Sadnata04 Tools Documentation

EFence:

Classes:

EFencePropertyPage.java

Overview: This class should give the user the option to set the flags for the runtime part of the "Compile & Run" command. As such it is a GUI, and it extends the PropertyPage class. Its extension point (in the .xml file) is the property pages.
The class should give the user a way to express his wishes for the runtime part. This includes *controllers* such as radio buttons, check boxes and text fields. These are declared in the file under "controllers".

After setting the options, the user should be able to store and read these setting from the storing area. This is achieved by using the setPersistentProperty / getPersistentProperty methods.

Also on startup, and with pressing of every button, appropriate functions should be called to correspond to the events arising by the java mechanism.

Methods:

Constructor: Default (super).

createContents(): This function creates a new *composite* and *grid*, sets their parameters, then it calls the function "addEfenceSection()" that draws the controllers on the specific composite and grid. (More on that function later). Then it checks to see if this is the first time the user has used this property page. If it is, it should use the default values, thus calling the "performDefaults()" method. Then it reads the properties stored (default if first time, else if not) and presents them on screen.

Void performDefaults(): This function sets the default parameters in case this is the first time the user has used this specific property page (instance). Then it calls "storeProps()" in order to cache the decisions to disk.

Boolean readEFenceRadios(): This function reads the state of the radio buttons and transfers it to memory, while presenting the right status on the screen. Returns true if successful.

Boolean storeEFenceRadios(): This function stores the state of the radio buttons and transfer it to the disk. Returns true if successful.

Void readButton(Button btn, QualifiedName id): This function reads the state of the button represented in the repository by "id" and updates the button btn in the GUI.

Void readString(Text t, QualifiedName id): This function does the same, only with texts.

Void storeButton(Button btn, QualifiedName id): This function stores the state of the button btn in the repository.

Void storeString(Text t, QualifiedName id): This function stores strings.

Boolean readProps(): This function reads the properties from the repository and sets the controllers to the state they should be. Returns true if successful.

Boolean storeProps(): This function stores the properties in the repository.

Public Boolean performOk(): When the user presses the "OK" button, we should save his preferences.

Boolean isFirstTime(): Checks if it is the first time the user has opened this property page. Returns true if it is, false if it isn't the first time. It works by checking a special place in the repository - has anyone wrote to it yet?

CompileSingleFile.java

Overview: This class should compile and run a single file. As such it should have access to the repository. That is why we defined the QualifiedNames. Generally speaking, the mechanism is as follows: First we compile the file using gcc, and linking it with efence. After that we're building a script for the execution part. The script is named after the file with the extension of .efscript. This is preferred because the user may edit this script any way he wishes. Afterwards we give execution permission for the script and run it (chmod +x). The output should resolve to the debug console.

Note: This class and the "CompileProject.java" class are very similar. The only difference will be discussed in the next section.

Void setContents(File aFile, String aContents): This function writes a string of characters (aContents) to the file aFile). It is used to build the script.

Void createScript(): This function creates the script read from the repository.

Void createDefaultScript(): This function creates the script corresponding the default flags. It is true that in the runtime of this class the defaults are already in memory, but since the defaults never change it is more efficient to just write them straightforward.

Void runScript(): This function starts the runtime part. First it compiles the file. Then it tries to give the script executing permissions. If it fails it lets the user know. Finally it runs it.

Void selectionChanged(IAction action, ISelection selection): This function's job is to keep track on the file that the user selected from the navigator view.

CompileProject.java

Overview: This class is very similar to “CompileSingleFile.java”. The only difference is the “getFiles()” procedure, that returns a string consisting of all the files of the project. The files are arranged in a spaced array. After getting the files, the function gccs all of the files in a single command line.

Adding the Kcachegrind Plug-In to the Eclipse

In our workshop we added a new plug-In to the IBM's Eclipse software.
We added a profiling tool **Calltree** and the profile data visualization **Kcachegrind**.
Calltree uses runtime instrumentation by the use of Valgrind framework for its cache simulation and call tracing.
We added this Plug-In by using the plug-In tools of the Eclipse.
At first we worked with the Eclipse's wizards and then we started digging in to the basic code in order to add all the necessary features for our tool.
We learned all about the Eclipse by reading its on-line documentation and also we had a glance in the Eclipse book.
We have also found some more information in the relevant sites in the net, here is a list of some of them:
http://eclipse.org/
http://eclipse-plugins.2y.net/eclipse/index.jsp
http://www.myeclipseide.com/index.php
http://www.eclipseplugincentral.com/displayarticle187.html

Creating the Plug-In:

In order to create the plug-In we created the following java files:

- KcacheAction.java
- SamplePropertyPage.java
- KcachePlugin.java

And the following classes:

- KcacheAction.class
- SamplePropertyPage.class
- KcachePlugin.class

And also the following xml file:

- plugin.xml

The file content:

- **SamplePropertyPage.java:**

In this file we inserted the code that will enable the user to add the paths of his Valgrind framework and the Kcachegrind software, so the plug-in actions could load that software. We also add some of the Calltree options to the property page so the user can choose the way he wants his file/project profiled.

The Kcachegrind property page includes the following options:

File paths:

- Please Enter Full Path For Kcachegrind:
- Please Enter Full Path For Valgrind:
- Please Enter Full Path For The Executable:

Calltree options (string field):

- --dumps=

--dumps=<count> Dump trace each <count> BBs [0=never]

- --fn-skip=

--fn-skip=<function> Ignore calls to/from function?

- --dump-after=

--dump-after=<func> Dump when leaving function

- More Flags You Wish To Add:

Calltree options (checkbox):

- Trace-jump

Trace (conditional) jumps in functions?

- Collect-state

Start with collecting events

The main functions in this file, that we worked on are:

createContents(Composite parent)

The main function that creates the property page layout

addFirstSection(Composite parent)

Adds the 3 text fields for the valgrind, kcachgrind and the executable pathes.

addSecondSection(Composite parent)

Adds the text fields and the check boxes for the valgrind flags

performDefaults()

Sets the defaults values for the above fields

And other functions, witch are described in the EFence files documentation (readButton, readString, storeButton, storeString, readProps(),storeProps(),isFirstTime(),performOk())

- **KcacheAction.java:**

In this file we added the necessary code for running Valgrind with the Calltree skin.
We took the data that was inserted in the property page and run the software accordingly.
If the user still hasn't entered the property page the plug-in will give him a message that he should fill in the property page before running the action.
If the user didn't insert the correct data to the property page the software won't run.
The action combined from two run (execv) commands. One for the Valgrind with the Calltree skin and with the relevant options that the user had chosen in the property page and one for the Kcachegrind with the output file that was created by the Calltree.
The run execution structure is as following:

☒ Valgrind with Calltree skin:

<VALGRIND_PATH>" --skin=calltree --logfile="<PROJECT_NAME>"."<INDEX> <CMD> <EXCUTABLE_PATH>

☒ KCachegrind with the Calltree output file:

<KCACHGRING_PATH>" –r " <PROJECT_PATH>"/"<PROJECT_NAME> "." <INDEX>".pid*"

In order to locate the suitable output file we created a persist property that will change for every run and we added this property to the output file name we could locate the specific output file each time although we don't get it's process number (of the Calltree run).

The main functions in this file that we worked on are:

Run()

The main function in this action, it is called when the user clicks on the action button .
In this function we build the command lines for calling valgrind and then kcachegrind by getting the project properties that the user set in the property page.

isFirstTime()

Checks whether this is the first we profile this project. It is used to set a "file index" to keep previous profile data outputs, in a logical order.

selectionChanged(IAction action, ISelection ss)

this function is called whenever the user chose an element with the mouse. And then if the chosen element is a a file, folder or a project, it sets the global variable "project" to the chosen project.

- **plugin.xml:**

In this file all the plug-in extensions are defined and linked to the relevant java file. Also the name of the actions and property page is defined in this file.

We used the following extensions points:

org.eclipse.ui.propertyPages

org.eclipse.ui.actionSets

org.eclipse.ui.perspectiveExtensions

How to use the Kcachegrind plug-in:

- Make sure you have the Valgrind and the Kcachegrind software on your computer. If you don't have it you can install it from the following web sites:
http://valgrind.kde.org/downloads.html
http://kcachegrind.sourceforge.net/cgi-bin/show.cgi/KcacheGrindDownload
Also make sure that your working environment is KDE 3.0.
- Add the plug-in to the Eclipse, you can download it from our site and install it according to the installation guide.
- After the plug-in was installed you should choose the project property. There you'll see the Kcachegrind property page.
- In this page you should insert the full path of the Valgrind and the Kcachegrind software, and also the full path of your project executable.
- You can also insert the relevant data to the options of the Calltree and you can also add some more options to the Calltree.
- After filling the property page you should enter the project menu and click on the Memory Tools sub menu. In this sub menu press on the Run Kcachegrind option.
- The Kcachegrind shall open with the profiling data of your project.