EasyInterface User Manual

http://github.com/abstools/easyinterface

Jesus Doménech
Samir Genaim

http://www.envisage-project.eu/

Preface

How to Read this User Manual

Start with Chapter 1 in order to understand the overall architecture of the EasyInterface framework and the role of each component. Next read Chapter 2 and implement all the steps of the incremental example, after which you will probably have enough knowledge to integrate your own applications without further reading.

Contents

1 Overview of the EasyInterface Framework **4**
- 1.1 The Architecture of EasyInterface . . . 4
- 1.2 The Server Side . . . 4
- 1.3 The Client Side . . . 5

2 Quick Guide to EasyInterface **7**
- 2.1 Add Your First Application to the EasyInterface Server . . . 7
- 2.2 Passing Input Files to Your Application . . . 9
- 2.3 Passing Outline Entities to Your Application . . . 10
- 2.4 Passing Parameters to Your Application . . . 10
- 2.5 Using the EasyInterface Output Language in Your Application . . . 13
  - 2.5.1 Printing in the Console Area . . . 13
  - 2.5.2 Adding Markers . . . 14
  - 2.5.3 Highlighting Code Lines . . . 15
  - 2.5.4 Adding Inline Markers . . . 16
  - 2.5.5 Opening a Dialog Box . . . 16
  - 2.5.6 Adding Code Line Actions . . . 17
  - 2.5.7 Adding OnClick Actions . . . 18

3 EasyInterface Server **19**
- 3.1 Configuring the EasyInterface Server . . . 19
  - 3.1.1 Name and Path of the Configuration File . . . 19
  - 3.1.2 The Syntax of the Configuration File . . . 19
- 3.2 Communicating with EasyInterface Server . . . 28

4 EasyInterface Clients **29**
- 4.1 Web-Client . . . 29
  - 4.1.1 Generate Outline . . . 29
- 4.2 Eclipse Plugin . . . 29
- 4.3 Remore shell . . . 29

5 The EasyInterface Output Language **31**
- 5.1 Brief description . . . 31
- 5.2 Details XML output . . . 31

1 | Overview of the EasyInterface Framework

The EasyInterface framework provides a simple way to build interfaces, e.g., a web-interface or an Eclipse plugin, for tools written in (almost) any programming language. Moreover, it does not require the programmer to be familiar with any GUI library or web programming. Roughly, the only requirement is that the application can be executed from a command-line and that its output goes to the standard output.

The goal of EasyInterface is to provide developers with a toolkit to *build their applications once and get several interfaces for free*. EasyInterface was originally developed for building a common frontend for program analysis tools developed in the Envisage[1] project. This is why, as the reader will notice later, its graphical user interfaces are basically developing environments that allow editing programs, etc.

In the rest of this chapter we overview the different components of EasyInterface, and explain how they are combined to achieve the above goal.

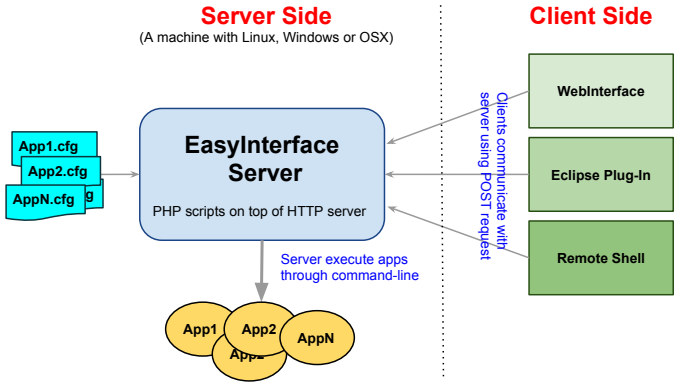

Figure 1.1: The Architecture of the EasyInterface Framework

1.1 The Architecture of EasyInterface

The architecture of EasyInterface is depicted in Figure 1.1. It includes two main components: (1) *server side*: a machine with several applications (the circles `App1`, `App2`, etc., in Figure 1.1) that can be executed from a command-line and their output goes to the standard output. These are the applications that we want to make available for the outside world, i.e., execute them as services on the internet; and (2) *client side*: several clients that make it easy to communicate with the server side to execute an application, etc. In what follows, we first explain the inner components of the server side and which problems they solve, and then we explain the client side.

1.2 The Server Side

The problem that we want to solve at the server side is:

> Provide a uniform way for remotely accessing locally installed applications as services.

This problem is solved by the EasyInterface server, which is collection of PHP programs that run on top of an HTTP server. This server allows specifying how a local application can be executed and which parameters it takes using simple configuration files (`App1.cfg`, `App2.cfg`, etc., Figure 1.1). For example, the following is a snippet of such configuration file:

[1] http://www.envisage-project.eu

```
<app id="myapp" visible="true">
  ...
  <execinfo method="cmdline">
    <cmdlineapp>/path-to/myapp.sh _ei_parameters</cmdlineapp>
  </execinfo>
  <parameters prefix = "-" check="false">
    ...
    <selectone name="c">
      <option value="1" />
      <option value="2" />
    </selectone>
  </parameters>
</app>
```

This XML defines an application that has a unique identifier **myapp**. The cmdlineapp tag is a template that describes how to execute the application from a command-line. Here **_ei_parameters** is a template parameter that will be replaced by an appropriate value. The parameters tag includes a list of parameters accepted by the application. For example, there is a parameter called "c" that can take one of the values 1 or 2. Once the configuration file is installed on the EasyInterface server, anyone can access the application using an HTTP POST request that includes the following text:

```
{
  command:  "execute",
  app_id:  "myapp",
  parameters: {
     c:  ["1"],
     ...
  },
  ...
}
```

When the EasyInterface server receives such a request, it generates a corresponding command-line (according to what is specified in the configuration file), executes it, and redirect the standard output back to the client.

1.3 The Client Side

Although we now have a relatively easy way to execute applications on the server side, it is still not as easy as we aimed at. Our aim is to simplify this process further by providing (graphical) user interfaces that automatically (1) connect to the EasyInterface server and ask for the list of available applications; (2) let the user choose an application to execute and set the values of the corresponding parameters; (3) generate a corresponding request and send it to the EasyInterface server; and (4) shows the returned output to the user. The EasyInterface framework provides three such interfaces: a *web-interface* that can be executed in a browser and looks like a developing environment (see Figure 4.1 on Page 30); an Eclipse-plugin that runs within the Eclipse IDE; and a remote-shell that can be used from a command-line.

Since the web-client and the Eclipse plugin are GUI based developing environments, EasyInterface provide also, to an application, the possibility to generate output that has some graphical effects, e.g., open dialog-boxes, highlight code lines, add markers, etc. To use this feature, the applications should be modified to use the EasyInterface output language. The following is a snippet of such output:

```
<highlightlines dest="/Examples_1/iterative/sum.s">
  <lines> <line from="5" to="10"/> </lines>
</highlightlines>
...
<oncodelineclick dest="/Examples_1/iterative/sum.c" outclass="info" >
  <lines><line from="17" /></lines>
```

```
  <eicommands>
    <dialogbox boxtitle="Hey!">
      <content format="text">
        Click on the marker again to close this window
      </content>
    </dialogbox>
  </eicommands>
</oncodelineclick>
```

The **highlightlines** indicates that lines 5–10 of the file `/Examples_1/iterative/sum.s` (which is opened in the editor) should be highlighted. The **oncodelineclick** tag indicates that when clicking on line 17, a dialog-box with a corresponding message should be opened. Note that the application is only modified once to produce such output, and will have similar effect in all interfaces that support this output language.

2 | Quick Guide to EasyInterface

This chapter provides a quick introduction on *how to integrate an application in the* EasyInterface *framework*. In particular, we develop a simple application, integrate it in the EasyInterface server, and try it out through the web-client. The presentation is incremental, we start with a simple application and in each step we add more features to demonstrate the different parts of the EasyInterface framework. In our explanation we assume that a Unix based operating system is used, however, we comment on how to do the analog operations on Windows when they are different. Note that in this chapter we only use the web-client, for other clients refer to Chapter 4. We assume that EasyInterface is already installed and working, which can be done following the instructions available in `INSTALL.md`.

Let us start by trying some demo applications that are available by default in the web-client. If you visit `http://localhost/ei/clients/web`, you should get a page similar to the one shown in Figure 4.1 on Page 30. At the top part of this page you can see a button with the label `Apply`, and to its right a combo-box with several items `Test-0`, `Test-1`, etc. These items correspond to applications available in the web-client, and we will refer to it as the applications menu. To the left of `Apply` there is a button with the label `Settings`, if you click it you will see that each `Test-i` has also some parameters that can be set to some values. Note that, by default, the web-client is configured to connect to the EasyInterface server at `http://localhost/ei/server` and ask for all applications, together withe their corresponding parameters, that are available at that server. Note also that application `Test-i` actually corresponds to the bash-script `server/bin/default/test-i.sh`, and that its configuration file is `server/config/default/test-i.cfg` (later you will understand the details of such configuration files).

If you select an application, from the combo-box, and click on `Apply`, the web-client sends a request to the server to execute this application. The request includes also the current values of the parameters (those in the settings section) and the file that is currently active in the code editor area. The server, in turn, executes the corresponding program, i.e., the bash-script `server/bin/default/test-i.sh` in this case, and redirects its output back to the web-client. The web-client will either print this output in the console area, or view it graphically if it uses the EasyInterface output language. Execute the demo applications just to get an idea on which graphical output we are talking about (e.g., highlight text, markers). In the rest of this chapter we explain, step by step, how to add your own application to EasyInterface.

2.1 Add Your First Application to the EasyInterface Server

When we add an application to the EasyInterface server it will appear automatically in the applications menu of the web-client (unless you have changed the configuration of the web-client already!). Let us add a simple "Hello World" application.

We start by creating a bash-script that represents the executable of our application (it could be any other executable). We will place this bash-script in the directory `server/bin/default` together with the `test-i.sh` scripts, however, this is not obligatory and it can be placed anywhere in the file system as far as the HTTP server can access it. Create a file `myapp.sh` in `server/bin/default` with the following content:

```
1 #!/bin/bash
2
3 echo "Hello World!"
```

As you can see, it is a simple program that prints "Hello World" on the standard output. Later we will see how to pass input to this application and how to generate more sophisticated output. Change the permissions of `myapp.sh` by executing the following (on Windows this is typically not needed):

```
> chmod -R 755 myapp.sh
```

Execute `myapp.sh` (in a shell) to make sure that it works correctly before proceeding to the next step.

Next we will configure the server to recognize our application. Create a file `myapp.cfg` in the directory `server/config/default` with the following content (we could place this file anywhere under `server/config` not necessarily in `default`):

```
<app visible="true">
  <appinfo>
    <acronym>MFA</acronym>
    <title>My First Application</title>
    <desc>
      <short>A simple EI application</short>
      <long>A simple application that I've done using the EasyInterface Framework</long>
    </desc>
  </appinfo>
  <apphelp>
    <content format='html'>
      This is my first <b>EasyInterface</b> application!
    </content>
  </apphelp>
  <execinfo>
    <cmdlineapp>./default/myapp.sh</cmdlineapp> %   are used to escape
  </execinfo>
</app>
```

Let us explain the meaning of the different elements of this configuration file. The **app** tag is used to declare an EasyInterface application, and its **visible** attribute tells the server to list this application when someone asks for the list of installed applications. Changing this value to **"false"** will make the application "hidden" so only those who know its identifier can use it. The **appinfo** tag provides general information about the application, this will be used by the clients to show the application name, etc. The **apphelp** tag provides some usage information about the application, or simply provide a link to another page where such information can be found. The actual content goes inside the **content** tag, which is HTML as indicated by the **format** attribute (use 'text' for plain text). The most important part is the **execinfo** tag, which provides information on how to execute the application. The text inside **cmdlineapp** is interpreted as a command-line *template*, such that when the server is requested to execute the corresponding application it will simply execute this command-line and redirect its output back to the client. Later you will understand why we call it *template*. Note that before executing the script, the server changes the current directory to `server/bin` and thus the command-line should be relative to `server/bin`.

Next we add the above configuration file to the server. This is done by adding the following line to `server/config/default/apps.cfg` (inside the **apps** tag):

```
<app id="myapp" src="default/myapp.cfg" />
```

Here we tell the server that we want to install an application as defined in `myapp.cfg`, and we want to assign it the *unique* identifier **"myapp"**. This identifier will be mainly used by the server and the clients when they communicate, we are not going to use it anywhere else. Note that in `default/apps.cfg` we used **"default/myapp.cfg"** and not **"myapp.cfg"**. This is because the server looks for configuration files starting in `server/config`. Note also that the main configuration file of the EasyInterface server is `server/config/eiserver.default.cfg`, and that `default/apps.cfg` is imported into that file (open `server/config/eiserver.default.cfg` to see this).

Let us test our application. Go back to the web-client and reload the page, you should see a new application named `MFA` in the applications menu. If you click on the `Help` button you will see the text provided inside the **apphelp** tag above. Select this application and click on the `Apply` button, the message "Hello World!" will printed in the console area.

2.2 Passing Input Files to Your Application

Applications typically receive input files (e.g., programs) to process. You must have noticed that the web-client provides the possibility of creating and editing such files. In this section we explain how to pass these files, via the server, to our application when the `Apply` button is clicked.

When you click on the `Apply` button the web-client passes the currently opened file (i.e., the content of the active tab) to the server, and if you use the `Apply` option from the context menu of the file-manager (select an element from the files tree on the left, and use the mouse right-click to open the context menu) it passes all files in the corresponding sub-tree. What is left is to tell the server how to pass these files to our application. Let us assume that `myapp.sh` is prepared to receive input files as follows:

```
> myapp.sh -f file1.c file2.c file3.c
```

In order to tell the server to pass the input files (that were received from the client) to `myapp.sh`, open `myapp.cfg` and change the command-line template to the following:

```
<cmdlineapp>./bin/default/myapp.sh -f _ei_files</cmdlineapp>
```

When the server receives the files from the client, it stores them in a temporary directory, e.g., in `/tmp`, replaces `_ei_files` by the list of their names, and then execute the resulting command-line (now you see why we call it command-line template). It is important to note that only `_ei_files` changes in the above template, the rest remain the same. Thus, the parameter "`-f`" means nothing to the server, we could replace it by anything else or even completely remove it — that depends only on how our application is programmed to receive input files.

Let us now change `myapp.sh` to process the received files in some way, .e.g., to print the number of lines in each file. For this, replace the content of `myapp.sh` by the following:

```
1  #!/bin/bash
2
3  . default/parse_params.sh
4
5  echo "I've received the following command-line parameters:"
6  echo ""
7  echo "  $@"
8
9  echo ""
10 echo "File statistics:"
11 echo ""
12 for f in $files
13 do
14    echo " - $f has " `wc -l $f | awk '{print $1}'` "lines"
15 done
```

Let us explain the above code. At line 3 we executes an external bash-script to parse the command-line parameters, the details are not important and all you should know is that it leaves the list of files (that appear after `-f`) in the variable `files`. Lines 5-7 print the command-line parameters, just to give you an idea how the server called `myapp.sh`, and the loop at lines 12-15 traverses the list of files and prints the number of lines in each one.

Let us test our application. First run `myapp.sh` from a shell passing it some existing text files, just to check that it works correctly. Then go back to the web-client, reload the page, select `MFA` from the applications menu, open a file from the file-manager, and finally click the `Apply` button. Alternatively, you can also select an entry from the file-manager and choose `Apply` from its context menu. You should see the output of the application in the console area.

2.3 Passing Outline Entities to Your Application

In the web-client, the area on the right is called the outline area (see Figure 4.1 on Page 30). Since EasyInterface was designed mainly for applications that process programs, e.g., program analysis tools, this area is typically dedicated for a tree-view of program entities, e.g., method names, class names, etc. The idea is that, in addition to the input files, the user will select some of these entities to indicate, for example, where the analysis should start from or which parts of the program to analyze. Next we explain how we can pass these selected entities to an application.

By default the web-client is configured to work with C programs, and thus if you open such a program (from the file-manager) and then click on the `Refresh Outline` button, you will get a tree-view of this program entities, e.g., method names (if you use `Refresh Outline` from the context menu in the file-manager you will get a tree-view of program entities for all files in the sub-tree). Note that to generate this tree-view the web-client actually executes a "hidden" application that is installed on the server, namely `server/bin/default/coutline.sh`, but this is not relevant to our discussion now (see Section 4.1.1 for more details). Note also that `coutline.sh` is limited and will not works perfectly for any C program: it simply looks for lines that start with `int` or `void` followed by something of the form `name(...)`. This script is provided just to give an idea on how an application that generates an outline is connected to the web-client (see Section 4.1.1 for more details).

As in the case of input files, the web-client always passes the selected entities to the server when the `Apply` button is clicked, and it is our responsibility to indicate how these entities should be passed to our application. Let us assume that `myapp.sh` is prepared to receive entities using a "`-e`" parameter as follows:

```
> myapp.sh -f file1.c file2.c file3.c -e sum.c:main sum.c:sum
```

In order to tell the server to pass the entities (that were received from the client) to our application, open `myapp.cfg` and change the command-line template to the following:

```
<cmdlineapp>./bin/myapp.sh -f _ei_files -e _ei_outline</cmdlineapp>
```

As in the case of files, before executing the above command-line the server will replace `_ei_outline` by the list of received entities. Let us now change `myapp.sh` to process these entities in some way, e.g., printing them on the standard output. Open `myapp.sh` and add the following lines at the end:

```
1 echo ""
2 echo "Selected entities:"
3 echo ""
4 for e in $entities
5 do
6    echo "- $e"
7 done
```

This code simply print the entities in separated lines. Again, the external script `parse_params.sh` parses the command-line and stores the list of entities in the variable `entities`.

Go back to the web-client, reload the page, select some files and entities and execute the `MFA` application to see the result of the last changes. It is always recommended to check that the application works correctly from a shell first.

2.4 Passing Parameters to Your Application

In addition to input files and outline entities, real applications receive other parameters to control different aspects. In this section we explain how to declare parameters in the EasyInterface framework such that (1) they automatically appear in the web-client (or any other client) so the user can set their values; and (2) the selected values are passed to the application when executed.

Let us start by modifying `myapp.sh` to accept some command-line parameters: we add a parameter "`-s`" to indicate if the received outline entities should be printed; and "`-c W`" that takes a value `W` to indicate what to count in each file — here `W` can be "`lines`", "`words`" or "`chars`". For example, `myapp.sh` could then be invoked as follows:

```
> myapp.sh -f file1.c file2.c file3.c -e sum.c:main sum.c:sum -s -c words
```

To support these parameters, change the content of `myapp.sh` to the following:

```
1  #!/bin/bash
2
3  . default/parse_params.sh
4
5  echo "I've received the following command-line parameters:"
6  echo ""
7  echo "  $@"
8
9  echo ""
10 echo "File statistics:"
11 echo ""
12
13 case $whattocount in
14     lines) wcparam="-l"
15     ;;
16     words) wcparam="-w"
17     ;;
18     chars) wcparam="-m"
19     ;;
20 esac
21
22 for f in $files
23 do
24     echo " - $f has " `wc $wcparam $f | awk '{print $1}'` $whattocount
25 done
26
27 if [ $showoutline == 1 ]; then
28     echo ""
29     echo "Selected entities:"
30     echo ""
31     for e in $entities
32     do
33        echo "- $e"
34     done
35 fi
```

Compared to the previous script, you can notice that: we added lines 13-20 to take the value of "`-c`" into account when calling `wc` at line 24; and in lines 27-35 we wrapped the loop that prints the outline entities with a condition. Note that `parse_params.sh` sets the variable `whattocount` to the value of "`-c`", and sets `showoutline` to 1 if "`-s`" is provided in the command-line and to 0 otherwise. Before proceeding to the next step, execute the script from a shell to make sure that it works correctly.

Our goal is show these parameters in the web-client (or any other client), so the user can select the appropriate values before executing the application. The EasyInterface framework provides an easy way to do this, all we have to do is to modify `myapp.cfg` to include a description of the supported parameters. Open `myapp.cfg` and add the following inside the **app** tag (e.g., immediately after **execinfo**):

```
<parameters prefix = "-" check="false">
  <selectone name="c">
    <desc>
```

```
      <short>What to count</short>
      <long>Select the elements you want to count in each input file</long>
    </desc>
    <option value="lines">
      <desc>
        <short>Lines</short>
          <long>Count lines</long>
      </desc>
    </option>
    <option value="words">
      <desc>
        <short>Words</short>
        <long>Count words</long>
        </desc>
    </option>
    <option value="chars" >
      <desc>
        <short>Chars</short>
        <long>Count characters</long>
      </desc>
    </option>
    <default value="lines"/>
  </selectone>
  <flag name="s">
    <desc>
      <short>Show outline</short>
      <long>Show the selected outline entities</long>
    </desc>
    <default value="false"/>
  </flag>
</parameters>
```

Let us explain the different elements of the above XML snippet. The tag **parameters** includes the definition of all parameters. The attribute **prefix** is used to specify the symbol to be attached to the parameter name when passed to the application, for example, if we declare a parameter with name "c" the server will pass it to the application as "-c". Note that this attribute can be overridden by each parameter. The attribute **check** tells the server to check the correctness of the parameters before passing them to the application, i.e., that they are valid values, etc. The tag **selectone** defines a parameter with **name** "c" that can take one value from a set of possible ones. For example, the web-client will view it as a ComboBox. The **desc** environment contains a text describing this parameter and is used by the client when viewing this parameter graphically. The **option** tags define the valid values for this parameter, from which one can be selected, and the **default** tag defines the default value. The **desc** environment of each **option** contains a text describing this option, e.g., the **short** description is used for the text in the corresponding ComboBox. The tag **flag** defines a parameter with name "s". This parameter has no value, it is either provided in the command-line or not, and its **default** value is **"false"**, i.e., not provided. For the complete set of parameters supported in EasyInterface see [PARAMETERS] in Chapter 3.

Go to the web-client, reload the page, and click on the `Settings` button and look for the tab with the title `MFA`. You will now see the parameters declared above in a graphical way where you can set their values as well. When you click on the `Apply` button, the web-client will pass these parameters to the server, however, we still have to tell the server how to pass these parameters to `myapp.sh`. Open `myapp.cfg` and change the **cmdlineapp** template to the following:

```
<cmdlineapp>./bin/myapp.sh -f _ei_files -e _ei_outline _ei_parameters</cmdlineapp>
```

As in the case of `_ei_files` and `_ei_outline`, the server will replace `_ei_parameters` by the list of received parameters before executing the command-line. Execute the `MFA` application from the web-client with different values for the parameters to see how the output changes.

2.5 Using the EasyInterface Output Language in Your Application

In the example that we have developed so far, the web-client simply printed the output of `myapp.sh` in the console area. This is the default behavior of the web-client if the output does not follow the EasyInterface Output Language (EIOL), which is a text-based is language that allows generating more sophisticated output such as highlighting lines, adding markers, etc. In this section we will explain the basics of this language by extending `myapp.sh` to use it, for more details see Chapter 5.

An output in EIOL is an XML structure of the following form:

```
<eiout>
 <eicommands>
    [EICOMMAND]*
 </eicommands>
 <eiactions>
    [EIACTION]*
 </eiactions>
</eiout>
```

where (1) **eiout** is the outermost tag that includes all the output elements; (2) [EICOMMAND]* is a list of commands to be executed; and (3) [EIACTION]* is a list of actions to be declared. An [EICOMMAND] is an instruction like: *print a text on the console*, *highlight lines 5-10*, *add marker at line 5*, etc. An [EIACTION] is an instructions like: *when the user clicks on line 13, highlight lines 20-25*, etc. In the rest of this section we discuss some commands and actions that are supported in EIOL, for the complete list see Chapter 5.

2.5.1 Printing in the Console Area

Recall that when the EIOL is used, the web-client does not redirect the output to the console and thus we need a command to print in the console area. The following is an example of a command that prints "Hello World" in the console area:

```
<printonconsole consoleid="1" consoletitle="Welcome">
  <content format="text">
    Hello World
  </content>
</printonconsole>
```

The value of the `consoleid` attribute is the console identifier in which the given text should be printed (e.g., in the web-client the console area has several tabs, so the identifier refers to one of those tabs). If a console with such identifier does not exist yet, a new one, with a title as specified in `consoletitle`, is created. If `consoleid` is not given the output goes to the default console. Inside **printonconsole** we can have several **content** tags which include the content to be printed (in the above example we have only one). The attribute `format` indicates the format of the content. In the above example it is plain 'text', other formats are supported as well, e.g., 'html'.

Let us change `myapp.sh` to print the different parts of its output in several consoles. Open `myapp.sh` and change its content to the following:

```
1  #!/bin/bash
2
3  . default/parse_params.sh
4
5  echo "<eiout>"
6  echo "<eicommands>"
7  echo "<printonconsole>"
8  echo "<content format='text'>"
9  echo "I've received the following command-line parameters:"
10 echo ""
11 echo "   $@"
```

```
12 echo "</content>"
13 echo "</printonconsole>"
14
15 echo "<printonconsole consoleid='stats' consoletitle='Statistics'>"
16 echo "<content format='html'>"
17 echo "File statistics:"
18 echo "<div>"
19 echo "<ul>"
20
21 case $whattocount in
22     lines) wcparam="-l"
23     ;;
24     words) wcparam="-w"
25     ;;
26     chars) wcparam="-m"
27     ;;
28 esac
29
30 for f in $files
31 do
32     echo " <li> $f has " `wc $wcparam $f | awk '{print $1}'` $whattocount "</li>"
33 done
34 echo "</ul>"
35 echo "</div>"
36 echo "</content>"
37 echo "</printonconsole>"
38
39 if [ $showoutline == 1 ]; then
40     echo "<printonconsole consoleid='outline' consoletitle='Outline'>"
41     echo "<content format='html'>"
42     echo ""
43     echo "Selected entities:"
44     echo "<ul>"
45     echo ""
46     for e in $entities
47     do
48       echo "<li> $e </li>"
49     done
50     echo "</ul>"
51     echo "</content>"
52     echo "</printonconsole>"
53 fi
54 echo "</eicommands>"
55 echo "</eiout>"
```

The output of `myapp.sh` is give in EIOL, because at Line 5 we start the output with the tag **eiout** which we close at line 55. At Line 6 we start an **eicommands** tag, inside **eiout**, which we close at Line 54. Inside **eicommands** we have 3 **printonconsole** commands: the first one is generated by lines 7-13; the second by lines 15-37; and the last one by lines 40-52. Note that the first command uses the default console, while the last two use different consoles. Note also that the content in the last two is given in HTML. Execute `myapp.sh` in a shell first to check that it works correctly, and then execute the `MFA` application from the web-client to see the effect of these changes.

2.5.2 Adding Markers

Next we explain a command for adding a marker next to a code line in the editor area. The following is an example of such command:

```
<addmarker dest="path" outclass="info">
```

```
  <lines>
    <line from="4" />
  </lines>
  <content format='text'>
    text to associated to the marker
  </content>
</addmarker>
```

The attribute `dest` indicates the *full path* of the file in which the marker should be added. The attribute `outclass` indicates the nature of the marker, which can be 'info', 'error', or 'warning'. This value typically affects the type/color of the icon to be used for the marker. The tag **lines** includes the lines in which markers should be added, each line is give using the tag **line** where the `from` attribute is the line number (**line** can be used to define a region in other commands, this is why the attribute is called `from`). The text inside the **content** tag is associated to the marker (as a tooltip, a dialog box, etc., depending on the client).

Let us modify `myapp.sh` to add a marker at Line 1 of each file that it receives. Open `myapp.sh` and add the following code snippet immediately before 54 of the previous script (i.e., immediately before closing the **eicommands** tag):

```
1 for f in $files
2 do
3   echo "<addmarker dest='$f' outclass='info'>"
4   echo "<lines><line from='1'/></lines>"
5   echo "<content format='text'> text for info marker of $f</content>"
6   echo "</addmarker>"
7 done
```

Lines 3-6 generate the actual command to add a marker for each file passed to `myapp.sh`. Execute `myapp.sh` in a shell first to check that it works correctly, and then execute the `MFA` application from the web-client to see the effect of these changes.

2.5.3 Highlighting Code Lines

The following command can be used to highlight code lines:

```
<highlightlines dest="path" outclass="info" >
  <lines>
    <line from="5" to="10"/>
  </lines>
</highlightlines>
```

Attributes `dest` and `outclass` are as in the **addmarker** command. Each **line** tag define a region to be highlighted. E.g., in the above example it highlights lines 5-10. You can also use the attributes `fromch` and `toch` to indicate the columns in which the highlight starts and ends respectively.

Let us modify `myapp.sh` to highlight lines 5-10 of each file that it receives. Open `myapp.sh` and add the following code snippet immediately before the instruction that closes the **eicommands** tag:

```
1 for f in $files
2 do
3   echo "<highlightlines dest='$f' outclass='info'>"
4   echo "<lines><line from='5' to='10'/></lines>"
5   echo "</highlightlines>"
6 done
```

Execute `myapp.sh` in a shell first to check that it works correctly, and then execute the `MFA` application from the web-client to see the effect of these changes.

2.5.4 Adding Inline Markers

Inline markers are widgets placed inside the code. They typically include some read-only text. The following command adds an inline marker:

```
<addinlinemarker  dest="path" outclass="info">
    <lines><line from="15" /></lines>
    <content format="text">
        Text to be viewed in the inline marker
    </content>
</addinlinemarker>
```

Attributes `dest` and `outclass` are as in the **addmarker** command. Each **line** tag defines a line in which a widget, showing the text inside the **content**, is added. Note that some client, e.g., the web-client, allow only plain 'text' content.

Let us modify `myapp.sh` to add an inline marker at line 15 of each file that it receives. Open `myapp.sh` and add the following code snippet immediately before the instruction that closes the **eicommands** tag:

```
1 do
2   echo "<addinlinemarker dest='$f' outclass='info'>"
3   echo "  <lines><line from='15' /></lines>"
4   echo "  <content format='text'> Awesome line of code!!</content>"
5   echo "</addinlinemarker>"
6 done
```

Execute `myapp.sh` in a shell first to check that it works correctly, and then execute the `MFA` application from the web-client to see the effect of these changes.

2.5.5 Opening a Dialog Box

The following command can be used to open a dialog box with some content:

```
<dialogbox outclass="info" boxtitle="Done!" boxwidth="100" boxheight="100">
  <content format="html">
     text to be shown in the dialog box
  </content>
</dialogbox>
```

The dialog box will be titled as specified in `boxtitle`, and it will include the content as specified in the **content** environments. The attributes `boxwidth` and `boxheight` are optional, they determine the initial size of the window.

Let us modify `myapp.sh` to open a dialog box with some message. Open `myapp.sh` and add the following code snippet immediately before the instruction that closes the **eicommands** tag:

```
1 echo "<dialogbox outclass='info' boxtitle='Done!'  boxwidth='300' boxheight='100'>"
2 echo "  <content format='html'>"
3 echo "    The <span style='color: red'>MFA</span> analysis has been applied."
4 echo "    You can see the output in the result in the text area and the corresponding"
5 echo "    program files."
6 echo "  </content>"
7 echo "</dialogbox>"
```

Execute `myapp.sh` in a shell first to check that it works correctly, and then execute the `MFA` application from the web-client to see the effect of these changes.

2.5.6 Adding Code Line Actions

A *code line action* defines a list of commands to be executed when the user clicks on a line of code (more precisely, on a marker placed next to the line). The commands can be any of those seen above. The following is an example of such action:

```
<oncodelineclick dest="/Examples_1/iterative/sum.c" outclass="info" >
  <lines><line from="17" /></lines>
  <eicommands>
    <highlightlines>
      <lines>
        <line from="17" to="19"/>
      </lines>
    </highlightlines>
    <dialogbox boxtitle="Hey!">
      <content format="html">
        Click on the marker again to close this window
      </content>
    </dialogbox>
  </eicommands>
</oncodelineclick>
```

First note that the above XML should be placed inside the **eiactions** tag (that we have ignored so far). When the above action is executed, by the web-client for example, a marker (typically an arrow) will be shown next to line 17 of the file `/Examples_1/iterative/sum.c`. Then, if the user clicks on this marker the commands inside the **eicommands** tag will be executed, and if the user clicks again the effect of these commands is undone. In the above case a click highlights lines 17-19 and opens a dialog box, and another click removes the highlights and closes the dialog box. Note that the commands inside **eicommands** inherit the `dest` and `outclass` attributes of **oncodelineclick**, but one can override them, e.g., if we add `dest="/Examples\_1/iterative/fact.c"` to the **highlightlines** command then a click highlights lines 17-19 of `fact.c` instead of `sum.c`.

Let us modify `myapp.sh` to open add a code line action, as the one above, for each file that it receives. Open `myapp.sh` and add the following code snippet immediately before the instruction that closes the **eiout** tag (i.e., after closing **eicommands**):

```
1  echo "<eiactions>"
2
3  for f in $files
4  do
5    echo "<oncodelineclick dest='$f' outclass='info' >"
6    echo "<lines><line from='17' /></lines>"
7    echo "<eicommands>"
8    echo "<highlightlines>"
9    echo "<lines><line from='17' to='19'/></lines>"
10   echo "</highlightlines>"
11   echo "<dialogbox boxtitle='Hey!'> "
12   echo "<content format='html'>"
13   echo "Click on the marker again to close this window"
14   echo "</content>"
15   echo "</dialogbox>"
16   echo "</eicommands>"
17   echo "</oncodelineclick>"
18 done
19
20 echo "</eiactions>"
```

Note that at line 1 we open the tag **eiactions** and at line 21 we close it. The rest of the code simply prints a code line action as the one above for each file. Execute `myapp.sh` in a shell first to check that it works correctly, and then execute the `MFA` application from the web-client to see the effect of these changes.

2.5.7 Adding OnClick Actions

OnClick actions are similar to code line actions. The difference is that instead of assigning the action to a line of code, we can assign it to any HTML tag that we have generated. For example, suppose that at some point the application has generated the following content in the console area:

```
<content format="html"/>
   <span style="color: red;" id="err1">10 errors</span> were found in the file sum.c
</content>
```

Note that the text "10 errors" is wrap by a span tag with an identifier `err1`. The OnClick action can assign a list of commands to be executed when this text is clicked as follows:

```
<onclick>
   <elements>
     <selector value="#err1"/>
  </elements>
  <eicommands>
    <dialogbox boxtitle="Errors">
      <content format="html">
        There are some variables used but not declated
      </content>
    </dialogbox>
  </eicommands>
</onclick>
```

It is east to see that this action is very similar to **oncodelineclick**, the difference is that instead of **lines** we now use **elements** to identify those HTML elements a click on which should execute the commands.

Let us modify `myapp.sh` to open add an OnClick action assigned to the list of files that it prints on the console. First look for the first occurrence of

```
1 echo "<ul>"
```

which should be at line 19, and change it by

```
1 echo "<ul style='background: yellow;' id='files'>"
```

This change will give the list of files that we print in the console the identifier `files`, and will change its background color to yellow. Next add the following code immediately before the instruction that closes **eiactions**:

```
1  echo "<onclick>"
2  echo "<elements>"
3  echo "<selector value='#files'/>"
4  echo "</elements>"
5  echo "<eicommands>"
6  echo "<dialogbox boxtitle='Errors'> "
7  echo "<content format='html'>"
8  echo "There are some variables used but not declated"
9  echo "</content>"
10 echo "</dialogbox>"
11 echo "</eicommands>"
12 echo "</onclick>"
```

Execute `myapp.sh` in a shell first to check that it works correctly, and then execute the `MFA` application from the web-client to see the effect of these changes. In particular, clicking on the list of files in the console area (anywhere in the yellow region) should open a dialog box.

3 | EasyInterface Server

This chapter describes the server side of the EasyInterface framework, that we refer to as the EasyInterface server (or simply the server). As explained in Section 1.2, the goal of this server is to provide a uniform way to access local applications and examples (i.e., those installed on the same machine where the server runs).

The EasyInterface server achieves the above goal by: (1) providing a way to describe, using XML based configuration files, how to execute a local application and which parameters it takes, as well as define sets of related examples; and (2) providing a JSON based protocol that can be used to request information on those applications and examples, execute applications, etc. Section 3.1 describes the syntax of the server configuration file, which covers point (1) above; and Section 3.2 describes the protocol that can be used to communicate with the server. This chapter does not cover issues related to installation, for such documentation see `INSTALL.md`.

3.1 Configuring the EasyInterface Server

This section describes how to configure the EasyInterface server. Section 3.1.1 explains where the configuration file show be placed, and Section 3.1.2 describes the (valid) content of this file. Before proceeding to the next sections, it highly recommended to read Chapter 2 to get a general idea on how the configuration file looks like, etc.

3.1.1 Name and Path of the Configuration File

By default the server looks for the configuration file `server/config/eiserver.cfg`, and if no such file exists it uses `server/config/eiserver.default.cfg`. The default installation comes with a predefined `sever/config/eiserver.default.cfg` that includes some demo applications and corresponding examples. It is very recommended not make substantial changes to `eiserver.default.cfg`, and instead create your own `eiserver.cfg`. This way you can always have a correct configuration file at hand from which you can copy, etc.

3.1.2 The Syntax of the Configuration File

The content of the configuration file should adhere to the [EISERVER] XML structure that is described below. Inside this tag you can define applications, examples, etc. The best way to understand how to do is follow the links in the definition of [EISERVER].

General comment about XML structures

For the purpose of better organization of the configuration files, any XML structure

```
<tagname ...>
  ....
</tagname>
```

can be also written as

```
<tagname src=[CFGFILENAME] />
```

where the file [CFGFILENAME] includes the actual XML structure (of the first form). However, if the XML structure has an attribute `id` then it must appear as well in the second form.

The main XML tag of the configuration file

EISERVER

```
<eiserver>
  [SETTINGS]?
  [EXAMPLES]?
  [APPS]?
</eiserver>
```

DESCRIPTION:

This XML tag is the root of the configurations file. The [SETTINGS] section is used for setting some global parameters; the [APPS] section defines which applications are available on the server; and [EXAMPLES] defines which sets of examples are available on the server.

General settings

SETTINGS

```
<settings>
</settings>
```

DESCRIPTION:

This tag is does not include anything yet. It is reserved for future use, where we plan to put general settings that are related to the server and not to specific application or examples set.

Examples settings

EXAMPLES

```
<examples>
  [EXSET]*
</examples>
```

DESCRIPTION:

This tag is used to declare sets of examples sets that are available in the server, where each such set is defined by one [EXSET].

EXSET

```
<exset id=[EXSETID]>
  [EXELEMENT]*
</exset>
```

DESCRIPTION:

This tag declare a set of examples, which are defined by collection of [EXELEMENT] (a file, a directory, or a link to a github repository). The attribute id is a unique identifier that is used to refer to this set when communicating with the server.

EXELEMENT

```
( [FILE] | [FOLDER] | [GITHUB] )
```

DESCRIPTION:

An example element, which can be a file [FILE], a folder [FOLDER], or a link to a github repository [GITHUB].

FILE

```
<file name=[FILENAME] url=[URL] />
```

DESCRIPTION:

This tags declares a file, where the name attribute is its name and url is a link to its content. Note that name is not necessarily the same as the one in url.

FOLDER

```
<folder name=[FOLDERNAME]>
  [EXELEMENT]*
</folder>
```

DESCRIPTION:

This tags declares a folder with name as its name. The content of this is a list of [EXELEMENT] tags, which in turn declare the inner files, folders, etc.

GITHUB

```
<github repo=[GITHUBREPO] owner=[GITHUBUSER] branch=[GITHUBBRANCH]? path=[GITHUBPATH]?
    />
```

DESCRIPTION:

Declares a reference to the public github repository repo which is owned by the user owner. Optionally one can also refer to a specific branch which `master` by default, and to a specific githubpath (a directory or a single file) which is a the root directory by default.

Applications settings

APPS

```
<apps>
  [APP]*
</apps>
```

DESCRIPTION:

Thhis tag declares a list of applications (to be added to the server). Each such application is defined by one [APP] environment.

APP

```
<app id=[APPID] visible=[BOOL]?>
  [APPINFO]
  [APPHELP]?
  [EXECINFO]
  [PARAMETERS]
</app>
```

DESCRIPTION:

This tag defines an application, where the meaning different parts is as follows:

- `id` is a unique identifier used to refer to this application when communicating with the server.
- `visible` indicates if this application should be listed when the list of available applications is requested — by default it is `true`. Note that even if an application is not visible, it can be used like any other application by those who know it `id`.
- [APPINFO] provides some general information about the application, e.g., title, logo, etc.
- [APPHELP] provides enough information on how the application can be used, etc. It is mainly used in the help sections of the different clients.
- [EXECINFO] defines how the application can be executed (e.g., from a command-line).
- [PARAMETERS] defines the set parameters accepted by the application.

APPINFO

```
<appinfo>
  [ACRONYM]?
  [TITLE]?
  [LOGO]?
  [DESC]?
</appinfo>
```

DESCRIPTION:

This tag provides general information for an application:

- [ACRONYM] is an acronym for the application, e.g., COSTA;
- [TITLE] is the full name of the application;
- [LOGO] is an image corresponding to the log of the application; and
- [DESC] is a description of the tool.

ACRONYM

```
<acronym>[TEXT]</acronym>
```

DESCRIPTION:

Plain text to be used as an acronym, e.g., COSTA.

TITLE

```
<title>[TEXT]</title>
```

DESCRIPTION:

Plain text deciding a title, e.g., for an application. It is typically more informative than an acronym (see [ACRONYM].

LOGO

```
<logo url=[URL] />
```

DESCRIPTION:

A link to an image — in some standard format such png, jpg or gif — to be used by clients as a logo (for an APP).

DESC

```
<desc>
  <short>[TEXT]</short>
  <long>[TEXT]</long>
</desc>
```

DESCRIPTION:

This is a description of some entities, e.g., of an application, a parameter, a parameter option, etc. It consists of two parts, the first one is a short description, and the second is a detailed description. In both cases it should be plain text. Clients will select one of them depending on the intended use.

APPHELP

```
<apphelp>
  [CONTENT]+
</apphelp>
```

DESCRIPTION:

A (formatted) text that provides enough information on how an application can be used, etc. It can be provided in several formats, e.g., html or plain text, by using several [CONTENT] tags. Clients are supposed to pick the appropriate format if more than one is available. It is recommended to always include a content in plain text since it can be view in any client.

CONTENT

```
<content format=[TEXTFORMAT]? >
  [TEXT]
</content>
```

DESCRIPTION:

A text given in a specific format, e.g., "text", "html", etc. If the attribute format is not provided, then it is assumed to be "text" format (plain text).

EXECINFO

```
<execinfo>
 [CMDLINEAPP]
</execinfo>
```

DESCRIPTION:

Provides information on how to execute an application. Currently it includes only a command-line template [CMDLINEAPP].

CMDLINEAPP

```
<cmdlineapp> [CMDTEMPLATE] </cmdlineapp>
```

DESCRIPTION:

Describes how to run an application, where [CMDTEMPLATE] is a template describing a command-line. It is best explained by an example. Consider the following template example

```
/path-to/app _ei_files -m _ei_outline _ei_parameters
```

In this template, anything that starts with `_ei` is a template parameter that will be replaced by some corresponding information, and `/path-to/app` is the application executable. When the server receives a request for executing the corresponding application, the request includes several data that should passed to the application. For example, the following are typical data that should be passed to an application:

1. files to be processed (e.g., program to be analyzed);
2. entities selected from the program outline (e.g., methods); and
3. values for the different parameters.

The server passes this data to the application by replacing the template parameters with corresponding data as follows:

1. the files are stored locally (e.g., in `/tmp`), and `_ei_files` is replaced by a list file names (each with an absolute path, separated by a space);
2. `_ei_outline` is replaced by a list of selected entities (e.g., method names); and
3. `_ei_parameters` is replaced by the list of parameters generated from those provided in the request.

This result in, for example, the following instance of the template:

```
/path-to/app a.c b.c -m a.main -v 1 -d 3 -a
```

which is then executed and its output is redirected to the client. The server does some security checks to guarantee that the command-line is not harmful.

The following is a list of template parameters that can be used:

- `_ei_files` is replaced by a list of file names (separated by space) in the local file system;
- `_ei_root` is replaced by the local temporary directory name, where all files have been saved. This file is of the form `/path-to-tmp/_ei_files` where `/path-to-tmp` depends on the operating system (e.g., `/tmp` in Linux);
- `_ei_outline` is replaced by a list of selected entities (separated by space);
- `_ei_parameters` is replaced by a corresponding list of parameters (see [PARAMETERS]);

- `_ei_sessionid` is replaced by a session identifier, this makes it possible to track the information of a user along several request;
- `_ei_clientid` is replaced by the client identifier, i.e., `webclient`, `eclipse`, etc., which makes it possible to provide output depending on the client. See Chapter 4 for a list of clients and their corresponding identifiers.

Application parameters

PARAMETERS

```
<parameters prefix=[PARAMPREFIX]? check=[BOOL]?>
  [PARAM]*
</parameters>
```

DESCRIPTION:

Defines a list of parameters that are accepted by a corresponding application. Each parameter is define by one [PARAM] environment. The prefix attribute is used to specify a string that will be attached to each parameter name when passed to the application. For example, if prefix="--" and there is a parameter called 'level' with value X, then '--level X' will be passed to the application. The default value of prefix is "-". It can also be set to an empty string if there is no need for a prefix. The check attribute is used to indicate if the server should verify that the values of the parameters are valid (w.r.t. the specified values). The default value of check is "true". The attributes prefix and check are inherited by each parameter [PARAM], which in turn can override then as well.

PARAM

```
( [SELECTONE] | [SELECTMANY] | [FLAG] | [TEXTFIELD] )
```

DESCRIPTION:

Defines a parameter accepted by a corresponding application. There are several types of parameters supported:

- [SELECTONE] defines a parameter that takes one value from a predefined set;
- [SELECTMANY] defines a parameter that takes several values from a predefined set;
- [FLAG] defines a parameter that either appear or not in the command-line; and
- [TEXTFIELD] defines a parameter that take a free-text value.

SELECTONE

```
<selectone name=[PARAMNAME] prefix=[PARAMPREFIX]? check=[BOOL]? widgetid=[WIDGETID]? >
  [DESC]
  [OPTION]+
  [DEFAULTVALUE]?
</selectone>
```

DESCRIPTION:

Defines a parameter that takes a *single* value out of a given list:

- name is the name of the parameter, it must be unique among all parameters of an app;
- prefix and check can be used to override the corresponding attributes of [PARAMETERS];
- [DESC] provide a description of this parameter;

- [OPTION]+ is a list of possible values for this parameter;
- [DEFAULTVALUE] specifies the default value. If not specified then the first [OPTION] is considered as the default one;
- widgetid specifies the preferred layout when used in a client with a graphical interface (e.g., combo-box, radio button, etc.). This is client depends, see Chapter 4 for more information.

SELECTMANY

```
<selectmany name=[PARAMNAME] prefix=[PARAMPREFIX]? check=[BOOL]? widgetid=[WIDGETID]? >
  [DESC]
  [OPTION]+
  [DEFAULTVALUE]*
</selectmany>
```

DESCRIPTION:

Defines a parameter that takes *several* values out of a given list. The meaning of the attributes and inner environments is as in [SELECTONE], except that in this case we can specify several [DEFAULTVALUE].

FLAG

```
<flag name=[PARAMNAME] explicit=[BOOL]? prefix=[PARAMPREFIX]? check=[BOOL]?
     widgetid=[WIDGETID]? >
  [DESC]
  [DEFAULTVALUE]?
</flag>
```

DESCRIPTION:

Defines a parameter that can take `true` or `false` values. The meaning of the attributes and inner environments is as in [SELECTONE]. In addition, the attribute `explicit` is used to specify how this parameter should be passed to the application. For example, assume the parameter name is `f`, then:

- when `explicit` is `true` the parameter is explicitly passed to the application, i.e., using "`-f true`" or "`-f false`"; and
- when `explicit` is `false` parameter is passed as "`-f`" if its value is `true` and not passed at all if its value is `false`.

The default value of `explicit` is `false`.

TEXTFIELD

```
<textfield name=[PARAMNAME] passinfile=[BOOL]? multiline=[BOOL]? prefix=[PARAMPREFIX]?
          check=[BOOL]? widgetid=[WIDGETID]? >
  [DESC]
  <initialtext>[TEXT]</initialtext>
</textfield>
```

DESCRIPTION:

Defines a parameter that can take free-text value. The meaning of the attributes and inner environments is as in [SELECTONE]. The initialtext tag includes a text to be shown in the corresponding text-area by default. The meaning of extra attributes is as follows:

- `multiline` is used to specify if the free-text should be single-line or multi-lines. By default its values is `false`, i.e., single-line.
- `passinfile` is used to indicate that the actual value should be saved into a file, and what is passed to the application is the file name instead of the actual text. This should be used for safety, when there is a risk that the free-text can be harmful to the command-line (although the server does some checks to avoid this).

OPTION

```
<option value=[PARAMVALUE]>[DESC]</option>
```

DESCRIPTION:

Defines an option (i.e., a possible value) for a parameter.

DEFAULTVALUE

```
<default value=[PARAMVALUE] />
```

DESCRIPTION:

Defines a default value for a parameter.

TEXTFORMAT

(`"text"` | `"html"` | `"svg"`)

PARAMVALUE

[a-z,A-Z,0-9,-,_]+

PARAMNAME

[a-z,A-Z,0-9,-,_]+

BOOL

(`"true"` | `"false"`)

APPID

[a-z,A-Z,0-9,-,_]+

EXSETID

[a-z,A-Z,0-9,-,_]+

WIDGETID

[a-z,A-Z,0-9,-,_]+

URL

A valid `http` or `https` URL.

PARAMPREFIX

Can be any string that matches [a-z,A-Z,0-9,-,_]+, typically `"-"` or `"--"`.

TEXT

Free text.

GITHUBPATH
A path to a file or a directory in a github repository (relative to the root of the repository).

GITHUBBRANCH
A valid branch name for a github repository.

GITHUBUSER
A valid github username.

GITHUBREPO
A valid github repository.

FOLDERNAME
[a-z,A-Z,0-9,-,.,_]+

FILENAME
[a-z,A-Z,0-9,-,.,_]+

CFGFILENAME
A path to a configuration file. Should be relative to `server/config`.

CMDTEMPLATE
The explanation is given in [CMDLINEAPP]

3.2 Communicating with EasyInterface Server

At the moment we write an example with all the things that you can write.

```
var jsonParams = {
     "command": "execute",//command
     "app_id": "costa",//app id
     "parameters": {
         "lsa":["true"],//flag parameter
         "flag2":["false"], //flag parameter
         "sone":["sla"], //selectone parameter
         "smany":["slaa","tru"], //selectmany parameter
         "_ei_files": [
             {
                 "name": "prueba",
                 "type": "directory",
             },
             {
                 "name":"f1.txt",
                 "type": "file",
                 "content": "I'm f1.txt file\\n"
             },
             {
                 "name":"f2.txt",
                 "content": "I'm f2.txt file\\n"
             }
         ]//array of files and directories
     }//end parameters
 };// end jsonParams

  $.post("eiserver.php",
   {
      eirequest: JSON.stringify(jsonParams)
   },
   function(data) { });
```

4 | EasyInterface Clients

4.1 Web-Client

4.1.1 Generate Outline

The Outline App returns an XML with the outline tree.

OUTLINE

```
<category text=[NODENAME]
    value=[NODENAME] selectable=[BOOL]?
    icon=[PATH]?>
        [OUTLINE]*
</category>
```

Description:

Defines a list of points to be analysed.

- `text` is the name to be show.
- `value` is the internalname .
- `selectable` indicates if this node can be selected.
- `icon` indicates an alternative path for the icon of the node.

4.2 Eclipse Plugin

4.3 Remore shell

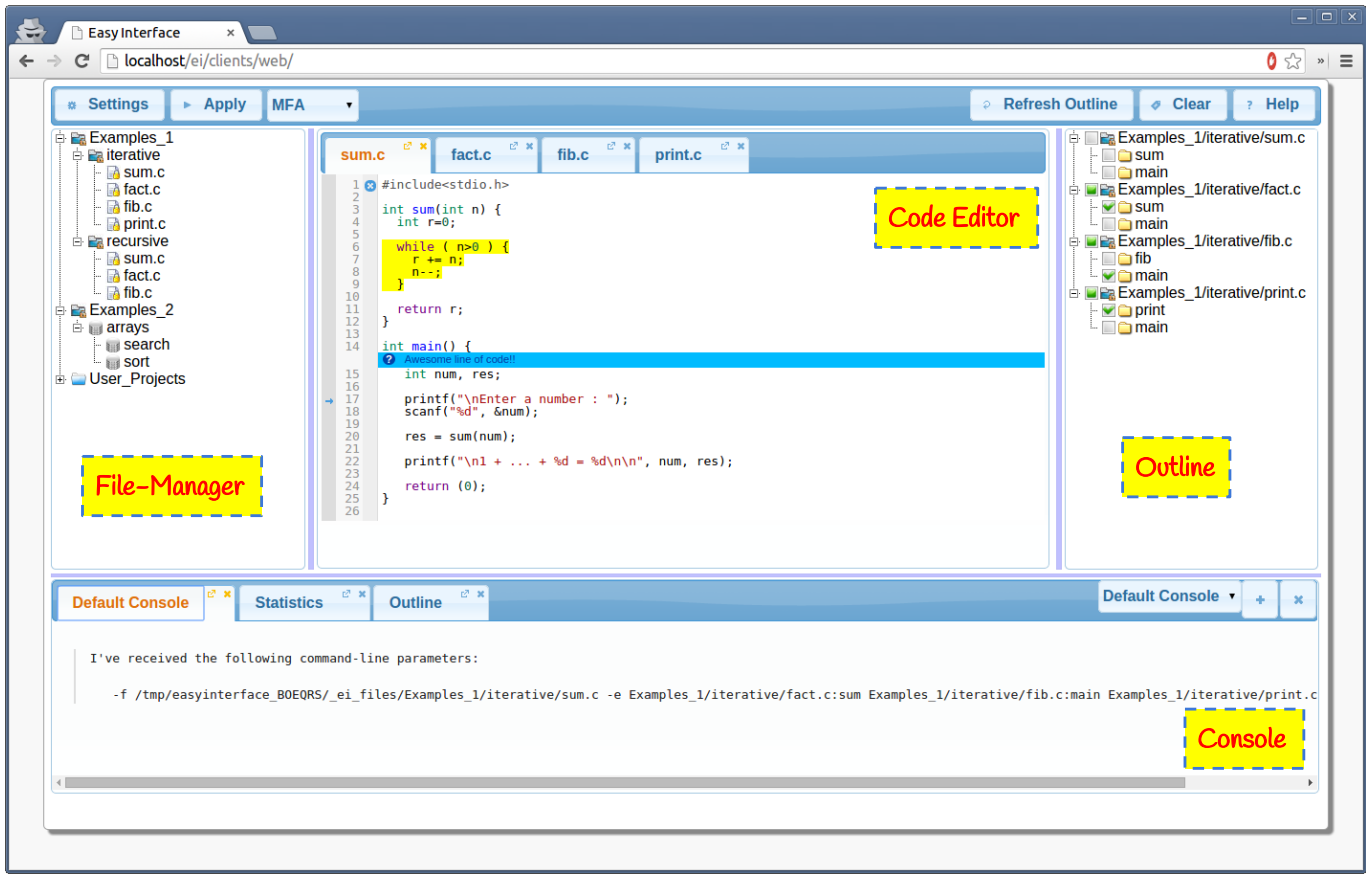

Figure 4.1: EasyInterface Web Client

5 | The EasyInterface Output Language

In this chapter we describe a text-based output language that allows application to view their output in a graphical way, e.g., highlighting lines, adding markers, defining on-click actions, etc. Some clients, e.g., the web-client, interpret this language and render the effect of the corresponding commands in the respective environment.

5.1 Brief description

5.2 Details XML output

EIOUT

```
<eiout version=[VERSION]? >
  [EICOMMANDS]*
  [EIACTIONS]*
</eiout>
```

Description:

This is the main environment of the output, it includes several list of commands environment [EICOMMANDS], and several list of actions actions [EIACTIONS]. Commands are executed first, in the given order, and then actions are executed in the given order as well. The `version` attribute indicates the version of the output language that is used, which is 1.0 by default.

EICOMMANDS

```
<eicommands outclass=[OUTCLASS]? dest=[PATH]?>
  [EICOMMAND]*
</eicommands>
```

Description:

A list of commands to be performed. The attribute `dest` is the destination file on which the command is applied (if needed). E.g., when highlighting a line we might want to highlight a line in one file or another. If `dest` is not specified, then the commands will be applied to the file that is currently active, e.g., if the client have a code editor with several tabs, one for each file, the command will be applied to the active tab. If none is active then the behavior is not specified. The attribute `outclass` specifies the *output class* of the commands in this environment, that is, the nature of the corresponding output generated by the commands, e.g., error, information, warning, etc. All commands inside this environment inherit the values of `outclass` and `dest`, and each can overwrite them.

EIACTIONS

```
<eiactions dest=[PATH]? autoclean=[BOOL]?>
  [EIACTION]*
</eiactions>
```

Description:

A a list of actions to be declared. An action typically executes a list of [EICOMMANDS] when the user interacts with the interface in some predetermined way, e.g., *when the user clicks on line 30, highlight lines number 12 and 16*. We say the an action is *performed* as a response to the user interaction. If the user interacts again with the interface, according to what is specified in the action, then the action is *unperformed* if possible (when the corresponding commands support the *undo* operation), e.g., in the above example if the user clicks again on line 30 the highlights of lines 12 and 16 are turned off.

Before *performing* an action, the last *performed* action is *unperformed* first. This behavior can be disabled by setting the autoclean attribute to ``false''. All actions inside this environment inherit the value of autoclean, and each can overwrite it. The attribute dest and outclass are as in the case of commands (see the description of [EICOMMANDS]).

EICOMMAND

```
(
  [PRINTONCONSOLECOMMAND]
| [HIGHLIGHTLINESCOMMAND]
| [DIALOGBOXCOMMAND]
| [WRITEFILECOMMAND]
| [SETCSSCOMMAND]
| [ADDMARKERCOMMAND]
| [ADDINLINEMARKERCOMMAND]
)
```

DESCRIPTION:

A command in the EASYINTERFACE output language, briefly:

- [PRINTONCONSOLECOMMAND] can be used to print on the console.
- [HIGHLIGHTLINESCOMMAND] can be used to highlight lines in the code editor.
- [DIALOGBOXCOMMAND] can be used to open a dialog window with a corresponding message.
- [WRITEFILECOMMAND] can be used to add a file (and a corresponding content) to the files tree.
- [SETCSSCOMMAND] can be used to change the CSS properties of some elements.
- [ADDMARKERCOMMAND] can be used to add a marker next to a line in the code editor.
- [ADDINLINEMARKERCOMMAND] can be used to add a line widget (an inlined marker) in the code editor.

EIACTION

```
(
  [ONCODELINECLICKACTION]
| [ONCLICKACTION]
)
```

DESCRIPTION:

An action in the EASYINTERFACE output language, briefly:

- [ONCODELINECLICKACTION] can be used to perform an action when the user clicks on a line in the code editor.
- [ONCLICKACTION] can be used to perform an action when the user clicks on a DOM element.

PRINTONCONSOLECOMMAND

```
<printonconsole outclass=[OUTCLASS]? consoleid=[CONSOLEID]? consoletitle=[STR]?>
  [CONTENT]+
```

```
</printonconsole>
```

DESCRIPTION:

Print the content of the [CONTENT] environments on the console with identifier consoleid. If consoleid is not specified, the output goes to the default console. If consoleid is specified but there is no console with such an identifier, the console is created and consoletitle (if specified) is used as its title. The attribute outclass is as described in [EICOMMANDS].

HIGHLIGHTLINESCOMMAND

```
<highlightlines outclass=[OUTCLASS]? dest=[PATH]?>
  [LINES]*
</highlightlines>
```

DESCRIPTION:

Highlight the lines specified by [LINES] in the file dest. The attribute outclass is as described in [EICOMMANDS].

DIALOGBOXCOMMAND

```
<dialogbox outclass=[OUTCLASS]? boxtitle=[STR]? boxwidth=[INT]? boxheight=[INT]? >
  [CONTENT]+
</dialogbox>
```

DESCRIPTION:

Open a dialog box with the content specified by the [CONTENT] environemnts. The value of boxtitle, if specfied, is used as a title for the dialog box. The attribtes boxwidth and boxheight can be used to set the size of the window. The attribute outclass is as in [EICOMMANDS].

WRITEFILECOMMAND

```
<writefile filename=[PATH] overwrite=[BOOLEAN]>
  [TEXT]
</writefile>
```

DESCRIPTION:

Create a new file and place it in the files view, using the path specified by filename. The file is initialized with the all text insider this tag. If the file exists, and overwrite is true the content is replaced otherwise a new file is created with a new name. The default value of overwrite is false. This command does not support *undo*.

SETCSSCOMMAND

```
<setcss>
  [ELEMENTS]
  [CSSPROPERTIES]
</setcss>
```

DESCRIPTION:

Change the CSS properties, as specified bt [CSSPROPERTIES], of all elements that match the selector in

[ELEMENTS]. There must be exactly one [ELEMENTS] environment and one [CSSPROPERTIES] environment. The elements are typically selected from those generated by other commands.

ADDMARKERCOMMAND

```
<addmarker outclass=[OUTCLASS]? dest=[PATH]? boxtitle=[STR]? boxwidth=[INT]? boxheight=
    [INT]?>
  [LINES]
  [CONTENT]*
</addmarker>
```

DESCRIPTION:

Add a marker next to each line that is specified in [LINES]. The column information from each [LINE] in [LINES] is ignored. All markers are associated with the content given by the [CONTENT] environments, as a tooltip. If the client allows expanding the tooltip to a dialog window, the the attributes boxtitle, boxwidth and boxheight can be used to set the properties of the corresponding window (see [DIALOGBOX]). The attributes dest and outclass are as described in [EICOMMANDS].

ADDINLINEMARKERCOMMAND

```
<addinlinemarker outclass=[OUTCLASS]? dest=[PATH]?>
  [LINES]
  [CONTENT]
</addinlinemarker>
```

DESCRIPTION:

Add an inline marker (a line widget) for each line that is specified by [LINES]. All line widgets will include the content specified by the [CONTENT] environments. In some clients, the supported content might be only of text format. The attributes dest and outclass are as described in [EICOMMANDS].

ONCODELINECLICKACTION

```
<oncodelineclick dest=[PATH]? autoclean=[BOOL]? outclass=[OUTCLASS]?>
  [LINES]
  [CONTENT]*
  [EICOMMANDS]
</oncodelineclick>
```

DESCRIPTION:

Add markers at the the code lines specified by [LINES], such that when any is clicked the commands in [EICOMMANDS] are performed. The content given by the [CONTENT] environemts is associated with the markers. The attributes dest and outclass are as described in [EIACTIONS]. Moreover, the above [EICOMMANDS] environment inherits the dest and outclass attributes of this environment.

ONCLICKACTION

```
<onclick outclass=[OUTCLASS] autoclean=[BOOL]? >
  [ELEMENTS]
  [EICOMMANDS]
</onclick>
```

DESCRIPTION:

A click on any DOM element that matches the selector of [ELEMENTS], will execute the commands declared in [EICOMMANDS]. The attributes dest and outclass are as described in [EIACTIONS]. Moreover, the above [EICOMMANDS] environment inherits the dest and outclass attributes of this environment.

LINES

```
<lines>
    [LINE]+
</lines>
```

DESCRIPTION:

A group of lines, typically used to specify the lines affected by an [EICOMMAND] or an [EIACTION].

LINE

```
<line from=[INT] to=[INT]? fromch=[INT]? toch=[INT]? />
```

DESCRIPTION:

A region (of lines) typically used to specify the region on which the effect of an [0]r an [I]s applied:

- from is the start line.
- to is the end line.
- fromch is the where the first line starts.
- toch is the character (i.e., column number) where the last line ends.

The default value of to is as the value of from. The default value of colfrom is 0, and of colto is the end of the line.

ELEMENTS

```
<elements>
 [SELECTOR]*
</elements>
```

DESCRIPTION:

Set of elements

SELECTOR

```
<selector value=[STR] />
```

DESCRIPTION:

The attribute value musy be valid selector as in JQuery. It is used to match some DOM elements.

CSSPROPERTIES

```
<cssproperties>
  [CSSPROPERTY]*
</cssproperties>
```

DESCRIPTION:

A set of CSS properties.

CSSPROPERTY

```
<cssproperty name=[STR] value=[STR] />
```

DESCRIPTION:

A CSS property. The attributes `name` and this `value` should correspond to valid CSS properties.

CONSOLEID

([a-z,A-Z,0-9,-,_]+ | new | default)

The value 'new' means a new console, no reference to this console is saved. The value 'default' means the default console of the client.

PATH

a path to file, including the file name, starting from the root /_ei_files/

VERSION

x.*y*, where *x* is the major version number and *y* is the minor one, e.g. 1.0, 1.1, etc.

OUTCLASS

none | info | warning | error