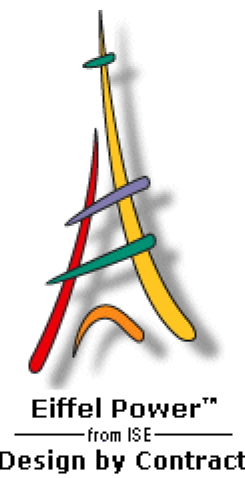
Advanced Topics in Object Technology

By Bertrand Meyer
Professor of Software Engineering at ETH Zurich



EiffelStudio: A Guided Tour

By Karine Arnout – ETH assistant, Ph.D. student.

1. About EiffelStudio *3*
2. Before getting started *3*
3. Starting a project with EiffelStudio *3*
4. Executing a system *7*
5. About the project directory *7*
6. Browsing facilities *9*
- 6.1 EiffelStudio's development window 10
- 6.2 Browsing by class name 10
- 6.3 Browsing from Cluster tree 12
- 6.4 Browsing from history 12
- 6.5 Class views 14
- 6.6 Pick and Drop 32

7. Editing a class text *34*
- 7.1 Clipboard 35
- 7.2 History 35
- 7.3 Search 35
- 7.4 Automatic completion 36

8. Debugging *38*
8.1 Setting breakpoints **38**
8.2 Executing with breakpoints **39**
9. Producing extensive documentation *42*
9.1 Generating multi-format documentation **42**
9.2 Generating XMI **46**
10. Graphics-based design with EiffelStudio *47*
Appendix: About compilation modes *54*
References *54*

1. About EiffelStudio

EiffelStudio is the core of ISE Eiffel. It goes beyond any existing programming environment since it covers the entire software lifecycle: it provides powerful support from analysis and design to implementation and maintenance, including a graphical tool for complete reverse engineering.

This guided tour aims at helping you know the basis of working with EiffelStudio, including:

- Starting a project (either from a wizard or a control file or retrieving an existing project);
- Compiling a project and running the generated system;
- Discovering EiffelStudio browsing facilities – including graphical representation;
- Using Pick and Drop;
- Recompiling and editing;
- Debugging;
- Producing documentation;
- Generating metrics about the software.

A free version of EiffelStudio – to be used for personal work only – is available online for download from Eiffel Software Web site: http://www.eiffel.com. It retains the main features of the enterprise version, except the Diagram and Metrics tool.

2. Before getting started

To fully benefit from this guided tour, you need to have ISE Eiffel installed on your machine and precompiled the EiffelBase library (this is an installation option). If you are running Windows, the installation procedure automatically registers the required environment variables.

However, you need to do this by hand if your platform is either Unix/Linux or VMS. The two variables you need are:

- **ISE_EIFFEL**, which is the path to your ISE Eiffel delivery folder (for instance, C:/Eiffel51)
- **ISE_PLATFORM**, which is either windows, unix or vms.

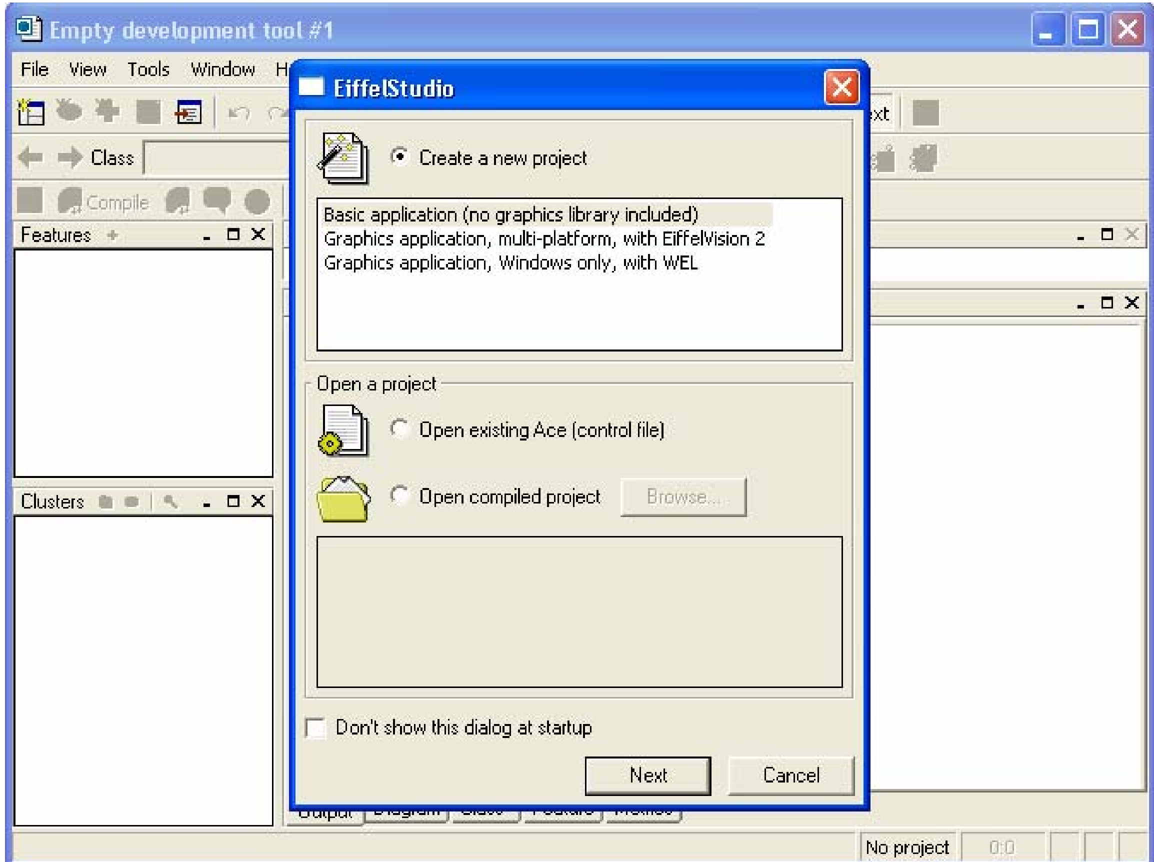
You also need to update your **Path** variable with the link to ISE Eiffel executable files ($ISE_EIFFEL/studio/spec/windows/bin).

3. Starting a project with EiffelStudio

To launch EiffelStudio on Windows, you can simply click the corresponding icon, which was automatically added during the installation procedure. You can also launch it as any other program, i.e. (on Windows XP):

Start → All Programs → ISE Eiffel → EiffelStudio 5.1
On Unix/Linux or VMS, you use the command line: **estudio**.

When launching EiffelStudio, it first comes up with a window (the environment window) and a dialog box on top of it:



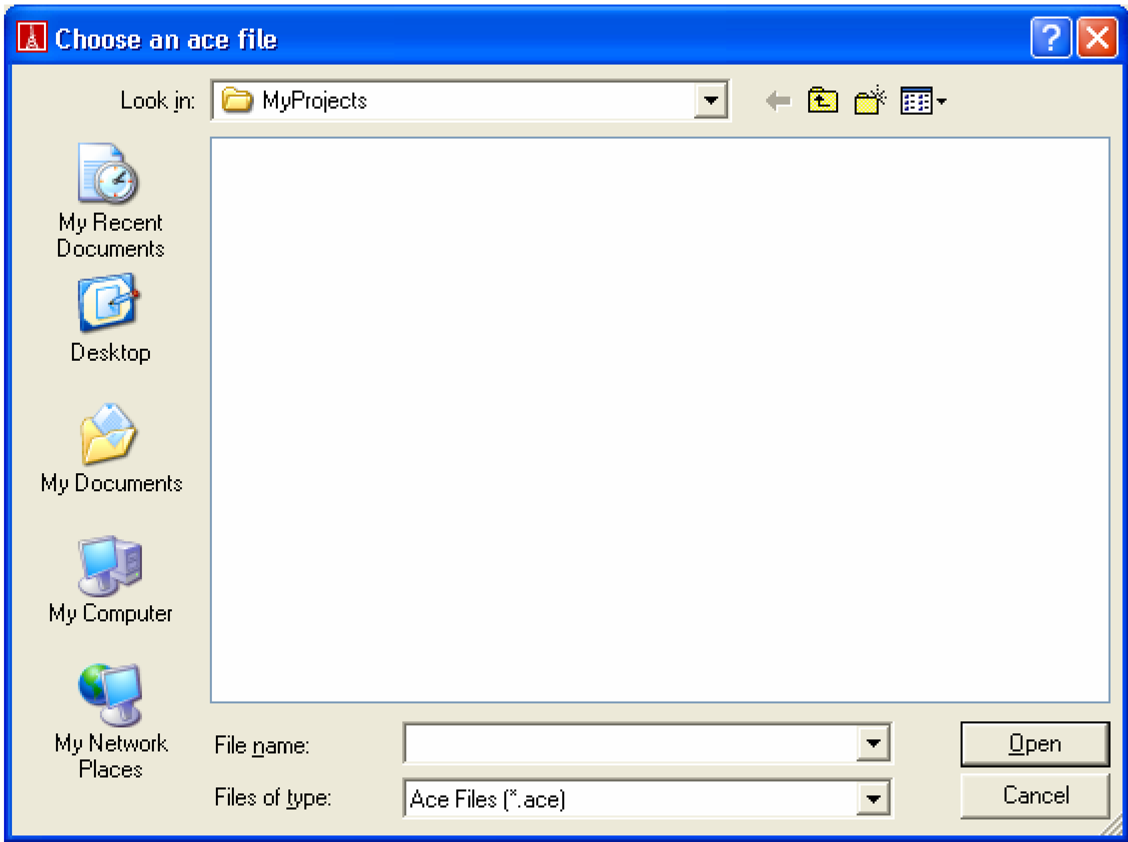
You then have the choice between three possibilities:

- **Create a new project**, which opens a wizard (depending on the kind of project you selected – basic application, graphical Windows application or graphical multi-platform application) helping you build your first ISE Eiffel application.
- **Open existing Ace** (control file), which opens a dialog asking you to select an Ace file (*.ace), meaning *Assembly of Classes in Eiffel*. This option is for advanced users.
- **Open compiled project**, to retrieve and open an existing project.

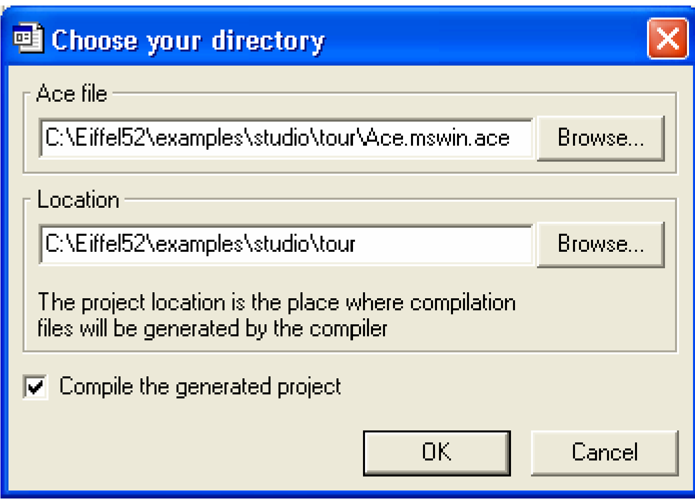
As a beginner, you would probably choose the first option (and you would be right!), which guides you through the different steps of starting a new ISE Eiffel project.

But as a guided tour, this introductory paper would like to go a little further in the use of EiffelStudio. Therefore we will use an example provided with your ISE Eiffel delivery – the *Tour* example – which is located in **$ISE_EIFFEL/examples/studio/tour**. This example is not meant to be a best-of-breed example of what you can do with EiffelStudio. It is a very simple example – the same kind of the well-known *HelloWorld* sample – whose only merit is to be already written and ready to be compiled!

Thus, we will choose the second possibility – open **existing Ace (control file)** and then click **Next**. This opens a file explorer dialog inviting you to choose an Ace file. We select the Ace file located in the Tour example folder: either **Ace.mswin.ace** (for Windows) or **Ace.unix.ace** (for UNIX, Linux or VMS):

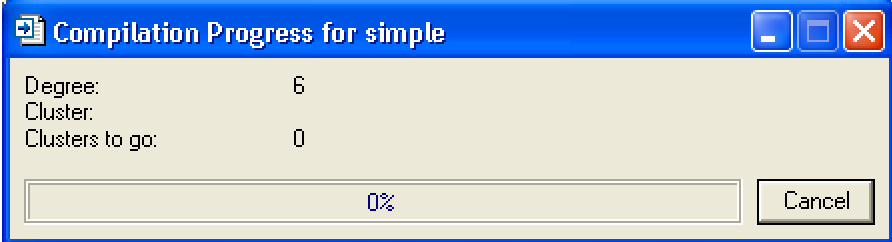


Another dialog then asks you to choose a **project folder**, i.e. the directory where your system will be generated – the default folder is the one containing the selected Ace file:



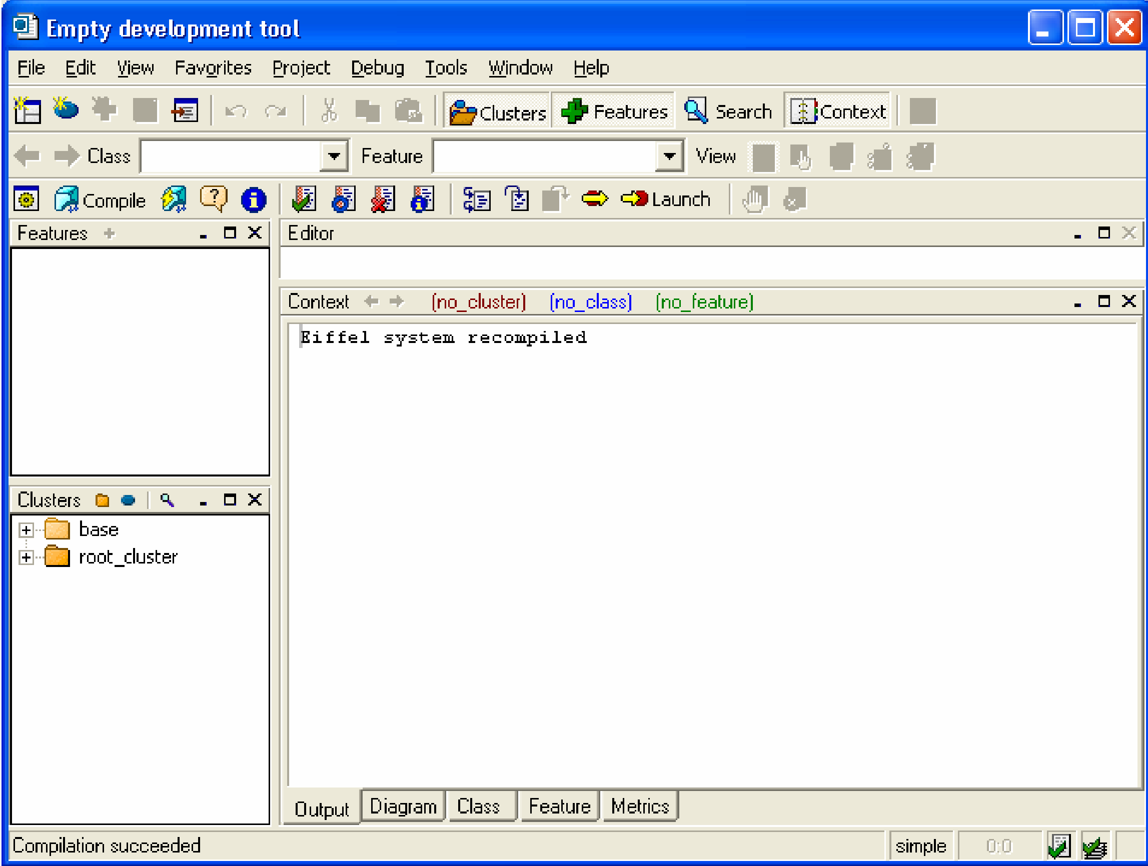
At this stage, you can also choose whether you want EiffelStudio to compile your system – the option is checked by default, but it may be convenient for you not to have compilation start at this point if you want to finalize a system for example (this terminology will be explained later). Then click **OK** to confirm.

We can simply keep the default settings. Thus, compilation starts. During this process, a progress bar appears, telling you about the successive compilation steps, usually called *degrees*:



The compilation is almost instantaneous here since the EiffelBase library had already been precompiled at the installation time. Thus there are only a few classes to compile. But even if you had not precompiled this core library before, it would not have taken a lot longer since ISE Eiffel compiler is very fast (just a few seconds).

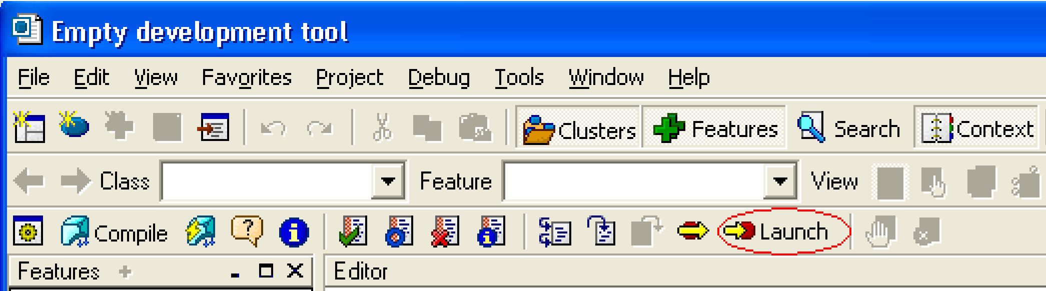
You are notified when the compilation process is complete:



You are now ready to run this extraordinary *Tour* system!

4. Executing a system

To run the compiled system, you simply have to click on the execution icon (*Run application and stop at breakpoints*) in the project bar or press F5:



We could also use the neighboring icon on the left (*Run application without stopping at breakpoints*) since we haven't set any breakpoints yet.

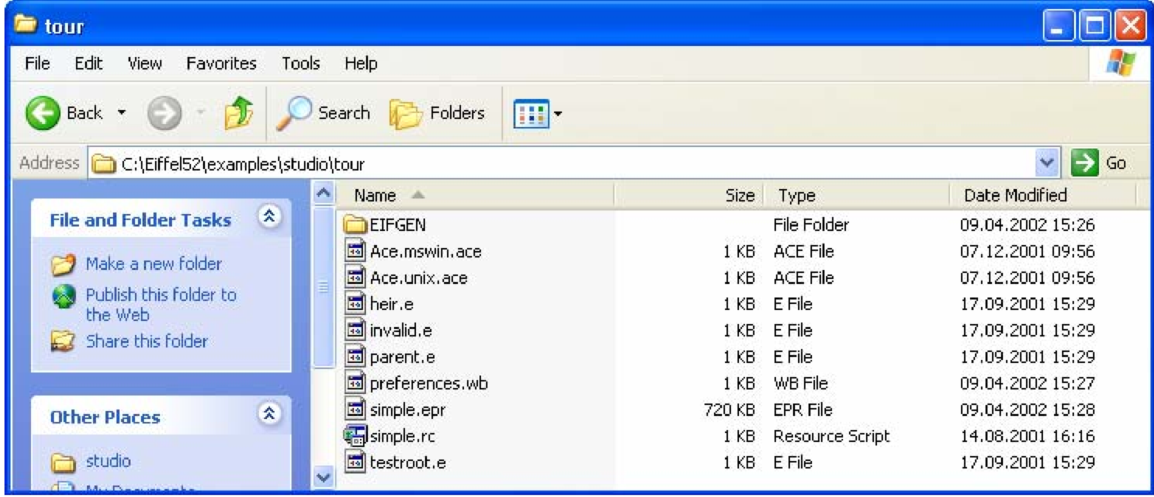
Here is the fancy result!

```
C:\Eiffel52\examples\studio\tour\EIFGEN\W_code\simple.exe

 ISE Eiffel spoken here
--------------------------------

In class HEIR
-------------------
Calling a routine of class HEIR
Showing an attribute of class HEIR

THIS IS SOME ADDED TEXT
In class PARENT
---------------
Message number 1


Press Return to finish the execution...
```

Note that the line Press *Return to finish the execution* would not appear if you executed the system from outside of EiffelStudio.

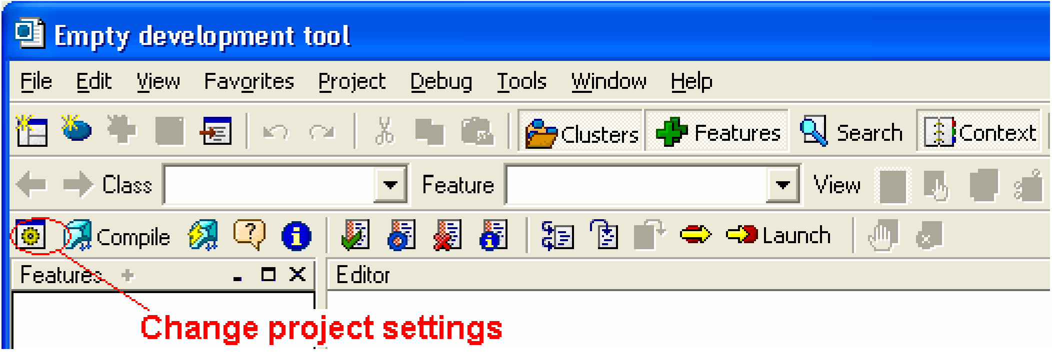
5. About the project directory

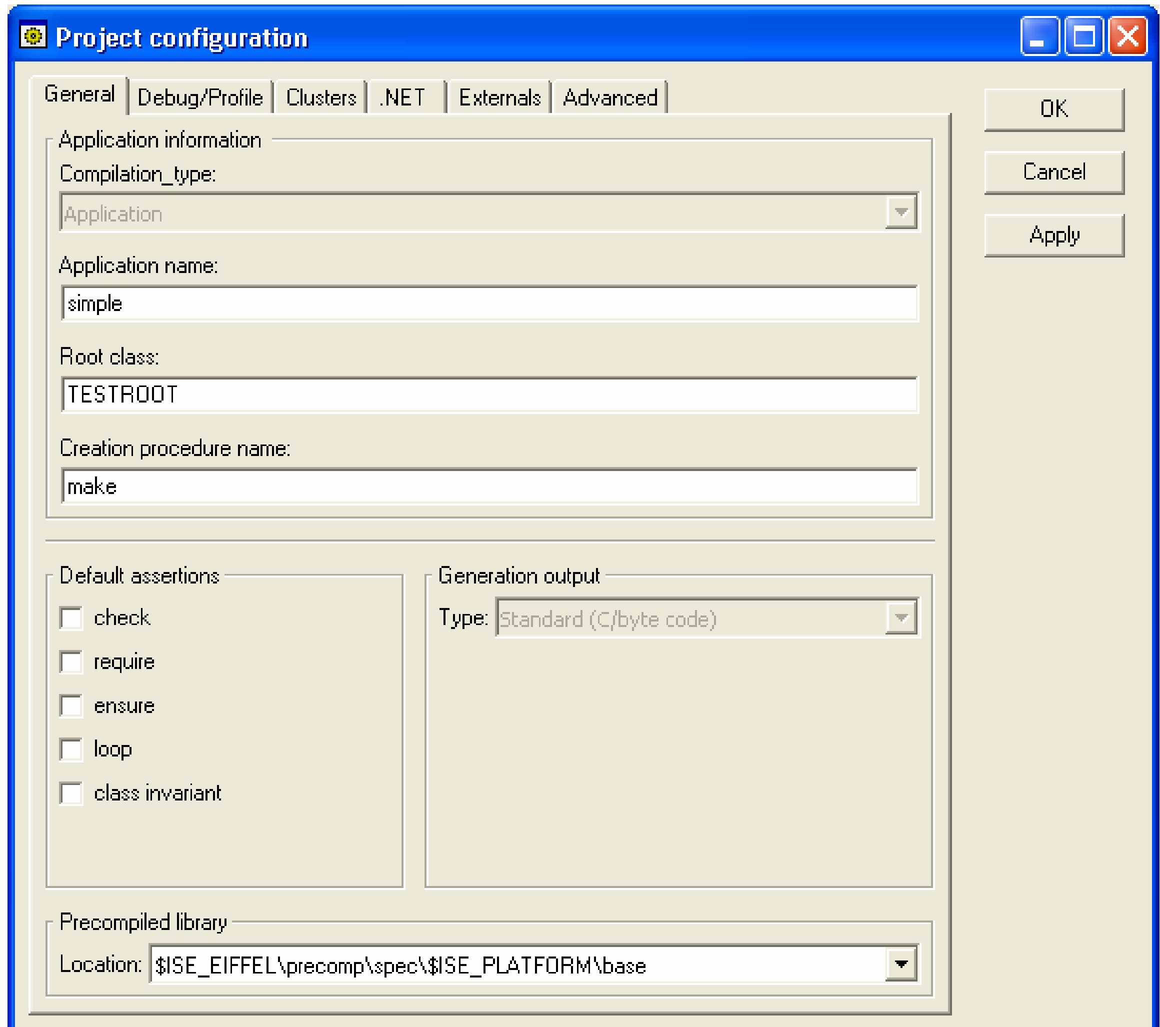
Now that we know what our Tour system is about, it may be interesting to have a closer look at the files generated during the compilation process. So let's move to our project directory: $ISE_EIFFEL/examples/studio/tour.



- The files with the extension .e – for "Eiffel" – (*heir.e*, *invalid.e*, *parent.e*, *testroot.e*) are the **Eiffel source files**. The file called *heir.e* contains the class HEIR, the convention being that the filename is the lower-case version of the class name (although this is not compulsory).
- The two files with .ace extension are the **control files** we mentioned before: the **Ace files** (one for Windows and one for Unix). They are written in an Eiffel-like notation called Lace (Language for Assembling Classes in Eiffel). These files contain all the information needed by ISE Eiffel compiler to build the system.

  Nevertheless it is seldom that you have to edit it. As a matter of fact, this file is automatically generated if you choose to create a new project with a wizard (see 3. Starting a project with EiffelStudio) and then you can modify it very easily through the **project settings** dialog:





- The file simple.epr is the **Eiffel Project Repository** that you will select in the initial screen to retrieve the corresponding project. Each successful compilation generates a .epr file.
- The subfolder called **EIFGEN** (for "EIFfel GENeration") is also a product of the compilation. It is generated and maintained by the compiler. Thus you should not change anything in this directory. If your curiosity brings you into this folder, you will notice that EIFGEN itself contains three subfolders: COMP, F_code and W_code. The W_code directory ("W" meaning "Workbench") contains the generated code. The compilation can produce three other subfolders named **Diagrams**, **Documentation** and **Metrics**. This will be discussed later in the tour.

6. Browsing facilities

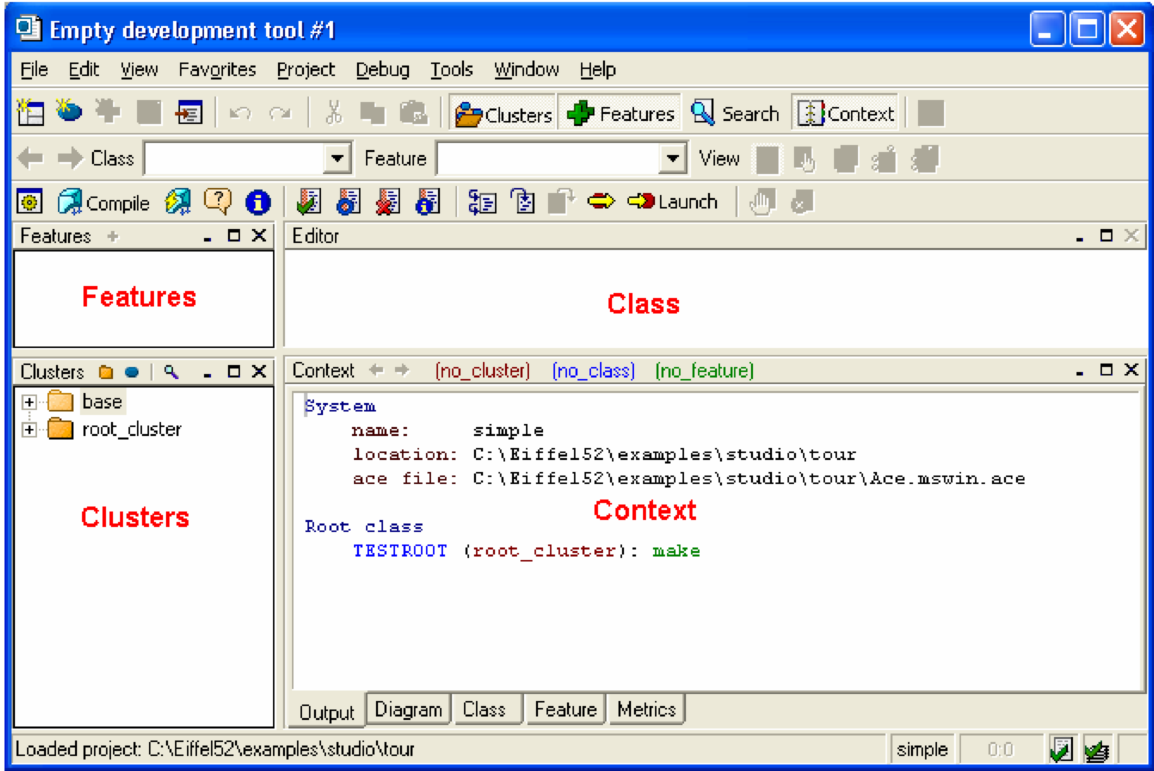
EiffelStudio's browsing facilities are some of the most innovative aspects of the tool.

You can browse (i.e. traverse the structure) whatever you do, whenever you want (or need) to: while editing, debugging, etc.

EiffelStudio provides different kind of browsing facilities. You may want to display a class text from its **name** or look at it from the **cluster** tree or just use **hyperlinks** to go to a software element.

EiffelStudio can also produce different **views** of the objects (the full text of a class, its interface only, exported features, etc.), including graphical views. We will come to this point later.

6.1 EiffelStudio's development window

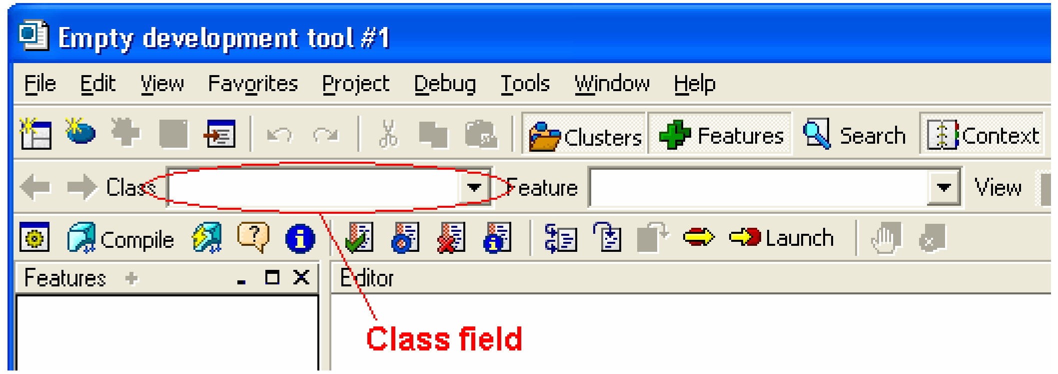


As suggested by the screenshot above, EiffelStudio's development window is divided into four parts: **Features**, **Class**, **Clusters** and **Context**. This is the default layout. You may also want to display a search pane or remove some of the previous ones. The class pane is the only one you cannot remove.

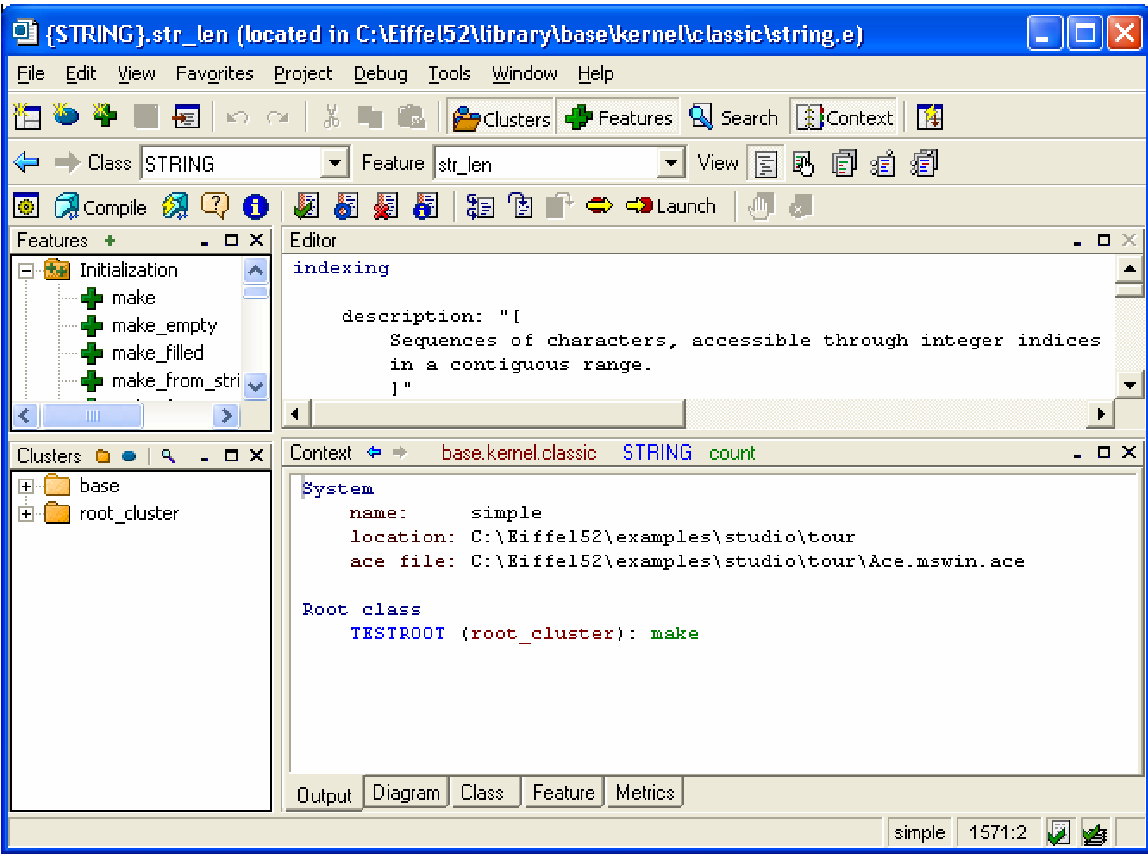
If you find that one development window is not enough, you can open other ones by selecting **New window** in the **File** menu or by using the Ctrl+N shortcut.

6.2 Browsing by class name

If you want to display a class text and you know its class name, you can just type its name into the Class field to retarget to this particular class:



Let's use the class STRING from the Kernel library of EiffelBase. Typing STRING (or the lower-case variant or a mix of them: EiffelStudio will automatically change it to the upper-case form) retargets the development tool to the specific class STRING:



Note that this browsing facility is independent from any file location: you do not need to worry about where the class STRING (or any other one) is stored in the files of your computer.

If you look closer to the development window, you can see that the four panes have changed:

- The class pane contains the full text of class STRING (this part is editable).

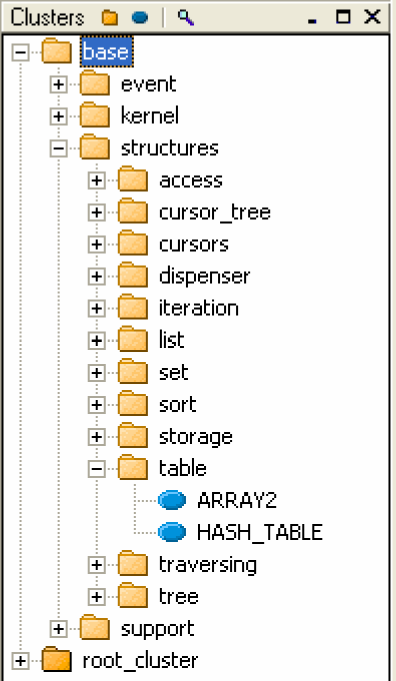
- The features pane contains the features of class STRING sorted by feature clauses. Each of these features can be retargeted to display the specific feature text.
- The clusters pane is a tree view of the clusters in your compiled system.
- The context pane displays some general information about your system. In particular, the root class and root class creation routine are "pick-and-dropable" (see next section for more details). The tabs that you see (Output, Diagram, Class, Feature and Metrics) give you different views of the system. The tour comes to this later.

6.3 Browsing from Cluster tree

In the previous section, we assumed that we knew the name of the class of which we wanted to display the source text. How can we then do when we know nothing about the system?

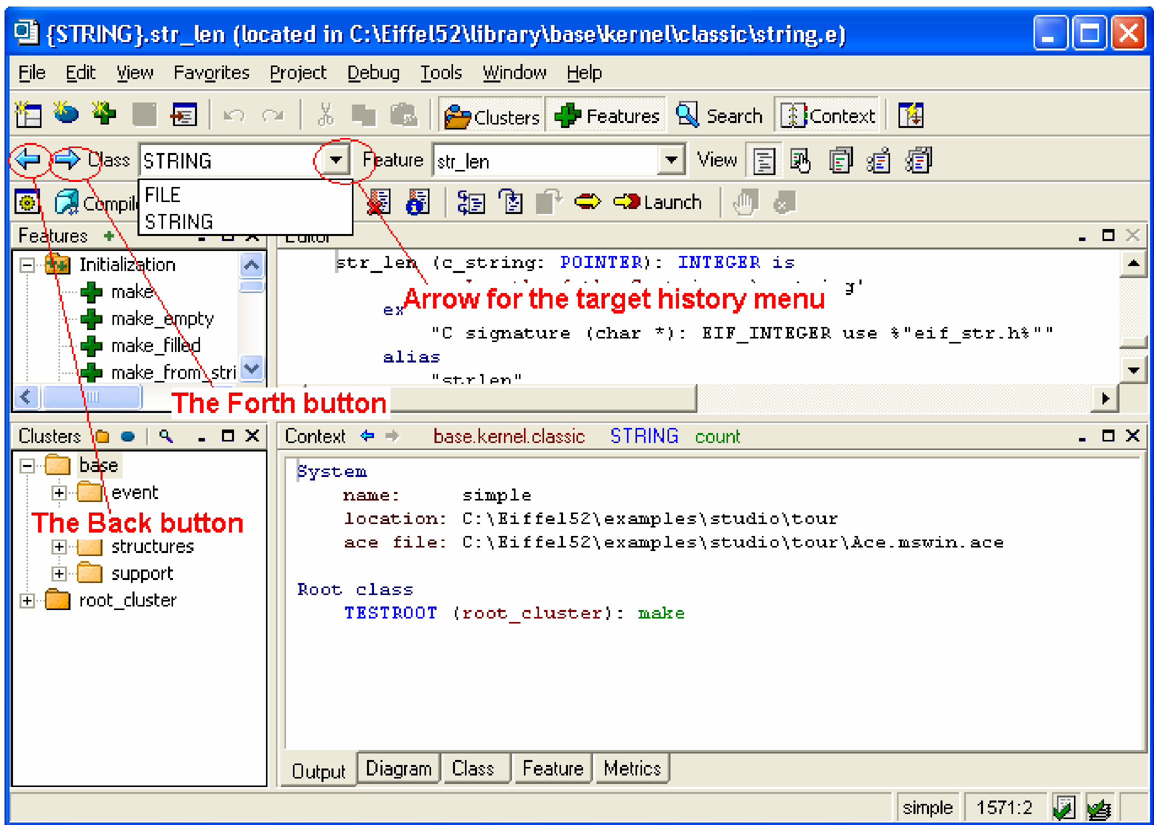
One possible answer is to use the cluster tree. This is a kind of "file explorer", which enables you to browse through the clusters of the system and the classes it may contain. You just have to click the little + sign to the left of each cluster name to expand it and discover its content. The clusters may contain either subclusters or classes (represented as ellipses or "bubbles", as in the Business Object Notation – BON).

The following screenshot gives you part of the structure of the EiffelBase library:

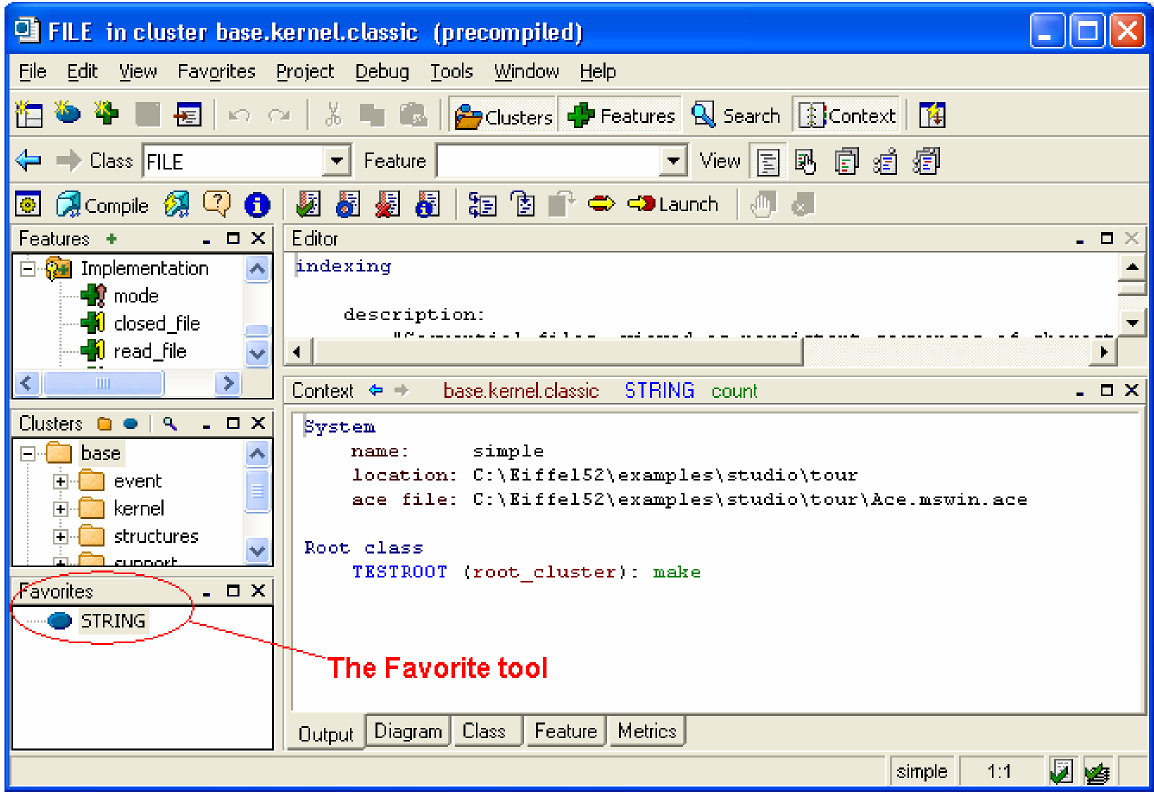


6.4 Browsing from history

EiffelStudio also provides you with very useful facilities to move back and forth between classes. You can use the **Back** and **Forth** buttons to retarget the tool to the class you visited previously. If you want to go directly to a specific class you have already visited, then you can use the **target history** menu as shown by the following figure, which remembers the last ten visited classes:



Besides, EiffelStudio provides the ability to save targets to your "Favorites" or bookmarks in a very simple way: just go to the **Favorites** menu and select **Add to favorites**. To go back to a favorite class, just select the item **Show favorites** in the same *Favorites* menu, which brings the Favorites tool (see next page). Then click the "bubble" corresponding to the class you want to display.

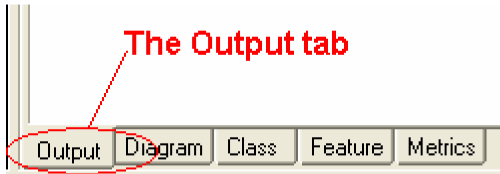


Remark: You can start a new development window on a particular class by finding the class somewhere in the interface and control-right-click on it. This may be useful if you want to keep track of several things in several classes.

6.5 Class views

As mentioned before, the context tool (more precisely, the tabs of the context tool) gives access to several class views.

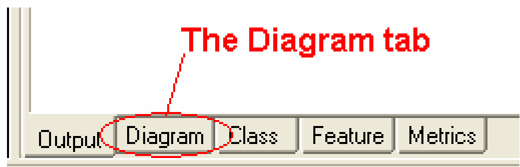
6.5.1 The Output tab



The Output tab gives you access to general information about the system (i.e. system name, project directory, location of the Ace file, the root class name, cluster and creation routine name).

It also displays the compiler output (hence its name), that is either "Eiffel system recompiled" (see the appendix for further explanations about Eiffel compilation) or useful information to find out eventual errors in the software.

6.5.2 The Diagram tab

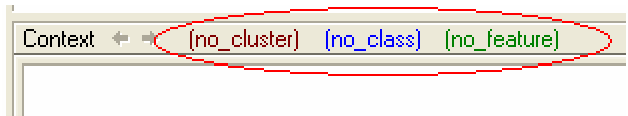


The Diagram tab gives you access to the **Diagram Tool**, which provides you with graphical views of your system. You can use it both to *examine* systems already built or under development, and to *build* these systems graphically (see 12. Graphics-based design for more details).

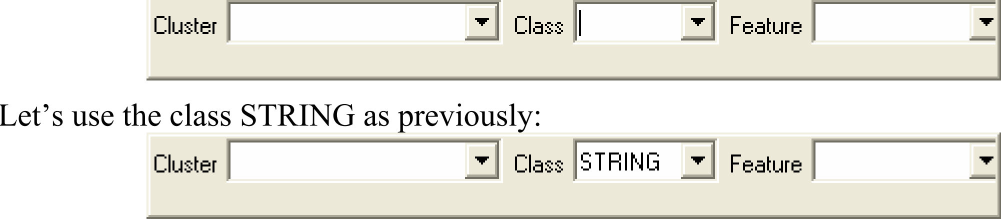
As a matter of fact, EiffelStudio supports **complete seamlessness** and **reversibility** between the Diagram Tool's graphical view and the text view of other EiffelStudio tools: whenever you make a change or addition with the graphical tool, EiffelStudio updates the text; and when you change the text, EiffelStudio updates the Diagram Tool's representation on the next compilation.

The Diagram Tool uses the BON notation (*Business Object Notation*) to represent Eiffel systems; BON is a simple, clear and self-explanatory notation that you will learn in a few minutes just by looking at some examples.

To visualize the hierarchy a particular class belongs to, just click the context bar shown below:

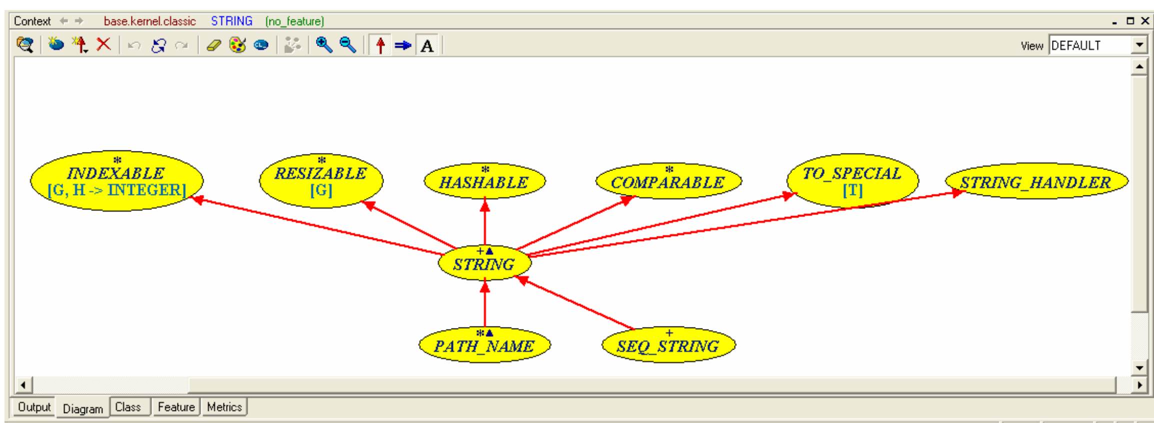


The following bar appears, inviting you to type the name of the class (or cluster, or feature) you want to examine:



Let's use the class STRING as previously:

The corresponding inheritance diagram appears in the context tool:



Notation:

An object **class** is represented by an ellipse (sometimes referred to as "bubble"), which contains the class name and formal generics of the Eiffel class it represents:

It also displays information about properties of a class. These properties are:

- *Deferred*: The class is declared as deferred.

- *Effective*: The class is not deferred, but has at least one deferred parent, or redefines one ore more features.

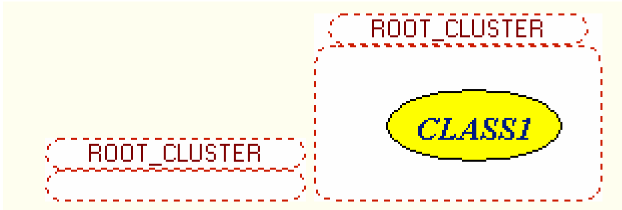
- *Persistent*: The class inherits STORABLE or has indexing tag persistent.

- *Interfaced*: The class has one ore more external features.

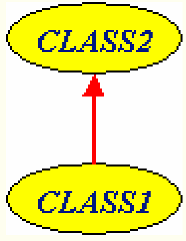
- *Reused*: The class resides in a precompiled library or in a library cluster.

- *Root class*: The class has the program's entry point.

A **cluster** is a collection of classes and subclusters. It is represented by a stippled, rounded rectangle. The cluster name is in a similar rectangle, by default located at the top of the cluster just outside of it. A cluster can be iconified, which means it is minimized to take up as less space as possible, by double clicking on the cluster label as shown by the following figure:

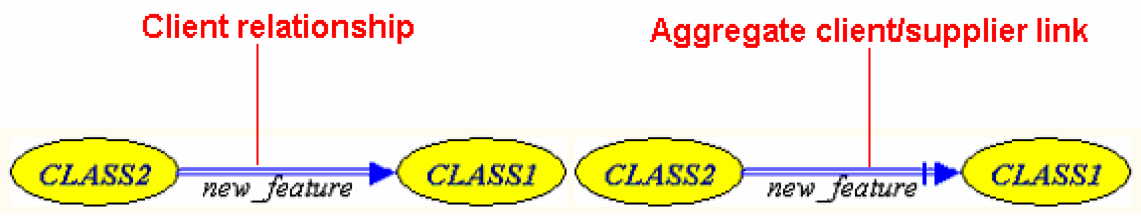


Inheritance between classes is represented by a red arrow from a class to its parent. Typically, inheritance links point upwards:



Client relationships between classes are represented by a blue arrow with a double line from the client to the supplier. Typically, these links point to the right.

A special kind of relationship is the aggregate client/supplier link. Aggregate means that the supplier is an integral part of the client. In the class text, it corresponds to expanded features. In the diagram, an aggregate link has a little line along the back side of the arrow.

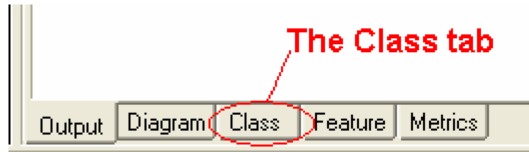


The Diagram toolbar:

- *Target to cluster or class*: Drop a class or a cluster on this icon to build the corresponding diagram.
- *Target to enclosing cluster*: Re-centers the diagram on the parent cluster.
- *Create a new class*: Pick from this button and drop on the diagram to add a new class.
- *Create new inheritance links*: Selecting this button allows you to create inheritance links by picking from one class and dropping on the other.
- *Create new client-supplier links*: Selecting this button allows you to create client-supplier links by picking from one class and dropping on the other.
- *Create new aggregate client-supplier links*: Selecting this button allows you to create aggregate client-supplier links by picking from one class and dropping on the other.
- *Delete*: Drop items in this hole to remove them from the system.
- *Undo last action*: Click to undo the last action.
- *History tool*: Click to open the **History tool**.
- *Redo last action*: Click to redo the last undone action.
- *Remove figure*: Drop figures in this hole that you want to remove from the view only.
- *Change color*: Drop class bubbles in this hole to change their color.

- *Change class name and generics*: Drop class bubbles in this hole if you want to change a name and/or formal generics.
- *Put handles on a link*: Click on this button to toggle right angles on all links in the view. Drop a link on it to bring up the link tool for that link.
- *Select depth of relations*: Click to bring up the **Select depth** dialog.
- *Include all classes of cluster*: Drop a cluster in this hole to show all classes of the cluster in this view.
- *Zoom in*: Enlarges the diagram.
- *Zoom out*: Shrinks the diagram.
- *Show/hide inheritance links*: Toggles visibility of all inheritance links in the view.
- *Show/hide client-supplier links*: Toggles visibility of all client-supplier links in the view.
- *Show/hide labels*: Toggles visibility of all client link labels in the view.
- *Delete current view*: Click to remove the current view from the list of views for current class or cluster.
- *Export diagram to PNG*: Click to generate a .PNG file from current view.

6.5.3 The Class tab



The Class tab, which you find at the bottom of the Context tool, gives you access to many forms of information about the current class.

The default view shows the ancestors of the current class:

Context base.kernel.classic STRING (no_feature)

Ancestors of class string

```
STRING
     INDEXABLE [G, H -> INTEGER]
          TABLE [G, H]
               BAG [G]
                    COLLECTION [G]
                         CONTAINER [G]
                              ANY
     RESIZABLE [G]
          BOUNDED [G]
               FINITE [G]
                    BOX [G]
                         CONTAINER [G]...
     HASHABLE
          ANY
     COMPARABLE
          PART_COMPARABLE
               ANY
     TO_SPECIAL [T]
          ANY
     STRING_HANDLER
          ANY
```

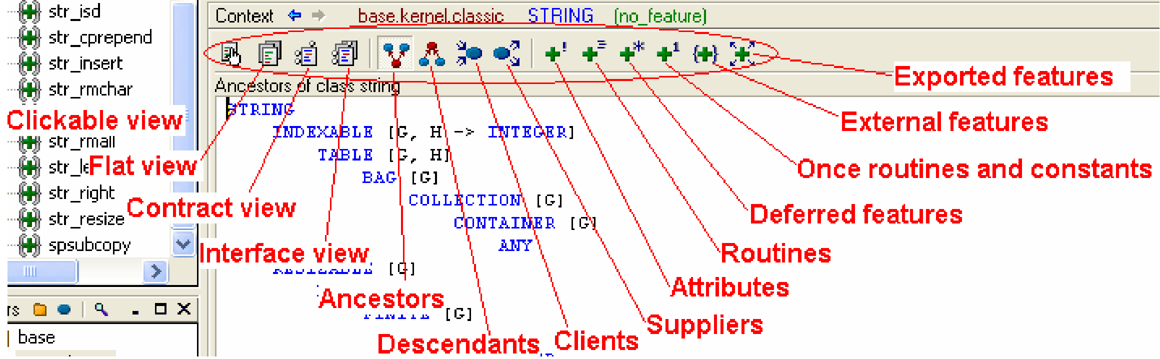
Output | Diagram | Class | Feature | Metrics

This shows that STRING is an heir of INDEXABLE and RESIZABLE and HASHABLE, etc. (example of multiple inheritance) and that INDEXABLE is itself an heir of TABLE (repeated inheritance).

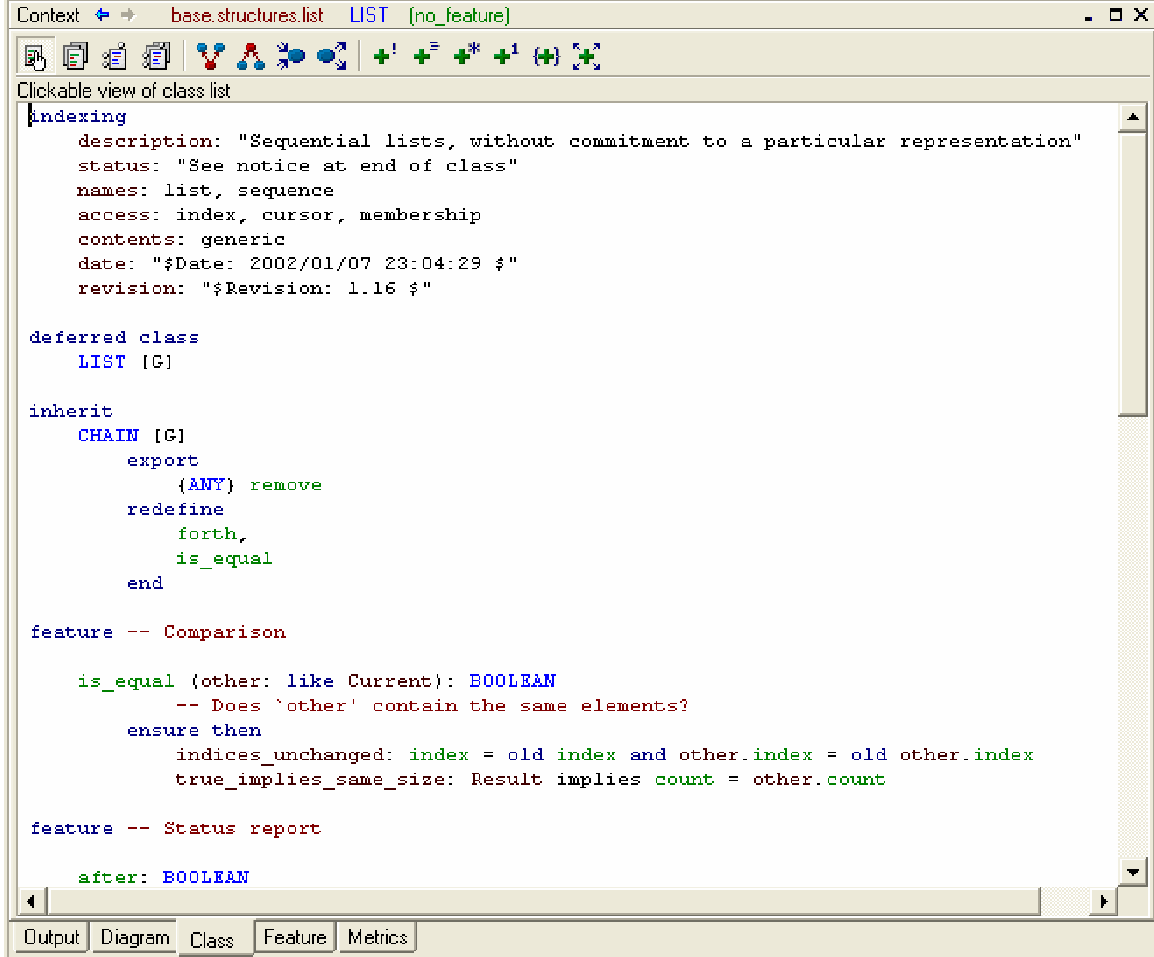
If a class appears several times (because or multiple or repeated inheritance), the Class tool only displays its ancestry once. This is materialized by the three dots, …, as illustrated here for the second occurrence CONTAINER.

Remark: The class names that appear in the different views are hyperlinks. Thus, you can use them to retarget the development window to the corresponding class.

The following figure summarizes the various views available from the toolbar:



- The **Clickable view** is mostly the same as what you can find in the editor, i.e. the class text. The main differences are that you cannot edit it and that it does not display the features body. Besides, the clickable view is automatically formatted and uses nice colors and fonts to distinguish keywords, identifiers and other syntactical elements:

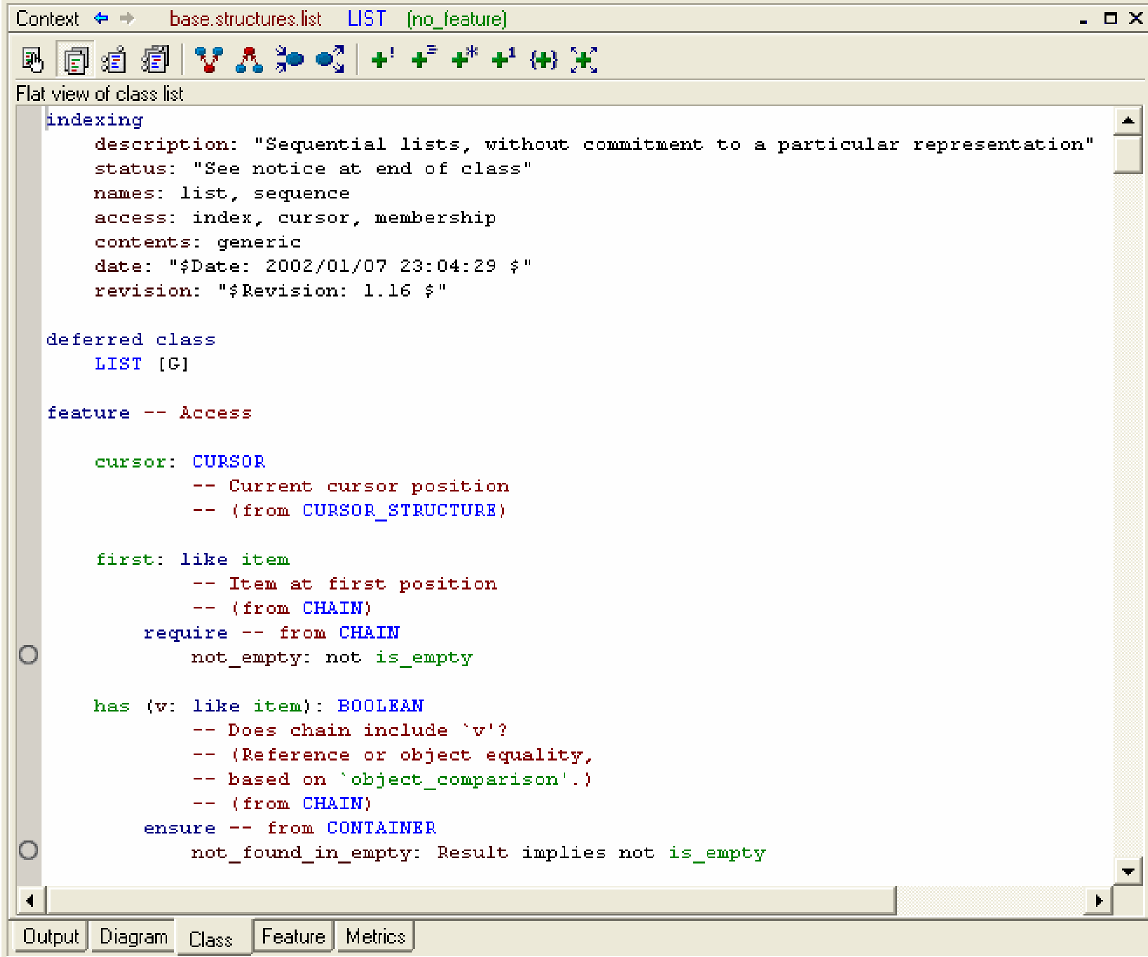


Remark: This view is called "clickable" since every syntactical element is a hyperlink (i.e. "clickable"), which you can use for browsing.

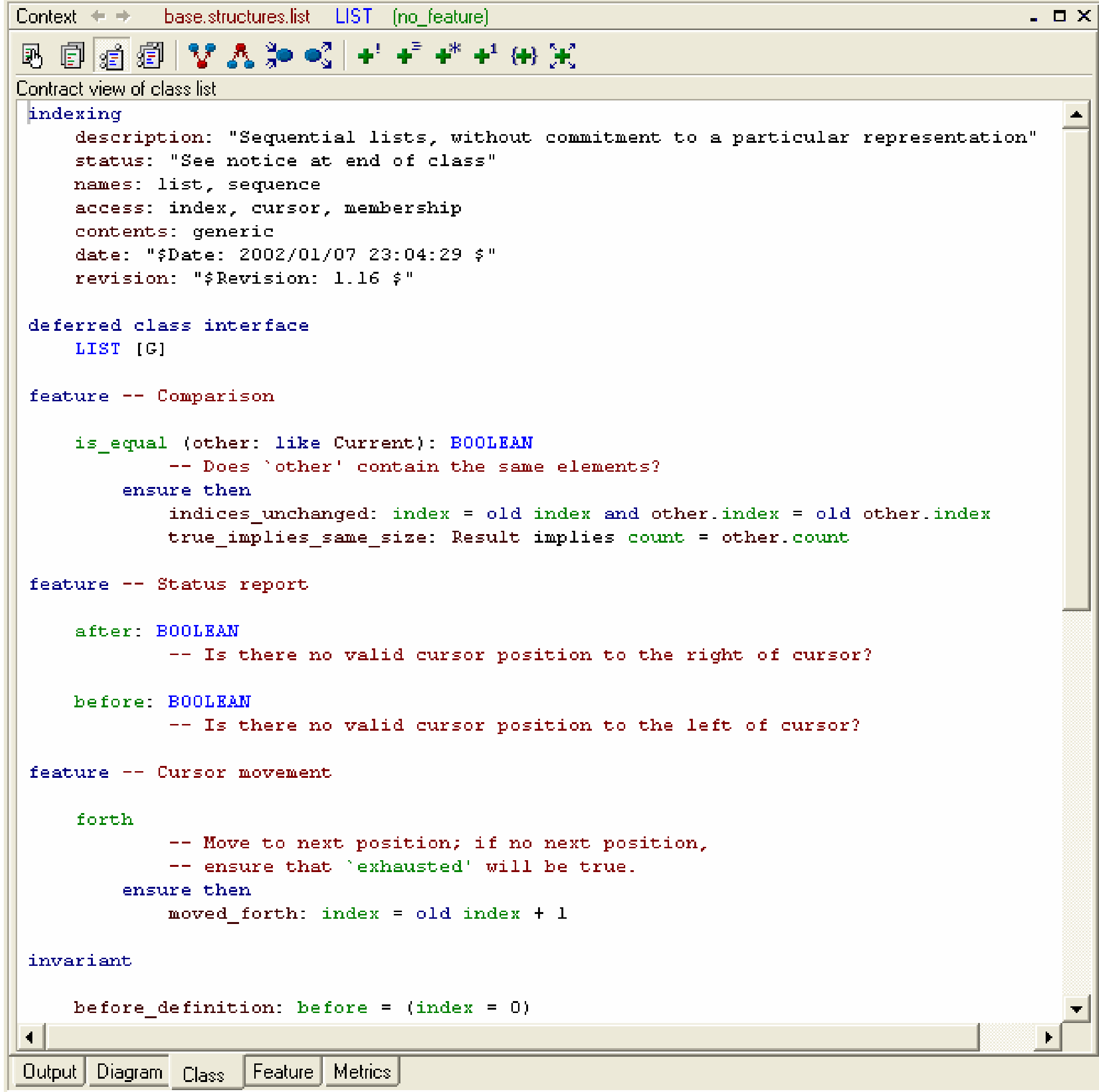
- The **Flat view** displays all the features for the current class, i.e. including both written-in and inherited features as well as contracts: the flat form's invariants includes all clauses from ancestors' invariants and the preconditions and postconditions are expanded to take *require else* and *ensure then* clauses into account.

  The first two features appearing in the next figure, *cursor* and *first*, are indeed inherited from ancestors; they are not declared in class LIST itself. Note that EiffelStudio explicitly shows the origin of non-immediate features (i.e. inherited features) by adding a line of the form: -- (from CLASS_OF_ORIGIN).

  Remark: The flat view is also the only view, which displays breakpoints (see section 9 for more details about debugging facilities).



- The **Contract view** (also known as **short form** of a class) displays the contracts of all written-in features of the current class: it gives the interface properties of a class. It discards any non-exported feature and for the remaining ones, it discards the implementation – the *do* or *once* clauses but keeps the feature header (its signature), the comments and contracts (expected contracts referring to a non-exported feature).
  The next screenshot gives the contract view of class LIST.



- The **Flat Contract view** or **Interface view** (or **flat-short form**) applies the same rules as the Contract view. The only difference is that it both displays the contracts of all written-in and inherited features of the current class.

The Contract tool of EiffelStudio also provides some useful information about the **features** of the class. In particular, the **exported features** view is really interesting whenever you need a client/supplier relationship between classes. Here is the example for class LIST:

Context base.structures.list LIST (no_feature)

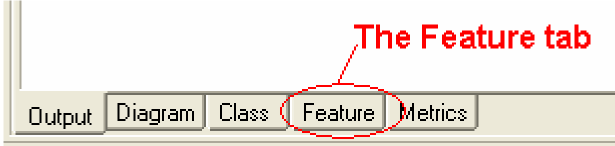
Exported features of class list

```
Class CONTAINER [G]:

     changeable_comparison_criterion: BOOLEAN
     compare_objects
     compare_references
     empty: BOOLEAN   (obsolete)
     object_comparison: BOOLEAN

Class COLLECTION [G]:

     extendible: BOOLEAN
     fill (other: CONTAINER [G])
     is_inserted (v: G): BOOLEAN
     prunable: BOOLEAN
     wipe_out

Class BAG [G]:

     extend (v: G)
```

Output Diagram Class Feature Metrics

As you may have guessed, the highlighted classes and features names are "clickable" hyperlinks, which is one of the nicest benefits of such views.

6.5.4 The Feature tab



The **Feature tab** gives access to basic information about the selected feature.

The default view of the chosen feature is **flat**. Like the flat view of a class, it gives the text of a feature with no implementation but includes comments and both immediate and inherited contracts (use of **require else** and **ensure then**).

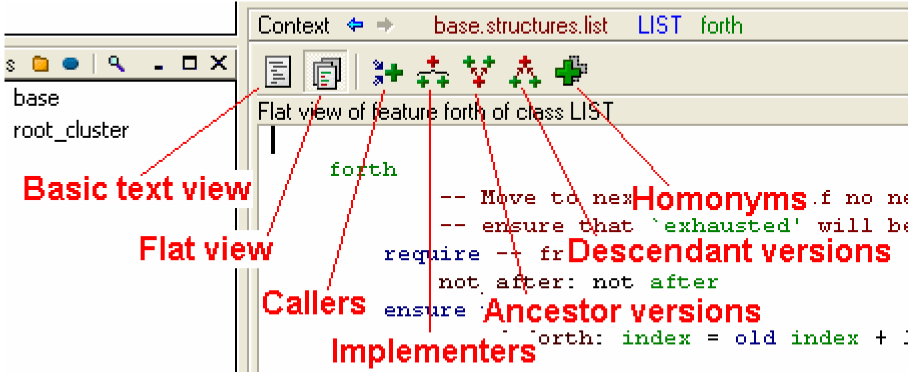
Context base.structures.list LIST forth

Flat view of feature forth of class LIST

```
     forth
             -- Move to next position; if no next position,
             -- ensure that `exhausted' will be true.
         require -- from LINEAR
             not_after: not after
         ensure then
             moved_forth: index = old index + 1
```

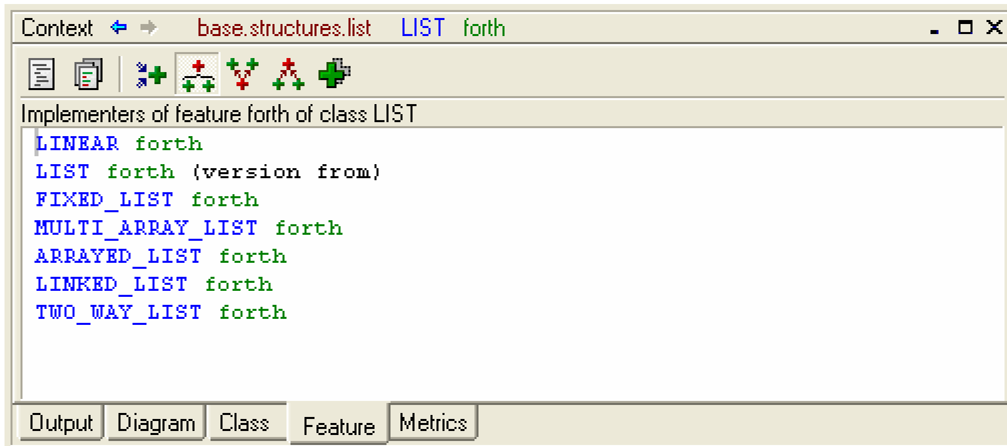
Output Diagram Class Feature Metrics

The following figure shows the other **feature views** available from the Feature tab:

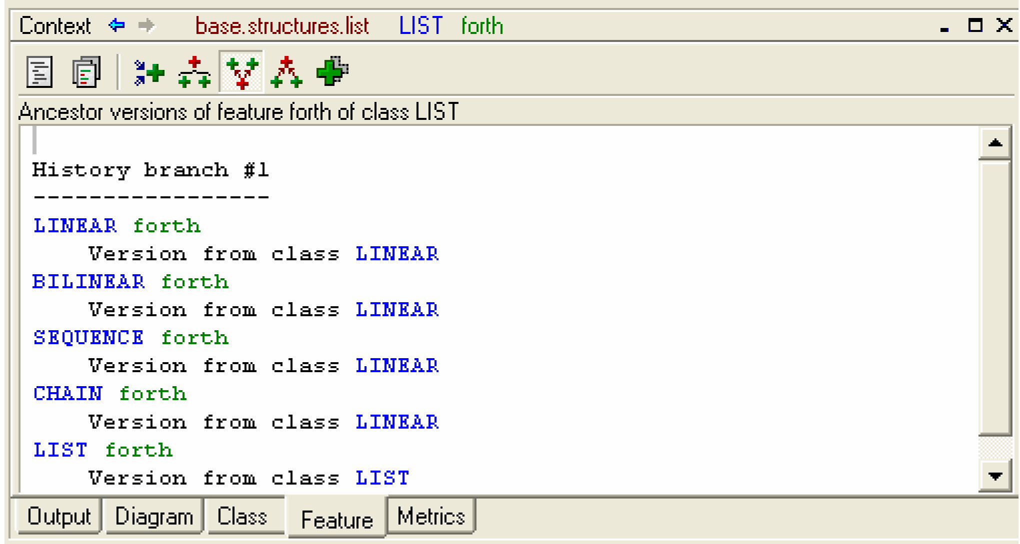


- **Callers**: This view shows all the places in the system that call the routine or one of its redefinitions – which is very useful for debugging.

- **Implementers**: This view shows all the ancestors and descendants of the class defining the current feature, which provide a separate version of this feature. Here is the example of feature *forth* defined in class LIST:

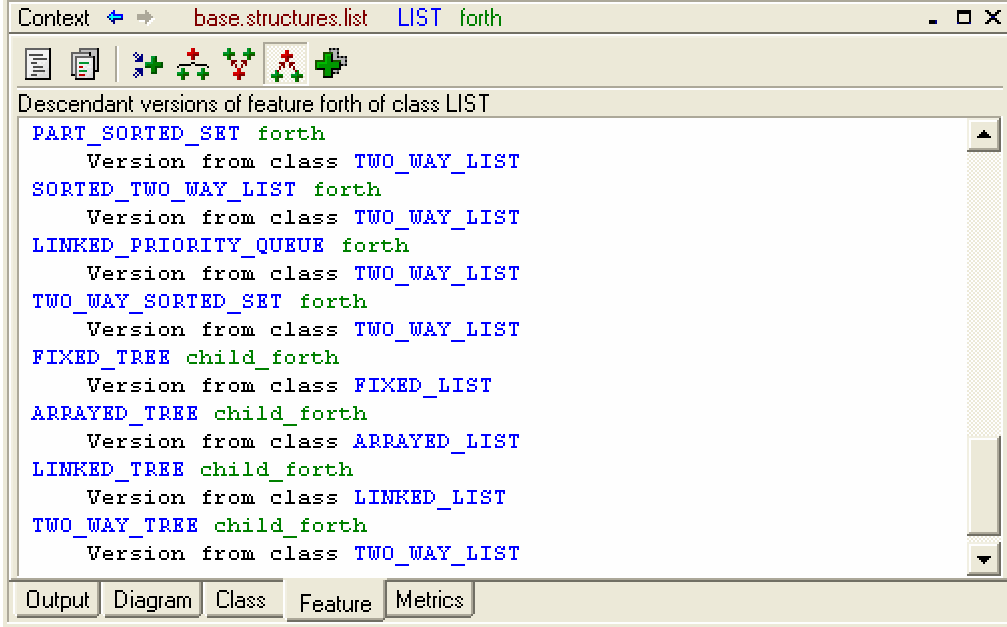


- **Ancestor versions**: For each ancestor of the current class (let's say LIST) that has a version of feature *forth*, this view indicates the name of that feature, which could be different from *forth* because of the renaming mechanism although it is not the case in our example – as shown by the figure below.



Remark: This view mentions the history branch number, since there could be several of them in case of feature merging (combining different features inherited from different parents).

- **Descendant versions**: In the same way as the ancestor versions view, this one gives the names of the selected feature in the descendants of the current class. For instance, the feature *forth* of class LIST is renamed *child_forth* in class FIXED_TREE:



- **Homonyms**: This view displays all the features of the system, which have the same name as the current feature (whether they are related or not).

6.5.5 The Metrics tab

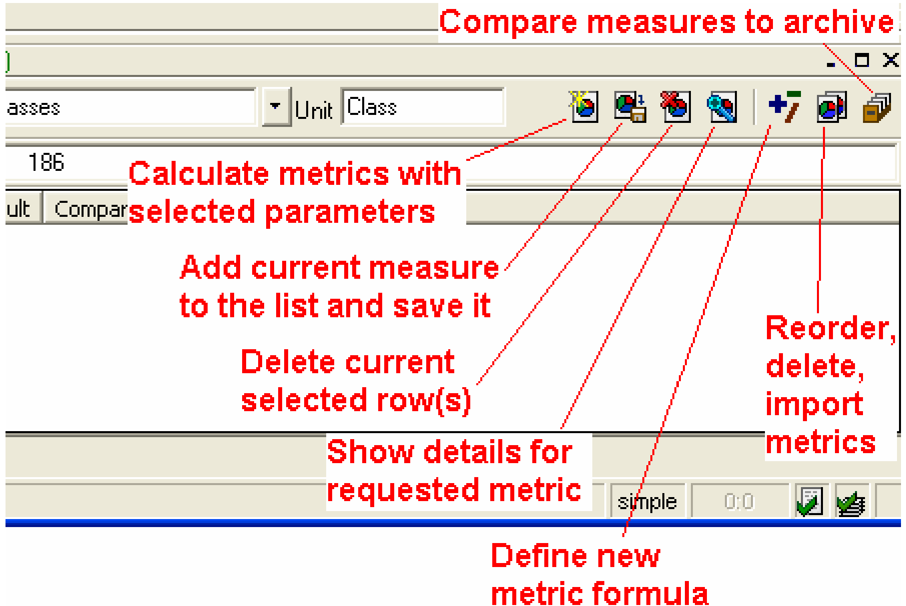
The Metrics tab gives access to the **Metric Tool**, which provides many facilities like computing measures over a project, including smaller scopes, defining new metrics according to users' needs and handling archives to compare projects.

Definitions:

- A **metric** is a quantitative property of software products or processes whose possible values are numbers. Each metric is relative to a certain **unit**, specified as part of its definition.
- A **measure** is the value of a metric for a certain product or process.
- Any metric applies over one or more scope types. A **scope type** is a type of product or process over which the metric is measured. A **scope** is an instance of a scope type. For instance, a given cluster is an instance of the scope type *cluster*.
- To **compute a measure** is to apply a certain metric over a certain scope of an applicable scope type.

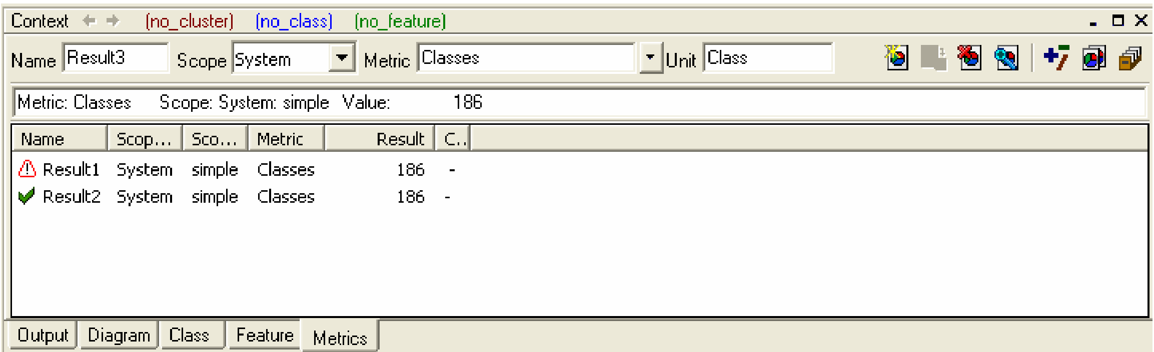
Measuring:

To perform a measure, you simply choose a metric and a scope, and request the computation, by clicking the first button of the toolbar ("***Calculate*** *metrics with the selected parameters*"; this operation allows evaluating the selected metric over the selected scope):



The measure is then displayed in the provided text field, but it is not saved yet. To save the measure, just click the second button ("*Add current measure to the list and* ***save*** *it*").

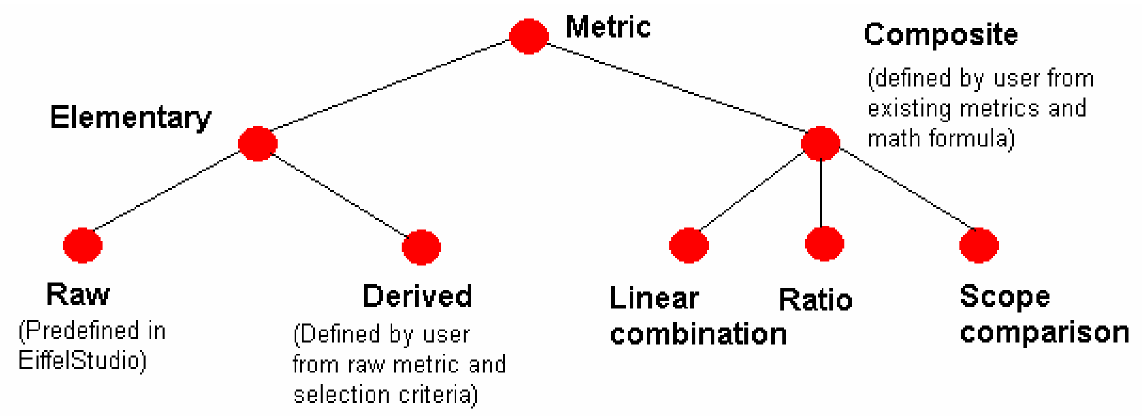
The last calculated measure appears in the multi-column list and is saved in a file automatically generated by the interface. This file (*metric_file.xml*) is located in the **Metric subfolder** of the current project location.



Remark: After having recompiled a project, the recorded measures may no longer be accurate. The icon ⚠ replaces ✔ to inform users that measures have been calculated before the last recompilation, and therefore may be obsolete. Updating measures is possible by right clicking on the multi-column list and choosing an item from the context menu. Measures will be updated both in the multi-column list and in the file they were saved in.

Defining new metrics:

The EiffelStudio metric framework distinguishes between different kinds of metrics. It provides a number of predefined metrics but also enables users to define their own metrics in terms of the predefined ones according to a taxonomy illustrated as follows:



Metrics are divided into elementary and composite:

- An **elementary** metric measures the number of occurrences of a certain pattern in the product or process. (For instance, the number of precondition clauses in a class.)
- A **composite** metric, defined by a user of EiffelStudio, applies a mathematical or logical formula involving other metrics (elementary or previously defined metrics.)

Elementary metrics are themselves divided into raw and derived metrics:

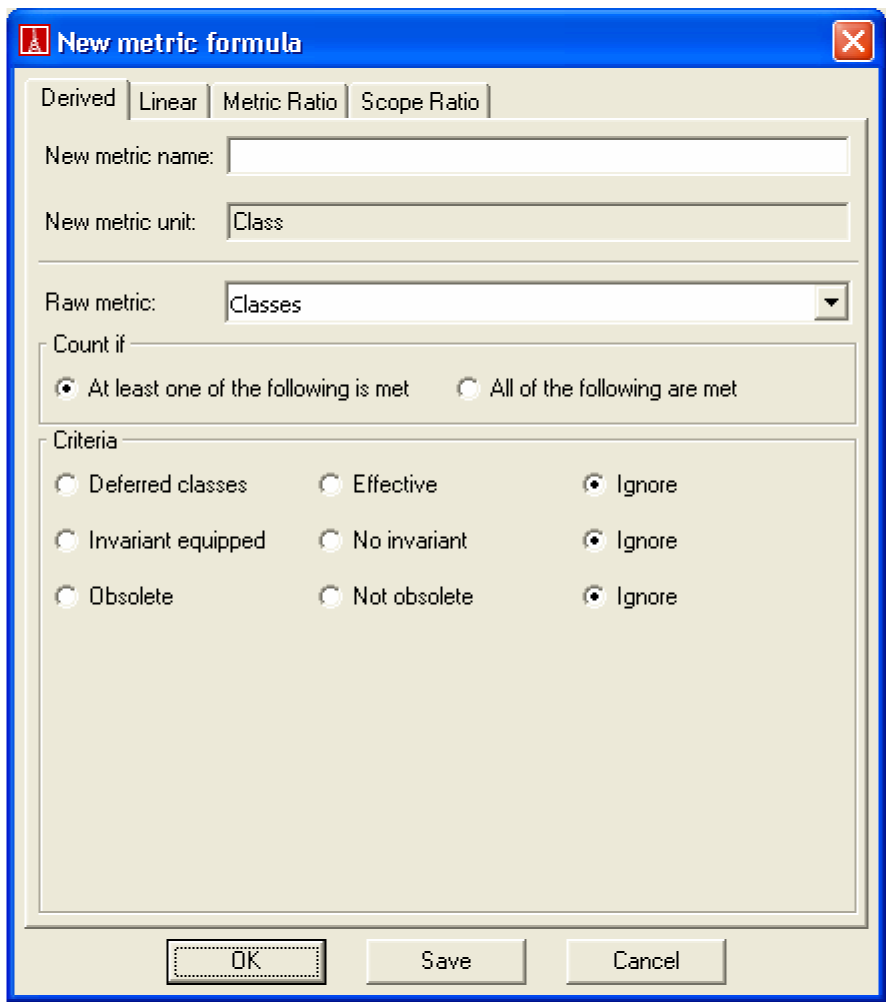
- **Raw** metrics are simple counts, built-in into EiffelStudio, of occurrences of certain basic elements. (For example, *Classes*, the number of classes, is a raw metric).
- A **derived** metric is an elementary metric defined from a raw metric by counting only the patterns satisfying a certain combination of its selection criteria. (A **selection criterion** for a raw metric is a property, with a fixed set of possible values – two or more, characterizing the patterns or events being counted by the metric.)

A **composite** metric is a metric whose values are defined by a mathematical formula involving other metrics (elementary or previously defined composite metrics). They are classified into three categories: linear, ratio and scope comparison metrics:

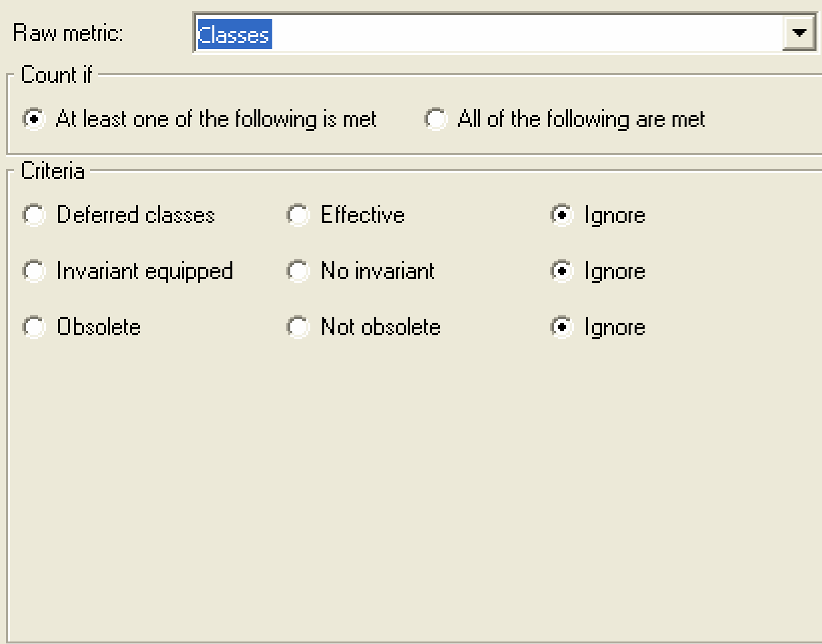
- **Linear** metrics are of the form $\sum k_i . m_i$, where k_i are real values and the m_i existing metrics with the same unit, other than RATIO.
- **Ratio** metrics are of the form m_1 / m_2 where both m_i are previously defined metrics, not necessarily with the same unit, neither of which a RATIO. The resulting unit is a RATIO.
- **Scope comparison** metrics measure the ratio of the value of a given non-ratio metric over two different scope types. (For example, by choosing the metric *Classes* and the scope types “cluster” and “system” we can measure the proportion of classes in a system that belong to the current cluster.)

Let’s now examine how to define a **new derived metric**:

To define a new metric, just click the “*Define new metric formula button*”. A new dialog comes up inviting you to select the kind of metric you want to define and the corresponding relevant criteria (see next page).



First, enter a name in the "*New metric name*" field. Then, select a raw metric. Regarding the selected metric, you will get one of the following panes in the "*Criteria*" frame:



Raw metric: Dependents

Count if

- At least one of the following is met
- All of the following are met

Criteria

Include self (dependent or not)	Exclude self unless dependent	
Direct clients	Direct and indirect clients	Ignore
Direct suppliers	Direct and indirect suppliers	Ignore
Direct heirs	Direct and indirect heirs	Ignore
Direct parents	Direct and indirect parents	Ignore

Raw metric: Features

Count if

- At least one of the following is met
- All of the following are met

Criteria

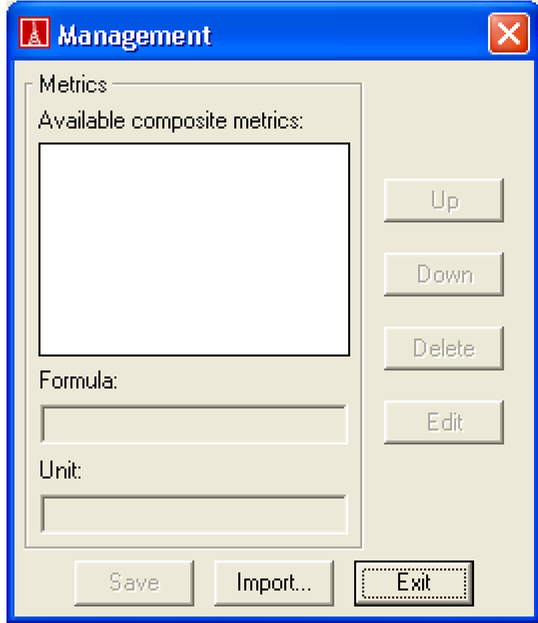
Attributes	Routines	Ignore
Queries	Commands	Ignore
Functions	Not functions	Ignore
Deferred features	Effective	Ignore
Exported	Restricted	Ignore
Inherited	Not inherited	Ignore
Precondition equipped	No precondition	Ignore
Postcondition equipped	No postcondition	Ignore

Note the large number of selection criteria applicable to the raw metric Features (number of features), reflecting the many angles under which it is possible to classify features.

If you want to save the metric definition without exiting the interface, click *Save*, otherwise, click *OK* to save and exit. If you want to define another derived metric, click to reset.

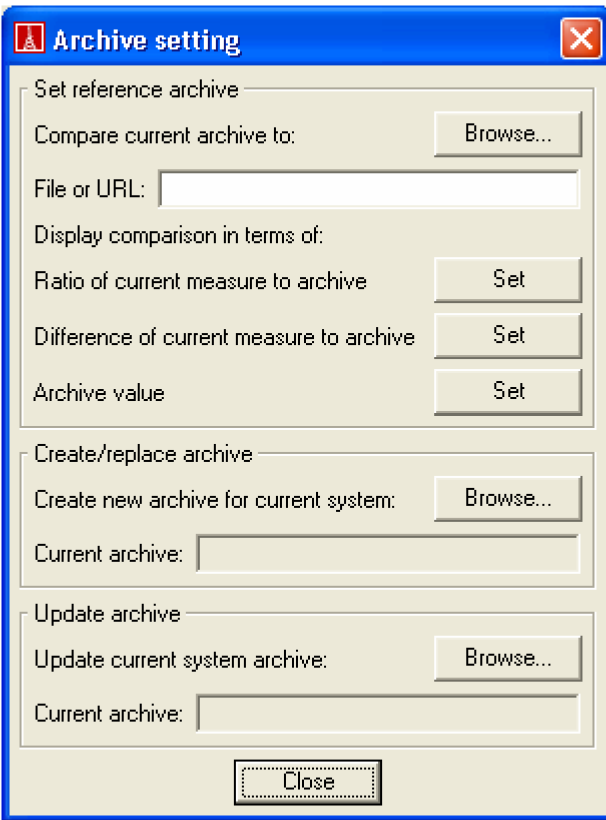
There are still two buttons of the metric toolbar that we have not covered yet:

- gives access to the **metric management** tool:



You can use it to **reorder** the metrics you have defined (by using the *Up* and *Down* buttons) or **delete** any of them. The *Formula* field shows the definition of the selected metric, namely arithmetic or boolean. The *Import* button enables you to **import** some metrics that you have already defined in other projects (by selecting the corresponding XML file).

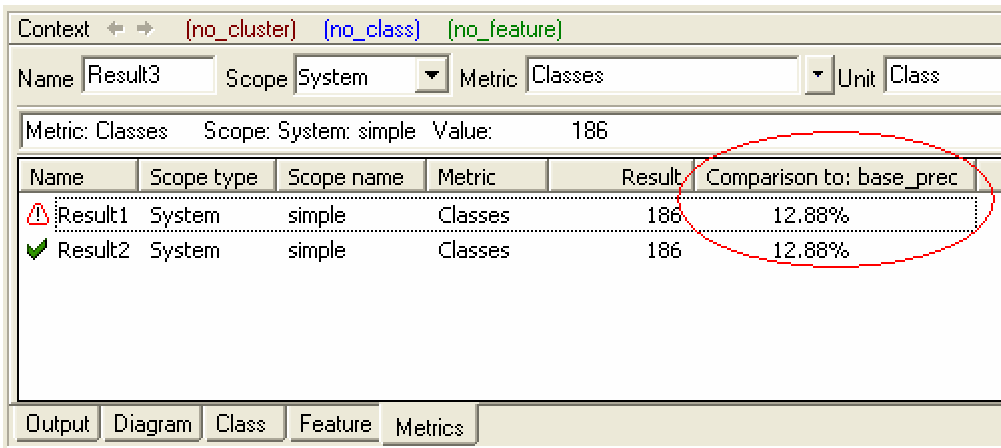
- brings up the **archive comparison** dialog:



Handling archives is a really specific operation: it allows comparison between system scopes. Indeed, it is actually easy to compare smaller scopes (such as features, classes and clusters) one to another among the same project. But this is different for systems since one project contains one and only one system. That is why archives have been created, to store measures for a given system when it is loaded and retrieve them later on when another system is being loaded.

The metric tool allows three kinds of facilities concerning archives: creation of your own archive, update of an existing one or comparison to another system whose measures have been archived. Since the first two options are almost self-explanatory, we will focus on the most interesting one, i.e. **compare two archives**:

You can either click *Browse…* to select a local archive (an XML file) for comparison or enter the following URL in the "*File or URL*" field (on condition that you have access to the Web!): http://metrics.eiffel.com/eiffelvision.xml. Then click the *Set* button next to "*Ratio of current measure to archive*" to select the format for displaying the comparisons, and click Close. This updates the display of computed measures as shown by the following figure:

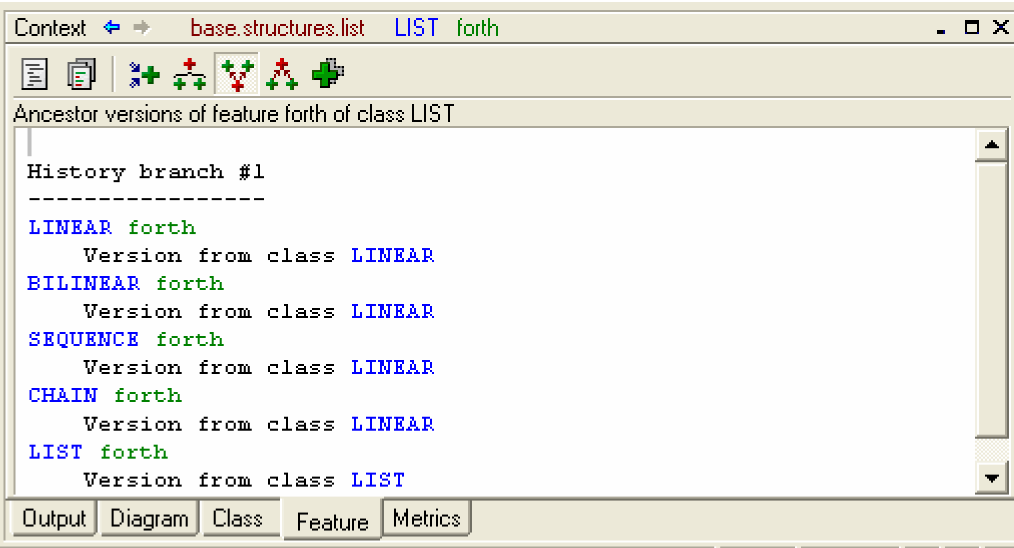


<u>Remark</u>: All future measures that you perform will be immediately compared to the values in the selected archive if the metric is available on both sides.

6.6 Pick and Drop

The previous paragraphs guided you through several browsing facilities of EiffelStudio. But this tour could not be complete without mentioning one of the most important browsing mechanism and EiffelStudio's specificity: "Pick-and-Drop".

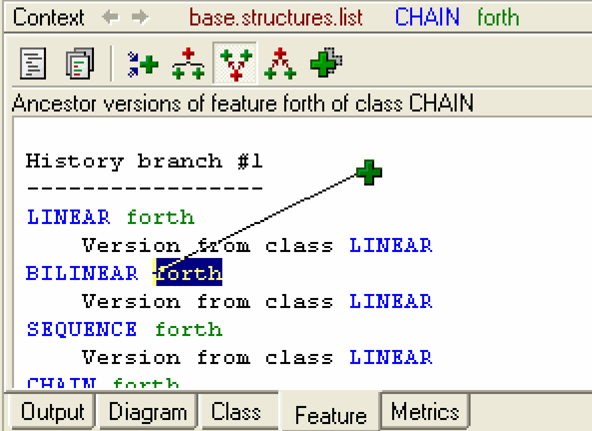
Let's start again with the ancestor versions of feature forth of class LIST (i.e. Feature tab of the Context tool):



The second entry mentions the version of feature *forth* in class BILINEAR. Let's assume that you want to see what that version actually is.

As seen before, you could control-right-click on the feature name to create a new Development Window targeted to feature *forth* from class BILINEAR. But you do not necessarily want to have one more window (in fact, in most cases, you will not want). Instead you can use the **Pick-and-Drop** facility to retarget **the current development window**. Here is how it works:

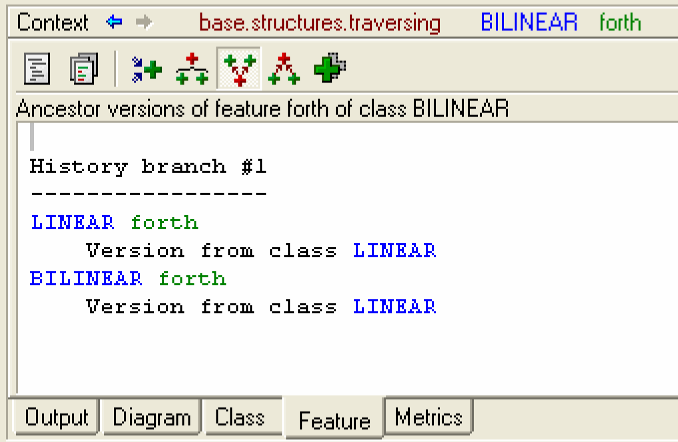
Right-click on the desired feature reference and *release the mouse button immediately* (yes, just a simple click without maintaining the button down) and **move** the mouse a trifle *without pressing any button down*:



The cursor changes into a cross representing the type of development object that you have **picked**, namely a feature (it would be an ellipse for a class, etc.).

Now you need to **drop it** – concretely just **right click again** – at any appropriate place to retarget the corresponding tool.

You can drop it exactly where it is: you will obtain the ancestor versions of feature forth from class BILINEAR. (The retargeting affects both the Editing Tool and the Context Tool, which keeps its current view: ancestor versions in the Feature tab.)



Remarks:

- During the move step, you can **cancel, at any time, the whole operation** by a simple **left-click**.
- The move step is in fact optional in case the current position is a valid drop target: you can drop immediately after the pick without moving the mouse.

The metaphor of pebbles and holes:

In the same way Eiffel is a typed object-oriented language, the Pick-and-Drop mechanism is **typed**: you can only drop a "**pebble**" (the cursor, when its shape has changed depending on the type of the currently selected development object) into a compatible "**hole**" (which can be a window, a tree view entry or a hole-shaped icon).

The type **compatibility** does not necessarily mean identity but **conformance**, i.e. to an entity of type LIST, you may assign not only an expression of the same type but also one of any type, which inherits – conforms to – class LIST.

The power of Pick-and-Drop:

The power of Pick-and-Drop mostly comes from its connection with the various views of the Context tool – Class views, feature views, diagram view. As a matter of fact, we already highlighted the point that all the feature and class names or other graphical representations that appear in these views are clickable: this means you can "pick-and-drop" any of them.

Among places you can use as a target of pick-and-drop, you have the class bubbles in a Diagram view and the icons representing classes and features in the Cluster tree, Feature tree and Favorites list.

7. Editing a class text

EiffelStudio includes a powerful editor, which allows you to read and modify the text of Eiffel classes. It offers usual clipboard, history and search facilities. More advanced, class text specific functionality is also available.

7.1 Clipboard

First, EiffelStudio's editor provides common clipboard functionality, i.e. **Copy, Cut** and **Paste** commands. You can either use the corresponding entries in the **Edit** menu or use the familiar keyboard shortcuts **Ctrl+C**, **Ctrl+X** and **Ctrl+V**.

7.2 History

The key properties of an interactive system – **Undo** and **Redo** commands – are also part of EiffelStudio's editor.

Indeed, you can cancel your latest changes by choosing **Undo** from the **Edit** menu or using the common shortcut **Ctrl+Z**. This process can be repeated as many times as you want during an EiffelStudio session – the history is lost when you close EiffelStudio.

Likewise, you can **redo** an undone command by typing **Ctrl+Y** or selecting the corresponding entry in the **Edit** menu.

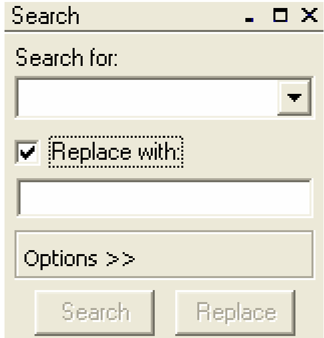
7.3 Search

EiffelStudio's editor also provides search facilities: it enables you to **search** for text and **replace** occurrences, individually or globally.

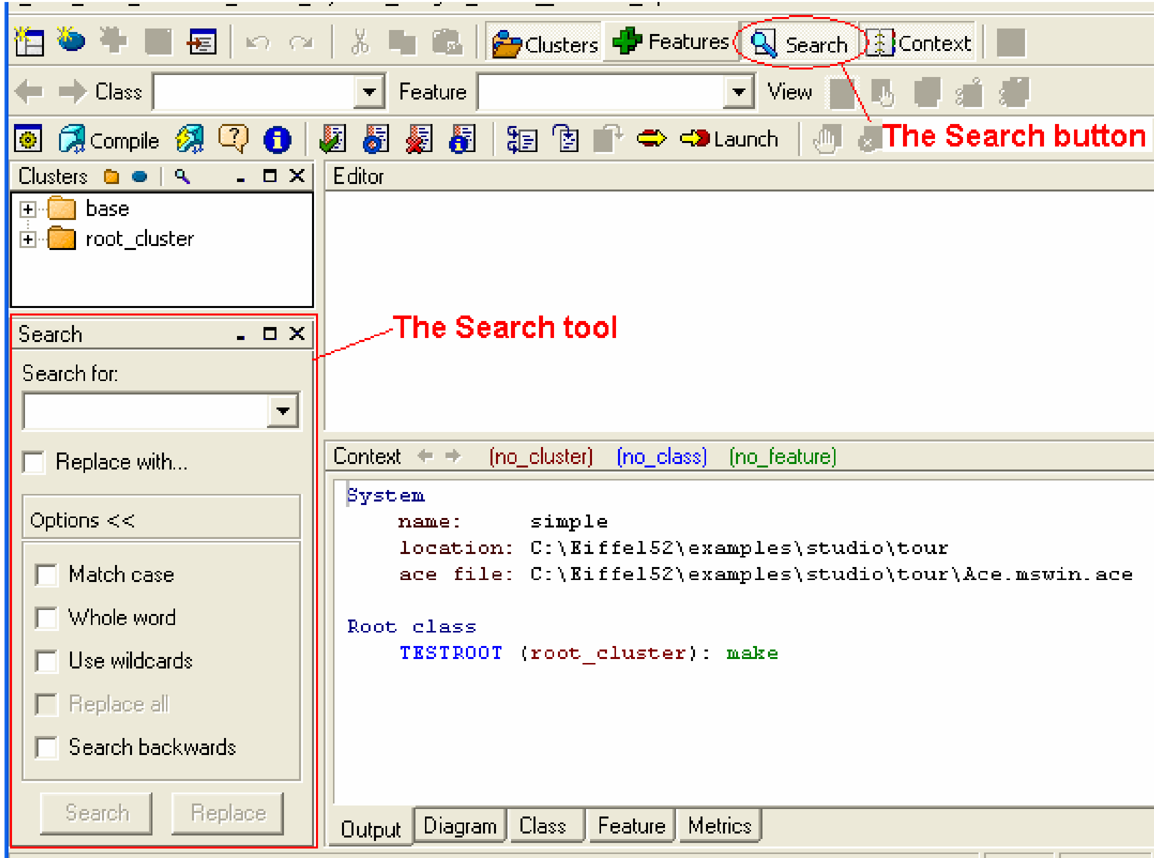
To start a search, you need to make the **Search tool** active by clicking the **Search button** in the top toolbar of EiffelStudio. It is also accessible from the **Edit** menu (**Find** menu entry) or by typing **Ctrl+F** in a text or Context tool.

The Search tool of EiffelStudio provides some self-explanatory options (see figure on next page) but also less obvious advanced ones, we will now review:

- **Use wildcards**: If you select this option, two characters will be treated specially in the **Search for** field: a question mark '**?**' will match any character and an asterisk '*****' will match any sequence of characters.
- The **Search for** field has an associated menu (a "search **history**"), which lets you reuse a recently entered search without retyping it.
- The **Replace with** box – if selected – makes a replacement field to appear:



At this point, you can select to replace only the last found occurrence or all occurrences at once.



7.4 Automatic completion

The editor of EiffelStudio also includes features which were designed to help write Eiffel code. **Automatic completion** is one of those. This feature is available in two contexts: when the user types some expression defined in the Eiffel language or when the user enters a feature call.

As, regarding automatic completion, expectations vary from one individual to another, this feature is highly **customizable** – through the **Preferences** in the **Tools** menu – and can be **disabled**.

Particularly interesting for most people, EiffelStudio editor recognizes the Eiffel keywords and **automatically completes** the basic **control structures** through their opening keyword, such as “if” or “from” for a loop:

```
display is
        -- Display basic information.
    do
        from
        |
        until

        loop

        end
    end
```

The editor also generates the basic **structure of a routine** text – including comments – when you type a routine name followed by the "**is**" keyword and a Return:

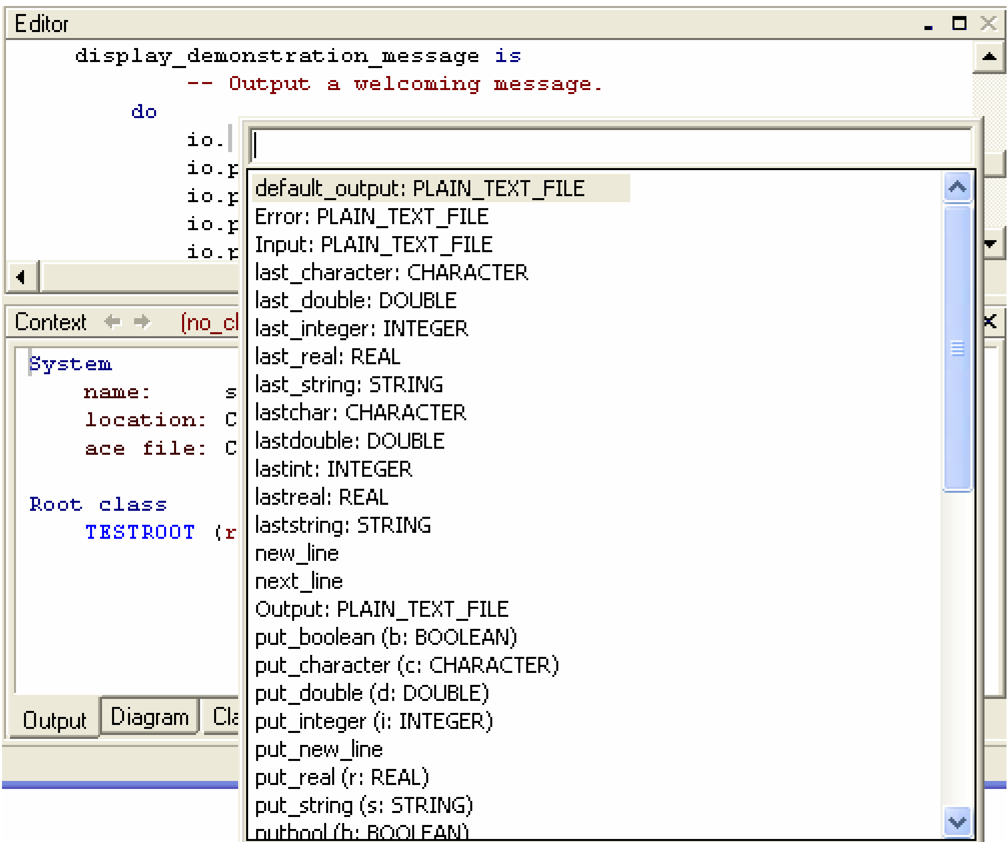
```
new_routine is
        --
```

Besides, it takes care of the **indentation** of **require**, **local**, **do**, **external** and **once clauses** and also inserts the appropriate "**end**" keyword.

Another interesting functionality is **feature completion**: when the user types "*an_identifier.an_incomplete_feature_name*" and then triggers the auto-complete, the editor proposes a list of possible feature names correct in this situation. To be more precise, a window pops up and displays the list of features that can be called on *an_identifier* and that match *an_incomplete_feature_name*. The user can accept the suggested name, choose another name (in the list or not), or simply cancel the auto-complete.

To trigger the auto-complete, press the key combination "Ctrl+Space" or click on **Complete word** in the **Advanced** sub-menu of the **Edit** menu.

If more than one completion is possible, you will get a menu of the possibilities:



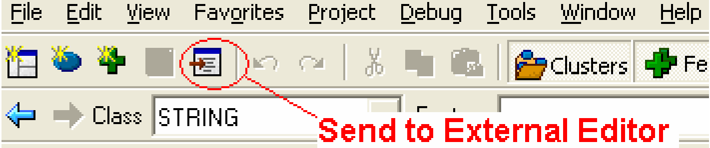
You can scroll through it with the up and down arrow keys or the mouse, and select one through Enter or double-click. You can also give up at any time through the Escape key.

Remarks:

- You can call the auto-complete without an identifier. The auto-complete window will then show the features of the current class.
- The keyboard shortcut for automatic completion can be changed in the editor preferences.
- Typing a non-alphanumeric character in the completion window automatically closes it. You can therefore type '.', ' 'or '(' to close the completion window and start typing the next token straight away.
- Only identifiers that were already defined at the time of last compilation can be completed, except for local entity names that can always be completed.
- Only compiled features appear in the auto-complete window.
- By default, features from class ANY will be ignored by the auto-complete window. This can be changed in the editor preferences.

External editor:

You may have a favorite editor and prefer to use it. You can access it from EiffelStudio through the **External Editor** button (in the top toolbar of EiffelStudio or **File** menu):



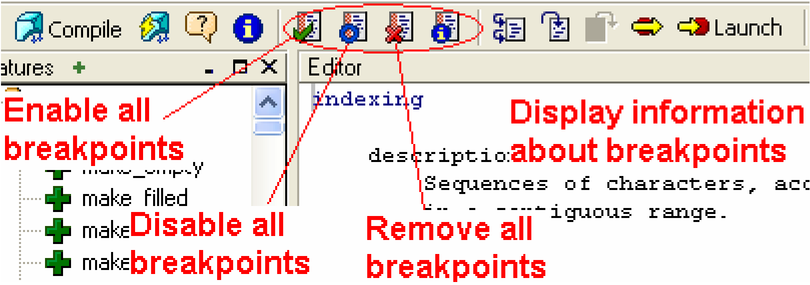
The default is Notepad on Windows and VI on UNIX and Linux.

8. Debugging

8.1 Setting breakpoints

EiffelStudio's debugger enables you to set breakpoints on individual routine instructions or on a routine as a whole or even on preconditions and postconditions.

The following figure shows the interesting icons in the **Project Toolbar**, which help control breakpoints:



The difference between disabling and removing is that disabling turns off breakpoints until further notice but remembers them (you can later re-enable them) whereas removing clears them for good

The *Display information* icon gives the list of breakpoints that you have already set.

How can we then set a breakpoint? You need to switch to the flat view of a class or feature. Let's take the example of the creation routine *make* of class TEXTROOT – the root class of our system:

```
Context  root_cluster  TESTROOT  make
Flat view of feature make of class TESTROOT

    make is
            -- Output messages tracing what's going on.
        do
            display_demonstration_message
            create {HEIR} o1
            create o2
            o1.display
            o2.display
        end

Output  Diagram  Class  Feature  Metrics
```

The small circles on the left side of the Flat form indicate possible breakpoint positions. Just (left) click on the circle where you want to set a breakpoint. Enabled breakpoints are marked by a circle filled with red:

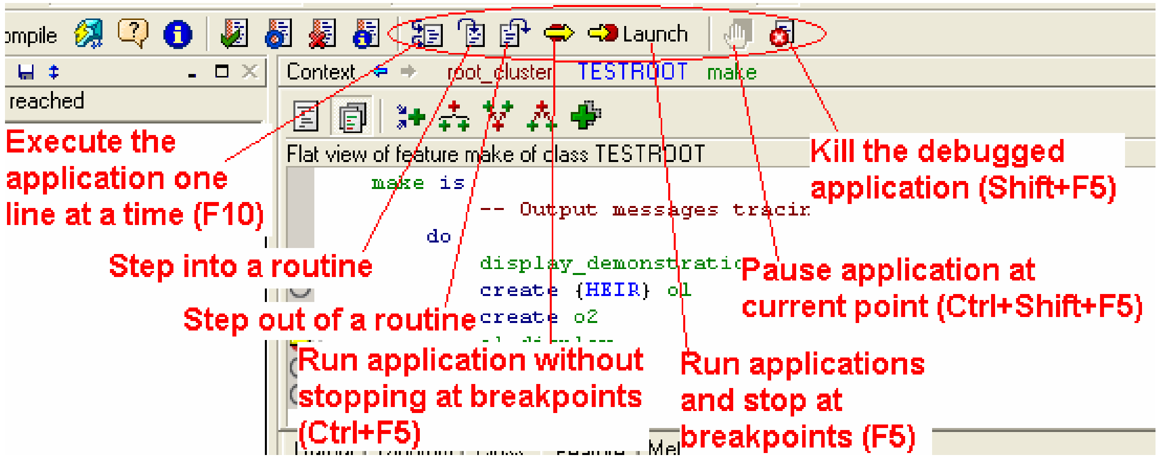
```
Context  root_cluster  TESTROOT  make
Flat view of feature make of class TESTROOT

    make is
            -- Output messages tracing what's going on.
        do
            display_demonstration_message
            create {HEIR} o1
            create o2
            o1.display
            o2.display
        end

Output  Diagram  Class  Feature  Metrics
```

If you left click one more time on the circle, the breakpoint will be removed.

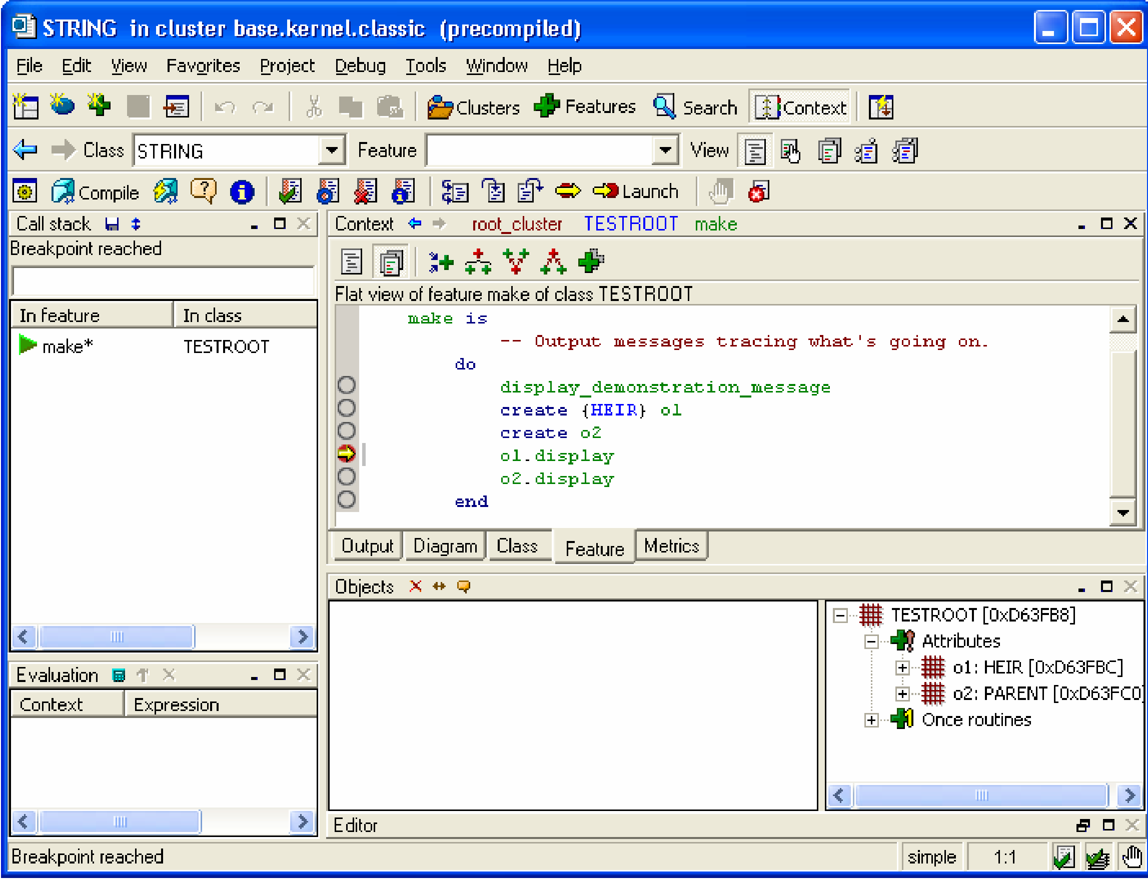
Remark: A non-filled red circle would indicate that a breakpoint is set but not enabled.

8.2 Executing with breakpoints

The figure below shows the buttons you will use to execute a system with breakpoints:



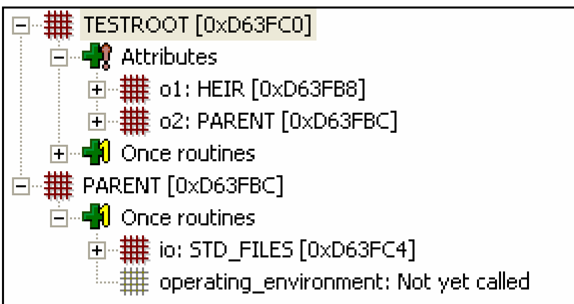
When starting execution of the system, the development window display switches to accommodate supplementary tools providing debugging information; the execution stops on the breakpoint you have enabled on the fourth instruction of procedure *make*:



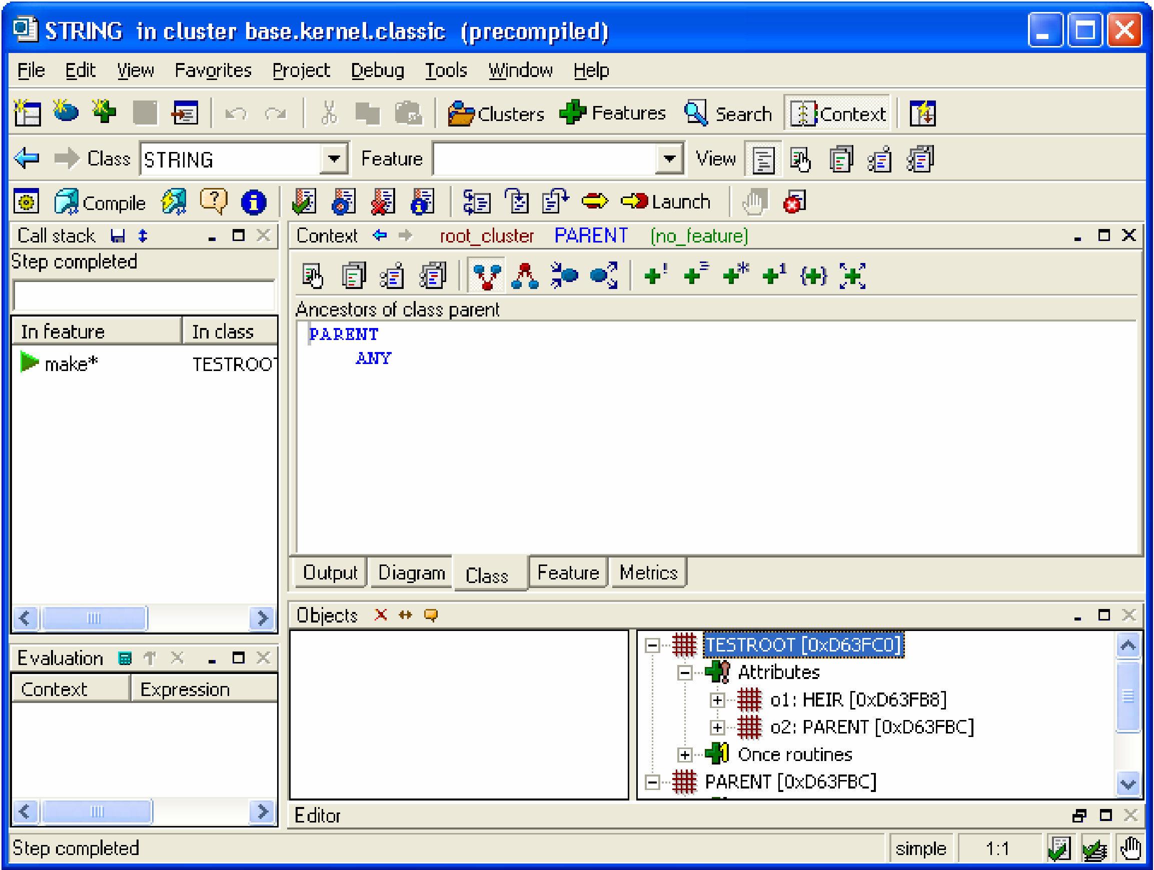
At the bottom right is a new tool: the **Object Editor**, which shows the content of current object and (later) related objects.

On the previous screenshot, you can see that the current object is an instance of type TESTROOT, which has two attributes o1 and o2 – o1 being of type HEIR and o2 of type PARENT – and once routines.

To see the details of an object, pick-and-drop its identifier in the Object Tool itself: this makes a new object entry to appear, showing the corresponding object information:



You can also pick-and-drop the instance of class PARENT to the Context tool to retarget it to that class – as if you had pick-and-dropped the class name PARENT:



To **terminate** execution, just click the corresponding button in the top toolbar or press **Shift+F5**. The specific execution tools (like the Object Editor) disappear and the display returns to its "original" form (i.e. like it was before execution).

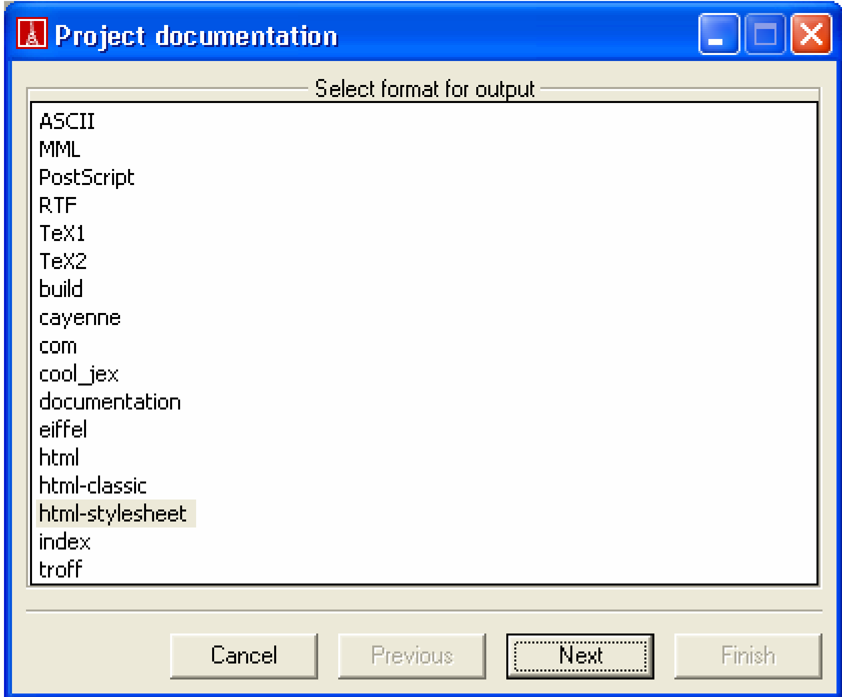
9. Producing extensive documentation

9.1 Generating multi-format documentation

EiffelStudio provides unique facilities to generate extensive documentation about your system using many different formats.

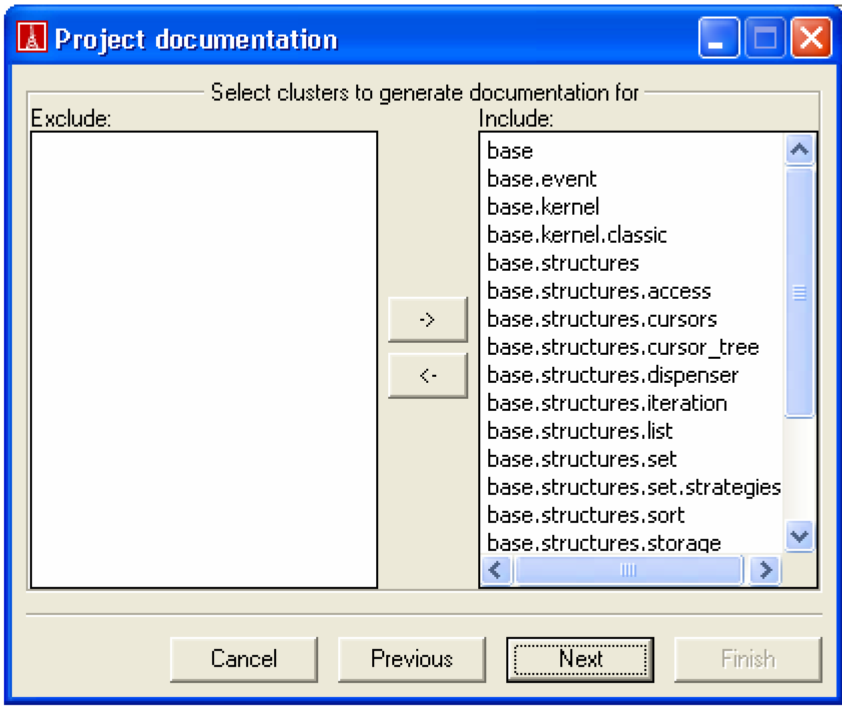
The **Documentation Wizard**, which helps you generate multi-format documentation, is available in the **Project** menu, under the entry **Generate Documentation…**.

The wizard starts with a list of available output formats, also known as filters:

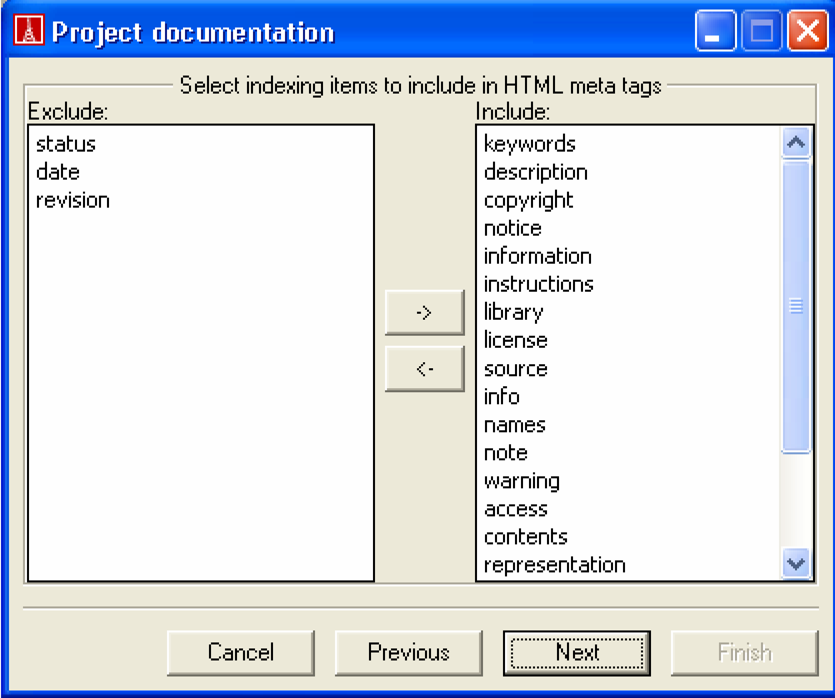


If you do not find the format you want in this list, you can also define your own filter by adding a file (with a .fil extension) in the following subdirectory of your ISE Eiffel installation: $**ISE_EIFFEL/studio/filters**.

Let's select the HTML format with stylesheets. The next window lets you select the clusters you want to produce documentation for:

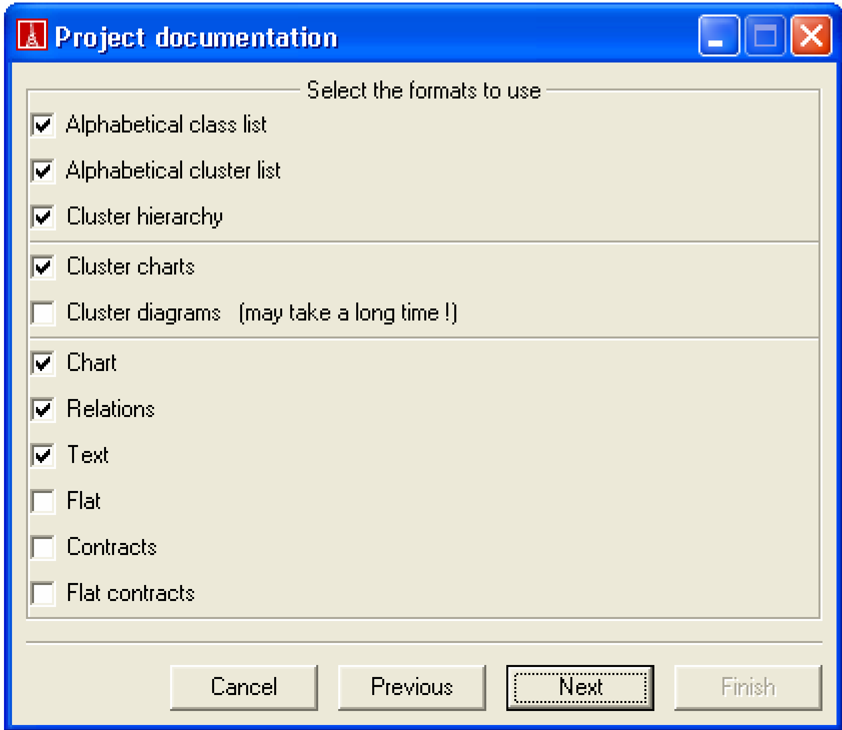


In case you selected HTML documentation generation, the next step is to specify indexing items on which HTML metatags will be based:

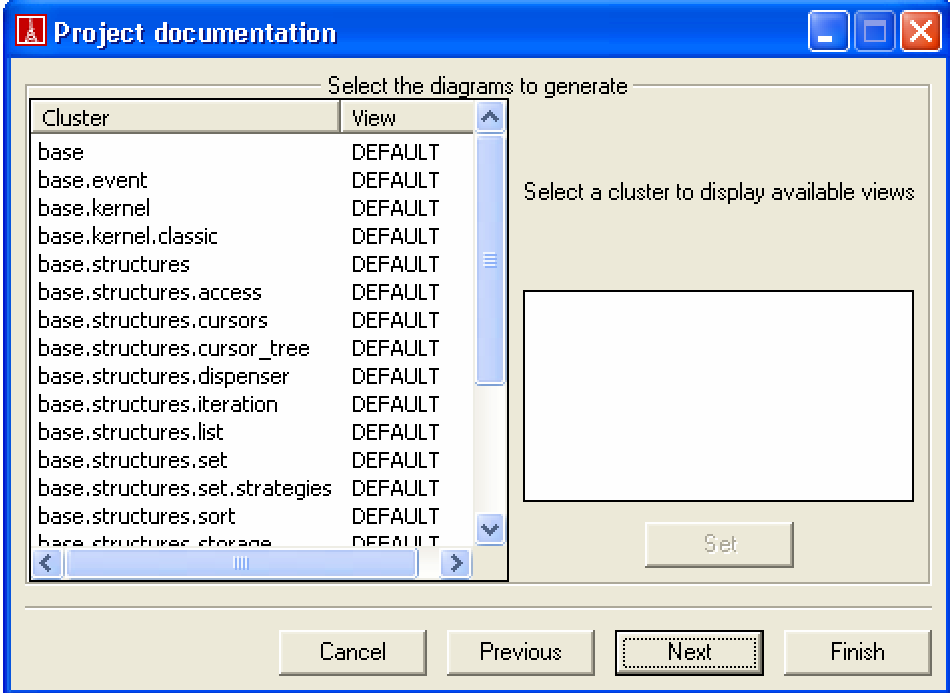


As a matter of fact, Eiffel classes usually include indexing clauses, such as "description", that enable developers to document the text. This is part of the Eiffel style guidelines. Such a concept is very similar to HTML metatags, which carry information that is available to search engines that explore and classify Web pages. Thus, it is quite appropriate to generate HTML documentation from indexing entries.

The documentation wizard then asks you to specify what kind of documents you want to generate. For example, you might need a file containing the **Cluster hierarchy** of your system or, for each class, a file containing the **Contracts** of that class. At this point you can also decide to generate BON diagrams for the clusters selected in the previous page.

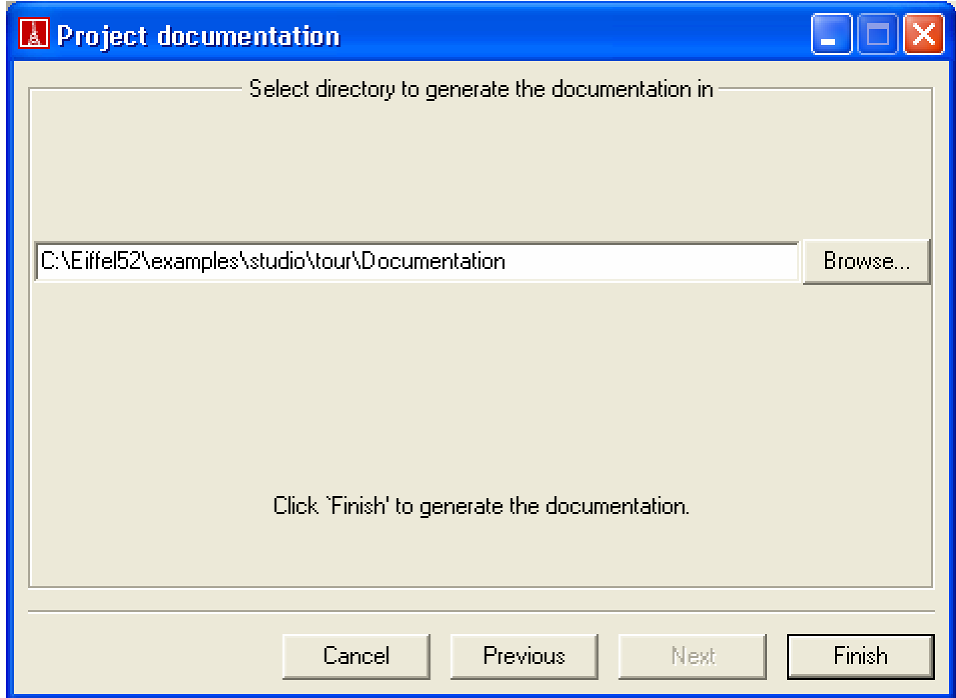


The following page only appears if you have checked **Cluster diagrams** in the previous page. This dialog enables you to choose a view for each diagram that is going to be generated. You have the choice between an automatically arranged view and all of the views you may have manually arranged with the Diagram Tool (see 6.5.2 The Diagram Tab):



Note that cluster diagrams are only available for HTML documentation.

The last page simply asks you to choose the directory where you want the documentation to be generated into:



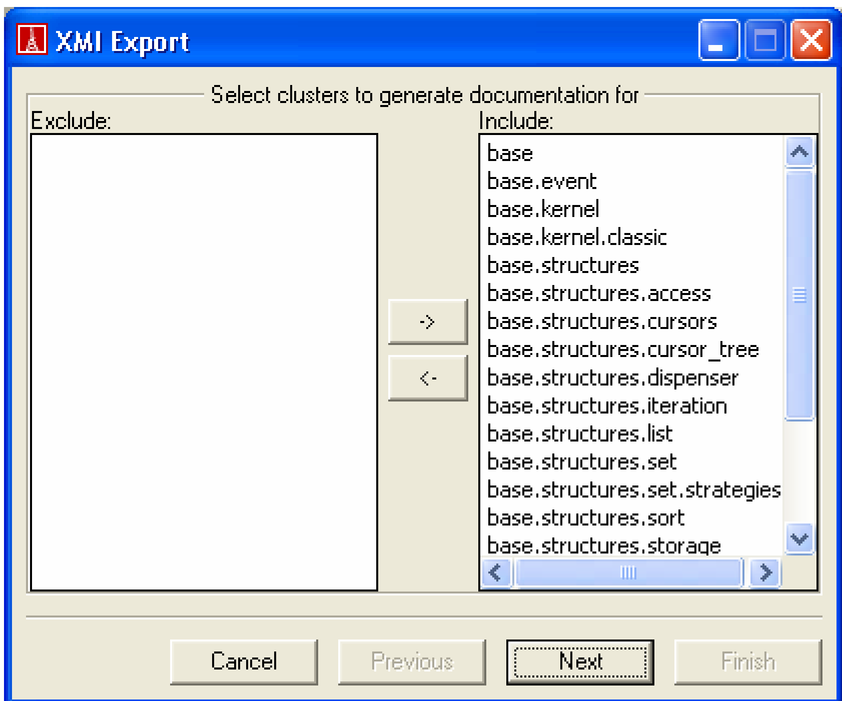
9.2 Generating XMI

EiffelStudio can also generate an **XMI** description of a system:

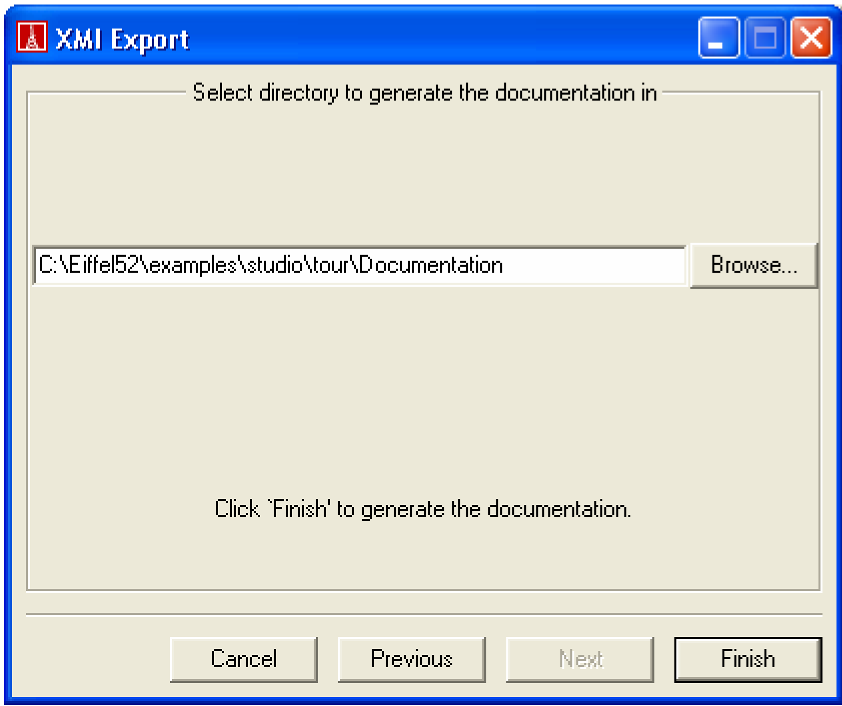
The **XMI** (*XML Metadata Interchange*) format is the new industry standard way to describe and exchange object-oriented systems.

For instance, you can import XMI to Rational Rose after installing a specific add-on (see http://www.rational.com/products/rose/forms/xmisupport/xmisupport.jsp for download).

The **XMI Export wizard** is available in the **Project** menu, under the entry **Export XMI…**. This wizard starts by letting you choose the clusters you want to document:



It then asks you to choose the directory where the XML file will be generated:



10. Graphics-based design with EiffelStudio

In the previous sections, we have used a previously defined example from ISE Eiffel delivery to explore the numerous possibilities of EiffelStudio. But we still miss one of the most attractive one, which is **graphics-based design**.

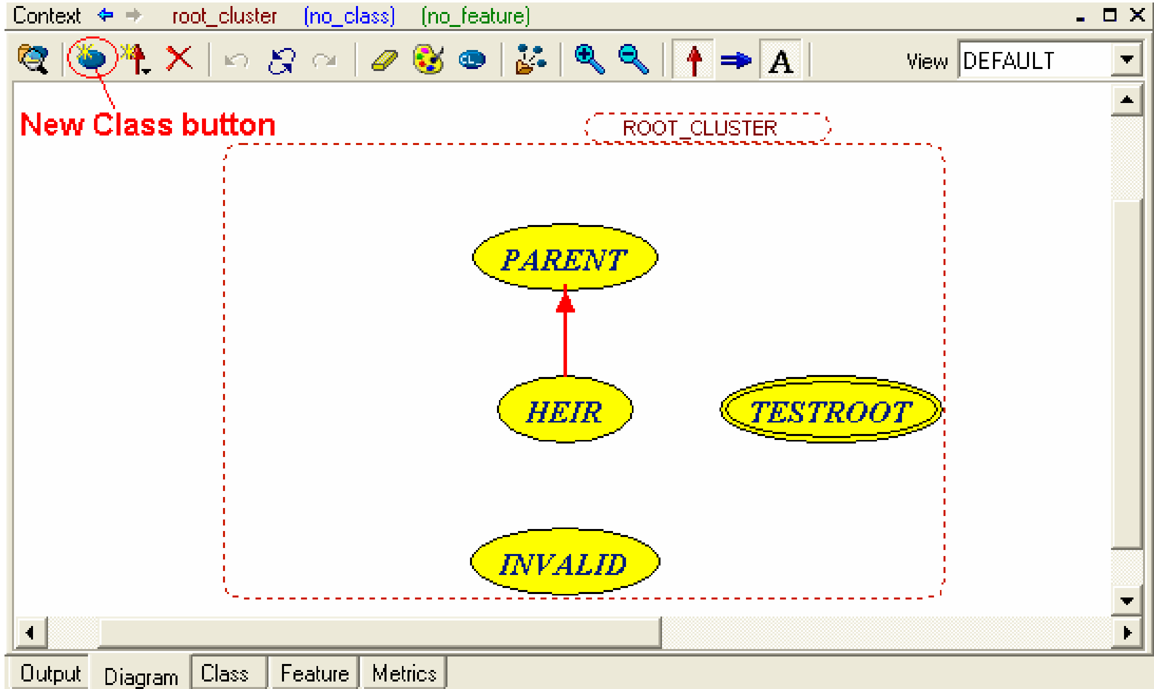
As a matter of fact, we already talked about the diagram tool and the different views it provides as well as the documentation you can generate from it. But one of the greatest specificity of EiffelStudio is that it also provides **true reverse engineering**.

Indeed, EiffelStudio's text-diagram interaction is bi-directional: when you make a textual modification to Eiffel classes, the next incremental recompilation updates the diagram; but you can also work directly from the diagram and the text will be generated or updated after each graphical operation.

This paragraph explores some of the possibilities provided by the diagram tool in matter of design.

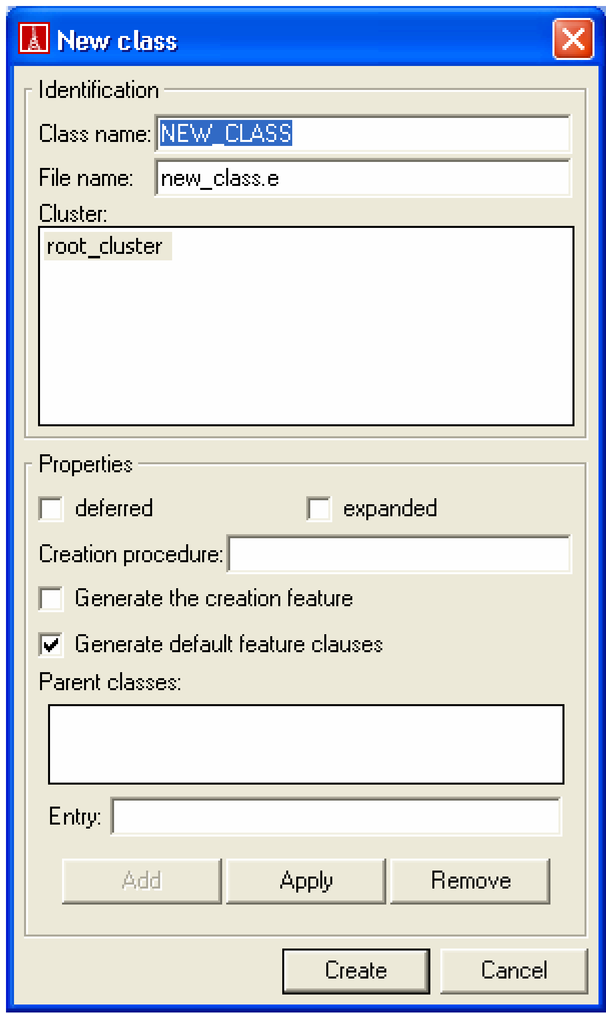
Let's start with the cluster view of our system (Diagram tab of the Context tool) and try to **add a new class graphically**. (This also means that you do not have to worry about creating and initializing a file: EiffelStudio will take care of the details.)

The **New class** button is the one you need to use:

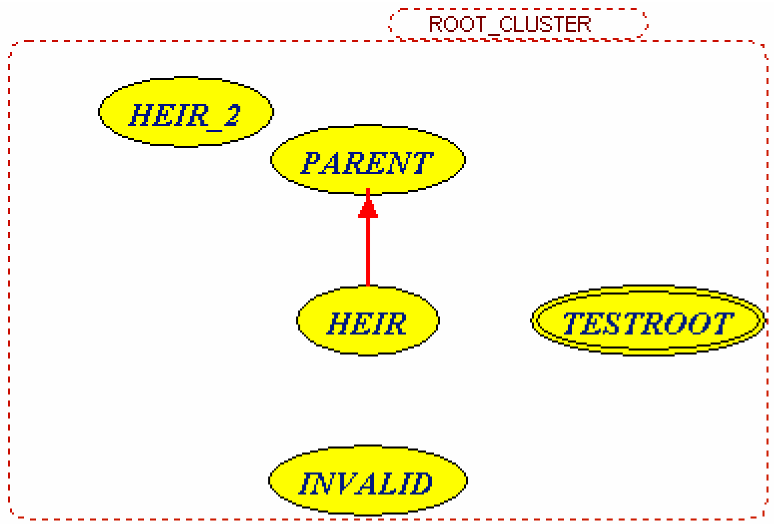


The shape of this button, i.e. a **pebble**, is not a coincidence: it means that you have to **drop** it into the root cluster diagram to add a class to your system.

A dialog pops-up asking you to name the class and set up some of general properties:

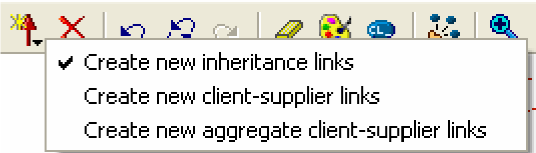


Just overwrite the default class name, i.e. NEW_CLASS, by HEIR_2 for instance. You do not need to change the file name in the second field: this will be updated automatically to `heir_2.e' by EiffelStudio. Then click *Create*: the new class is now part of the root cluster:

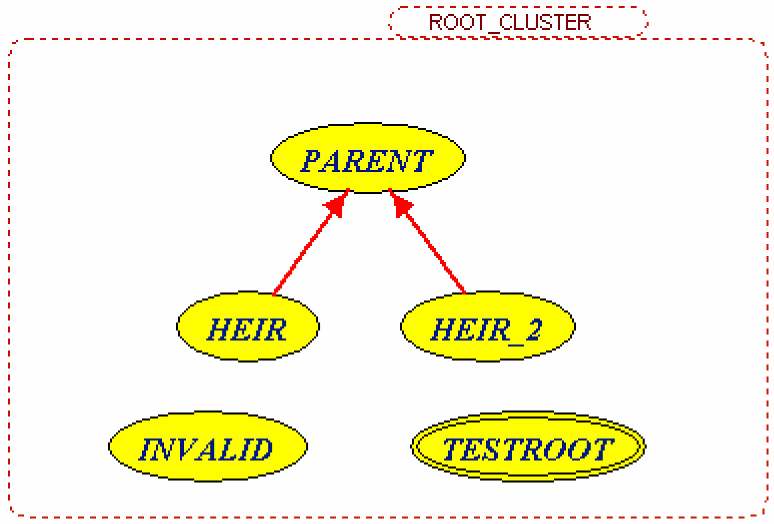


Then, you can **move** the bubbles by traditional **drag-and-drop** in order to get a better display and start adding links between classes.

For instance, it is likely you want to add an **inheritance link** between the newly created class HEIR_2 and the class PARENT. To do so, just select the kind of relation you want by clicking one of the three following possibilities:



Then **pick-and-drop** from the HEIR_2 bubble to the PARENT bubble to create the inheritance link:

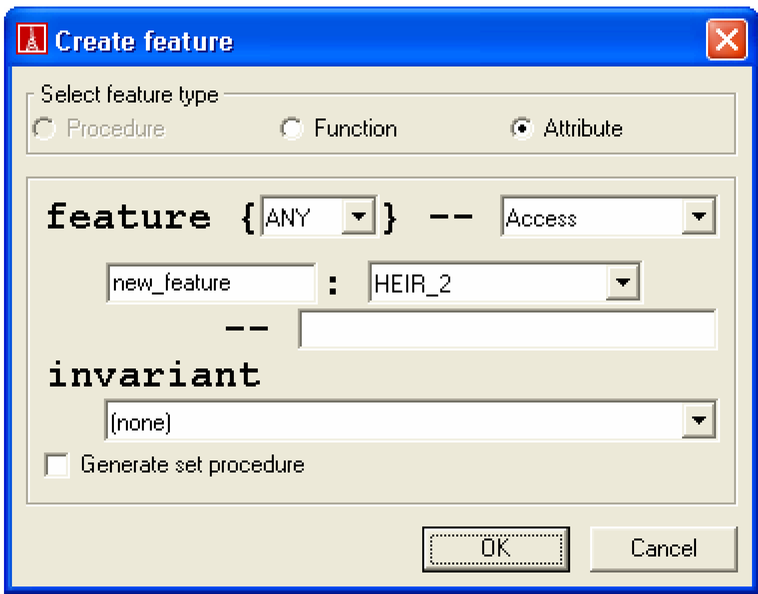


If you want to convince yourself this is not only drawing and check the automatically **generated code**, just **pick-and-drop** HEIR_2 bubble to the **Editor** Tool or control-right-click to the bubble to bring up a new Development Window on this class:

Editor

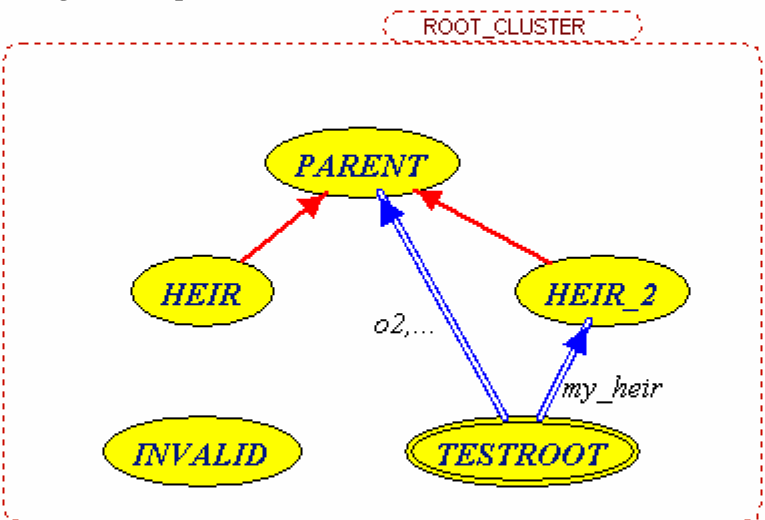
```
indexing
    description: "Objects that ..."
    author: ""
    date: "$Date$"
    revision: "$Revision$"

class
    HEIR_2

inherit
    PARENT

feature -- Access

feature -- Measurement

feature -- Status report

feature -- Status setting

feature -- Cursor movement

feature -- Element change

feature -- Removal

feature -- Resizing

feature -- Transformation

feature -- Conversion

feature -- Duplication

feature -- Miscellaneous

feature -- Basic operations

feature -- Obsolete

feature -- Inapplicable

feature {NONE} -- Implementation

invariant
    invariant_clause: True -- Your invariant here

end -- class HEIR_2
```

In the same way you have just added an inheritance link, you can also add client links between classes. This will bring up the following dialog, which asks you the kind of client relationship you want:



For instance, you can add an attribute *my_heir* of type HEIR_2 to the root class TESTROOT and specify this attribute has to be non *Void* by adding the appropriate invariant clause: *my_heir /= Void*. You can also ask EiffelStudio to add the corresponding set procedure by checking **Generate set procedure** in the dialog above.

The root cluster diagram is updated as follows:



The source text of class TESTROOT is also automatically updated:

Editor

```
class TESTROOT create

    make

feature -- Access

    my_heir: HEIR_2

feature -- Element change

    set_my_heir (a_my_heir: HEIR_2) is
            -- Set `my_heir' to `a_my_heir'.
        require
            a_my_heir_not_void: a_my_heir /= Void
        do
            my_heir := a_my_heir
        ensure
            my_heir_assigned: my_heir = a_my_heir
        end

feature

    o1, o2: PARENT
            -- Examples of attributes

    make is
            -- Output messages tracing what's going on.
        do
            display_demonstration_message
            create {HEIR} o1
            create o2
            o1.display
            o2.display
        end

    display_demonstration_message is
            -- Output a welcoming message.
        do
            io.put_new_line
            io.put_string (" ISE Eiffel spoken here")
            io.put_new_line
            io.put_string ("--------------------------------%N%N")
        end

        -- To get a typical compilation error, remove the two dashes
        -- at the beginning of the next line:
    -- inv: INVALID

invariant
    my_heir_not_void: my_heir /= Void

end -- class TESTROOT
```

Appendix: About compilation modes

EiffelStudio offers several forms – usually called modes – of compilation, which you can see in the entries of the **Compile** menu as well as keyboard shortcuts and, in some cases, buttons:

- **Melt** is a quick incremental recompilation, which does not optimize code for changed parts.
- **Freeze** is an incremental recompilation, which is not as fast as *Melt*, but generates more efficient code for changed parts.
- **Finalize** recompiles the entire system and generates highly optimized code.
- **Precompile** is to process an entire library, on which many systems can then rely without having to compile it. You can launch it either from the **Project** menu or from the item **Precompilation wizard** of the **Tools** menu)

References

- Bertrand Meyer, *Object-Oriented Software Construction*, Second edition, Prentice Hall, 1997.
- ISE Eiffel 5.2 documentation.
- Eiffel Software Web site at http://www.eiffel.com.