. If you need more information about these two steps, consult chapter 7 on page 22.

10.3.2 Make your refactoring do its work

You should now be able to select your refactoring in the Eclipse user interface. To make the selection do your refactoring, you have to do the following steps. First you have to implement a subtype of the Eclipse *AbstractHandler* class. Make sure that the class name matches the one given in the *plugin.xml* in the *handler* element. This class is responsible for initiating your refactoring. You should do this by facilitating the language toolkit for processor based refactoring. This means that you have to create subtypes of the following classes:

1. *DefaultRefactoringWizard* class of us, responsible for consolidating the following classes
2. *RefactoringInfo* class of us, carries information needed by the refactoring and the wizard itself
3. *UserInputWizardPage* of Eclipse, the window, where the user can do its input
4. *RefactoringPorcessor* of Eclipse, the place where the changes actually are generated

To find out more about this classes, read chapter 4 on page 9. If you implemented and wired them correctly, your refactoring should now be able to do its work.
If you intend to write a rename refactoring, you can extend the classes from the *abstractrefactoring.rename* package. If you do so, your only real concern is to write a subtype of the *RenameProcessor* class. In this case you should take a look at chapter 8 on page 25.
For writing a processor you should consider to reuse the classes explained in chapter 9 on page 28.
You find the description of example implementation in the next chapter.
If all this information is not enough, you can read the source code, which is still the most accurate documentation.

Chapter 11

An example Implementation: The rename interface refactoring

In this section we talk about the implementation of the *Rename Interface* refactoring. As its name suggest, the target of the refactoring is to rename a NesC interface. Since the refactoring is about renaming, it makes heavy usage of the infrastructure in the *abstractrefactoring.rename* package. This package is described in chapter 8 on page 25.

11.1 Until the processor starts its work

We assume that the developer is working in a NesC Project and has saved all his Editors. He wants to rename one of his Interfaces. To do that he double-clicks on the old name of the interface.

Eclipse marks it and fires a *Selection Changed* event. This triggers, among other things, the checks which entries are to be displayed in the refactoring menu.[1] This means that Eclipse has to call the *RefactoringsAvailabilityTester* property tester for each possible menu entry with the according property. When Eclipse comes to the *renameInterface* property, the global RefactoringsAvailabilityTester will call the *test()* method of the rename interface *AvailabilityTester*. This method will find the identifier the user marked and pass it to the *InterfaceSelectionIdentifier* class. This class checks if the identifier represents an interface. And yes, it does. The *AvailabilityTester* returns *true* and Eclipse knows, that the rename interface refactoring has to be part of the refactoring menu if the user opens it.

All this checks were done so fast that the user did not notice anything. He just selected an interface name. He opens the refactoring menu and Eclipse already knows what entries it has to show. The user clicks on *Rename Interface*.

Eclipse now creates an Object of type *ActionHandler*, defined in the *plugin.xml handler* tag[2] and calls the *execute()* method. The *execute* method initializes all the necessary LTK classes[3] and the appropriate rename interface classes. Among them the *Processor* and the *Wizard*. Then it starts LTK's *RefactoringWizardOpenOperation*.[4] It checks the initial conditions and shows the *RenameInputPage* to the user.

[1] Defined as described in 5.2 on page 16.
[2] As described in 5.1.4 on page 15.
[3] Described in 4.1 on page 9.
[4] Well illustrated by the interaction diagram4.3 on page 11.

The user who just clicked on *Rename Interface*, can now enter the new name he wants the Interface to have. While he does that, Eclipse check whether he enters a valid C identifier and allows him to click preview or OK, only if a possible string was written. Because the user entered a name, which generates no problems, he is allowed to click on preview. Eclipse checks the final conditions and calls the *createChange* method of the processor. The user can preview the changes and accepts them by clicking on OK.

11.2 The processor

Now we come to the point, where the real work happens. The class *tinyos.yeti.refactoring.entities.interfaces.rename.Pr* has to do three steps, to successfully rename an interface:

1. Find all identifiers affected by the renaming, most probably in several files.
2. Check that the new name does not collide with an existing one.
3. If there are no collisions, create the changes.

The processor extends the *tinyos.yeti.refactoring.abstractrefactorings.rename.RenameProcessor* class. It is explained in detail in chapter 8.2.5 on page 26. Therefore the processor implements the mini framework we mentioned in this chapter. The three framework functions perfectly match on our three steps:

1. Find all identifiers realised in function *initializeRefactoring*
2. Collision detection realised in function *checkConditionsAfterNameSetting*
3. Change creation realised in function *createChange*

We will now go on with explaining the actions taken in the different functions.

11.2.1 Find all identifiers affected by the renaming

The functionality described here is implemented in the *initializeRefactoring* function of our mini framework. In order to find all identifiers affected by the renaming, we do the following:

1. Find the source file, which defines the interface.
2. Check all project files for references to the interface definition.
3. Convert the found references to identifiers.
4. Check if the identifiers need to be renamed.
5. pack all identifiers in a map, mapped by the file in which they appear.

To find the source file, we are in the lucky position, that the selected identifier has to have the name of the Interface, which is to be renamed. A special case is it, when the selection is actually an interface alias, but for our example implementation, this case is not relevant. With the name of the interface to be renamed in our hands, we can use the class *ProjectUtil*. This class makes use of the facilities provided by the Yeti plug-in. It is able to get a *IDeclaration* class for the interface name out of the *ProjectModel* class. The declaration is actually an abstraction of the interface source file. At this point we have fulfilled our first step for the renaming.
In order to find all references to the interface, we proceed as follows. The declaration we found in

the first step has an *IASTModelPath*. We can use this path to gather all references of a specific file to the path. This is done via the Yeti *ProjectModel* again.
Now we come to the third step. We have now the references to the interface, but we are not able to rename based on the references. The reason is, that we have to do checks, if the identifiers behind the references are actually identifiers of interest, see the next step therefore. The references can be converted to *Identifer* AST nodes, which are also defined by Yeti. This conversion is done by means of our *AstPositioning* class. It takes an offset in a source file and gives you in turn the AST node at this position. You then can check if it is actually of the expected type, and your done with this step.
You have now the identifiers and the containing files. Unfortunately not all of this identifiers need to be renamed. First there is the possibility, that a reference spans more than one Identifier. Normally not both of them reference an interface and most probably not both the interface we are looking for. Second, also interface aliases reference the defining interface, but we are not interested in renaming aliases. This means that we have to filter our bunch of identifiers. To fulfill this task we use our *InterfaceSelectionIdentifier* class (chap. 8.2.2, page 26), which makes heavy use of the *AstAnalyzerClasses*(chap. 9.1, page 28).
Now we have all identifiers we are interested in according with the files, in which they appear. We then put this identifiers in a map, mapped by the file, to have access to it in the remaining two steps of our mini framework.
If there is a problem during this steps, this can be reported back to the user by adding a message to the returned value of type *RefactoringStatus*. An error at this stage will be reported as fatal error, which means that the user is not able to proceed with the refactoring, but has to abort it. The reason is, that an error at this stage probably means, that we did not find all affected identifiers, which will lead to compile errors, when we proceed.

11.2.2 Check for collisions

The functionality described here is implemented in the *checkConditionsAfterNameSetting* function of our mini framework. This function is called, after the user entered a valid new name and pressed the OK button. Its target is to find any possible collision with an existing name in the source files. First we have to check, if there is not already a project file with the given name. If there is one, we report this back to the user by adding a error message to the return value. The error message informs the user about the collision. The use of a error message instead of a fatal error, will allow the user to go back to the name input field, to choose another name. For each file, which is affected by the renaming, we make use of the so called *NesCComponentNameCollissionDetector* class. The collision detector in turn uses an appropriate *AstAnalyzer* type, to investigate the source files.
If the affected source file is a NesC module, then we just have to check, if there is a collision with an interface alias. But if we are talking about an NesC configuration, also the implementation has to be checked, if there are any component aliases, with the given name. If there is any problem, it will be reported with an error message. An error message especially allows the user to proceed, in the case he intended the collision for some special tweak.

11.2.3 Create the changes

When the *createchange()* method is called, a *CompositChange* is created. Then the method *addChange()* loops over all the identifiers in the *affectedIdentifiers* map and creates a *TextFileChage* to change the old name to the new name.

A special thing in NesC is that the interface file has the same name as the interface by

convention. That's why also a *RenameResourceChange* is added to rename the interface file to the new interface name. In the end the whole *CompositChange* is returned to the LTK Framework.

Glossary

AST AST stands for abstract syntax tree. The yeti NesC parser generates for each source file such an AST. It is a first abstraction of the source code, based on the NesC syntax.

Eclipse Is a so called integrated development environment engineered by IBM. An IDE facilitates program code writing.

NesC NesC (Network embedded systems C) is a component-based, event-driven programming language used to build applications for the TinyOS platform. TinyOS is an operating environment designed to run on embedded devices used in distributed Wireless Sensor Networks. nesC is built as an extension to the C programming language with components wired together to run applications on TinyOS.

OSGi The OSGi framework is a module system and service platform for the Java programming language that implements a complete and dynamic component model, something that does not exist in standalone Java/VM environments. Applications or components (coming in the form of bundles for deployment) can be remotely installed, started, stopped, updated and uninstalled without requiring a reboot; management of Java packages/-classes is specified in great detail. Life cycle management is done via APIs which allow for remote downloading of management policies. The service registry allows bundles to detect the addition of new services, or the removal of services, and adapt accordingly. (from Wikipdedia)

Yeti The TinyOS 2.x Plug-in for Eclipse, nicknamed Yeti 2, was developed by the Distributed Computing Group at ETH Zurich. The plug-in aims to provide developers with all the convenience functions expected from a modern development environment.