

# EiffelStudio: A Guided Tour



**Interactive Software Engineering**

## Manual identification

Title: *EiffelStudio: A Guided Tour*, ISE Technical Report TR-EI-68/GT. (Replaces TR-EI-38/EB.)

## Publication history

First published 1993 as *First Steps with EiffelBench* (TR-EI-38/EB) and revised as a chapter of *Eiffel: The Environment* (TR-EI-39/IE), also available as *An Object-Oriented Environment* (Prentice Hall, 1994, ISBN 0-13-245-507-2).

Version 3.3.8, 1995.

Version 4.1, 1997

This version: June 2001. Corresponds to release 5.0 of the ISE Eiffel environment.

## Author

Bertrand Meyer.

## Software credits

Emmanuel Stauf, Arnaud Pichery, Xavier Rousselot, Raphael Simon, Étienne Amodeo, Jérôme Bou Aziz, Gauthier Brillaud, Paul Colin de Verdière, Jocelyn Fiat, Pascal Freund, Savrak Sar, Patrick Schönbach, Zoran Simic, Jacques Sireude, Tanit Talbi, Emmanuel Texier, Guillaume Wong-So; EiffelVision 2: Leila Ait-Kaci, Sylvain Baron, Sami Kallio, Ian King, Sam O'Connor, Julian Rogers. See also acknowledgments for earlier versions in *Eiffel: The Environment* (TR-EI-39/IE)

Non-ISE: special thanks to Thomas Beale, Éric Bezault, Paul Cohen, Paul-Georges Crismer, Michael Gacsaly, Dave Hollenberg, Mark Howard, Randy John, Eirik Mangseth, Glenn Maughan, Jacques Silberstein.

## Cover design

Rich Ayling.

## Copyright notice and proprietary information

Copyright © Interactive Software Engineering Inc. (ISE), 2001. May not be reproduced in any form (including electronic storage) without the written permission of ISE. "Eiffel Power" and the Eiffel Power logo are trademarks of ISE.

All uses of the product documented here are subject to the terms and conditions of the ISE Eiffel user license. Any other use or duplication is a violation of the applicable laws on copyright, trade secrets and intellectual property.

Any third-party products mentioned in this document are hereby acknowledged as trademarks of their respective owners.

## Special duplication permission for educational institutions

Degree-granting educational institutions using ISE Eiffel for teaching purposes as part of the [Eiffel University Partnership Program](#) may be permitted under certain conditions to copy specific parts of this book. Contact ISE for details.

### About ISE

ISE (Interactive Software Engineering) helps you produce software better, faster and cheaper.

ISE provides a wide range of products and services based on object technology. ISE Eiffel is a complete development environment for the full software lifecycle. ISE's training courses, available worldwide, cover key management and technical topics. ISE's consultants are available to address your project needs at all levels.

ISE's TOOLS (Technology of Object-Oriented Languages and Systems) conferences, <http://www.tools-conferences.com>, are the meeting point for anyone interested in the software technologies of the future.

ISE originated one of the earliest .NET products and offers a full range of .NET services and training at <http://www.dotnetexperts.com>.

### For more information

Interactive Software Engineering Inc.

ISE Building, 360 Storke Road

Goleta, CA 93117 USA

Telephone 805-685-1006, Fax 805-685-6869

### Internet and e-mail

ISE maintains a rich source of information at <http://eiffel.com>, with more than 1200 Web pages including online documentation, downloadable files, product descriptions, links to ISE partners, University Partnership program, mailing list archives, announcements, press coverage, Frequently Asked Questions, Support pages, and much more.

Write to [info@eiffel.com](mailto:info@eiffel.com) for information about products and services. Go to [www.talkitover.com/eiffel](http://www.talkitover.com/eiffel) to subscribe to the ISE Eiffel user list.

### Support programs

ISE offers a variety of support options. See <http://support.eiffel.com> for details.

---

# EiffelStudio: A Guided Tour

This document is available both locally, as part of the ISE Eiffel delivery, and on the [eiffel.com](http://eiffel.com) Web site, in both HTML and PDF versions. See the [list of introductory documents](#).

This is **not** an introduction to the Eiffel method and language. Follow the preceding link for *Invitation to Eiffel* and a longer tutorial.

## 1 OVERVIEW AND PREREQUISITES

EiffelStudio is the central tool of ISE Eiffel, letting you design, develop, debug, document, measure, maintain, revise and expand systems using the full power of object technology and Design by Contract™.

This presentation introduces the essential properties of EiffelStudio. It will take you through a tour of the environment, using a pre-existing example system.

### *What will I achieve?*

Although it skips many specific or advanced facilities, this Tour will help you quickly become familiar with the way you can use the environment for your work. After reading it you will know the basics of working with EiffelStudio:

- Starting a project and retrieving an existing project.
- Entering new software elements (classes).
- Compiling your software.
- Making changes and having them immediately recompiled using the Melting Ice Technology™.
- Displaying a graphical representation of your software elements, and modifying the software through the graphical views (as well as through its text).

- Producing extensive documentation of your system, textual or graphical, under many different formats such as HTML, RTF, Postscript, XMI (for e.g. Rational Rose) and others.
- Browsing through simple or complex software systems, to find out their various components, properties and relationships.
- Measuring quantitative properties of the software, by applying metrics predefined in EiffelStudio as well as new ones that you define.
- Executing a compiled system, and controlling its execution through the debugging mechanisms of EiffelStudio.

### *About the scope of EiffelStudio*

The most important property to keep in mind as you are discovering EiffelStudio is that it is neither just a “programming environment” nor just a “CASE tool” (Computer-Aided Software Engineering) for analysis and design. It encompasses both of these functions and many others. Most system builders today are used to a dichotomy between the high end and the low end:

- At the analysis and design levels, graphical tools attempt to help you clarify your thinking about the system, interacting with customers and end users, and devise high-level system architectures, usually in diagrammatic form.
- At the low end, programming tools help you edit, compile and debug your programs.

Keeping these tools separate is, however, detrimental to the quality of the software process and the resulting products. If they are in the hands of different teams, communication problems may arise, leading to discrepancies between need and realization; this can be a source of bugs or even project failure. If the same people use both kinds of tools, they have to keep switching notations, tools and modes of thinking. The use of different frameworks at both ends makes it difficult to keep the high-level

model and the implementation consistent; too often, when a change is decided at the implementation level, it is not reflected back in the higher model. After a while, the system gets into the state of disorder and inconsistency that good tools are precisely meant to avoid.

EiffelStudio, in line with the principles of *seamless development* and *reversibility* of the Eiffel method, removes the gap by providing a single set of tools that accompany you throughout a project, from the most high-level initial stages to the most detailed aspects of implementation and debugging.

This is reflected throughout the environment by, for example, the dual use of text and graphics. As another example, you should think of the EiffelStudio *compiler*, not just as a tool for executing Eiffel software in its final form, but also, thanks to its extensive *validity checking* facilities, as a design consistency tool that performs many verifications commonly associated with CASE tools.

### *Learning by doing*

If you have access to EiffelStudio as you read this Tour, the best way to take the guided tour is to execute all the suggested operations as you read about them.

Please execute user actions, such as clicking, only when asked to do so.

### *What should I already know?*

This Tour assumes very little about what you know and what you don't.

It does assume that you can do simple manipulations on your platform of choice, such as: on Windows, finding and drag-and-dropping folders and files in the Windows Explorer; on Unix, changing to a certain directory (**cd**) and listing the files of a directory (**ls**).

The more you already know about object technology and object-oriented environments, the better. But remember, if you have used other environments before, keep a fresh outlook; EiffelStudio *is* different, and it may take a while before you fully understand why it does some things in a certain way.

## *A note on platform differences*

ISE Eiffel is one of the most portable environments in the industry, running in an almost identical fashion on Windows, on the new Microsoft .NET environment, on many variants of Unix, on Linux, on VMS.

Once an EiffelStudio session has been started, you can largely forget about the operating system. But a few operations — mostly at the beginning, to launch EiffelStudio — require platform-dependent mechanisms: starting a program, traversing the file structure, selecting a file. These cases will be marked accordingly below.

Windows users should particularly note the following two conventions of terminology:

- Operating systems store files into hierarchically nested structures called *folders* or *directories*. Although “folder” is the more common term for Windows, we will mostly speak of “directories”. It’s exactly the same thing.
- A file has a full path name, used to describe how to reach it from the root of its file system, as in `c:\d1\d2\` (Windows notation). Unix and Linux, as well as the Internet (for URLs), use the forward slash `/`, rather than the backward slash `\`, as a separator, as in `/d1/d2/`. Most file names in this manual appear in this Unix style. On Windows you will normally use the backslash convention, although EiffelStudio also accepts forward slashes. In any case you must be consistent: use one convention or the other, but don’t mix backward and forward slashes in the same path name. Also, note that some names, such as those of object files to be linked with your system, will be passed to outside tools — C compilers, loaders — that may not accept the forward slash.

VMS users may similarly use either the Unix convention or the specific VMS path naming convention.

If you are a one-platform person, just ignore, for the next few pages, all references to any platform other than your heart’s favorite. They will quickly go away.

## *What should I have done first?*

To run the example you must have installed ISE Eiffel and set up the environment. Check in particular the following:

- On Windows, you must have run the installation procedure; it will have put EiffelStudio in the Programs section of the start menu, subsection “ISE Eiffel *version*”, where *version* is the version number, e.g. 5.0.
- The environment variable **ISE\_EIFFEL** must be set to the installation directory, and the environment variable **PLATFORM** to the platform. On Windows this is taken care of automatically by the installation procedure, but on Unix/Linux and VMS you must update your path and environment manually. Throughout this discussion the notations **\$ISE\_EIFFEL** and **\$PLATFORM** will refer to the values of these variables — the installation directory, and the platform. (The Windows notation would be **%ISE\_EIFFEL%** and **%PLATFORM%**.)
- On Unix/Linux and VMS, your “path” must include the place where EiffelStudio executables reside. (On Windows the installation procedure takes care of this.)
- Also, the discussion assumes that as part of the installation you have included the EiffelBase library, in precompiled form. EiffelBase is automatically included if you have installed another precompiled library, such as WEL, the Windows Eiffel Library. The installation procedure takes care of precompiling EiffelBase.

## *Locating the example*

Please take a moment to locate the example files on your installation. They all appear in the following directory, part of the Eiffel delivery:

**\$ISE\_EIFFEL/examples/bench/tour**

(Windows users: remember that instead of the slash / your platform uses a backslash \. VMS users: this is to be replaced by the VMS path naming conventions.)

## 2 COPYING THE EXAMPLE FILES

If you are using Eiffel on a personal computer under Windows, you can work directly on the installation directory and don't need to make copies of files as per the present section. **Skip directly to the next section, 3.**

*Do* read the present section and apply its instructions if you work under Unix, or may have to share the Eiffel installation with other users, do not have write permissions on the installation, or want to keep the installation unchanged.

If you are going to work on a copy, choose or create a directory of your own; let's call it *YOURDIR* for the rest of the discussion.

To copy all the files of the example to *YOURDIR*:

- On Windows, open a Windows Explorer, go to *\$ISE\_EIFFEL\examples\bench\tour*, select all the files in that directory, and drag-and-drop them to *YOURDIR*.
- On Unix execute the shell command

```
cp $ISE_EIFFEL/examples/bench/tour/* YOURDIR
```

- On VMS execute the command

```
copy $ISE_EIFFEL:[examples.bench.tour]*.* YOURDIR
```

Once you have compiled the example under EiffelStudio, relying on precompiled EiffelBase library (the default), and without optimization, the contents of *YOURDIR* will take up less than one megabyte, including information on diagrams and metrics.

The final executable generated through “*finalization*” (optimized compilation) will take only about 300 kilobytes.

Without precompiled EiffelBase you would need about 3.5 megabytes, plus about 25 megabytes of generated C code and auxiliary files. The executable in that case will take up about 250 kilobytes,



## 3 STARTING EIFFELSTUDIO AND OPENING A PROJECT

In the rest of this Tour *YOURDIR* denotes the directory where the example resides (the original, [\\$ISE\\_EIFFEL/examples/bench/tour](#), or a copy). Launching will use the operating system's mechanism for starting a program, so we look separately at Windows and at Unix/VMS.

### *Launching EiffelStudio under Windows*

On Windows, you can launch EiffelStudio from the Start Menu under

**Programs → ISE Eiffel Version → EiffelStudio**

where *Version* is the version number, e.g. 5.0. Alternatively, you can double-click on the icon that the installation procedure will have added to your desktop (if you have selected that option during installation).

If this is the first time you are using EiffelStudio, you may get a dialog asking for an unlock code or inviting you to register the product. Please [contact ISE](#) for registration and purchase information.

### *Launching EiffelStudio under Unix or VMS*

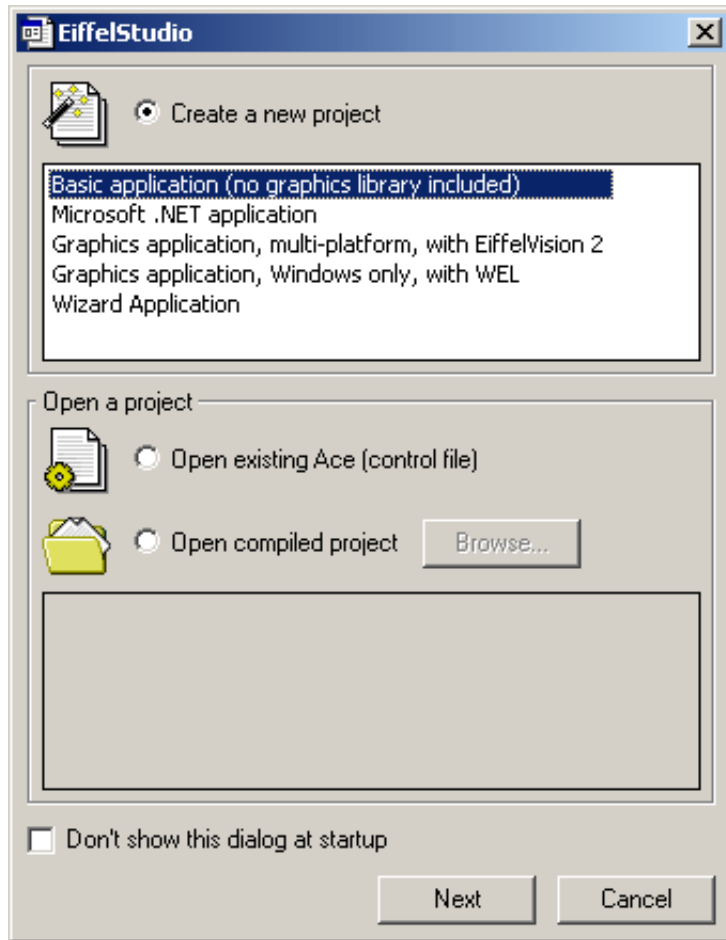
To launch EiffelStudio on Unix or VMS, change directory to *YOURDIR* and, from the command line, type

**estudio**

In general you can start EiffelStudio from any directory, but to make things simple for this Tour **please make sure** indeed to execute the **estudio** command from *YOURDIR*. (This will allow us to use relative rather than absolute names for some of the files involved.)

## 4 COMPILING AND EXECUTING A SYSTEM

EiffelStudio first comes up with a window and a dialog on top of it; the dialog looks like this (from here on the look-and-feel will be different on non-Windows platform, but the contents will be the same):



As this is our first project we want to **Create a new project**. The three possibilities are:

- **“Create a new project”**, which would let you select one among the common schemes — basic application, graphical Windows application, graphical multi-platform application, Microsoft .NET application — and set up everything for you.
- **“Open existing Ace (control file)”**, which finds the setup in a control file, called an *Assembly of Classes in Eiffel* or Ace.
- **“Open compiled project”**, which will work once you have compiled a project.

In future sessions you'll probably use the first option for a new project, as it takes care of generating everything for you, and the third option for an existing project. Right now, though, we want you (like an expert EiffelStudio user!) to use an existing Ace that exists precisely for the purpose of this Tour. So please select the second option

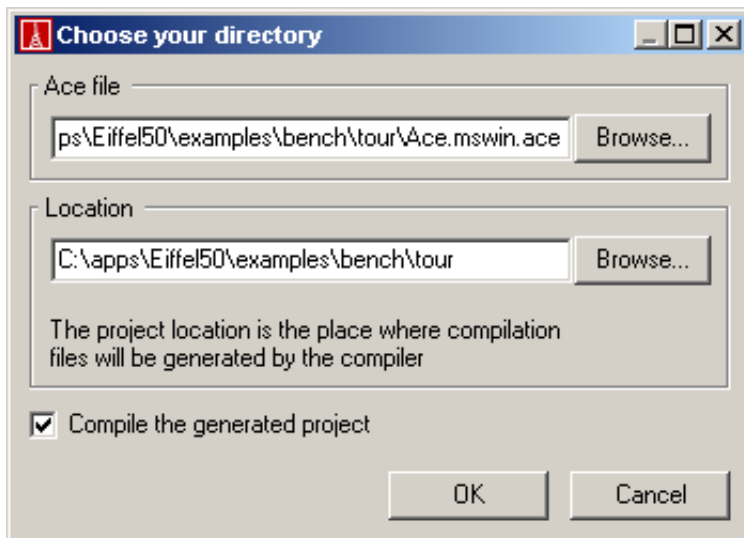
**Open existing Ace (control file)**

and click **Next**. This brings up a File Explorer inviting you to select an Ace file. The file you want is, depending on your platform, one of

**\$ISE\_EIFFEL/examples/bench/tour/Ace.mswin.ace**  
**\$ISE\_EIFFEL/examples/bench/tour/Ace.unix.ace**

(the first one for Windows, the second for Unix; remember that **\$ISE\_EIFFEL** stands for the location of the Eiffel installation, such as **C:\Eiffel50**; Ace files are normally marked by the **.ace** file extension). For VMS, select the Unix variant.

Use the File Explorer to go to the directory **\$ISE\_EIFFEL/examples/bench/tour/** and select the appropriate file, **mswin.ace** or **unix.ace**, for your platform. A confirmation window comes up:



Click **OK** to confirm. This selects the Ace file and starts compilation of your project.

During Eiffel compilation, a progress bar displays the successive compilation steps, or “degrees”. The bulk of our little project is the EiffelBase library, which the EiffelStudio installation procedure has precompiled; so just now there’s only a few extra classes to compile, and the process is almost instantaneous on a state-of-the-art computer. Even if you had to compile the EiffelBase classes, EiffelStudio compilation is so fast that you would hardly have the time to read the “degree” messages; you can see them later in compilations of bigger classes and systems.

On an IBM Thinkpad, Pentium III 850 MHz, 256 MB, running Windows 2000 Professional, Eiffel compilation takes about 9 seconds for the entire Guided Tour system including non-precompiled EiffelBase.

After Eiffel compilation completes you will see the message

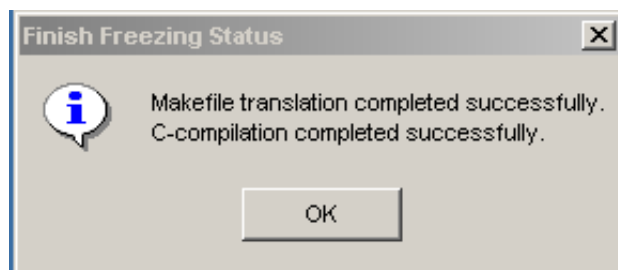
**System had to be frozen to include new externals.  
Eiffel compilation system recompiled.  
Background C compilation launched**

At this stage you can start using EiffelStudio, e.g. for browsing, but since C compilation won’t take long either (about 15 seconds on our test machine), it’s just as well to let it finish. C compilation messages will appear:

- On Windows, in a new console.
- On Unix, in the window from which you launched EiffelStudio.

Why C compilation? ISE Eiffel under some circumstances uses C as intermediate code for its compiler’s generated code, and relies on a C compiler to process the result. On Windows you will by default use the command-line Borland compiler (included with the ISE Eiffel delivery); you may replace it by another, such as Visual C/C++ from Microsoft (available directly from ISE) if you wish. On Unix you will use the resident C compiler.

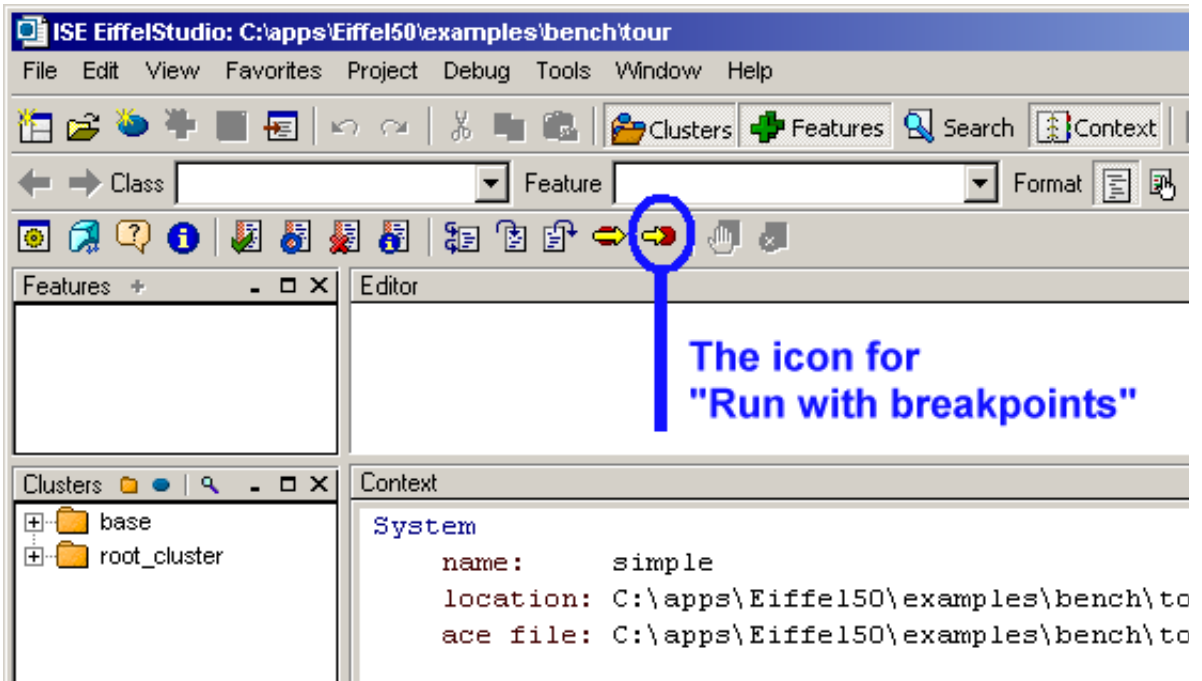
EiffelStudio reports that the C compilation has terminated:



Congratulations! You have successfully compiled your first Eiffel project. More precisely it's been both "melted" and "frozen". Strange terminology, you may think; in a little while we'll see why these compilation steps are called that way.

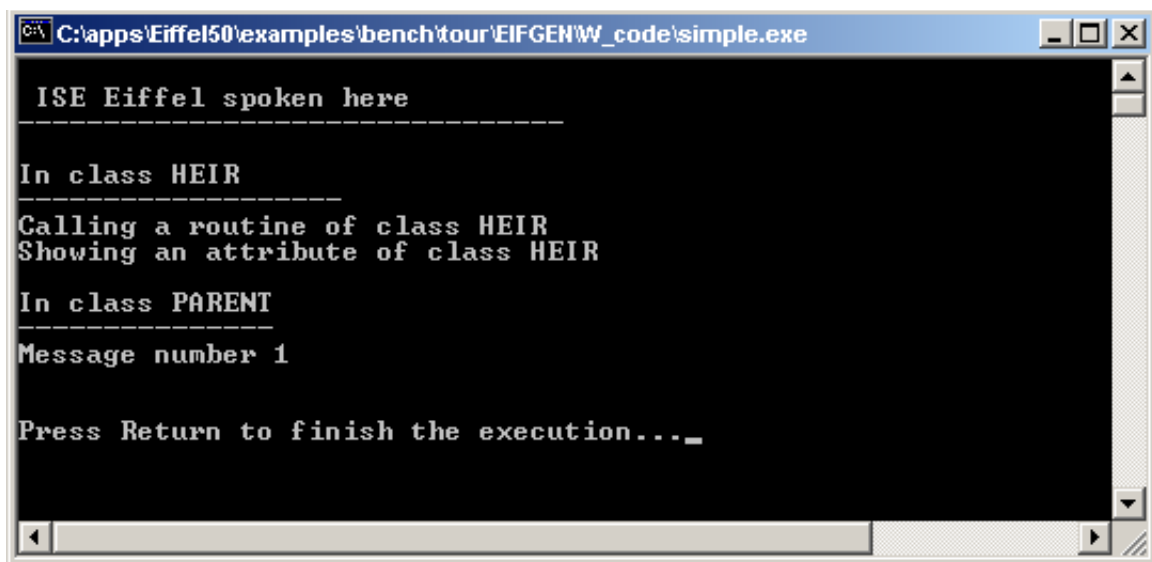
### *Executing the system*

The system doesn't do anything exciting, but let's execute it anyway. Find the execution icon ("Run with breakpoints") at the top of the EiffelStudio window:



It would be OK too to use the neighboring icon to the left, "Run without breakpoints", since we haven't set any breakpoints. Click either icon to execute the system.

This little application doesn't use any graphics or anything fancy but simply creates some objects and displays some information. Output done using the default Eiffel I/O (from the EiffelBase classes [ANY](#) and [STANDARD\\_FILES](#)) goes to a console. On Unix, Linux, VMS it's the window from which you started EiffelStudio. On Windows it's by default a new console window that comes up when and if the system does its first output operation, and stays up:



```
C:\apps\Eiffel50\examples\bench\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

In class PARENT
-----
Message number 1

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

The message “**Press Return to finish the execution**” would not appear if you executed the system from outside of EiffelStudio, for example from a command line. Its purpose within EiffelStudio is clear: to let you see the console output; without it, the console would go away at the end of execution. (None of this applies to Unix/Linux/VMS since there is no new console window to get rid of.)

If before closing the console window you look at the main EiffelStudio window (by moving away the console window) you will notice that it looks different from before, since it now shows the fields useful in monitoring execution and debugging. But we’ll look at this later. For the moment just dismiss the console by following the advice to “**Press Return**”: hit the Return or Enter key.

## 5 A LOOK AT THE PROJECT DIRECTORY

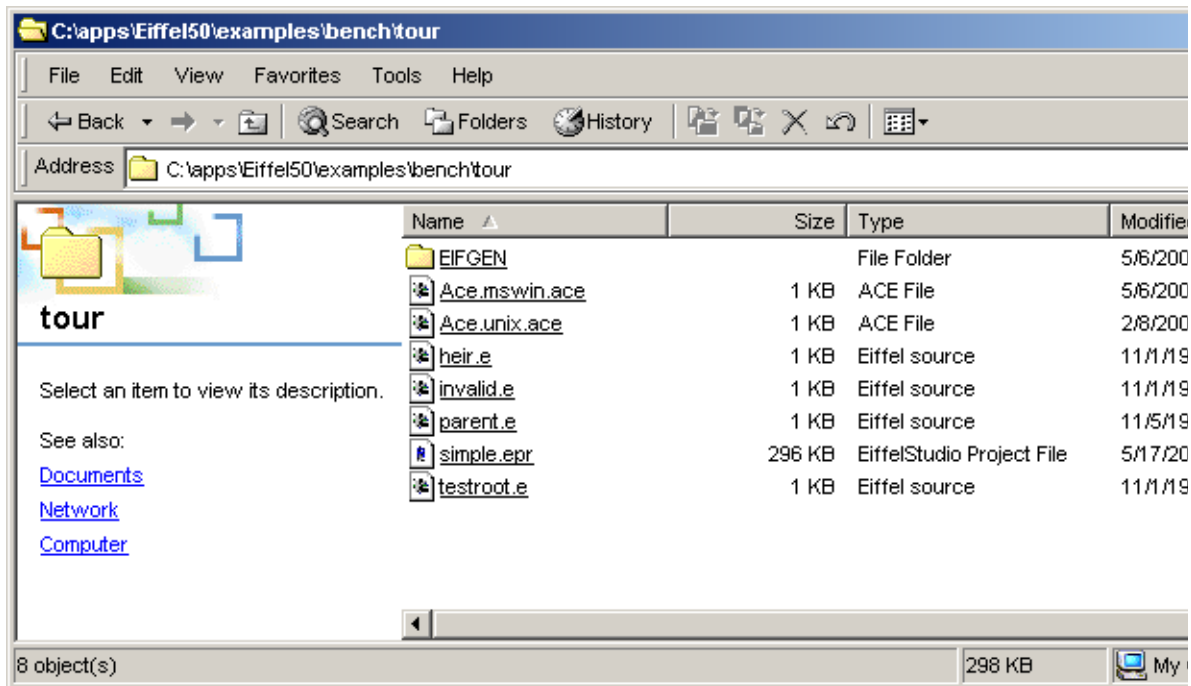
Before we proceed with the facilities of the environment, let’s take a look at the way EiffelStudio organizes the files of a project.

With EiffelStudio, you build project. Most projects yield an executable system, although you can also build a project just to define a library that will be used by executable systems.

Every session is relative to a project; you can start a new project from within EiffelStudio by going to the **File → New Project** menu, but please **don’t select that menu entry now** as we have many more things to do with our current project first.

Every project has a **project directory** which will contain the files generated and managed by EiffelStudio. The project directory may also host some of the source files containing your Eiffel classes, the Ace file, and external software written in other languages, but that is not required; the source files may reside anywhere. Some users, in fact, like to put nothing else than the EiffelStudio-generated files in the project directory; this separates user-managed and system-managed files, and can facilitate configuration management, backups and porting.

In this simple Tour, however, things have been set up so that all the files of interest, source texts as well as generated ones, will appear in the project directory *YOURDIR* (either `$ISE_EIFFEL/examples/bench/tour` or the copy that you have made). Go to that project directory using the Windows explorer or a `cd` command, and look at its contents (using `ls` on Unix/Linux):



The contents of this *YOURDIR* directory include the following:

- First you see a number of files with the extension `.e` (where `e` is for Eiffel): `heir.e`, `invalid.e` and others. These are the Eiffel source files, each containing one class. The recommended convention is, for clarity, to store a class of name `CLASS_NAME` into a file of name `class_name.e`, where `CLASS_NAME` is the lower-case version of `CLASS_NAME`; here, file `heir.e` contains the class `HEIR` and so

on. As you may remember, Eiffel is not case-sensitive, but the standard convention for class names is to write them in all upper case. The use of `class_name.e` as the file name is only a recommendation, not an obligation; but you *are* required to store one class per file. This keeps things simple and facilitate project management and configuration management.

- You also notice two files with an `ace` extension. These are the compilation control files. The reason there are two is that this example has a Windows version and a Unix/Linux version. As you remember, the Ace files for this example were available as part of the delivery; we used them to compile the project. In most practical cases, however, you won't need to build an Ace; if you use the “**Create a new project**” option of EiffelStudio (remember the first screen on page 10), EiffelStudio will build the Ace for you; if you change the Preferences during a session, EiffelStudio will update the Ace. When you become an experienced Eiffel developer you can also edit the Ace file directly; Aces are written in a simple Eiffel-like notation called **Lace** (Language for Assembling Classes in Eiffel). If you like you can take a peek at Lace by opening the relevant `.ace` file with a text viewer or editor now (but don't make any change); it shouldn't be hard to understand what it's all about. But that's not really necessary at this stage.
- Next you see a file called `simple.epr`. This is the Eiffel Project Repository, hence the extension. Any successful compilation of a project will produce a project of this kind, `system_name.epr`. To open an existing project when you start EiffelStudio (this was the option “**Open compiled project**” in the initial screen on page 10), you will just select the corresponding `.epr` file.
- Finally you will notice a subdirectory called **EIFGEN**, for “**EIF**el **GEN**eration”. **EIFGEN** is created and maintained by the compiler to store information about your project, including generated code for execution. EiffelStudio manages your project in such a way that **EIFGEN** can always be re-generated if need be; this means in particular that if things go wrong for any reason and you want to make a fresh start you can always delete this directory and the `.epr` file and recompile your system, recreating a new **EIFGEN** in the process. This also means that you should not add any files into this directory, or modify any of its files, since a later compilation is free to change or regenerate whatever it wants to in **EIFGEN**.

Later on, we will see that EiffelStudio may generate three more subdirectories of the project directory: **Diagrams**, if you produce graphical system diagrams; **Documentation**, if you request system documentation, for example HTML; and **Metrics**, if you perform measurements on your system. Other than these directories, **EIFGEN**, and `system_name.epr`, EiffelStudio will not touch anything in the project directory, so you may safely add and change whatever files and subdirectories you like.



You seldom need to look into **EIFGEN**, although you should know that it's there. Right now if you check the contents of the project directory **YOURDIR** (using the Windows Explorer on Windows, the **ls** command on Unix, or some equivalent mechanism), you will see that **EIFGEN** has been created, itself with some subdirectories, including **W\_Code** which contains the generated code (**W** for “Workbench” — we'll see the reason later). Feel free to browse through it if you like, but don't change anything.

By the way, we are done with any platform-specific instructions. Everything in the rest of this Tour, other than the graphical look-and-feel, will work the same across all EiffelStudio platforms.

## 6 STARTING TO BROWSE

It was important to take a look at how EiffelStudio stores your project, but unless your idea of fun is to poke around the file system to look at compiler-generated files that's not really the exciting part yet. Among the most innovative aspects of EiffelStudio is a unique set of facilities to *browse* through a software system.

### *Browsing styles*

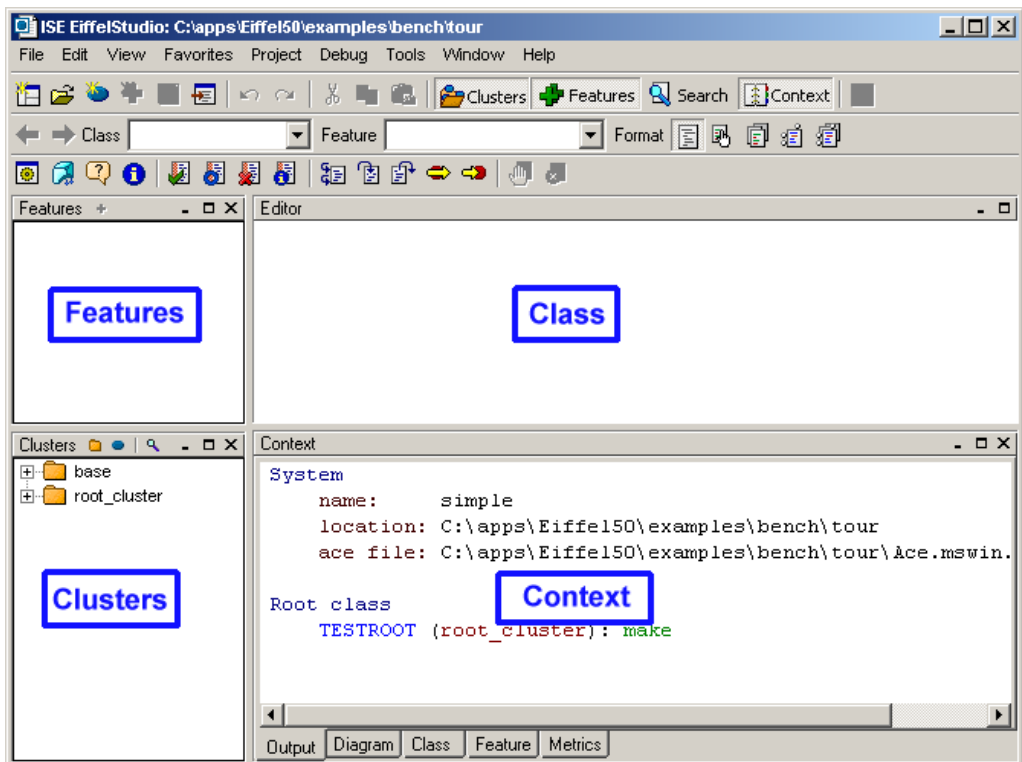
Browsing — traversing the structure — is particularly important in object-oriented development and especially in Eiffel because of the speed at which you can construct sophisticated class structures, making use of inheritance, genericity, the client relation and information hiding, and subjecting features to all kinds of adaptations — renaming, redefinition, undefinition, effecting — that are key to the expressive power of the software, but call for smart tools to keep track of what's going on. EiffelStudio's tools are second to none. Among their key properties:

- You can choose many different ways of browsing: sometimes you know the *name* of a class or feature, and will get to it just by typing its name; sometimes you want to traverse the system through its cluster-subcluster *structure*; often, you see a reference to element (class or feature) in the text of another element, and just want to get to it by following that reference, like a *hyperlink*. You'll be able to use all these techniques, and alternate freely between them.
- The browsing facilities are always available. There is no “browser” in EiffelStudio; you just browse when you want to, by looking at the information you need. You can do this while editing, debugging, or performing any other of the analysis, design, implementation, extension and maintenance tasks of system construction.

- Although classes are stored in files and clusters in directories, you can for the most part forget about the file system. Unlike most environments, which let you manipulate files containing software texts, EiffelStudio lets you concentrate on your *development objects* — the units that make sense for you: features, classes, clusters, systems. You think in terms of those conceptual units, and don't have to worry about where they are stored. Most of the time, you'll just forget about files and directories.
- You can produce many *views* of the development objects. For a class, you may see the full text, the interface only, the inheritance structure, the clients, the features, and many other views. You can even display *graphical* views along with textual ones. All these are fully browsable; you can go from one to the other as you please.

## A Development Tool

Let's see how this works. First, take a look at the EiffelStudio window:



If some parts are too small, just resize the window to arrive at something like what's on the figure. When it first comes up, EiffelStudio initially uses a fairly small window, because it's designed to run on a 800x600 display, although of course a bigger display is recommended. But as soon as you have resized it, EiffelStudio will come up, in the next session, with the size you've set.

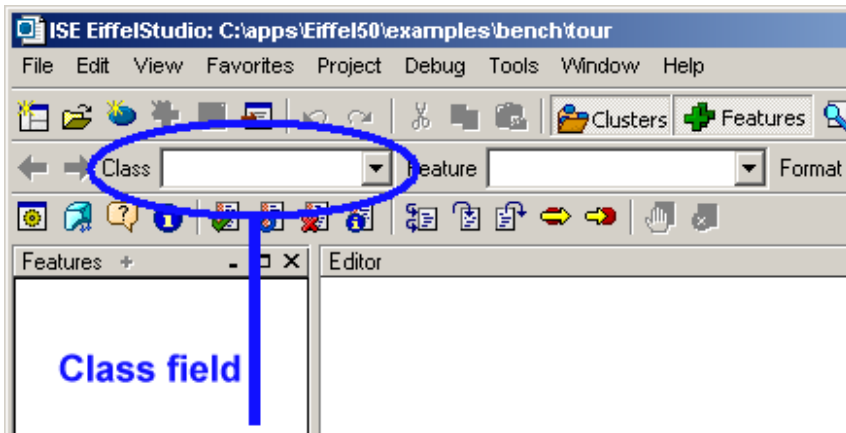
You can see four panes, marked on the figure: **Features**, **Class**, **Clusters**, **Context**. There will be others, such as **Search**, and you can remove any of them, except **Class**, at any time to make room for the others.

So far we have talked about “*the EiffelStudio window*”, but in fact that’s not correct. What you see is *one* “**Development Tool**”, of which you can have as many as you wish. Some people prefer to use a single development tool, avoiding screen clutter; others don’t think twice about having lots of windows, taking the “desktop metaphor” to its full conclusion (as many non-computer desktops are quite cluttered). There are many ways to start a new Development Tool; for example if you look at the entries in the **File** menu at the top left — don’t select any of these entries yet, just look — you’ll see, among others, **New window**, which would create a new Tool.

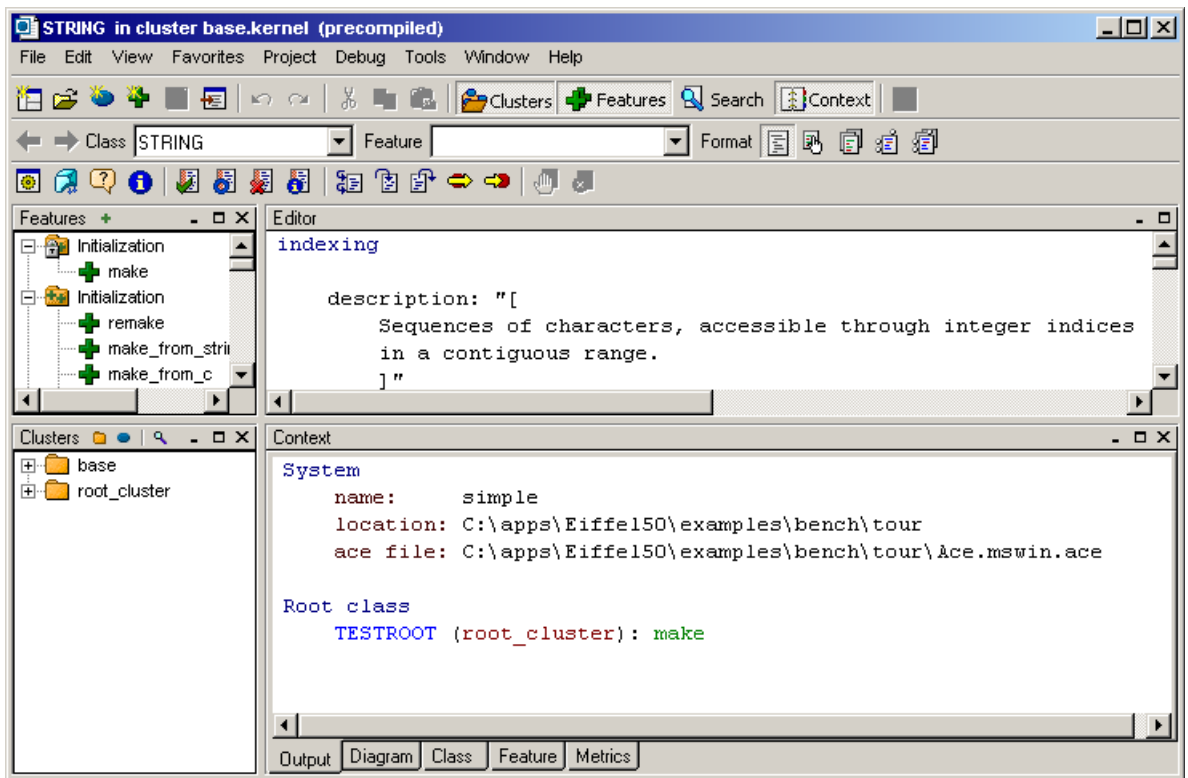
Whether you have one Development Tool or many, each one may have a **target**, which is an element of the system — system, cluster, class (the most common case), feature. This simply means that the tool displays information about that element.

### *Retargeting by name*

Our first example screen was targeted to the whole system. There are many ways to retarget it to a class. If you know the name of that class, you can just type it into the Class Field at the top left:



Let’s use one of the most basic classes, **STRING** from the Kernel Library of EiffelBase. Bring the cursor to the Class Field, click to make it active, type **string** (or **STRING**) followed by the Enter key. As shown on the next figure, this retargets the tool to class **STRING**. Note that you didn’t have to worry about where the class resides in the file system. Also, the Class Field will now show the target’s class name, **STRING**, in upper case since that is the standard Eiffel convention for class names. It doesn’t matter, when you enter the name into the field, whether you use lower or upper case (or some mix); EiffelStudio will show the name back in all upper case.

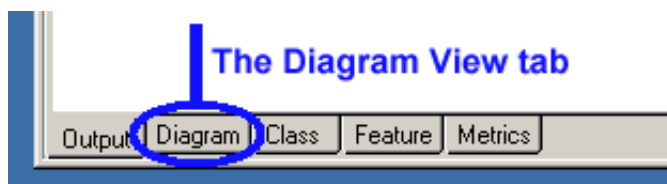


The Development Tool is now targeted to class `STRING`. Here's what the four panes show:

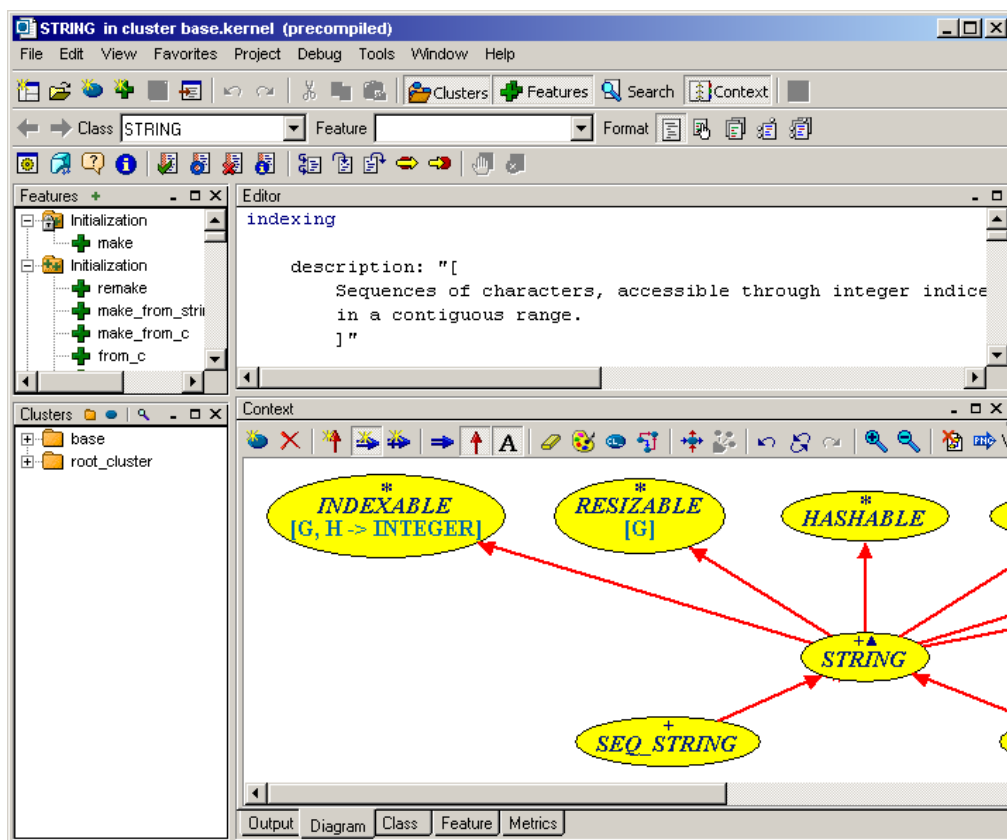
- Features pane, top-left: a tree view of all the features of the class, grouped by the feature categories (`Initialization`, `Access`, ...) as they appear in the class.
- Editing pane, top-right: the source text of the class, editable. For the moment the display is too small to display much of interest — it only shows the very first few lines — but we'll enlarge it when we need to modify class texts. (Not `STRING`, though, as it's not the kind of class you want to change during your first session with EiffelStudio, so just wait a bit.)
- Cluster pane, bottom-left: a tree view of the clusters of the system.
- Context pane, bottom right: a bag full of neat tricks. At the moment it just shows some general information about the system (where it resides, the name of its root class `TESTROOT`), but more generally it's there to give you all kinds of interesting views of the system, through the various tabs at the bottom: `Output`, `Diagram`, `Class`, `Features`, `Metrics`.

## A peek at diagrams

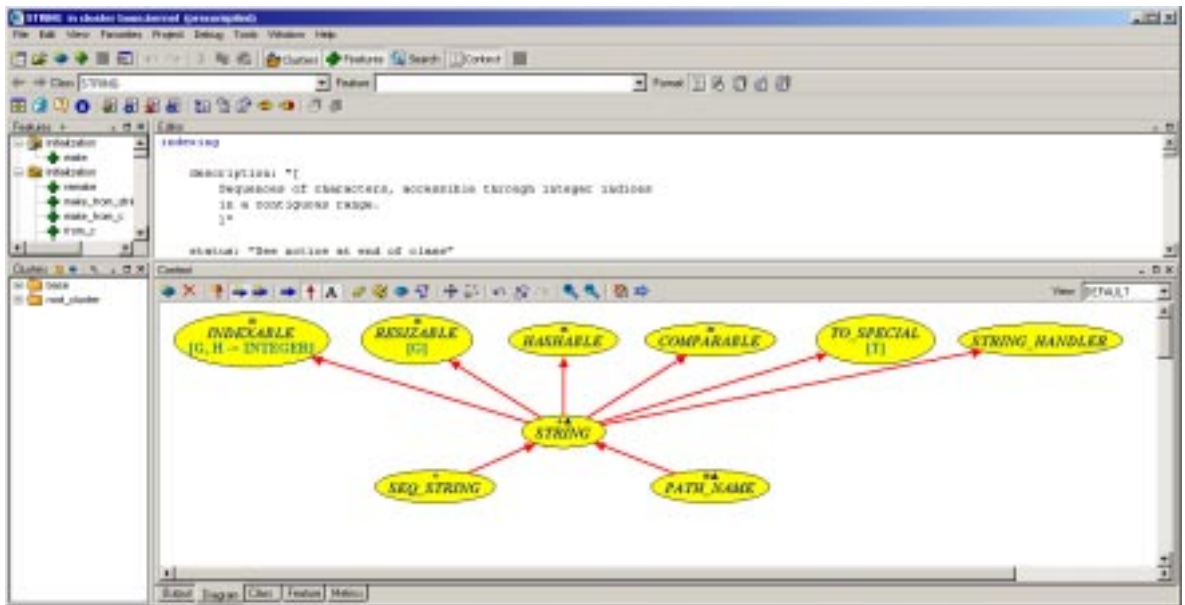
There's indeed a lot in the Context pane, too much to see right now, but to feed our curiosity let's just take a peek at the Diagram View. Click the **Diagram** tab at the bottom:



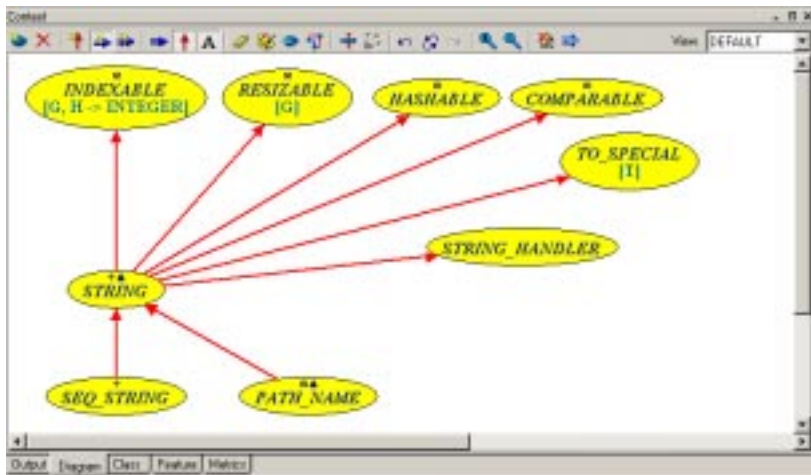
This displays a class diagram in the Context pane:



It doesn't show enough because it's cropped to the available area; to get the whole diagram, resize the window from the bottom-right corner. You can now see the inheritance structure:



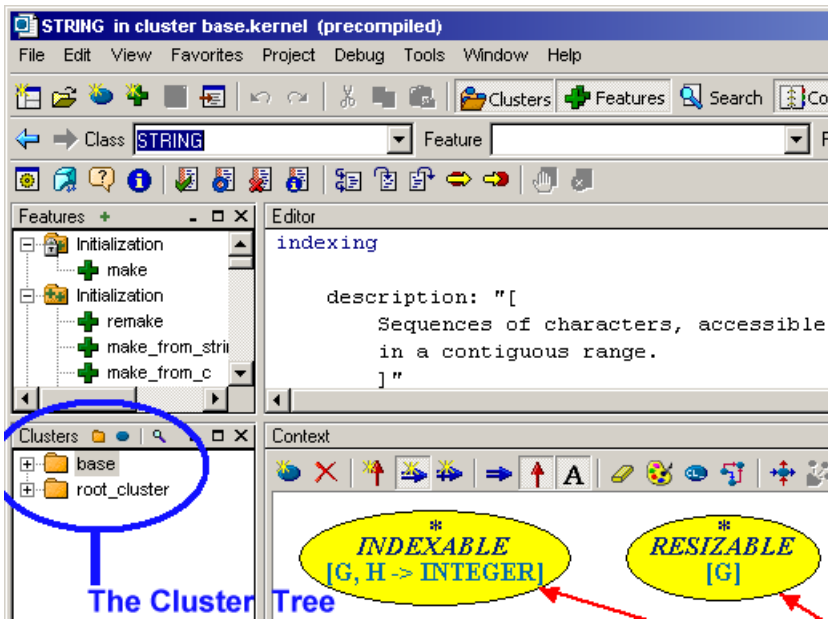
The red arrows show inheritance relations: *STRING* inherits from *INDEXABLE* and so on. You'll also be able to see the other key inter-class relation, client, but it's not particularly interesting for *STRING*. The default placement of the classes is quite good in this case, but if you don't like it you can change it; your changes will be retained the next time you bring up this diagram, in this session or a later one. (That's part of the information the **Diagram** directory keeps.) It will also be used in Web diagrams if you choose to generate an HTML form. To move a class bubble, just use standard drag-and-drop. For example you may move everything to the left and make the figure more compact:



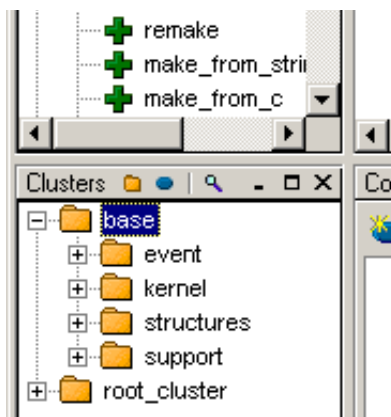
Later on you will learn how to use the Diagram View not just to display system structures graphically but also to build and modify systems. EiffelStudio indeed integrates the functionality of a CASE (Computer-Aided Software Engineering) workbench, seamlessly integrated with the programming facilities. But for the moment this peek at the Diagram View is enough, so let's get back to browsing.

### *Retargeting from the Cluster Tree*

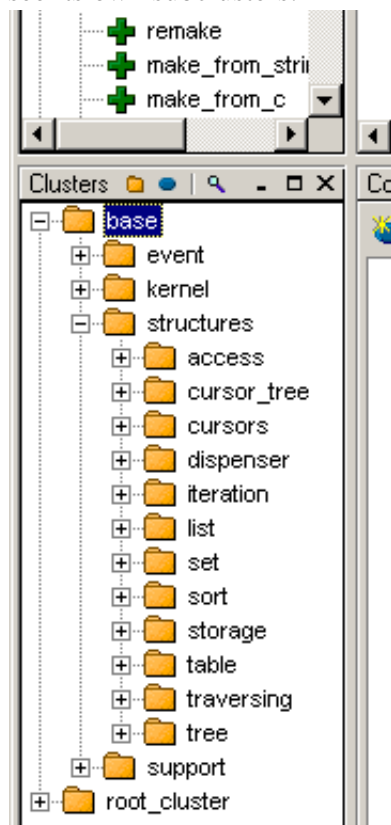
Your first browsing action used a class of which you knew the name, *STRING*. What if you don't know what's in the system and want to explore it? Among other techniques, you can let the Cluster Tree, in the Cluster pane at the bottom left of the Development Tool, guide you through the system's structure:



An Eiffel system, as you know, is organized into clusters, which you can structure hierarchically into subclusters. Here we see the top two clusters: *base*, containing the EiffelBase library; and *root*, containing the few classes specific to our Guided Tour system. Let's go into *base*, ISE's open-source library of fundamental reusable mechanisms. Click the little **+** sign to the left of its name; this expands the first level of the *base* cluster, to show its four subclusters: *event* for event handling, *kernel* for the Kernel Library, *structures* for the Data Structure Library, and *support* for additional supporting mechanisms.



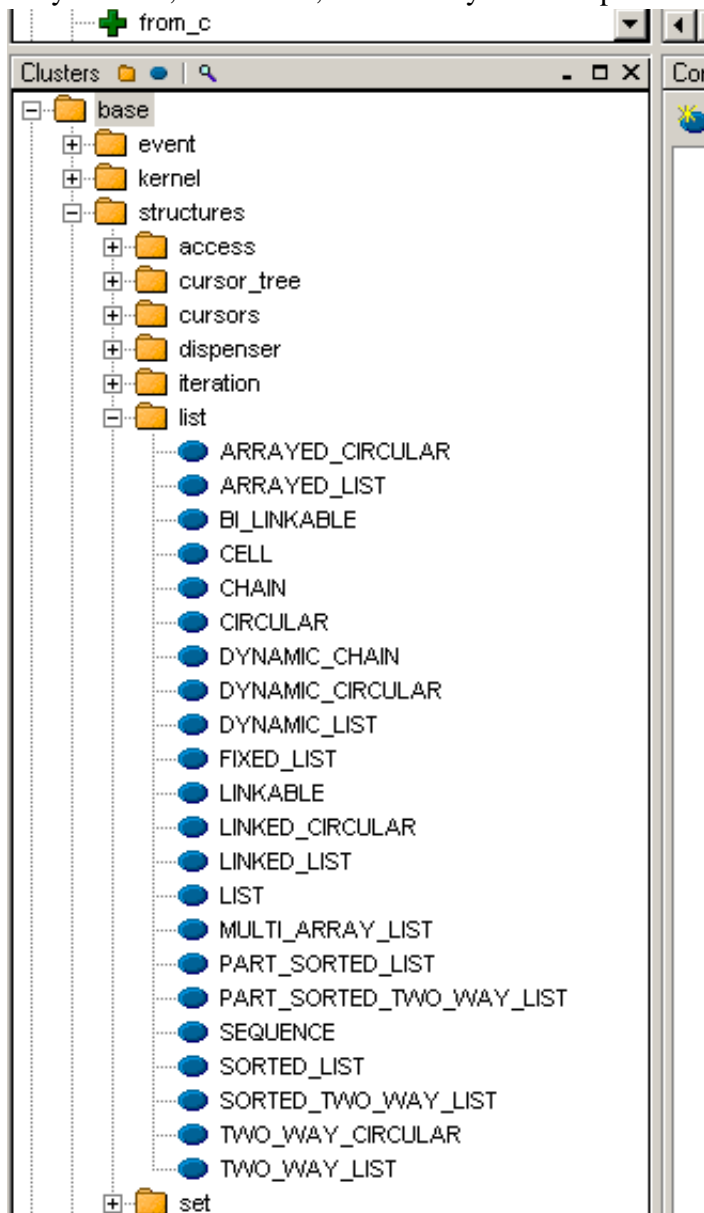
The most extensive of these EiffelBase libraries is [structure](#), which contains implementations of the major data structures and algorithms of computing science. Click the [+](#) next to [structure](#) to see its own subclusters:



If you initially don't see as many details as shown on this figure, you may get them by resizing the window, moving the vertical pane boundary, and possibly scrolling.

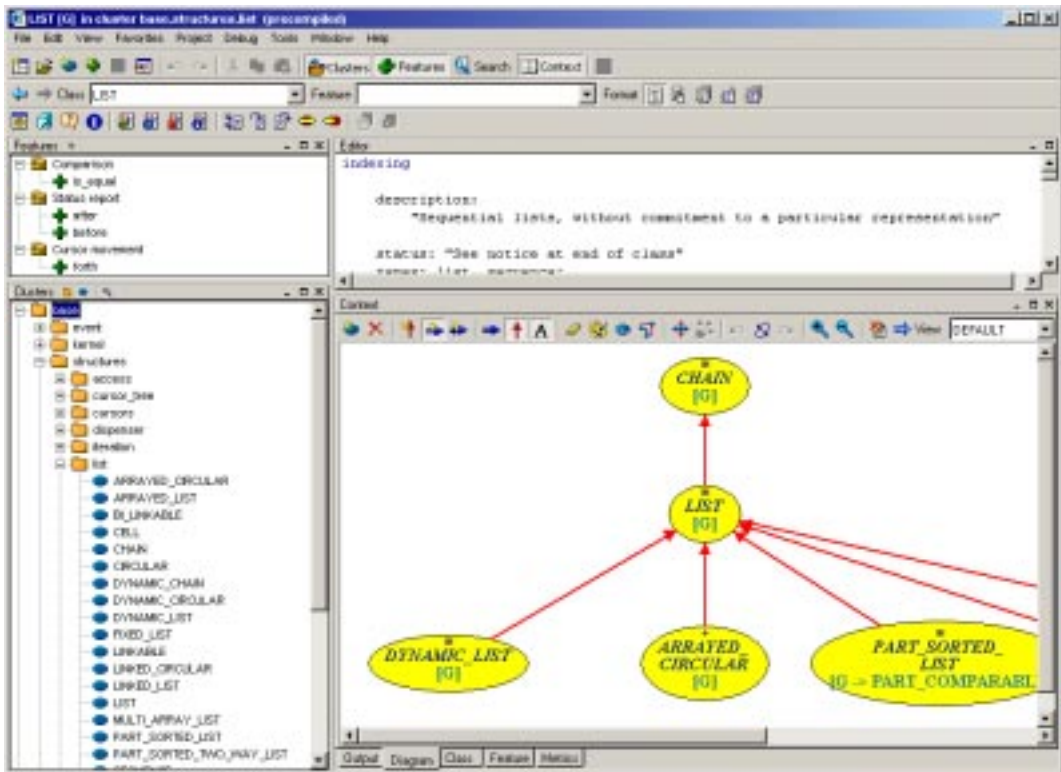


The EiffelBase Data Structure library and its subclusters are described in the book [Reusable Software](#). Let's go to one of the most frequently used subclusters, *list*, containing implementations of list structures. Click the **+** next to *list*. This time, since *list* is a terminal cluster, it's not subclusters you'll see, but **classes**, identified by small ellipses:



The ellipse, or “bubble”, is indeed throughout EiffelStudio, as in the Business Object Notation (BON, the underlying graphical conventions), the distinctive symbol for classes; remember the larger bubbles showing classes in the Diagram View a few moments ago.

Our second technique for retargeting a Development Tool to a class (other than typing the class name as we did before) is to click the class in the Cluster Tree. Do this now: click *LIST* in the tree. It doesn't matter whether you click on the class name or the adjacent bubble. This retargets the tool to class *LIST*. Because the Context Tool is still in *Diagram* view, it will display the inheritance structure for the new target class:

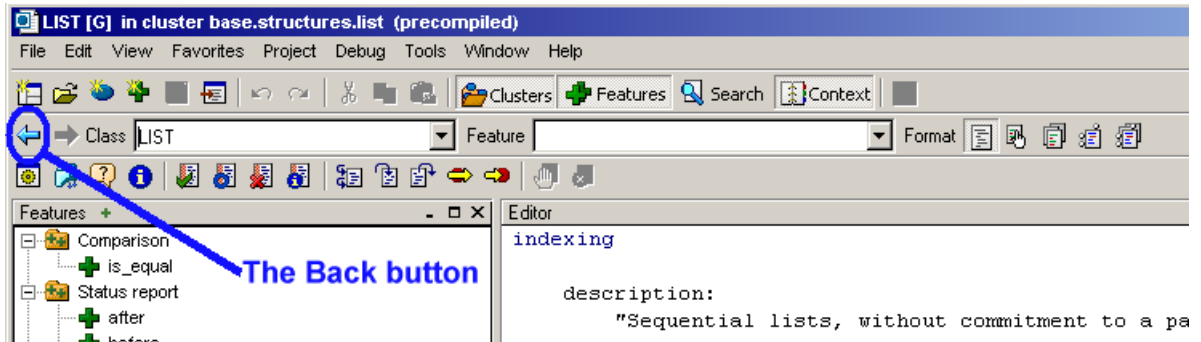


While this view is being produced you may see (or just get a glimpse of, if your machine is fast) messages indicating that it's producing the diagram for *LIST*. In a moment we'll switch views in the Context pane as we won't need the diagrams.

As the tool is now targeted to *LIST*, the Class Field at the top left now shows the name of that class, exactly as if we had typed that name, the way we did with *STRING* in the previous method of retargeting.

### *Moving back and forth*

Here now is a third way to retarget. Towards the top-left part of the Development Tool there are **Back** and **Forth** buttons, which will enable you to revisit classes seen during the current session:

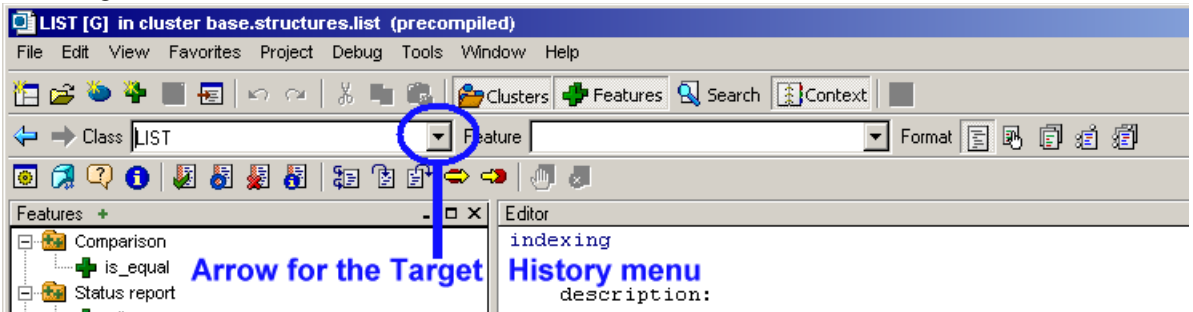


Click the **Back** button. This retargets the tool to the class you visited previously: **STRING**. The **Forth** button, immediately to the right of **Back**, becomes active. Click it to retarget back to **LIST**.

Note that all buttons of the interface have a “tooltip”: if you move the cursor on a button, **without clicking**, and wait a second or so, a small message comes up, explaining the purpose of the button. You may try this now on the **Back** and **Forth** buttons.

### The Target History

As a fourth way to retarget — there are more, but after this one we’ll stop counting — you can also use the Target History menu, which you can trigger through the little arrow to the right of the Class Field:

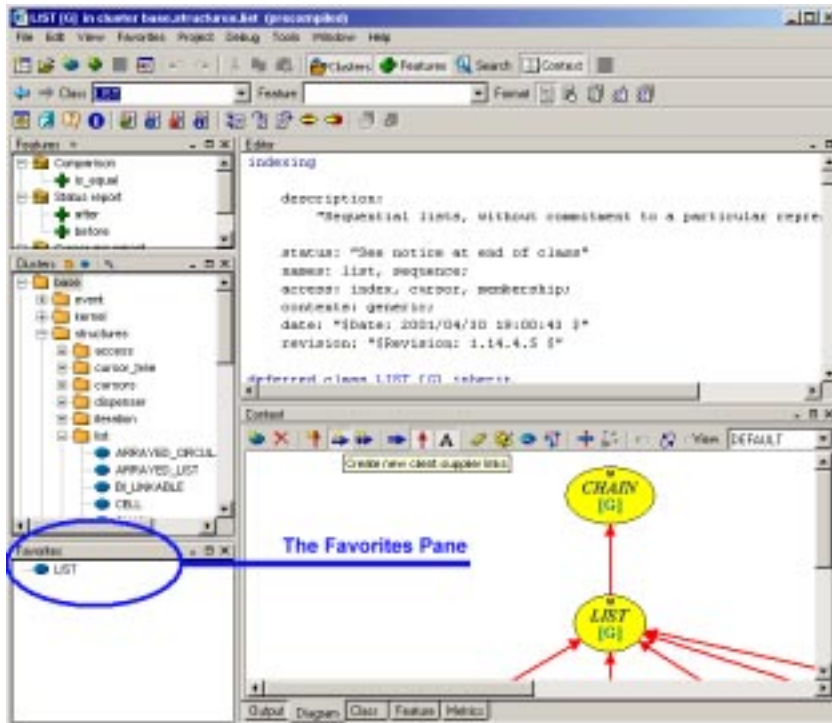


If you click this arrow — the little black triangle — you will see a menu of all your recent targets. Doing this now will only show the two classes visited so far, **STRING** and **LIST**, but later on there will be more entries. By default EiffelStudio remembers 10 classes; this is one of the settings you can change later if you wish, through the menu **Tools** → **Preferences**. (There’s no point in doing this now.)

### Adding to Favorites

If you find yourself often needing to examine a particular class, you can add it to your “Favorites”, similar to the favorites, also called *bookmarks*, which you use to retain interesting pages in a Web browser.

It's easy to add the current target — right now, *LIST* — to your Favorites. Do it now: go to the **Favorites** menu and select **Add to favorites**. Now display the favorites; one way is to go back to that same Favorites menu and select **Show favorites**. The Favorites pane appears below the Cluster Tree:



This also means one more way to retarget a Development Tool: click a class in the Favorites pane. *Two* ways actually, since even if you don't see the Favorites pane the class will appear in the **Favorites** menu and you can select it there.

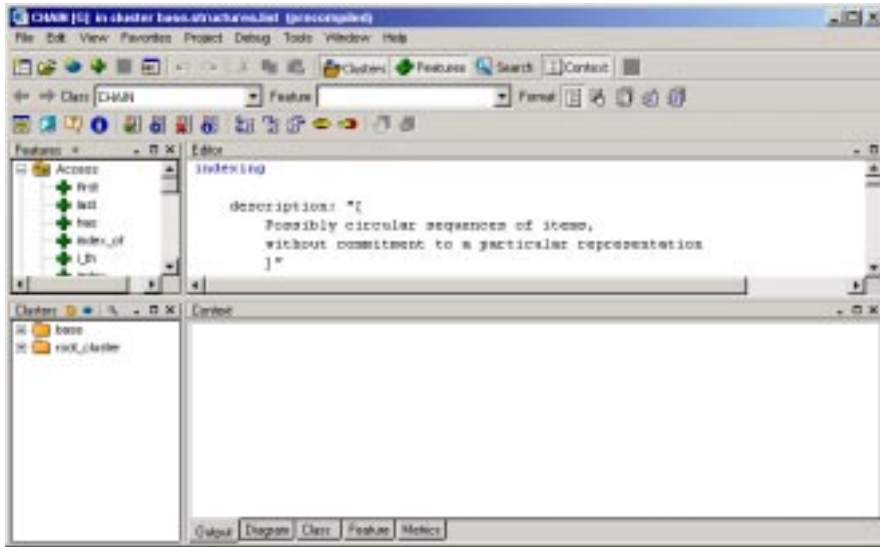
Right now we don't need the Favorites pane, so you can get rid of it by either selecting **Hide favorites** in the **Favorites** menu or clicking the little Close icon at the top right of the Favorites pane:



## Starting a new tool

With all the techniques seen so far, you have retargeted the current Development Tool to a new class. As noted, you may also wish to have two or more Development Tools targeted to different classes; this is useful to keep track of several things at one. A simple way to start a new tool on a class is to find the class somewhere in the interface and **control-right-click** on it.

Here, for example, the diagram in the Context Tool shows, at the top, a yellow bubble for the class *CHAIN*, a parent of *LIST*. (You can see it for example on the next-to-last figure.) Go to that bubble and control-right-click on it, that is to say, click with the rightmost button of the mouse while holding the CONTROL key on the keyboard. This starts a new Development Tool, targeted to the chosen class, *CHAIN*:



The place where we found the class *CHAIN* in the original tool (the one targeted to *LIST*) was the bubble representing the class in the Diagram View. But that's just one possibility. A general principle of EiffelStudio is **semantic consistency**: when you want to work with a development object — a class, a feature, a cluster — you can grab it wherever it appears in the interface, and in whatever format it is displayed: it may be in the Diagram View, in the text of a class in the Editing pane, in the Cluster or Feature Tree, or in any of the class documentation formats that we will soon see; and the form in which it appears may be text — the name of the class in a text document — or some graphical representation, such as a class bubble in a diagram. These variants don't matter: if the class or other development object catches your fancy, you can do whatever operations make sense for it, such as Control-right-click to start a new development object targeted to it, or any of the other operations we'll see next.

While you are at it, try a couple of other ways to create a new Development Tool. Go to **File → New window**; this creates a new tool, untargeted. The title bar says "**Empty development tool #1**". You can get the same effect by clicking the Create New Window icon, leftmost on the top row of buttons, just below "**File**". The corresponding keyboard accelerator is CTRL-N.

## 7 CLASS VIEWS

We haven't even looked at a class text yet, but it's important anyway to see how EiffelStudio provides you with numerous, complementary *views* of your software. The Context pane is the primary place to look for such views.

We'll need just one Development Tool for the moment, the one that was targeted to *LIST*. You can get rid of the others by closing their windows (through the top right cross mark on Windows and the equivalent in other window managers), or through **File → Close** — but don't select "Exit" which would take you out of EiffelStudio altogether!

If you don't see a tool targeted to *LIST*, just retarget one, as you know how to do this now, for example by typing the name followed by Enter in the Class Field at the top left.

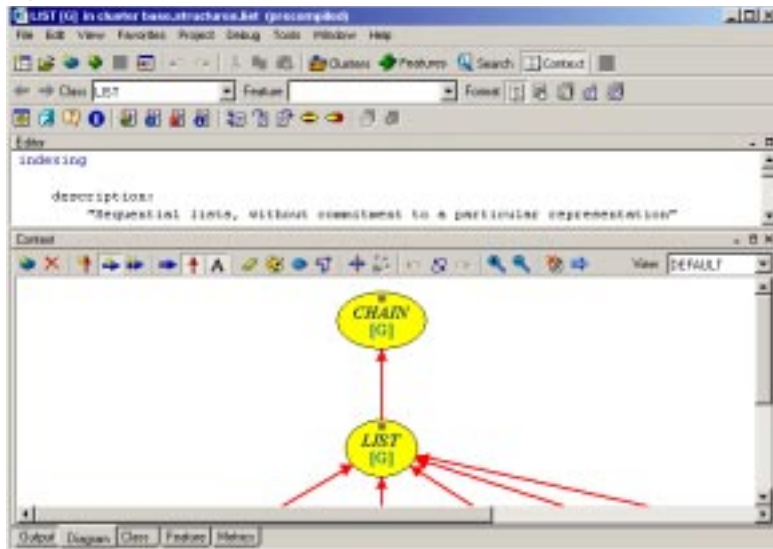
First let's give ourselves more space. Right now we don't need the Cluster Tree and Feature Tree panes. Get rid of them by clicking the corresponding buttons on the top toolbar:



You can get these panes back later by clicking the same buttons again.

Another way to hide a pane is to click its Close icon, the little cross mark highlighted (for the Features Tree pane) on the left in the last figure.

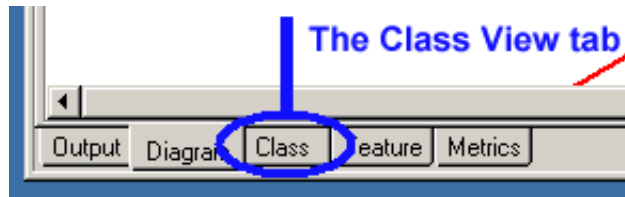
Two panes remain, Text and Context.



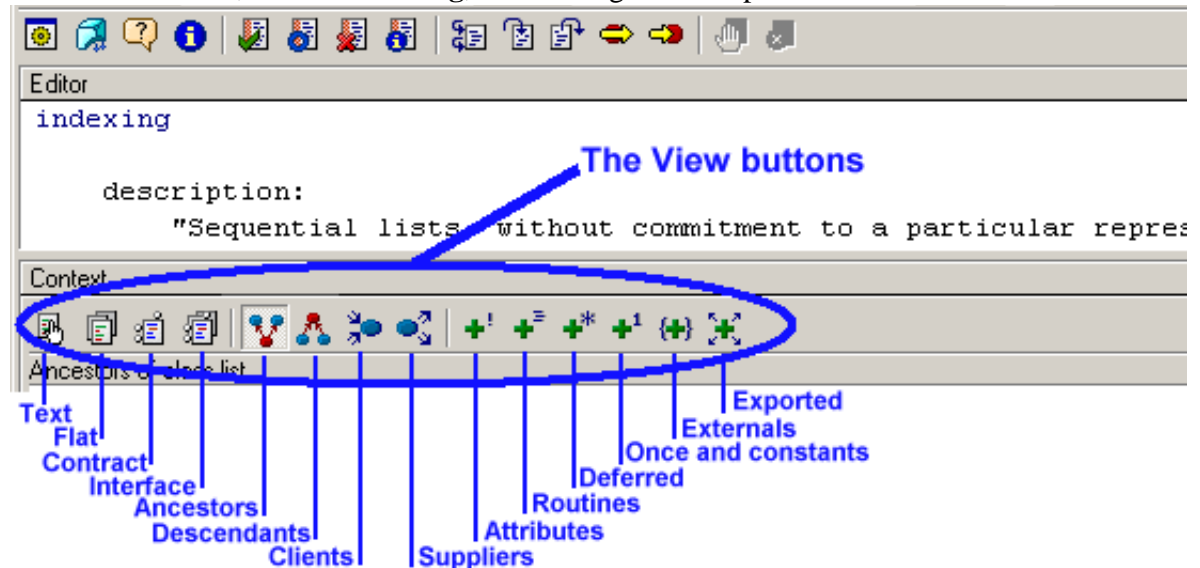
Make sure the Context pane is large enough; you can resize the window and, if necessary, narrow down the Editing pane since we don't need it for the moment. Don't worry, though, if the Context pane shows only part of the diagram, as it does on the last figure, since we will use the Context Tool to show other views.

### The Class View

The first view we'll look at is the Class View. You get it by clicking the corresponding tab at the bottom:

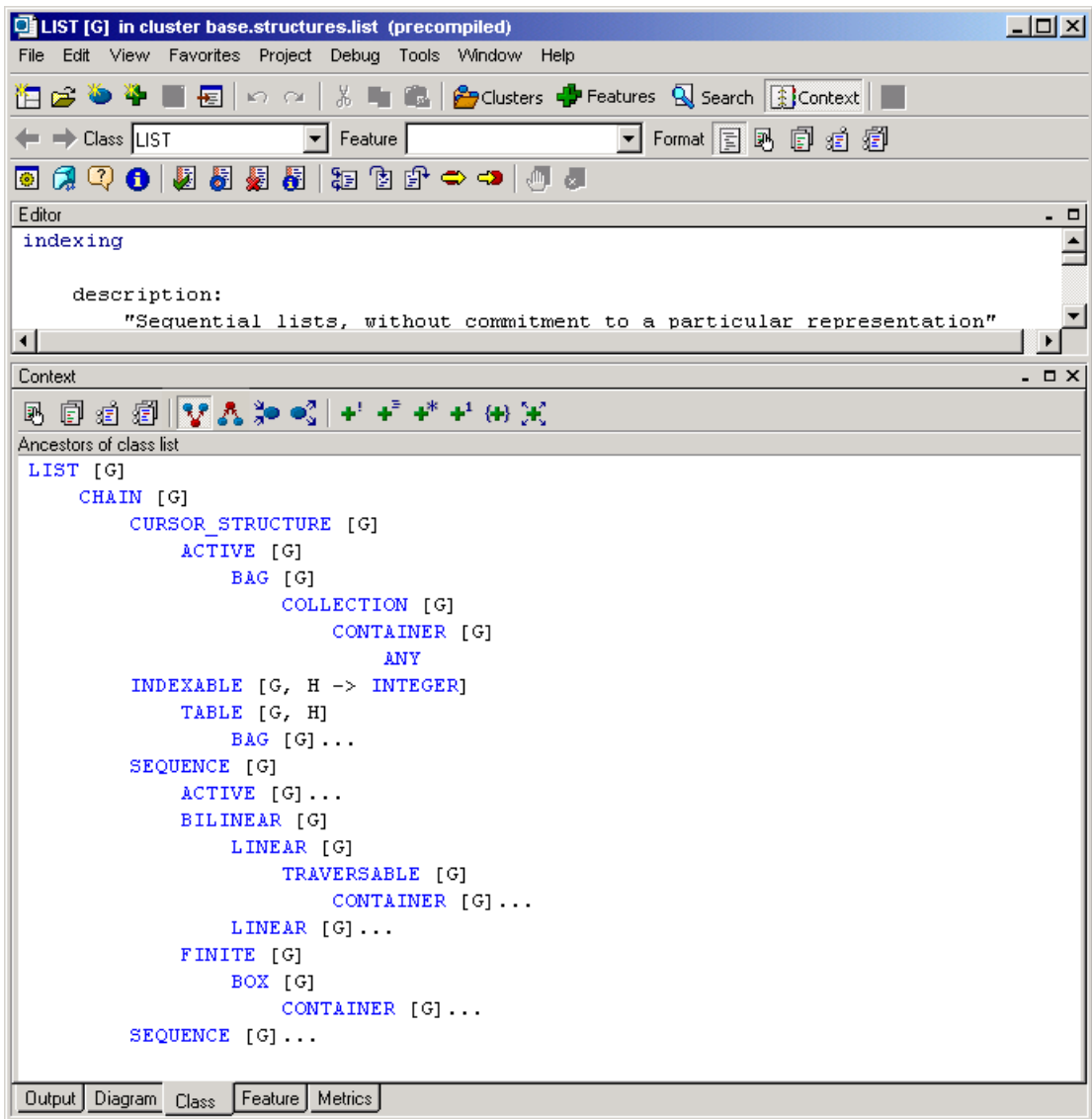


This gives you access to many forms of information about the current class — the target of the Development Tool. A set of buttons at the top of the Context pane enables you to display a number of **views** of the class. The currently highlighted button indicates the default view: **Ancestors**. You can see the others' names by moving the cursor over the various view icons, **without clicking**, and reading the tooltips.



The view currently displayed, **Ancestors**, shows the inheritance structure that leads to the current target, *LIST*:





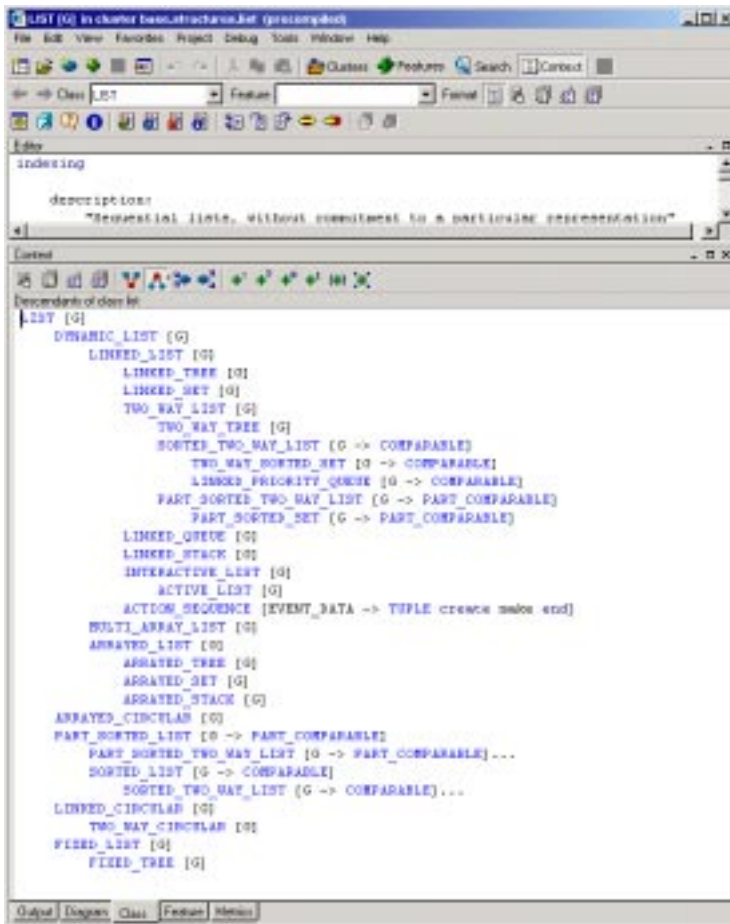
This shows that *LIST* is an heir of *CHAIN* which itself, as an example of multiple inheritance, is an heir of *CURSOR\_STRUCTURE*, *INDEXABLE*, and — twice, as an example of *repeated* inheritance — *SEQUENCE*. If, because of direct or indirect repeated inheritance, a class appears more than once, the display doesn't repeat its ancestry the second and subsequent times; the omitted repetition appears as just three dots, ..., as illustrated here for the second occurrences of *BAG*, *ACTIVE* and others.

As you may have guessed, all the class names that appear on this display, by default in blue, can function as hyperlinks: you can use any one of them to retarget the



Development Tool to the corresponding class. This will be another major retargeting mechanism. But let's not pursue it for the moment and instead continue looking at the documentation views.

Next to **Ancestors** is **Descendants**, which will give you the descendants of a class in a similar format:



The progeny of **LIST**, as you can see, is just as impressive as its ancestry.

Let's now look at the other formats starting from the left. The first button, **Clickable**, gives the class text. It's essentially the same information as appears in the top Editing pane (reduced to its bare minimum in the last few pictures, to show only the first three lines or so), but with some differences:

- The top Text view is editable. In fact it will be your primary tool for entering software texts. The bottom **Clickable** view is just a view; you can't change it.

- The Text view retains the formatting of the class text the way it was typed in; the **Clickable** view is automatically formatted — “*pretty-printed*” — according to the standard Eiffel layout rules.
- The **Clickable** view does not include comments inside routine implementations (**do** and **once** clauses), although it does retain features’ header comments.
- As part of the pretty-printing, the **Clickable** view uses colors and fonts to distinguish keywords, identifiers, comments and other syntactical elements. You can change the fonts and colors, like many other elements of the interface, through **Tools → Preferences**. (Now’s not the time.)

This view is called “clickable” because, as we’ll see later, every syntactical element on it is a hyperlink, which you can use for browsing. Here is the beginning of the *LIST* class text in **Clickable** view:

```

class
    indexing
        description:
            "Sequential lists, without commitment to a particular representation"
        status: "See notice at end of class"
        name: list, sequence
        access: index, cursor, membership
        contents: generic
        date: "Date: 2001/04/30 19:00:43 @"
        revision: "Revision: 1.18.4.5 @"

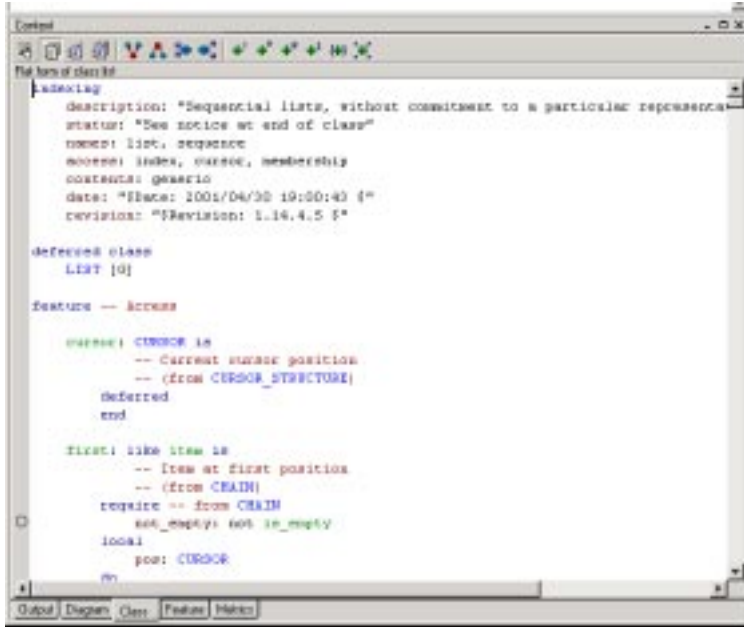
deferred class
    LIST [G]

inherit
    CHAIN [G]
    redefine
        forth,
        is_equal
    end

feature -- Comparisons

    is_equal (other: like Current): BOOLEAN is
        == Does 'other' contain the same elements?
    loop
        c1, c2: CURSOR
        do
            if Current = other then
                Result := True
            else
                Result := (is_empty = other.is_empty) and (object_comparisons = or
                    if Result and not is_empty then
                        c1 := cursor
                        c2 := other.cursor
                        check
                            c1 /= void and c2 /= void
                        end
                    end
                )
            end
        end
    end
end
  
```

After **Clickable** comes the **Flat** view button. The layout of the result is similar:



The flat form of a class is the reconstructed class text including not only what's declared in the class itself but also everything that it inherits from its ancestors, direct or indirect. This applies to the flat form's features, which include ancestor features, but also to contracts: the flat form's invariant includes all clauses from ancestors' invariants, and the preconditions are expanded to take **require else** and **ensure then** clauses into consideration. (The [Eiffel Tutorial](#) explains these notions in detail.)

As a result, the **Flat** view shows the class text as it might have been written had inheritance (what a horrible thought even to contemplate!) *not* been available to write it.

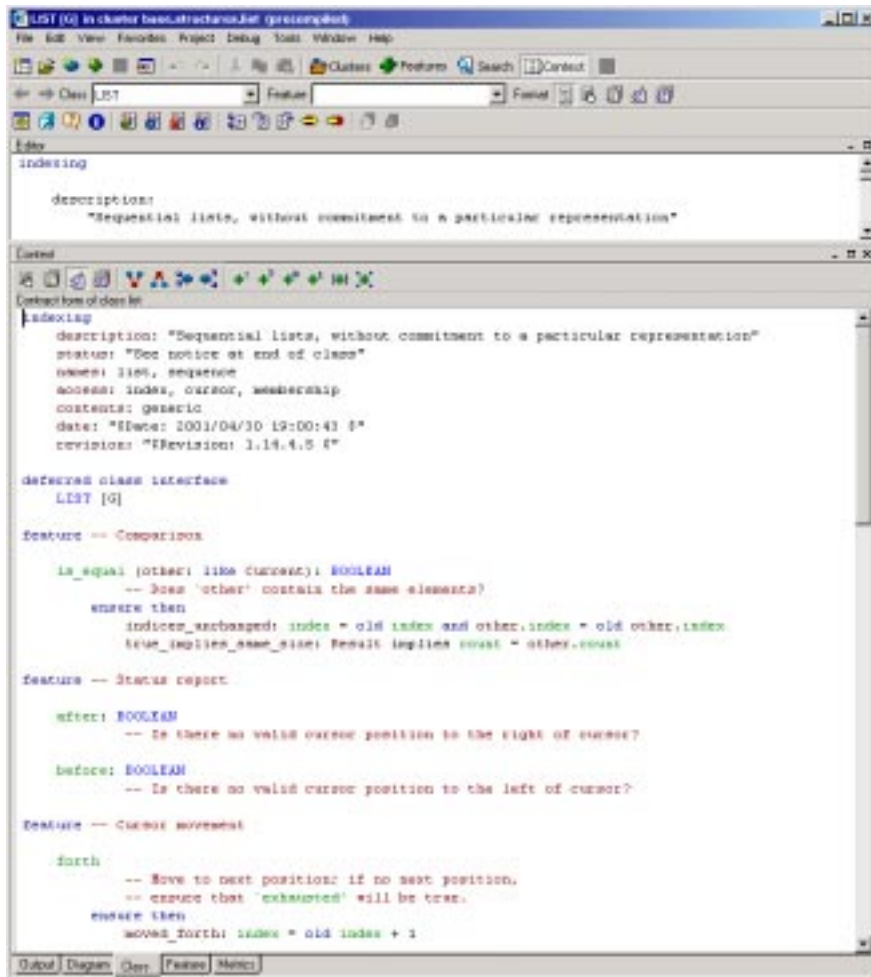
The first two features appearing in the above display, *cursor* and *first*, are indeed inherited from ancestors, rather than declared in *LIST* itself. Note how EiffelStudio, when producing the flat form, adds a line of the form -- (From *CLASS\_OF\_ORIGIN*) to the header comments of inherited routines, to document where they come from.

The flat form is an important notion of object technology, making it possible to understand a class by itself, regardless of the sophistication of the inheritance structure that led to it. Looking at the Flat view of *LIST*, you may note how little of the information comes from the class itself; most of the interesting work has been done in ancestors, and *LIST* just adds a few details.

If at any time you want to search for a certain pattern in the views displayed, click the **Search** button at the top of the window, or type CTRL-F. A self-explanatory Search pane will come up, with various options such as *Match case* and *Whole word*.

Next come two essential documentation views: **Contract** and **Interface**. Based on Eiffel's principles of Design by Contract, they document the interface properties of a class. Unlike the previous two, they do not show actual Eiffel texts, but information useful for client classes.

Here is the beginning of the **Contract** view for our example class *LIST*:

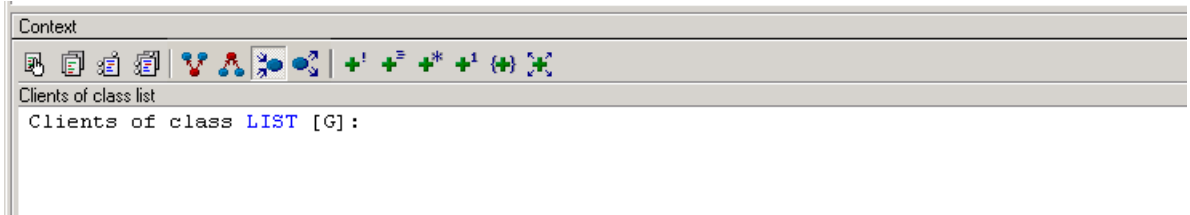


The contract form (also known as the **short form** of a class) is the class text deprived of any internal detail to retain interface information only. It discards any feature that's not exported (available to all clients); for the retained features, it discards the implementation — **do** or **once** clause — but retains the header (feature name, arguments, results), the header comment, and the contracts (precondition, postcondition, invariant) minus any contract clause that refers to a non-exported feature and hence would be useless to clients.

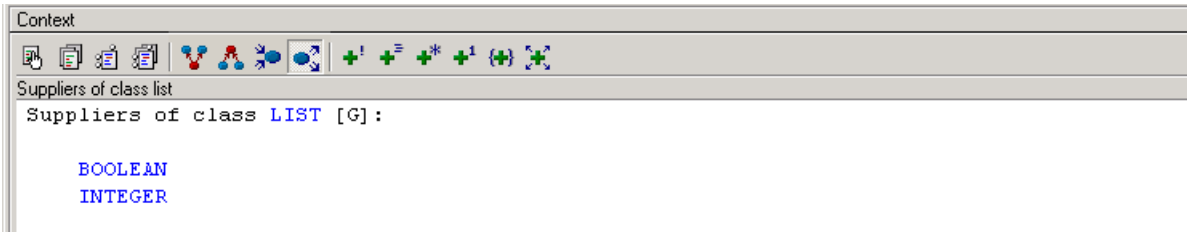
As you will know, especially if you have read the book [Object-Oriented Software Construction](#), the contract form is the preferred way of documenting software elements, especially reusable components, as it provides clients with just the right level of abstraction: precise enough thanks to the type signature and especially the contracts; clear enough thanks to the header comments; and general enough since it omits implementation details that are irrelevant to client programmers (and might lead them to write client code that won't work any more if the implementation changes).

In practice you will often want to use, instead of the **Contract** view, the next one, **Interface**, also known as “flat-short form”, which applies the same rules to the flat form rather than to the original class. This means it shows information on all the features of the class, immediate (defined in the class itself) as well as inherited, whereas the short form, non-flat, only considers immediate features. The **Interface** view provides the complete interface information for the class. Try it now on class **LIST**.

The next two buttons are for the **Ancestors** and **Descendants** views, which we have already seen, showing classes connected with the target through one of the two inter-class relations, inheritance. After them come **Clients** and **Suppliers**, to list the classes connected through the other relation, client. Clicking the **Clients** button shows the (empty) list of clients of **LIST**:



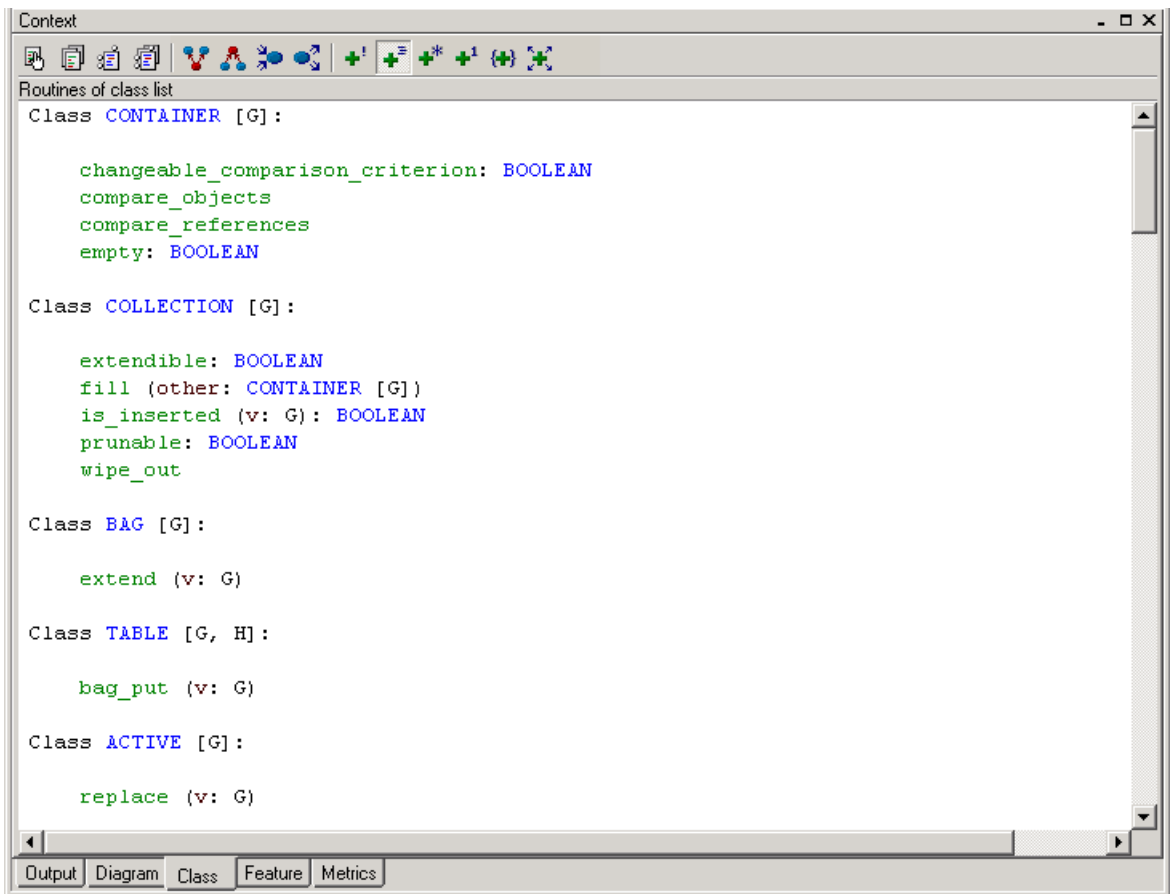
No class of this system directly uses **LIST** as client, although some use its descendant **ARRAYED\_LIST**. Now click the next button to see the **Suppliers** of **LIST**:



The only two classes that **LIST** needs for its own algorithms are basic types from the Kernel Library, **BOOLEAN** and **INTEGER**. In Eiffel, as you may remember, all types are defined by classes, even those describing such elementary values as integers and booleans.

## Feature information in the Class View

Let's resist the natural urge to go see now what the classes *INTEGER* and *BOOLEAN* look like, and instead continue our survey of views. The remaining views will all display information about the **features** of the class. The first of them, **Attributes**, lists the attributes. It's not very interesting for *LIST*, a deferred class with only one attribute — you can check this for yourself by clicking the **Attributes** button — so let's look at the next one. Click the **Routines** button now to display information about the routines of class *LIST*:



The sections of this display group routines according to the ancestors of *LIST* — including *LIST* itself — that first introduced them; for example (second and third sections) *extendible* originally comes from *COLLECTION* and *extend* from *BAG*. Much of the benefit of this display comes from its support for browsing: all the colored elements, representing classes and features, will be “clickable” hyperlinks.

The remaining Class View buttons all display information in the same format. Each selects a specific subset of the target class's features. The last two selected attributes and routines. You can now try any of the others by clicking the corresponding button:

- **Deferred** features: abstract features which don't have an implementation in the current class, only in eventual descendants. Try this for *LIST*; you'll see that this deferred class indeed has a number of deferred features.
- **Once and constants**: constant attributes, "once functions" which provide shared objects (close to the "singleton" pattern), and once procedure which provide a convenient initialization mechanism. *LIST* hasn't any.
- **External** features, implemented as calls to routines, macros or other elements implemented in other languages. *LIST* hasn't any.
- **Exported** features: all the features available to general clients. *LIST* has quite a few.

All the views you have now learned to produce were about *classes*. It's also very useful to obtain information about what happens to your *features* throughout the classes where they appear. This will be the purpose of the **Features** tab of the Context pane. But before we look at it let's see how to do more with the Class View facilities just seen, by exporting them to the outside world: to the Web, to a text processing tool, or in fact any other tool.

## 8 PRODUCING AND EXPORTING DOCUMENTATION

Software development is, most of the time, cooperative work. You must tell the rest of the team what you're up to, and find out what they can offer you. Bring in distributed development — increasingly common these days, with some people working at headquarters, others at home, others traveling, an offshore team half a world away... — and the problem becomes even more critical.

EiffelStudio provides unique facilities to make such distributed development possible in a safe, effective, harmonious way. Some of the key criteria are:

- You must be able to export the information easily to the World-Wide Web, the most general and widely available interaction mechanism, preferred by most people to all the alternatives.
- The documentation must be *faithful* to the software. Because of the ever-changing nature of software, this goal is impossible to satisfy unless the documentation is *extracted* from the software — as opposed to the traditional approach, still perpetuated by many CASE tools, of treating the two as separate.
- The task of updating the documentation after a software change must be straightforward and automatic.
- It's not enough to support HTML; many other formats are useful too.
- Users must have the ability to tune the mechanism to support *new* formats.
- For existing formats, they must have a way to tune the output easily to any specific style standards, company policies, local variants.

EiffelStudio's documentation generation satisfies all these requirements.

### *Documentation filters*

Let's see how documentation works by starting to generate it for our Guided Tour system — which really means for EiffelBase, since that's what it mostly consists of. The HTML result is available as part of the present documentation (we'll tell you where in just a minute), so you don't have to regenerate it unless you want to. Indeed we'll tell you when to click **Cancel** if you are happy with the pre-generated version. But let's get started anyway to understand the principles and possibilities.

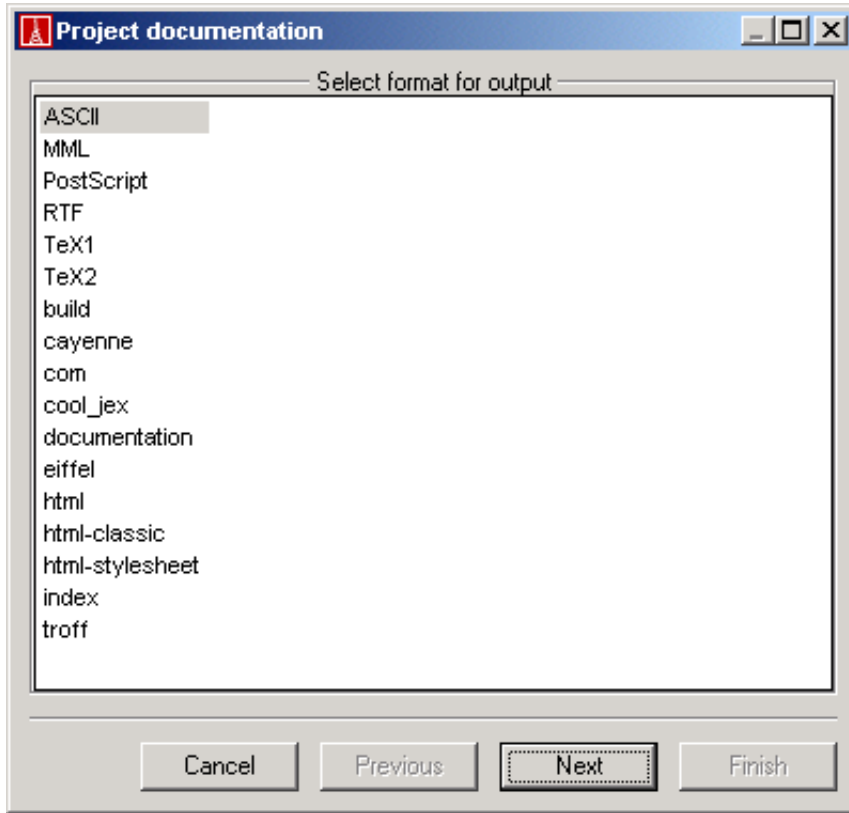
Click the following menu entry, used to generate documentation:

**Project → Documentation**

This is the next-to-last entry in the **Project** menu. The last one, by the way, **Export XML...**, is directly relevant too: it will make it possible to export information in the standard XML representation for UML, for consumption by third-party products such as Rational's Rose. But for the moment we choose the generate **Documentation** entry to start the Eiffel Documentation Wizard.



The Wizard starts with a list of available output formats, also called *filters*:



The filter names correspond to major documentation formats which EiffelStudio supports by default. Among the most important, listed here in rough order of appearance in the list:

- **ASCII**: plain text, no formatting codes.
- **eiffel**: essentially the same as ASCII; useful if you want EiffelStudio to pretty-print your class texts and replace the originals, as explained below.
- **MML**: internal format for Adobe FrameMaker.
- **Postscript**: to generate Adobe Postscript output, suitable for printing on a Postscript printer, display on a Postscript previewer such as Ghostscript, or distilling to Adobe PDF.

- **COM**: to generate class specifications in the form of an Interface Description Language (IDL) interface for Microsoft’s COM component model.
- **RTF**: Microsoft’s Rich Text Format, used in particular for Windows “Help” files.
- **TeX1**, **TeX2**: two variants for Donald Knuth’s T<sub>E</sub>X processing format.
- **COM**: to generate class specifications in the form of an Interface Description Language (IDL) interface for Microsoft’s COM component model.
- **troff**: if you already know what this is, congratulations (or condolences), you’ve been around the industry for a while. This is a traditional text-processing format available on Unix systems. Also works for the *gtroff* variant.
- **html-classic**: HTML, no style sheets. The next variant, *with* style sheets, is strongly recommended unless your colleagues will be reading your documentation with Mosaic 1, vintage 1993, or Netscape 2, Vintage 1995.
- **html-stylesheet**: HTML with style sheets. This is particularly attractive for Web publishing not only because the output makes full use of style sheet capabilities (fonts, colors, layout, formatting) but also because it becomes trivial to change the look-and-feel to support any style you or your users like, even *after* generation, simply by editing the style sheet file.

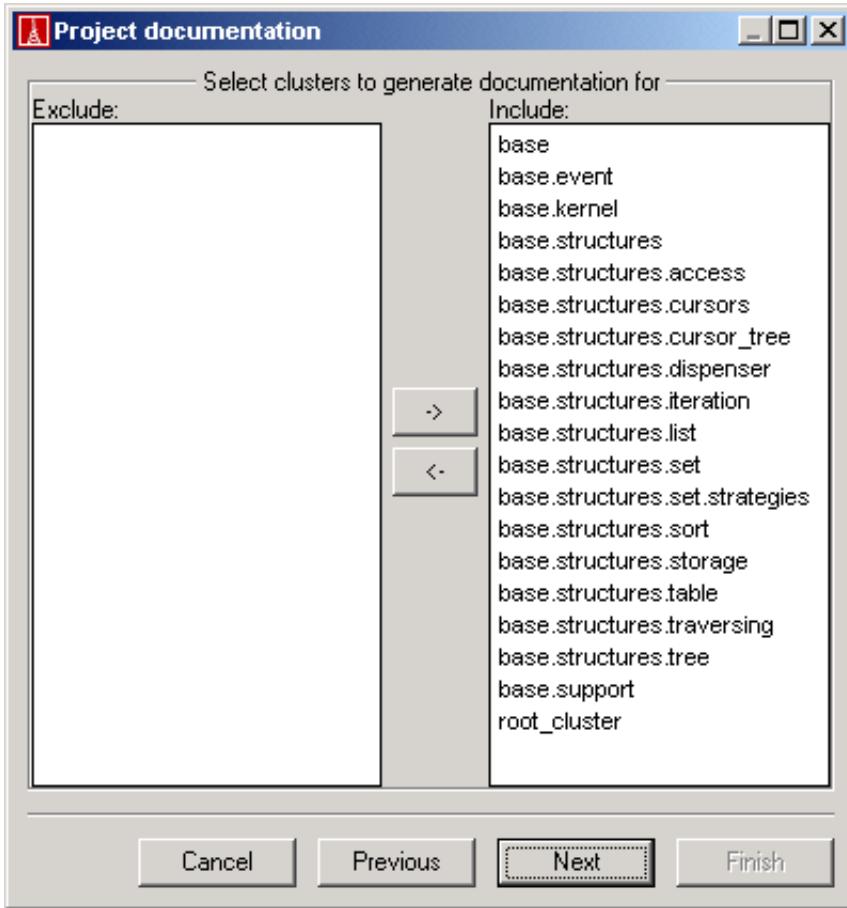
Not only do these predefined filters provide support for a number of important industry formats; best of all, if you want *another* format not represented on the list, or would like to adapt an existing format to your own style preferences, it’s easy to define a new filter. The list that EiffelStudio displays comes from the files with a **.eff** extension that it finds in a subdirectory of the installation:

**[\\$ISE\\_EIFFEL/examples/bench/filters](#)**

To define a new filter, simply add a file to this directory. Filters are expressed in a simple notation called EFF (*Eiffel Filter Format*), general enough to support a wide variety of tools for text processing, project management, Web publishing etc. The best way to define a new filter is usually to start from an existing one and adapt it. You will find the specification of EFF at the end of this manual, in **[“APPENDIX: WRITING DOCUMENTATION FILTERS WITH EFF, THE EIFFEL FILTER FORMAT”, 19, page 144.](#)**

### *Generating an HTML record of your project*

Let's select the most obviously attractive of the predefined filters: HTML with stylesheets. Click the line **html-stylesheet** in the list to make it active, then click **Next** at the bottom of the Documentation Wizard window. The next window appears:



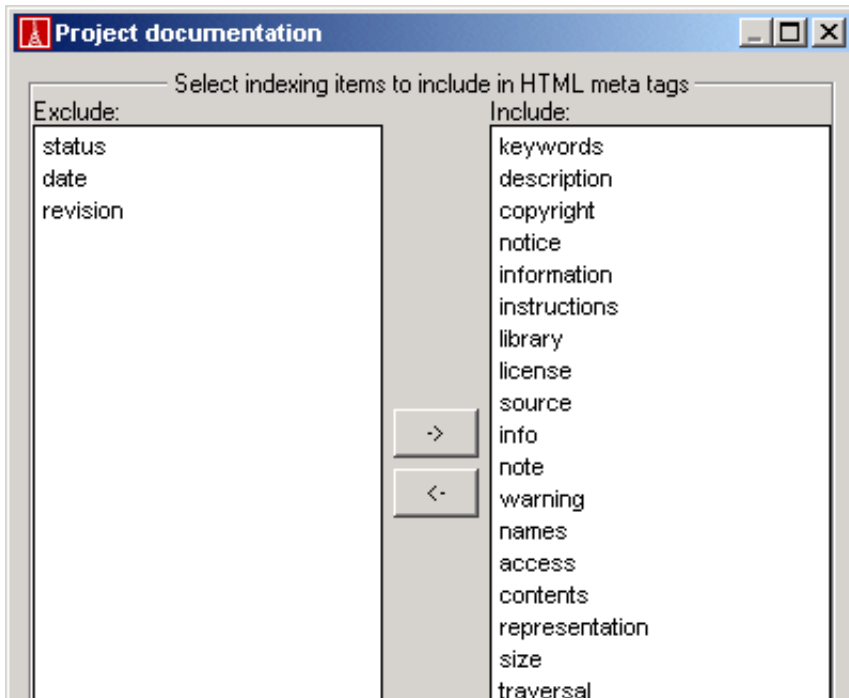
This is to let you decide which clusters of your system the documentation will include. Initially all clusters — down to the level of nested subclusters, for example **base.structures.list** — appear in the **Include** list on the right side; but you might want to exclude some standard libraries or other clusters from the documentation.

To move a cluster from the right column to the left one, click it to select it, and click the left arrow button; for the reverse, use the right arrow. Or double-click on any item to move it to the other column.

You can play with moving a couple of clusters back and forth, but for this Tour we'll want to generate everything, including EiffelBase, so make sure that in the end all clusters appear in the right column, as on the last figure. Then click **Next**.

### *Generating Metatags from Indexing entries*

The next step of the documentation wizard asks you to select indexing entries:



Eiffel classes, as you know, may start with an **Indexing** entry that enables class authors to include documentary information in any category they like. It is standard (and part of the official style guidelines) to include at the very least an entry of the form **description: Descriptive text** in every class. The earlier displays of class **LIST** showed that entry, which read “**Sequential lists, without commitment to a particular representation**”.

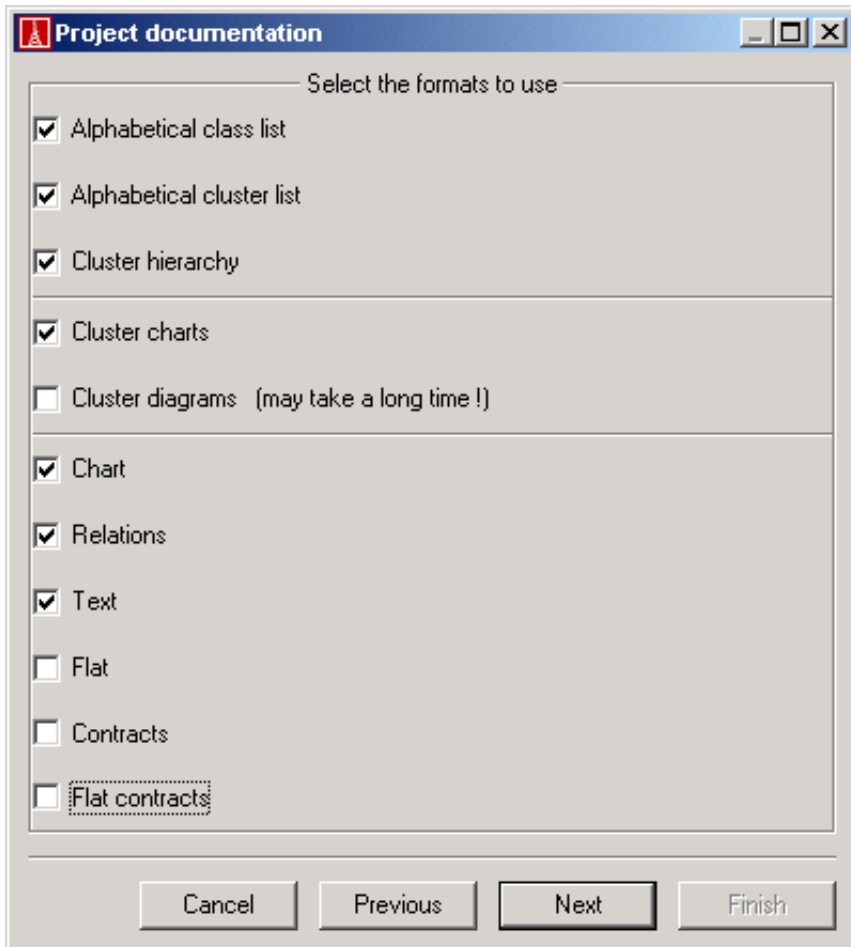
You may have noted that the purpose of Eiffel's **Indexing** clauses is, conceptually, similar to that of **metatags** in HTML. Metatags carry information which Web page visitors do not normally see in the browser; this information is available, however, to search engines and other tools that explore and classify Web pages. So it seems quite appropriate to generate metatags from **Indexing** entries.

The dialog illustrated in the last figure lets you select the entries you wish to transform into metatags. It appears only if you have selected an HTML filter. It lists all the **Indexing** tags found anywhere in the system; those on the right will be retained for metatags. Initially the **Exclude** list on the left contains three tags conventionally used — at ISE and other Eiffel sites — for interfacing with configuration management tools, and hence of internal interest only.

There is no need to change the default selection, so please leave everything unchanged and click **Next**.

### *Choosing a level of detail*

The next step of the Documentation Wizard lets you specify what kinds of documents you want to generate:



This is a very important facility since it gives you control over how much you want to publish about the properties of the software:

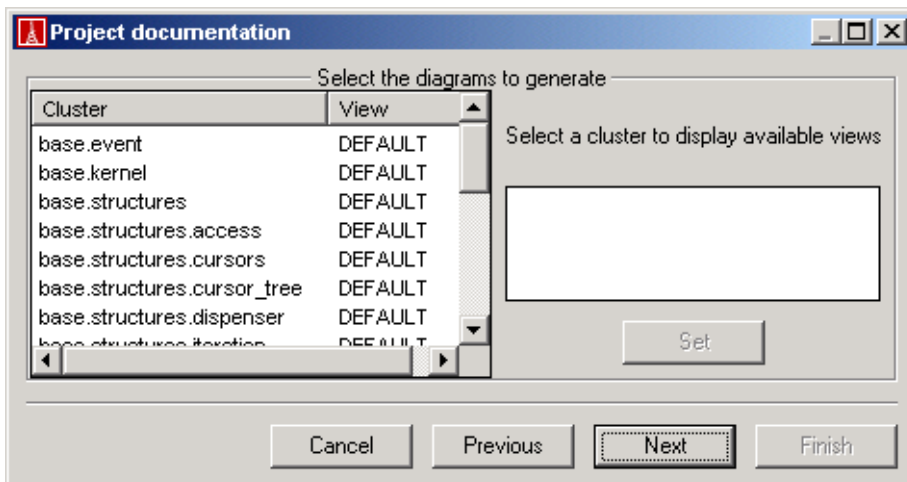
- You may want to publish *everything*, source included, for example on your Intranet for a group of developers working closely together on the same classes, or on the Internet for open-source software.
- You may want to publish only the *interfaces* (contract form, flat-short interface form). This is not necessarily to protect proprietary information; even if you don't care about showing your source code, it is usually too detailed for client programmers, especially in the case of libraries. If various teams work on separate parts of a project, what each releases to the other should usually be the specification, not the implementation.
- You may of course want to publish *both* the text and the interface, and let the recipients use the version that best suits their needs for each use.
- You may want to publish the *diagrams*, showing the structure in graphical form. Note the warning — which we are about to ignore — telling us this may take a while.
- The class list, cluster list, cluster hierarchy view, cluster chart (following the conventions of BON) are also optional.

The dialog shown on the last figure lets you specify the exact combination you wish. The figure shows the default options.

This time, if we generate anything, we'll generate everything. Please check **all** the checkboxes (the generation won't occur until the last step) and click **Next** to move to the next dialog of the Documentation Wizard.

### Specifying cluster views

The next dialog only appears when you have asked to generate diagrams:

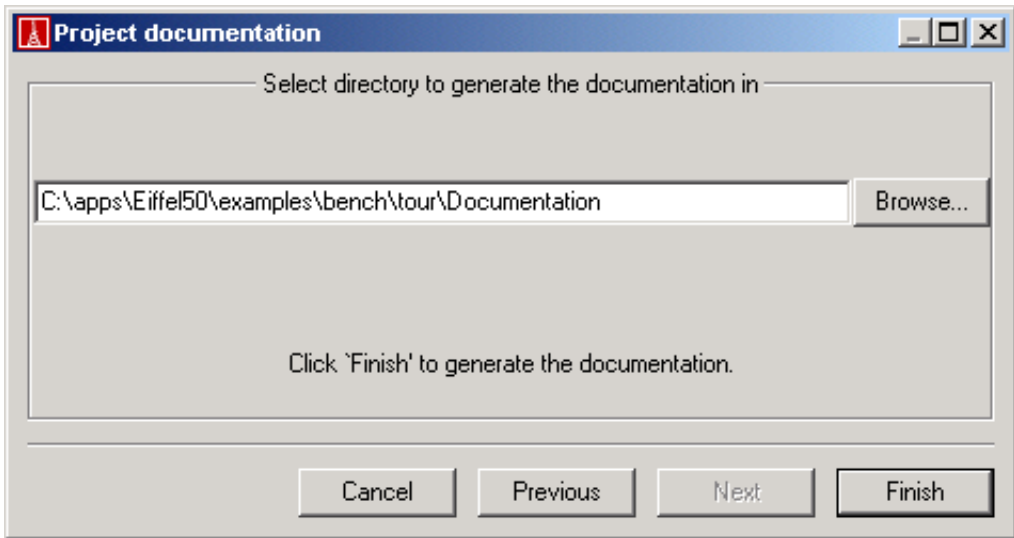


Although we didn't use this possibility yet, the Diagram view lets you define different subviews of any cluster. One view might show inheritance only, the other client links only; one might include all classes, the other hide some library classes. The last dialog shown will allow you, for any cluster, to select a subview other than the default for the generated diagram.

Here we have only used the default view, so just click **Next**.

## Generating

The last dialog simply asks you where you want to generate the result:



By default, as shown, EiffelStudio will produce the documentation in a subdirectory — created for the occasion, if it doesn't exist yet — of the project directory:

*Project\_directory/Documentation*

You may, however, select any other location you like. In the case of HTML generation, as here, EiffelStudio takes great care to use only **relative hyperlinks** so that you can move the **Documentation** directory around, for use either on a file system or on your Web site, with the guarantee that the hyperlinks will work — as long as you move the entire directory together.

To continue the Guided Tour, you do **not** need to complete the generation now unless you want to. The generated HTML is available in a subdirectory **Documentation** of the directory where you are reading this document (if electronically from the ISE Eiffel delivery), and also in the corresponding directory on the Eiffel site, at [http://www.eiffel.com/doc/manuals/getting\\_started/Documentation](http://www.eiffel.com/doc/manuals/getting_started/Documentation).

If you are happy with looking up one of these pre-generated directories rather than producing your own, click **Cancel** on the last dialog.

If you prefer to produce your own, click “Finish”. For our example system the process takes 6 minutes on the Thinkpad configuration mentioned earlier, and generates a 48 megabyte **Documentation** directory.

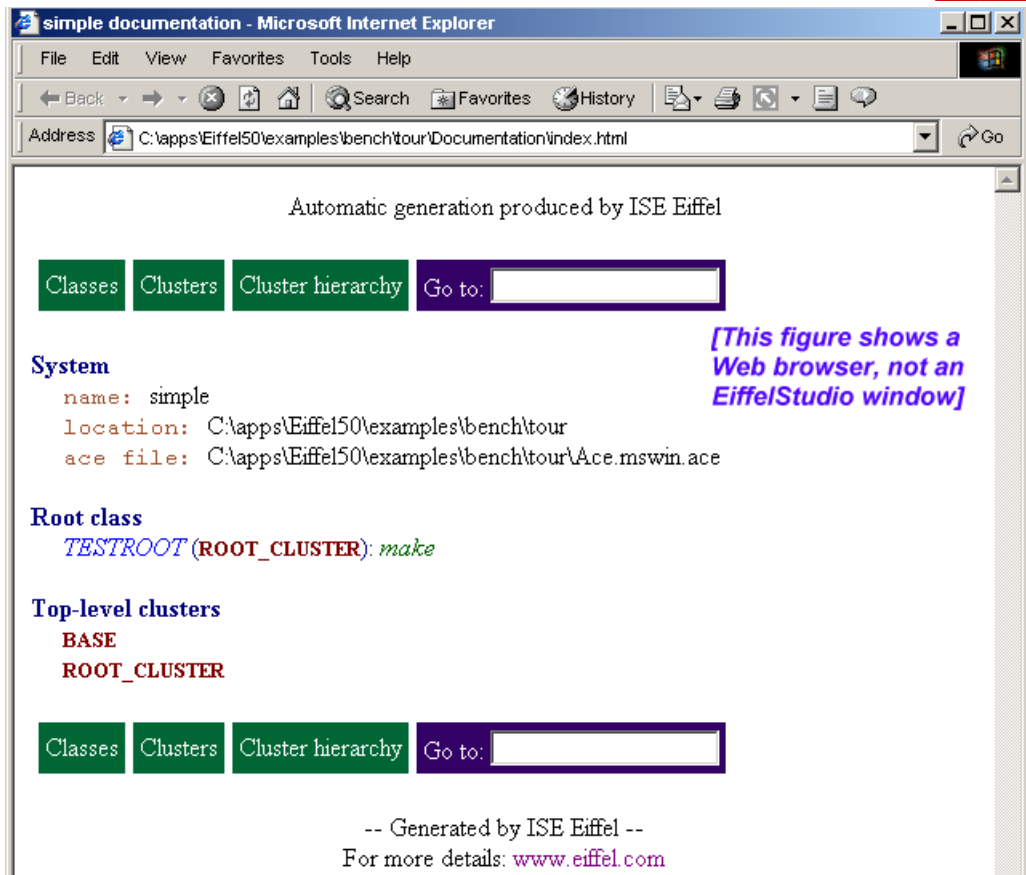
### *Browsing generated documentation*

Let’s take a peek at the generated documentation. If you are reading this electronically in HTML or PDF you can see the generated documentation in the **Documentation** subdirectory of the documentation. Since you can browse the HTML files of that subdirectory at your leisure we’ll just take a quick look to get familiar with the basics. Next to every figure you’ll find a link labeled **Browser link** enabling you to see the corresponding page directly in your browser. The link will open in a new browser page.

Make sure you have a browser with full support for style sheets, such as Netscape 4 or later, Internet Explorer 4 or later.

We start with the root of the generated documentation, **Documentation/index.html**:

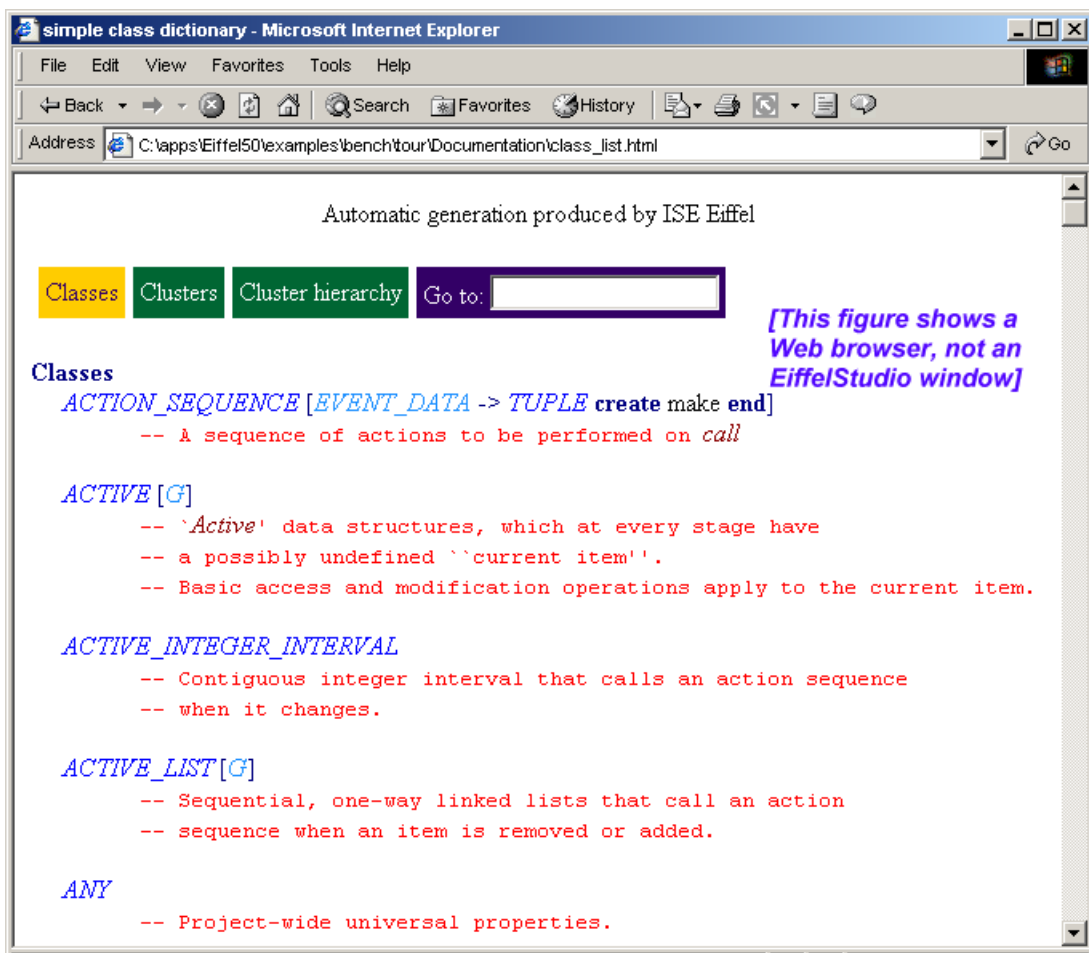
**Browser link**



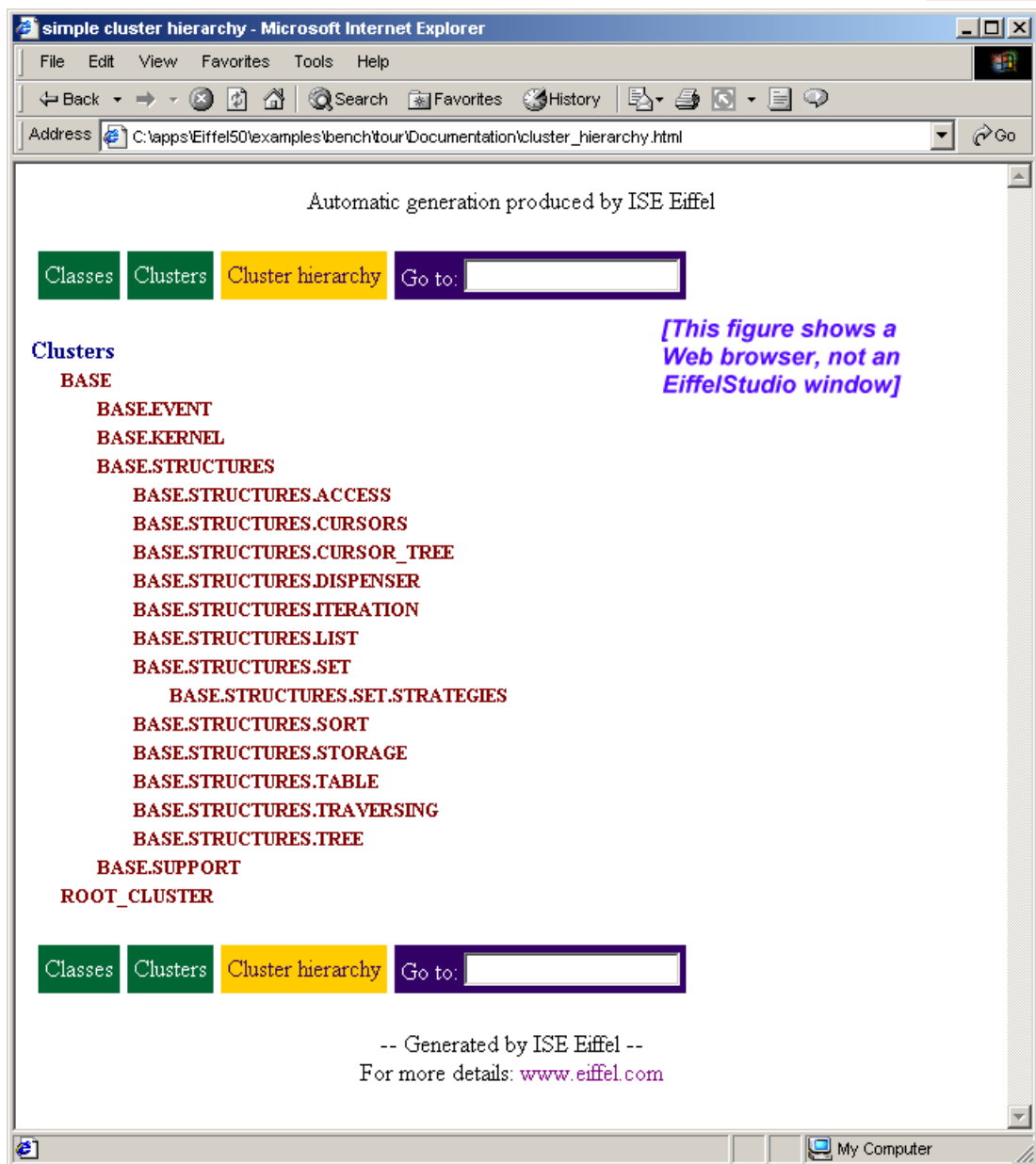


This root page shows overall information about the system. The top set of links, repeated at the bottom, enables you to browse the system from its list of classes, its list of clusters, or the cluster hierarchy; note the box labeled **Go to**, which provides a built-in search engine, enabling you to type any class list and go directly to the corresponding page. Let's look at the class list: click the box **Classes** at the top left.

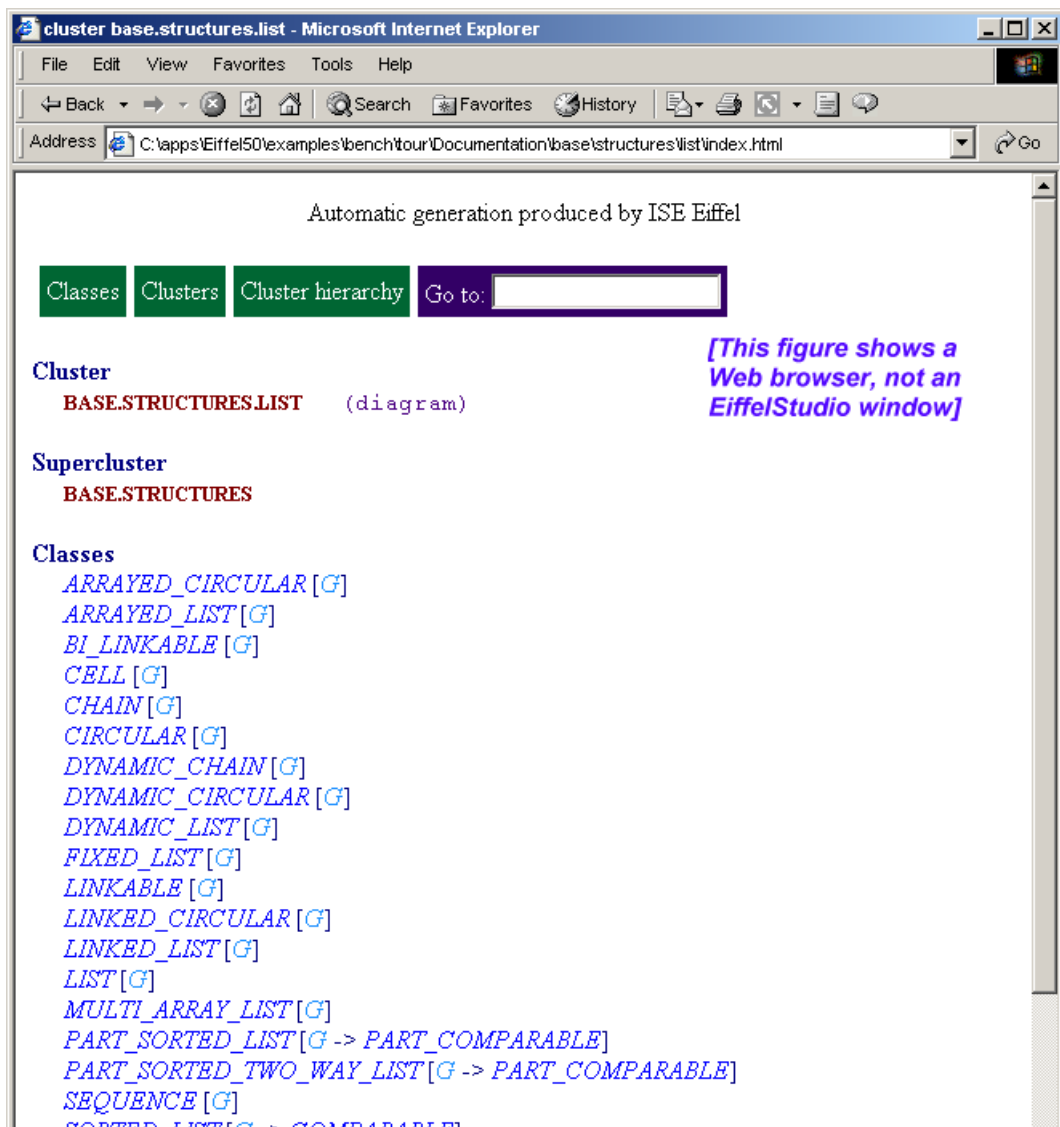
[Browser link](#)



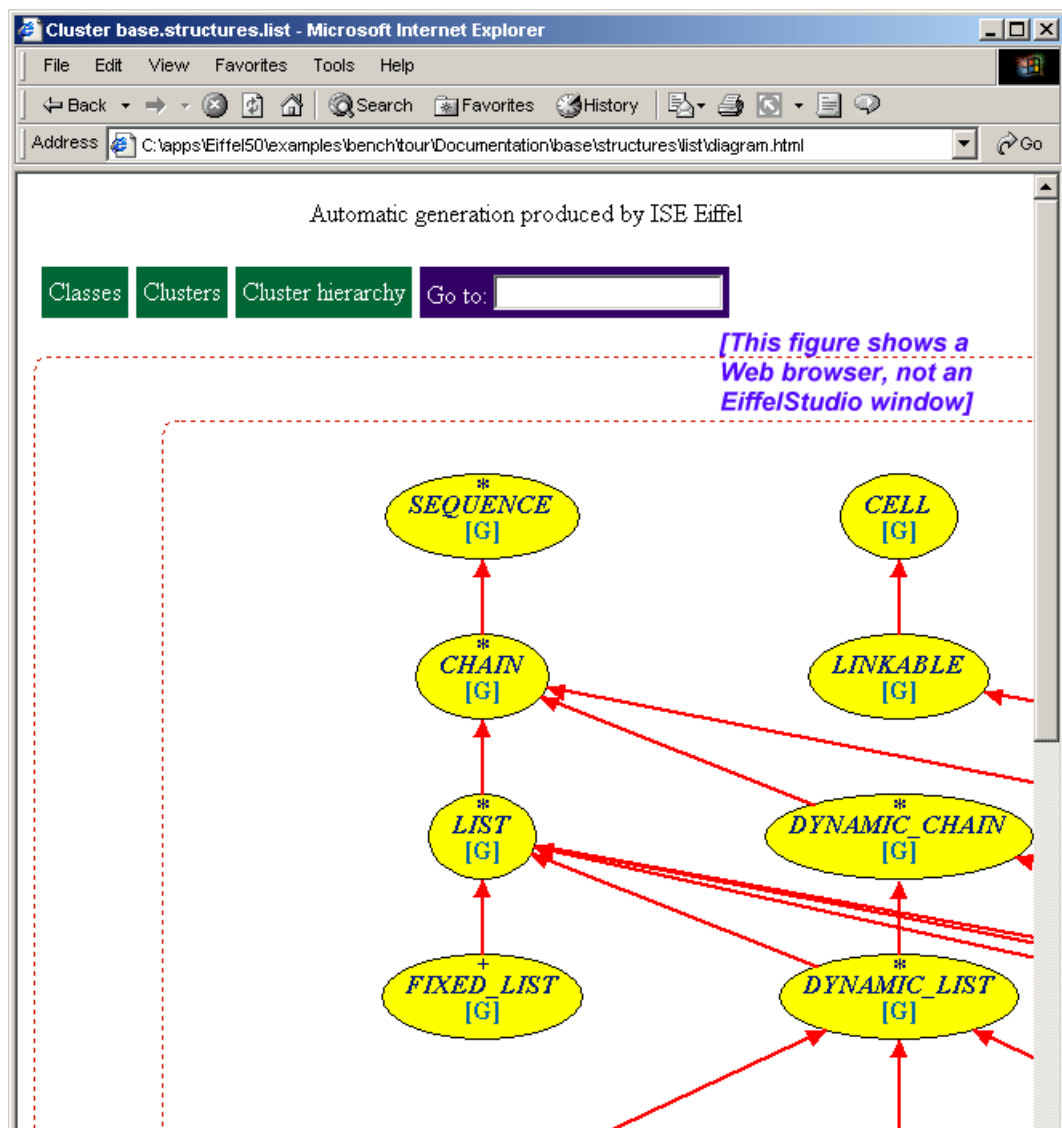
This shows the beginning of the list of classes, alphabetically sorted. You could click on any class to get the corresponding information. We'll look at individual classes in a moment; instead, click **Cluster hierarchy** to see the overall organization of the system into clusters:

[Browser link](#)

Note the convention for denoting nested clusters: **BASE**, **BASE.STRUCTURES**, **BASE.STRUCTURES.LIST**. Click **BASE.STRUCTURES.LIST** to see details of the List cluster of EiffelBase where we earlier (under EiffelStudio) found the class *LIST* used as example in the preceding sections:

[Browser link](#)

This indicates the relations of the cluster to others in the hierarchy, and its list of classes. Again you could click any class name but instead note the mention **(diagram)** next to the cluster name near the top. Remember that when generating the documentation we selected ([“Choosing a level of detail”, page 45](#)) to generate everything, diagrams included. Hadn’t we checked the corresponding check box, the **(diagram)** link wouldn’t be there. Click it now to get the generated diagrams:

[Browser link](#)

The output is a diagram showing graphically the classes of the cluster and their inheritance relations. All EiffelStudio-generated HTML diagrams use the PNG graphics format (*Portable Network Graphics*), supported by all recent browsers.

The class bubbles in a diagram are all hyperlinks. To see the HTML documentation for our old friend the class *LIST* — which you could also obtain by clicking its name on one of the preceding diagrams, or typing it in the **Go to** field — just click its bubble (left on the figure, third from the top):

[Browser link](#)

LIST Chart - Microsoft Internet Explorer

File Edit View Favorites Tools Help

Back Forward Stop Home Search Favorites History Print Copy Paste

Address C:\apps\Eiffel50\examples\bench\tour\Documentation\base\structures\list\list\_chart.html

Automatic generation produced by ISE Eiffel

Classes Clusters Cluster hierarchy **Chart** Relations Text Flat Contracts Flat contracts Go to: LIST

**deferred class**  
[LIST\[G\]](#)

**General**  
 cluster: **BASE.STRUCTURES.LIST**  
 description: "Sequential lists, without commitment to a particular representation"

**Ancestors**  
[CHAIN\[G\]](#)

**Queries**  
 infix "@": (*i*: [INTEGER](#)): [[like item](#)] [G](#)  
 after: [BOOLEAN](#)  
 before: [BOOLEAN](#)  
 changeable\_comparison\_criterion: [BOOLEAN](#)  
 count: [INTEGER](#)  
 cursor: [CURSOR](#)  
 duplicate (*n*: [INTEGER](#)): [[like Current](#)] [LIST\[G\]](#)  
 exhausted: [BOOLEAN](#)  
 extendible: [BOOLEAN](#)  
 first: [[like item](#)] [G](#)  
 for\_all (test: [FUNCTION](#)[[ANY](#), [TUPLE](#)[[G](#)], [BOOLEAN](#)]): [BOOLEAN](#)  
 full: [BOOLEAN](#)  
 has (*v*: [[like item](#)] [G](#)): [BOOLEAN](#)  
 i\_th (*i*: [INTEGER](#)): [[like item](#)] [G](#)  
 index: [INTEGER](#)  
 index\_of (*v*: [[like item](#)] [G](#); *i*: [INTEGER](#)): [INTEGER](#)

[This figure shows a  
 Web browser, not an  
 EiffelStudio window]

The display shows key information on the list, in a form called the “Chart format” listing the ancestors and then the features, divided into **Queries** (shown in part on the figure) and **Commands**. Note that all class names and feature names are hyperlinks, which could lead you to the appropriate place in a class text.

The top row of hyperlinks now includes class formats corresponding to those we discovered under EiffelStudio ([“CLASS VIEWS”, 7, page 30](#)): **Relations** (covering ancestors, descendants, clients, suppliers,), full **Text**, **Contracts**, **Flat contracts**. Click **Flat contracts** to see the full interface of the class:

[Browser link](#)

Automatic generation produced by ISE Eiffel

Classes Clusters Cluster hierarchy Chart Relations Text Flat Contracts Flat contracts Go to: LIST

**indexing**

description: "Sequential lists, without commitment to a particular representation"

status: "See notice at end of class"

names: list, sequence

access: index, cursor, membership

contents: generic

date: "\$Date: 2001/04/30 19:00:43 \$"

revision: "\$Revision: 1.14.4.5 \$"

*[This figure shows a Web browser, not an EiffelStudio window]*

**deferred class interface**

*LIST[G]*

**feature** -- Access

cursor: CURSOR

-- Current cursor position

-- (from CURSOR\_STRUCTURE)

first: like item

-- Item at first position

-- (from CHAIN)

require -- from CHAIN

not\_empty: not is\_empty

We'll stop this brief review here but you may continue browsing through the HTML pages if you like. Note how closely the appearance of the class texts, flat forms, contract forms, diagrams and other forms of documentation matches the corresponding formats under EiffelStudio.

Although we suggest staying with the standard, you can easily change any convention that doesn't match your own preferences:

- For the EiffelStudio appearance, use **Tools → Preferences**.
- For the HTML appearance, assuming that you know about Cascading Style Sheets (CSS) for HTML, edit the style sheet **default.css**. You will find this file in the generated **Documentation** directory; alternatively, to ensure the changes are applicable to the generated documentation of all future projects, edit **defaults.css** in the directory

**\$ISE\_EIFFEL/examples/bench/filters**

after backing it up. For more profound changes in the structure of the generated HTML, you may also backup and edit the Eiffel Filter Format file **html-stylesheet.eff** in the same directory. EFF is described in an appendix ([19, page 144](#)).

The documentation generation mechanisms, using HTML or other formats, let you publish your designs, at the level of detail you desire, on an Intranet, the Internet, or as part of documents you release. They are an important part of the power of ISE Eiffel for quality software development.

## 9 BROWSING FEATURES

Let us get back to EiffelStudio. We won't need a browser any more for this Tour, so you may close any browser window (other than the *current* window if you are reading this in HTML!) opened to look up the generated documentation in the previous section.

Before studying the documentation generation we saw how to display properties of *classes*. It's also interesting to explore the properties of *features*. Let's look at this now, through the Feature View.

Your Development Tool should still be targeted to class **LIST**, from the last view, **Routines**, that you displayed on it (page [38](#)). If you've lost it, just retarget a Development Tool to this class.

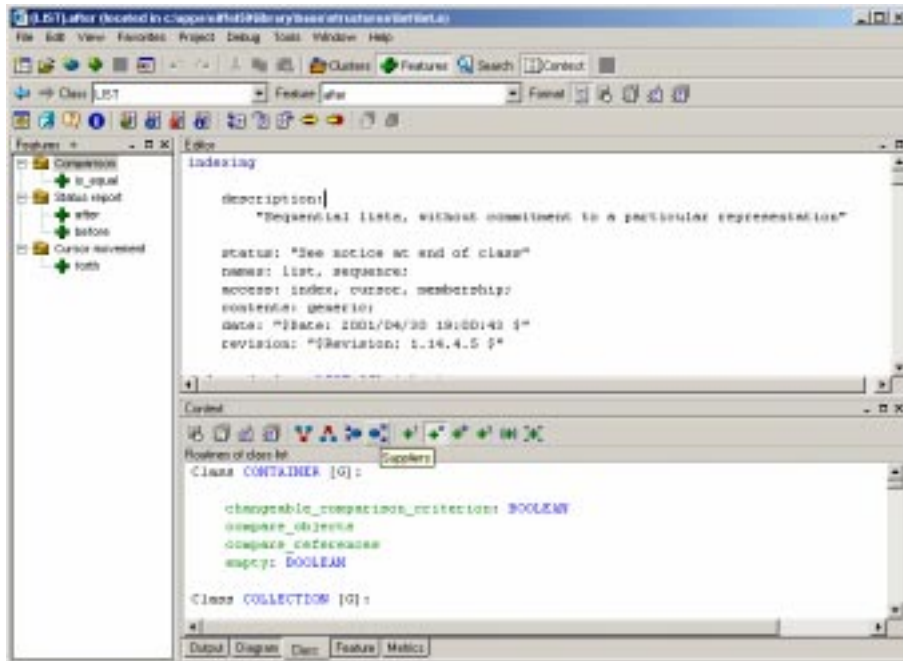
If the Features pane is not visible, bring it back by clicking the **Features** button on the top toolbar, used earlier to remove it:



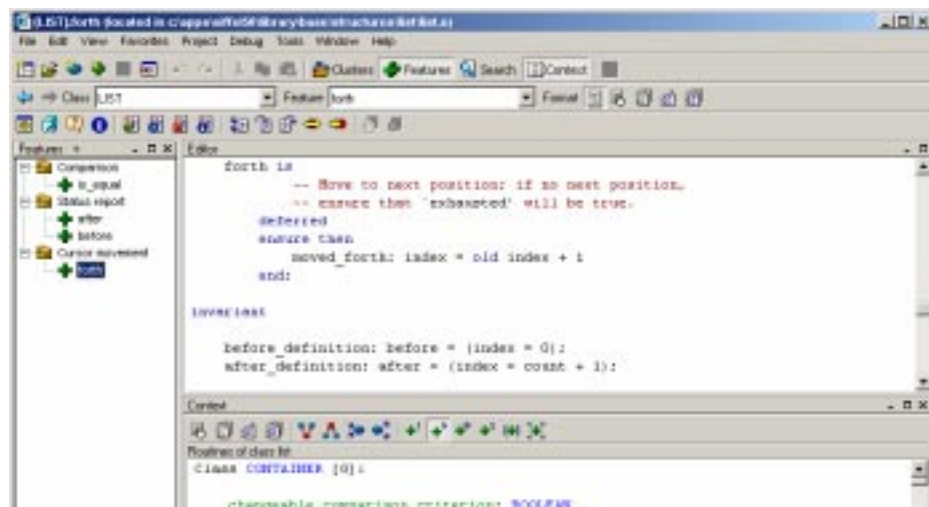
(Another way is through the menu entry **View → EiffelStudio bar**.)

## Targeting to a feature

The list of features, organized by feature clauses, appears on the left:



Note that the class only has a few immediate features because most of its interesting features are inherited. Make sure the Editing pane is tall enough (as on the above figure) and click the feature *forth*, the last one, in the Feature Tree on the left. This makes the feature the tool's current target, and scrolls the text to its declaration:





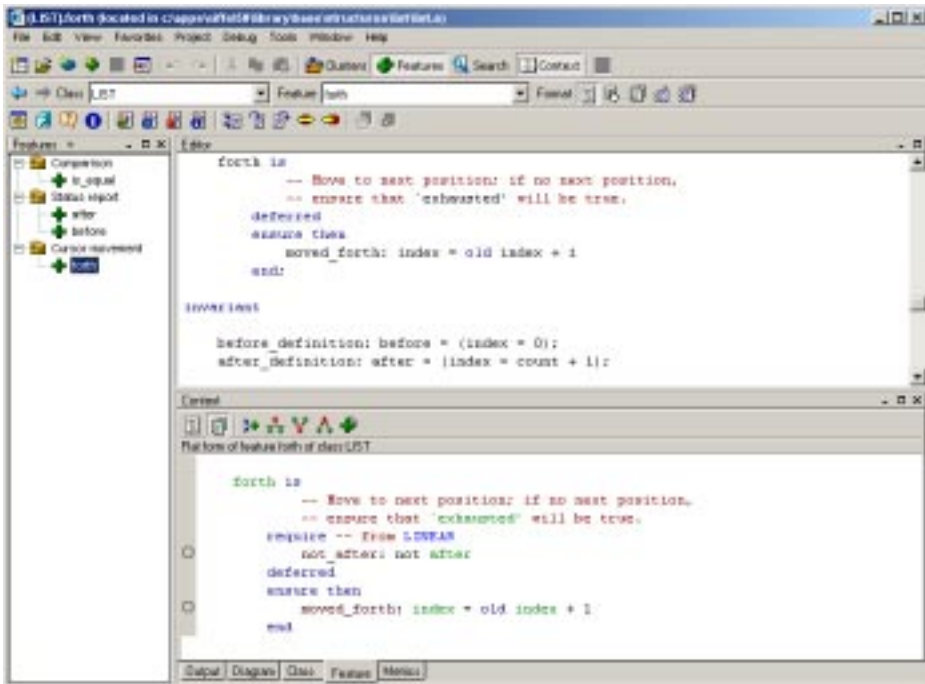
Note how both of the top target fields are now filled: the first one shows the target class, *LIST*, and the second one shows the target feature, *forth*.

### Basic feature information

Now let's look at the feature views. Click the Feature View tab at the bottom of the Context pane:

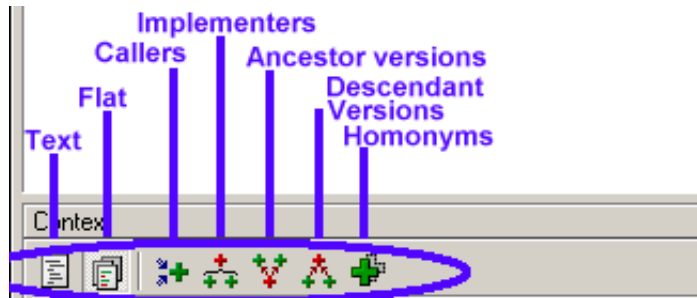


This brings up basic information on the selected feature in the Context pane:



The bottom Context pane shows one of the Feature views, by default **Flat**. The flat view of a feature, similar in concept to the flat view of a class (page 35), gives the full text of a feature, taking into account any inherited precondition or postcondition clauses. Here the feature as declared in the class appears in the top Editing pane, with no precondition and an **ensure then** postcondition clause. But it's a redefinition of an inherited feature; the flat view in the bottom Context pane shows the full precondition, inherited from the ancestor *LINEAR*, as well as the postcondition from *LIST*.

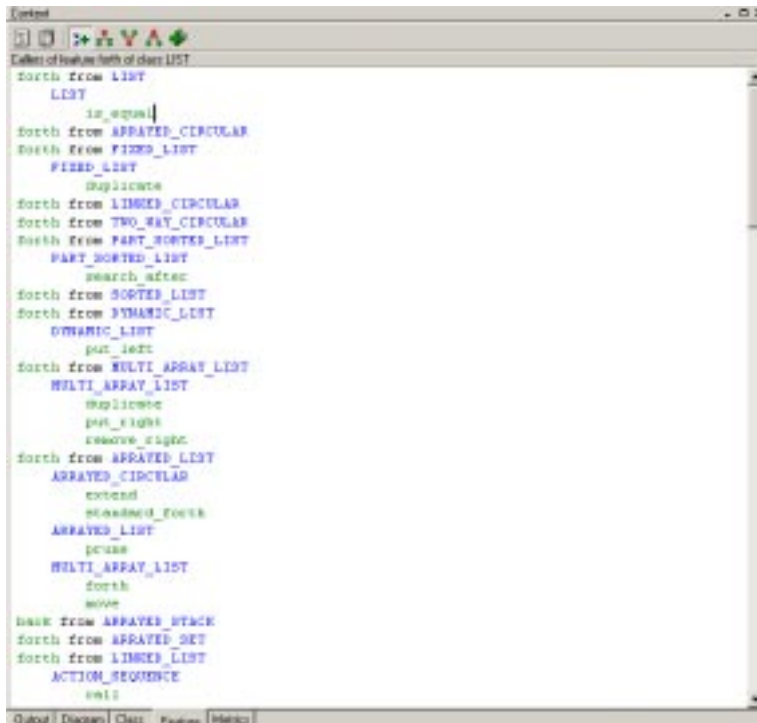
**Flat** is just one of the available Feature Views, shown by the buttons on the Feature View toolbar:



**Text** gives the feature text, fully clickable.

### *Who calls this feature?*

Next to **Flat** is **Callers**. Try it now by clicking the corresponding button (the following figure and the next only show the Context pane, where the views appear):

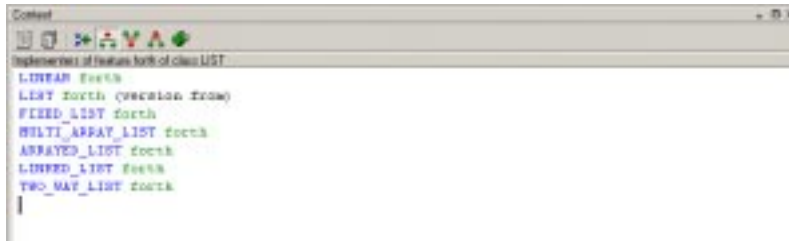


This view shows all the places in the system that call the routine, or one of its redefinitions. Such information can be invaluable for debugging in particular. The successive paragraphs correspond to the various versions of *forth* in class *LIST*, its ancestors and its descendants. Reading from the top we see that:

- The version from *LIST* is called in *LIST* itself by the function *is\_equal*.
- The version from *LIST*'s descendant *ARRAYED\_CIRCULAR* is not called directly in this system, although it **could** be called through dynamic binding (on an entity declared of type *LIST* but dynamically attached to an instance of *ARRAYED\_CIRCULAR*).
- About 60% down, *forth* from *ARRAYED\_LIST*, a version in another descendant of *LIST*, is called by two routines of *ARRAYED\_CIRCULAR*, one routine of *ARRAYED\_LIST*, and two routines of *MULTI\_ARRAY\_LIST*.
- Also note, in the following entry, that after renaming *forth* is called *back* in the descendant *ARRAYED\_STACK*.

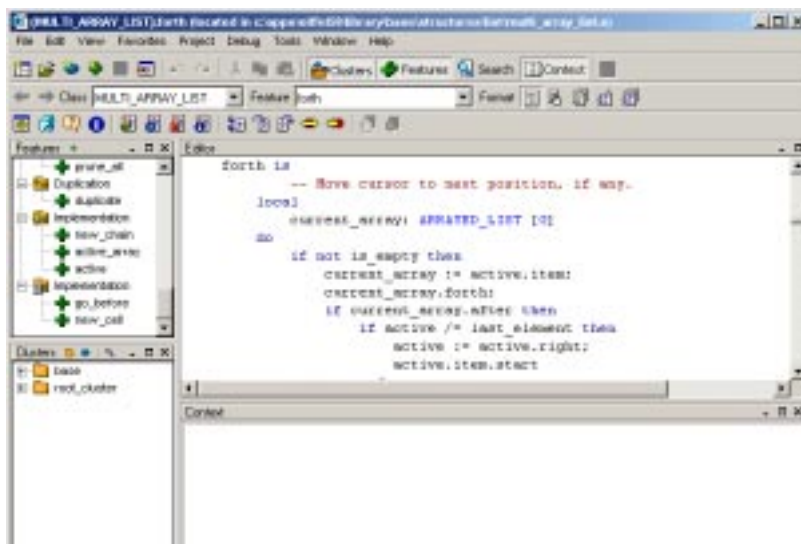
### *What happens to my feature through the inheritance hierarchy?*

Now click the next view button, **Implementers**:

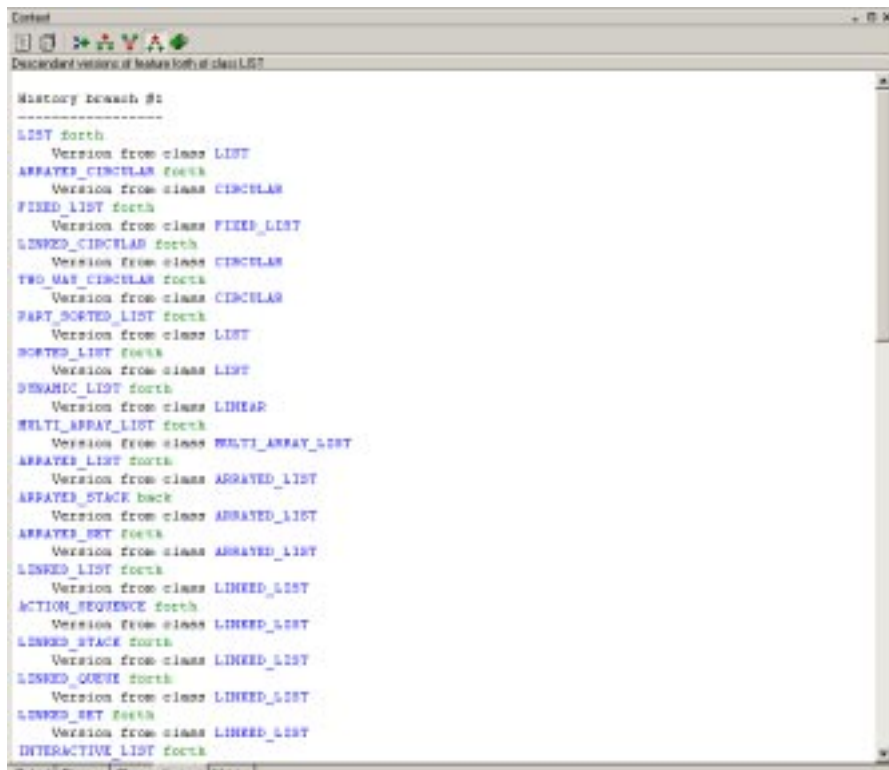


This is a very useful view, showing all the ancestors and descendants of *LIST* that provide a separate version of *forth*, including the original introduction of this feature in *LINEAR* and subsequent redeclarations (redefinitions or effectings). The mention **(version from)** marks the entry corresponding to the version applicable to the current class, here *LIST*.

Since all class and feature names on these views is a hyperlink, you can display any of the listed versions in a new Development Tool by control-right-clicking it (we will see shortly how to display it in the *same* tool). Control-right-click on the feature name *forth* on the line that reads *MULTI\_ARRAY\_LIST forth* (line 4). This brings up a tool targeted to the routine *forth* from *MULTI\_ARRAY\_LIST*, so that you can see the implementation of the routine in that class:



You can get rid of this new Development Tool (just close it, but don't exit EiffelStudio!) and come back to the original Development Tool, where we still have two unexplored views, **Ancestor versions** and **D descendant versions**. Click the first of these to obtain the ancestor versions of *forth* from *LIST*:



The format is self-explanatory: for each ancestor of *LIST* that has a version of *LIST*'s *forth* feature, it indicates the name of that feature — which could be something else than *forth* as a result of renaming, although here this happens only in descendants, not ancestors — and the version of the feature applicable to the given class.

The list is labeled **History branch #1** because in the case of feature merging (combining several features inherited from different parents, along the rules of the language) there could be more than one history branch.

The next button, **Descendant versions**, similarly tells you all that happens to a feature in the descendants of the current class.

### *Who has the same name?*

The last button, **Homonyms**, displays all the features of the system which, related or not to the current feature by redeclaration, have the same name. You can then explore any such feature to see if the relationship is more than casual.

In any system or library making extensive use of inheritance and its associated mechanisms — renaming, redefinition, effecting, undefinition, multiple and repeated inheritance, polymorphism, dynamic binding — the feature browsing facilities are invaluable to track what happens to features. What makes them even more precious is their connection with the rest of the browsing and documentation capabilities, especially the pick-and-drop studied in the next section.

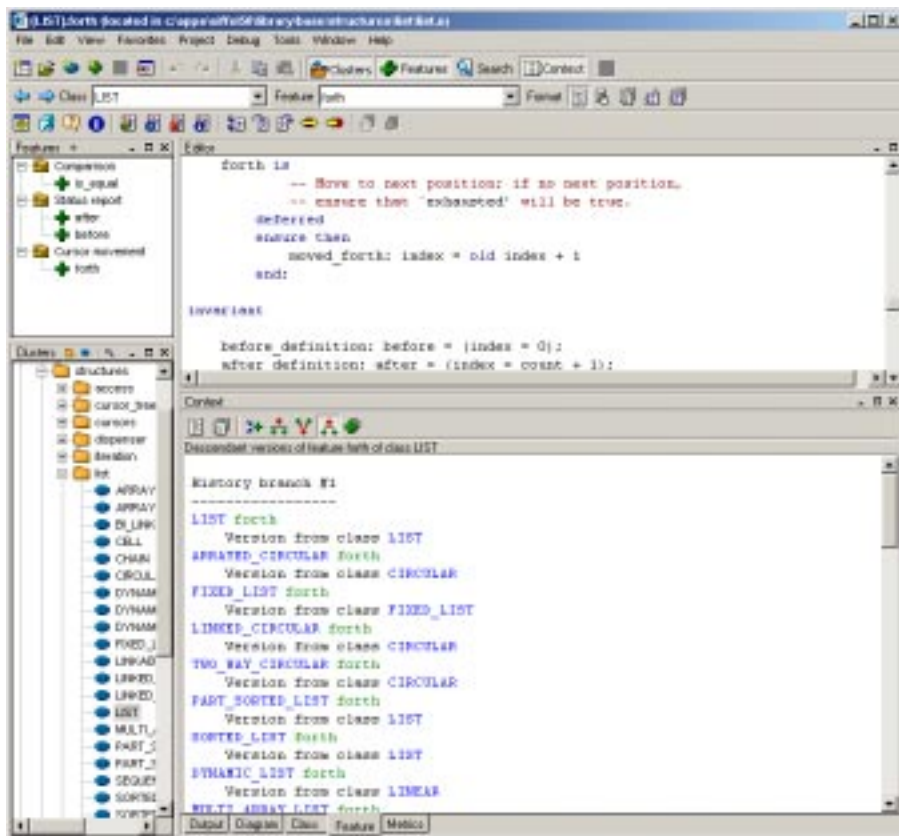
## 10 RETARGETING THROUGH PICK-AND-DROP

We have seen various ways of retargeting a Development Tool to a “development object” — a class or a feature — but have put aside one of the most important, which lets you pick a development object that you see anywhere in the display, and retarget the current tool, or another, to it.

### *Trying Pick-and-Drop*

We restart from the last state, with a Development Tool target to feature *forth* of class *LIST*. The next figure shows the whole window; it should be exactly what you see as a result of the last operations. We'll use the **Ancestor versions** view of the **Feature** tab.

If for some reason the window doesn't look like the next figure, it's easy to reconstruct it: make sure both the Cluster tree and the Feature tree are visible (if not, click the corresponding buttons as recalled on page 55); target the tool to class *LIST*; target it to its feature *forth* by clicking its name in the Feature tree; make sure both the top-right Editing pane and the bottom-right Context pane are visible; in the Context pane, select the **Feature** tab and its **Ancestor versions** format.



In the Context pane at the bottom right, the second entry reads

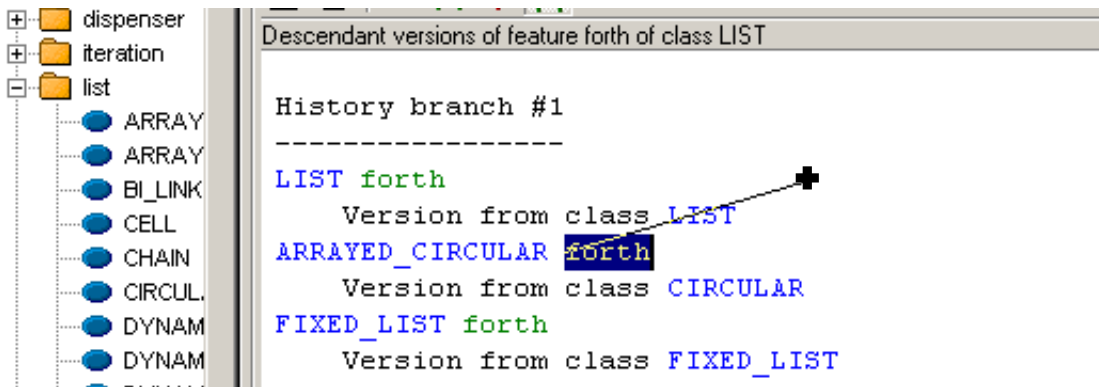
`ARRAYED_CIRCULAR forth`

referring to the version of feature `forth` in class `ARRAYED_CIRCULAR`. Now let's assume you want to see what that version actually is. It suffices to retarget the tool to it. Of course you could type the class name `ARRAYED_CIRCULAR` in the Class field at the top of the window, and the feature name `forth` in the adjacent Feature field. But this is too much work; after all, you have just seen a reference to the feature, through its name as it appears in the Ancestors version format, so it's natural to use it directly from the graphical interface.

As we've seen before, you could control-right-click on the feature name at the place where it appears; this would create a new Development Tool targeted to `forth` from `ARRAYED_CIRCULAR`. But you don't necessarily want a new window. Instead you can use Pick-and-Drop to retarget the current window.

Here is how it works. Position the cursor on the desired feature reference (the word *forth* in the line `ARRAYED_CIRCULAR forth`). Right-click, that is to say click the rightmost mouse button, and **release the button immediately**. That’s right: you use a simple click, and do **not** maintain the button down.

Now move the mouse a trifle, *without pressing any button*:



The cursor has changed into a new shape, a cross representing the type of development object that you have picked, a feature. For a class, as you may have guessed, it would be a small ellipse (“bubble”). Each kind of development object that you may create and manipulate during your work with EiffelStudio has its distinctive icon.

The display reflects that you have “*picked*” the feature *forth*. Now you can *drop* it at any appropriate place to retarget the corresponding tool. In fact you can drop it right where it is, in the Context pane of the current Development tool. To drop, just **right-click** again. (That is to say, as before, press the rightmost mouse button and release it immediately.) This achieves a “drop”, and retargets the Development Tool to the chosen feature, *forth* from `ARRAYED_CIRCULAR`. The retargeting affects both the Editing pane and the Context pane, which keeps its current view (**Ancestor versions** in the **Feature** tab). We’ll see shortly how to give them separate targets if that’s preferred.







### *Pebbles, holes, drop targets and type compatibility*

The Pick-and-Drop mechanism relies on the metaphor of **pebbles and holes**. When you pick a development object, the cursor changes into a “pebble” whose shape represents the type of the development object: class, feature, run-time object... You may then drop it into a “hole”, which can be a window, a tree view entry, or a hole-shaped icon. This performs the appropriate action such as retargeting a tool.

In the same way that Eiffel is a typed object-oriented language, the Pick-and-Drop mechanism is typed: you can only drop a pebble into a compatible hole. For example you may drop a class pebble into a Development Tool, to retarget it to the chosen class.

In Eiffel, type compatibility is not necessarily type identity, but is governed by *conformance*, based on inheritance and polymorphism: to an entity of type *POLYGON*, you may assign not only an expression of that same type, but also one of type *RECTANGLE*, if class *RECTANGLE* inherits from — conforms to — class *POLYGON*. Similarly, EiffelStudio considers that the development type “feature” conforms to “class”; this means you may drop a feature into a Development Tool targeted to a class; this will retarget the tool to the feature’s class and the feature itself, with the text of the class scrolled to the position of the feature.

In the Pick-and-Drop example — for *forth* of *ARRAYED\_CIRCULAR* — you did not have to go to a new target: the current window was a valid drop target, so you just dropped right away. In such a case you don’t even have to move the mouse; Pick-and-Drop is just a matter of two right-clicks.

This is similar to a **double-click**, a commonly used interaction technique, but without the stress of the usual double-click, which requires you to wait no more than a specified time — typically half a second or so — between the two clicks. With Pick-and-Drop the effect is the same whether the second click follows the first after one tenth of a second or two days.

## *Multiple tools*

In the previous example we pick-and-dropped a feature to its own tool. You can also pick-and-drop to a different tool.

Try this now. Bring up a new Development Tool by choosing the menu entry **File → New window** (unless you already have a second Development Tool open, in which case you can simply reuse it). Make sure the two Development Tool do not overlap too much, so that you can see enough of each. In the first Development Tool, pick a class (right-click on it). Move the mouse to the Editing pane of the second Development Tool. Drop the class by right-clicking again. The tool retargets itself to the chosen class.

Many people like to take advantage of this possibility to keep two or more Development Tools open, and pick-and-drop frequently from one to the other when they see a development object of interest and want to know more about it, without losing the original context.

## *Clickable formats*

A good deal of the power of Pick-and-Drop comes from its connection with the various views of the Context pane — Class Views, Feature Views, Diagram View. As was mentioned when we saw these views, all the feature and class names or other graphical representations that appear in these views are **clickable**; this means that you can select any of them as the source of a Pick-and-Drop.

This means that you can quickly traverse a system and get to its essential properties by displaying the information of a class in any of the many available views — the contract form of a class, its routines, its attributes, its clients, its ancestors, the ancestor and descendant versions of a feature, and so on — and, wherever you see a feature or class name, follow the corresponding link. This proximity-based form of browsing, combined with the other techniques seen earlier, provides considerable help when you are dealing with a large, possibly complex system, and want to master its intricacies, be it for development, testing, debugging, maintenance or revision.

Other places where you can pick development objects include the class bubbles in a Diagram View, or the icons representing classes and features in the Cluster Tree, Feature Tree and Favorites list.

## *Semantic consistency*

An important property of the pick-and-drop mechanism, shared by its cousin the right-click mechanism, has already been mentioned in this chapter: semantic consistency, which guarantees that the operations you can perform on a class, such as pick-and-drop, only depend on the *development object* to which you are applying the operation. It doesn't matter where you picked the object — in any development tool under any view — and in what form: textual, as a class or feature name; graphical representation, as a class bubble in a Diagram View; or an icon, for example in the Cluster Tree, Feature Tree, Favorites list.

The pebble that you see during the Move step of Pick-and-Drop represents the underlying development object — such as a class or a feature — regardless of how you got to it.

## *Behind the Pick-and-Drop conventions*

Pick-and-Drop works differently from the usual Drag-and-Drop present on many computing platforms. The usual Drag-and-Drop retains a small role within EiffelStudio (to move class bubbles around in the Diagram view) and you may of course have to use it when relying on for operating system functions such as copying files. But the key EiffelStudio operation is Pick-and-Drop. This technique is motivated by careful consideration of ergonomics — human engineering and user comfort. In particular:

- Pick-and-Drop is much less stressful. Drag-and-Drop requires you to maintain pressure throughout the move, being careful not to drop on the wrong place. With Pick-and-Drop there is no stress: you click and release; get a drop of coffee if you like (optional step); move the cursor with no pressure from your fingers or on your mind; make sure, at your leisure, to find the right drop place; and right-click again on it. At the end of the day, after many such operations, the stress reduction can make a real difference.
- With Drag-and-Drop, it's easy to lessen the pressure involuntarily and drop on the wrong place. The consequences can be damaging, especially since in such a case you may well *not know* where you dropped the element; after all, that wasn't intentional. It's possible, for example, to lose files that way. With Pick-and-Drop this is much less likely to happen.
- Pick-and-Drop makes it easy to cancel the operation if you change your mind: just left-click anywhere. With Drag-and-Drop you have to find an invalid place to drop; this may be difficult, or even impossible! (Sometimes pressing the Escape key works, but this is not universal.)

If you are new to ISE Eiffel you may find Pick-and-Drop surprising at first. We trust you will join the ranks of ISE Eiffel users who consistently rate it among the most convenient features of the environment.

## *Isolating the context*

In all the examples so far, the Context pane was targeted to the same target as the enclosing Development Tool and its Editing pane. You can retarget the Context pane separately, for example by pick-and-dropping a class or feature into it; but the next time you retarget the Development Tool as a whole the Context pane will follow.

This is called *merged* behavior, meaning that the target of the Context pane is merged with the target of the enclosing Development Tool and Editing pane. It's the default behavior, appropriate for your first steps with EiffelStudio.

As you become more proficient with the environment, you might not want this behavior any more: you might prefer the ability to retarget the tool and its Editing pane while keeping the Context targeted to its previous target. This enables you to see, in the same tool, information on two different classes, and is called *isolating* the Context.

The choice between merged and isolated behavior is a matter of taste; let's see how you can change it, so that you can set your own preference.

To change the behavior, choose the menu entry

**View → Isolate context tool**

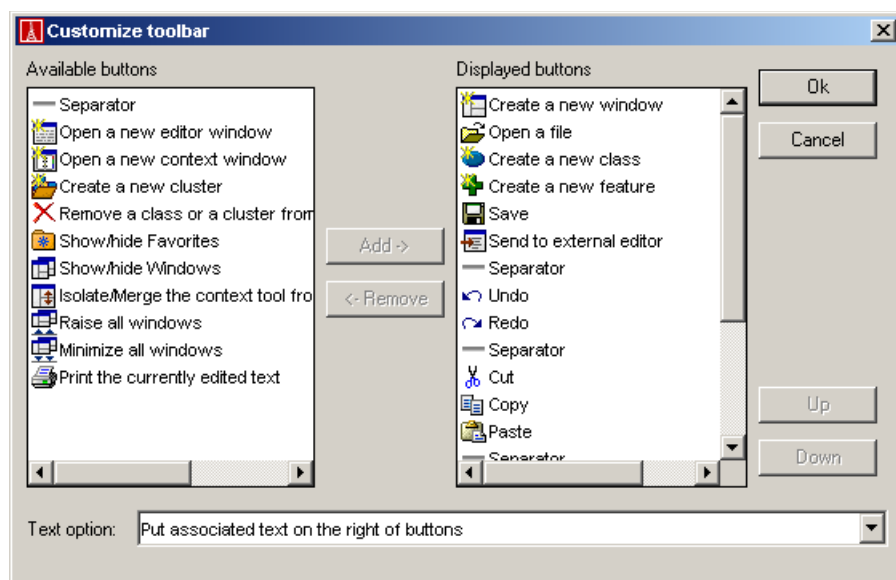
After this, the same entry will change to **View → Merge context tool**, so that you can later revert to merged behavior. Under isolation behavior, try pick-and-dropping a class or feature into the top Editing pane; then pick-and-drop a class or feature into the bottom Context pane. You will notice that either of these operations retargets the affected pane, but not the other.

## *Customizing the view*

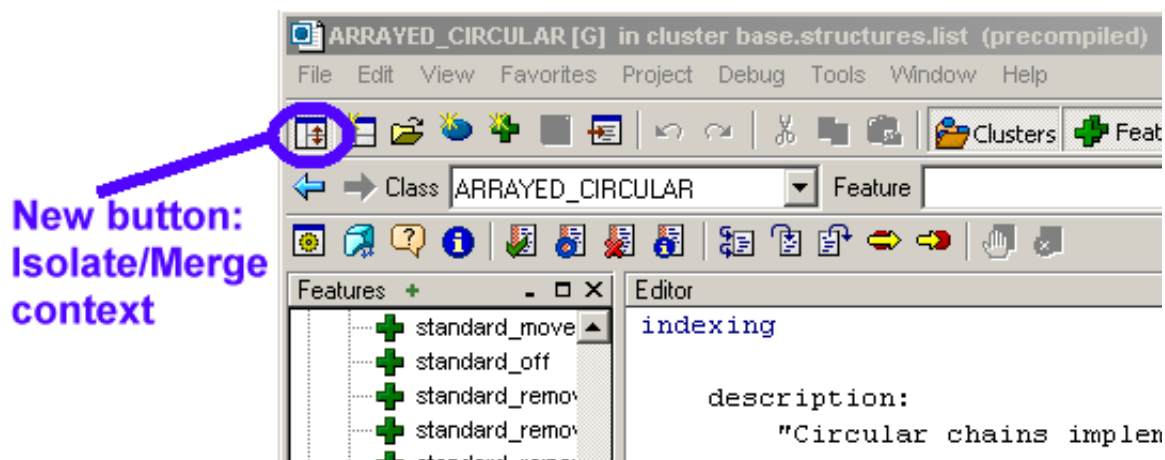
If you will often alternate between the merged and isolated behaviors, you can use the **Merge/Isolate** button of the top toolbar. This button is *not* present by default on the toolbar, so this is a good opportunity to take a quick look at the user interface customization mechanism, which you can use later to tailor the interface to the exact form you need. Select

**View → Toolbars → Customize standard toolbar**

(The adjacent entry is **Customize project toolbar** which provides complementary capabilities under a similar form.) You see a list of available buttons:



The icons in the list on the right are currently displayed in the toolbar, but not those on the left. Among the latter you see (fourth from the bottom on the left-side list) **Isolate/merge the context tool**. Select it by clicking; this makes the **Add ->** button active. Click this button to move the Isolate/Merge icon to the list of displayed icons. It becomes the first item of the list, which is fine for the moment. (Later on you can change the order of buttons in the toolbar if you like, by using the **Up** and **Down** buttons.) Click **OK**. The toolbar of your development tool has a new button:



You can now use this button to switch, for the enclosing Development Tool, between the isolated and merged behaviors of the Context pane.

## Context memory

If you start repeatedly retargeting the Context pane — especially under “isolated” behavior — you will notice the following properties:

- In most cases, pick-and-dropping a *class* to the Context switches the view to the **Class** tab, and pick-and-dropping a *feature* switches to the **Feature** tab.
- The view displayed in each case — for example **Ancestors** for the **Class** tab and **Flat** for the **Feature** tab — is default view for the corresponding tab.

## The many paths to retargeting

As a conclusion to this review of Pick-and-Drop let’s recapitulate the various ways we’ve seen for retargeting a whole Development Tool or a pane to a class:

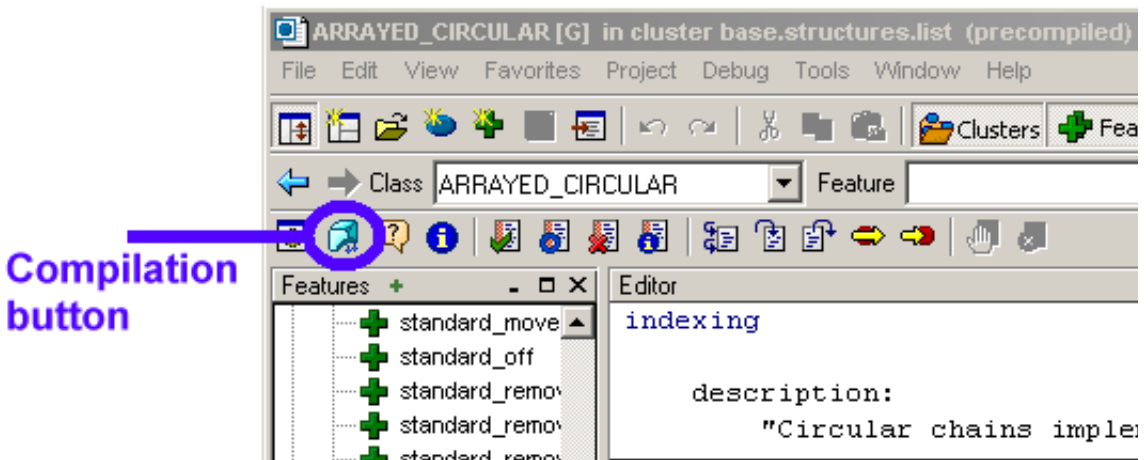
How to retarget	Same window/ pane, or new?	Where described
Type class name, then Enter, in class field at top-left of tool	Same	<a href="#">“Retargeting by name”, page 19</a>
Choose class in Cluster tree	Same	<a href="#">“Retargeting from the Cluster Tree”, page 23</a>
Choose class in Favorites	Same	<a href="#">“Adding to Favorites”, page 27</a>
“Back” button	Same	<a href="#">“Moving back and forth”, page 26</a>
“Forth” button	Same	<a href="#">“Moving back and forth”, page 26</a>
Pick class from history list	Same	<a href="#">“The Target History”, page 27</a>
Pick-and-drop: right-click on class name or graphical representation found in any tool, then move, then right-click.	Existing window/pane (of drop target)	<a href="#">“How Pick-and-Drop works”, page 64</a>
Control-right-click on class name or graphical representation found in any tool	New	<a href="#">“Starting a new tool”, page 28</a>

## 11 RECOMPILING AND EDITING

So far we have relied on existing class texts. Fascinating as it may be to explore excellent software such as EiffelBase, you probably want to write your own too (with the help of the reusable components in the Eiffel libraries). EiffelStudio provides a built-in editor — as well as the ability to use some other editor if you prefer — and sophisticated compilation mechanisms.

### *Recompiling*

When we started, we compiled the example system. Let's recompile it, just to see. We'll see compilation entries in the **Project** menu, but the easiest for the moment is to use the compilation button in the Project toolbar, the lower toolbar in the Development Tool:

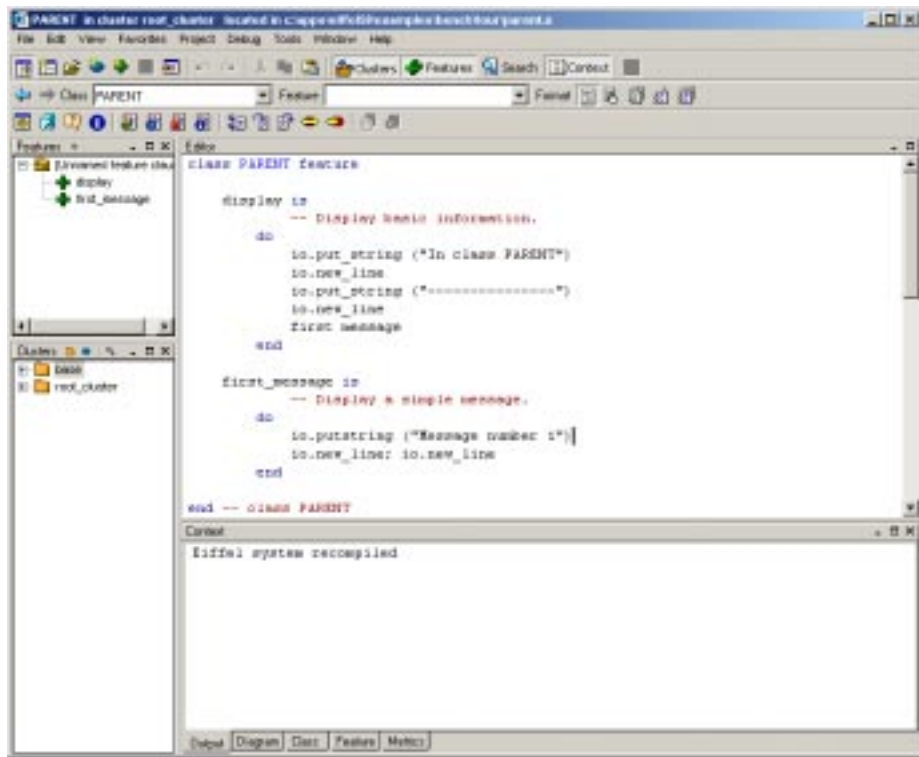


Click this button. You haven't changed anything in the project since it was compiled (you were not supposed to!), so EiffelStudio will very quickly detect this and finish compilation. On our test platform this takes less than a second. Now of course we should see what happens if you do change something.

### *Editing*

We don't want to touch EiffelBase classes (and in fact can't, since it is used in precompiled form), so let's focus on classes of our small root cluster. In the Cluster tree on the left, expand cluster **root\_cluster** and click class **PARENT** to retarget the Development Tool to it.

Make sure that the Editing pane is big enough to display the text of the class:



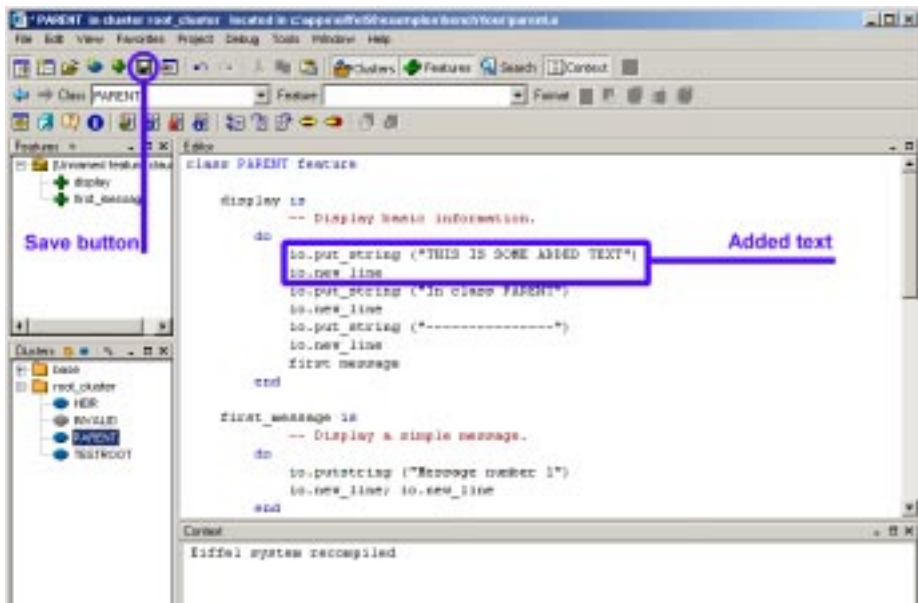
The Editing pane hosts a text editor which you can use to change the class text. Here the routine *display* starts by outputting a simple message; let's precede it by another line of display to check that we affected the outcome. We'll want to add the following two lines just after the **do**, before the first two instructions of the routine:

```

io.put_string ("THIS IS SOME ADDED TEXT")
io.new_line
  
```

They are very similar to the current first two lines of the routine, so you can just use copy-paste: select the first two lines with the mouse, copy them using **CTRL-C** (or **Copy** from the **Edit** menu), then paste them just after the **do** using **CTRL-V** (or **Paste** from the **Edit** menu). Add or remove tabs to align with the rest of the routine, and change the string to *THIS IS SOME ADDED TEXT* so that the result will look like what's shown on the next figure. This is all there is to change; the second line remains untouched. Please check the result and be careful not to introduce any mistakes; in the next section we'll study how EiffelStudio will report syntax and other errors, but right now we want to see what happens when everything is right!





Now save your changes; you may indifferently use CTRL-**S**, the **Save** entry from the **Edit** menu, or the Save button highlighted on the figure. (If you forget to save, the next compilation will tell you so, and ask you if you want all non-saved class edits to be saved automatically.)

### *Recompiling and executing after a change*

Next compile again, using the Compilation button (shown on the figure page 71). Some “degree” messages appear quickly; EiffelStudio has found out what class has changed and deduced what exactly to recompile — only a subset of the whole system. So this again will proceed very quickly.

Execute the system again now, using one of the execution buttons, with or without breakpoints, on the right in the bottom Project toolbar. You will see that the message output by the execution has changed to include the added string.

### *Views in the Editing pane*

In studying the Context pane we discovered a number of views of a class text (“**CLASS VIEWS**”, 7, page 30). For convenience, you can also display a number of these views in the Editing pane, although only the basic Text view is editable. A row of buttons next to the Class and Feature fields lets you choose between them:



To **comment out** a sequence of lines, select them and use **Edit → Advanced → Comment** or CTRL-**K**. Conversely, CTRL-Shift-**K** will uncomment. Also in the **Edit → Advanced** menu are “set to upper case”, with the keyboard shortcut CTRL-**U**, and to lower case, CTRL-Shift-**U**.

Other useful facilities of the **Edit → Advanced** menu are:

- **Embed in “if”**, or CTRL-**I**, which will create a conditional instruction and include the selected instructions in it.
- **Embed in “debug”**, CTRL-**D**, which will include the selected instructions in a **debug ... end** instruction, so that their execution becomes conditional on a Debug compilation option.

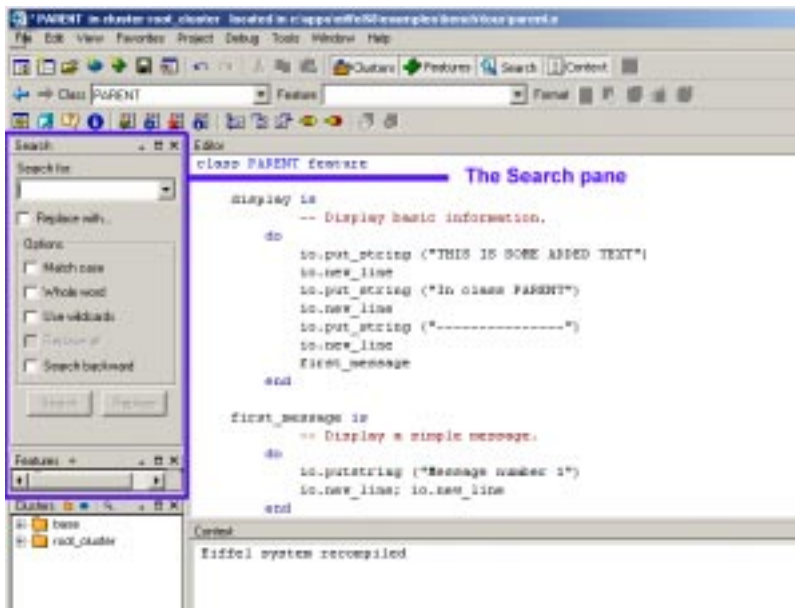
## Search and replace

The editor lets you search for text and replace occurrences, individually or globally. We assume you have seen a text search facility before, so we’ll just emphasize some of the least obvious features.

To start a search, make sure the Search pane is active by clicking the Search button in the top toolbar (this one we’ll let you find), using the **Edit → Find** menu entry, or type CTRL-**F** in a Text or Context pane.

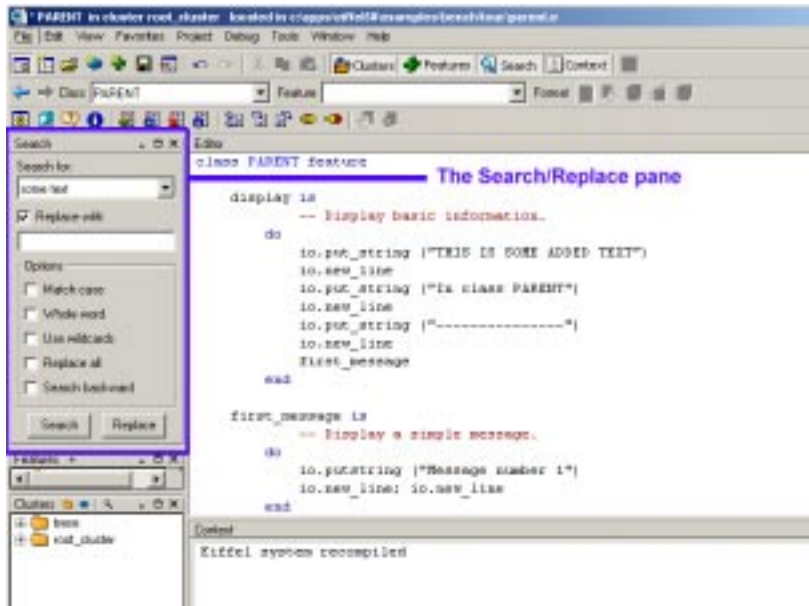
Note that although we are studying Search as part of the Editor, this function also applies to any textual form displayed in the Context pane; make sure to start a Search from the pane that you want to search.

The Search pane presents a number of self-explanatory options:



If you select **Use wildcards**, two characters will be treated specially if they appear in the **Search for** field: a question mark **?** will match any character, and an asterisk **\*** will match any sequence of characters.

If you check the **Replace with** box, a replacement field will appear:



Having filled the two fields, you can elect to replace the last found occurrence, or all occurrences at once.

The **Search for** field has an associated menu, so that you can reuse a recently entered search string without retyping it.

### *Let the editor do the typing*

Particularly interesting are the editor's **automatic completion** facilities. Well, particularly interesting for *most* people: maybe you like your editor to do the gruntwork for you, or maybe you don't. In the latter case — if you prefer to be in control of all the details — don't worry: through **Tools** → **Preferences** → **Editor** you can easily disable any facility that you don't like. The behavior described here is the default.

The EiffelStudio Editor knows about Eiffel syntax and will recognize syntactic elements as you type them. It will color them according to standard conventions: basic elements in black, keywords in blue, comments in dark red. You can change these conventions through Preferences.

If you start typing a control structure through its opening keyword, such as **if**, or **from** for a loop, the editor will automatically display the structure of the whole construct. Here for example is the result if you type the **from** followed by Return/Enter at the beginning of our example routine:

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

    loop

end
    io.put_string ("THIS IS SOME ADDED TEXT")
    io.new_line
```

This has produced the structure of an Eiffel loop: **from ... until ... loop ... end**. You can then fill in the blanks with the appropriate expression and instructions. The generated lines start with the appropriate number of Tab characters to support the standard Eiffel indenting conventions. If you want a more compact style, follow the **from** with a space rather than Return. Typing **if** followed by Return or a space will similarly produce the outline of a conditional instruction.

To start a routine, type the routine name followed by the keyword **is** and a Return. This generates the basic structure of a routine text:

```
my_routine is
    -- |
```

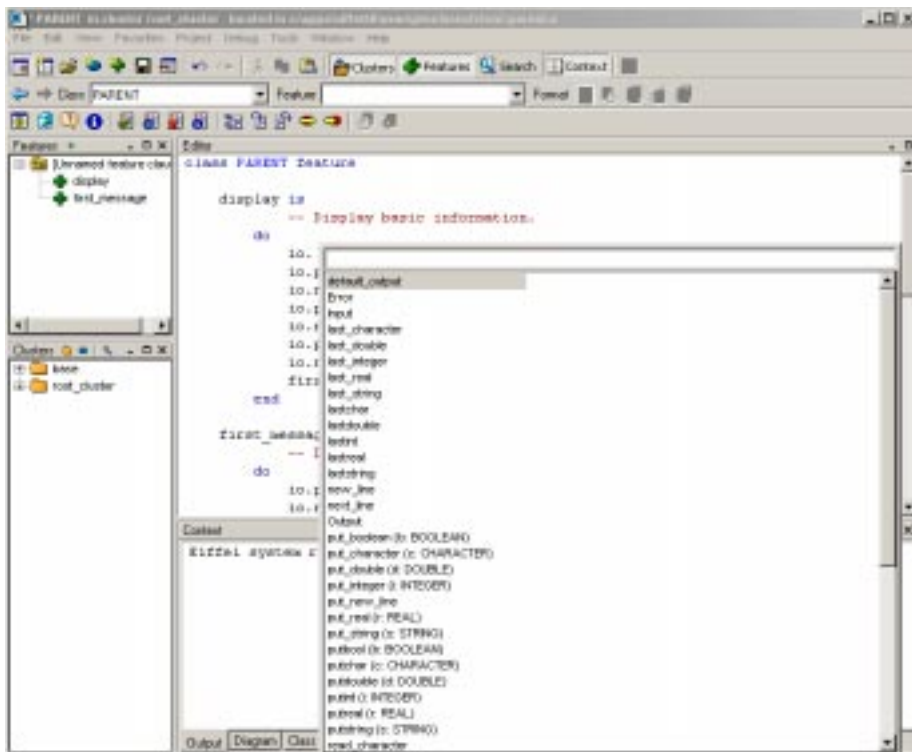
This prompts you to enter the header comment (no self-respecting Eiffel developer even *thinks* of writing a feature without a header comment). At the end of the header comment, type Return if the header comment continues, otherwise type a down arrow to continue with the indentation for the beginning of the routine, with one of the keywords **require**, **local**, **do**, **external**, **once**. Once you type **do**, followed by a Return or space, the completion mechanism will insert the appropriate **end**, but other than that it doesn't try to produce an entire routine structure because there are too many syntactical choices (precondition or not, postcondition or not, locals or not etc.).

Also interesting is **feature completion**, using the **CTRL-SPACE** key. It works at two levels:

- You can type the beginning of the name of a feature of the current class, then CTRL-SPACE to get possible completions.
- Once you have typed the name of a query (attribute or function), either all by yourself or aided by the previous completion technique, you can type a period followed by CTRL-SPACE to get the list of possible features to be applied, deduced from the list of features in the corresponding class (the type of the query).

In both cases, 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 through the Escape key.

Here for example is the menu you will see in the body of our example routine if you type *io.* followed by CTRL-SPACE, where *io* is the feature, coming from class *ANY*, that provides access to standard input and output facilities:



The following properties enhance the convenience of the completion mechanisms:

- If only one completion is possible, no menu appears; the completion is selected.
- If the cursor is just after the name of a query (which you have fully typed, or obtained through completion), typing CTRL-SPACE once more will produce a period, as if you had typed it.
- When a menu of possible completions is displayed, typing CTRL-SPACE will select the first of them.

The combination of these facilities means that you can often obtain what you want simply by typing CTRL-SPACE repeatedly.

Also note the following properties of automatic feature completion:

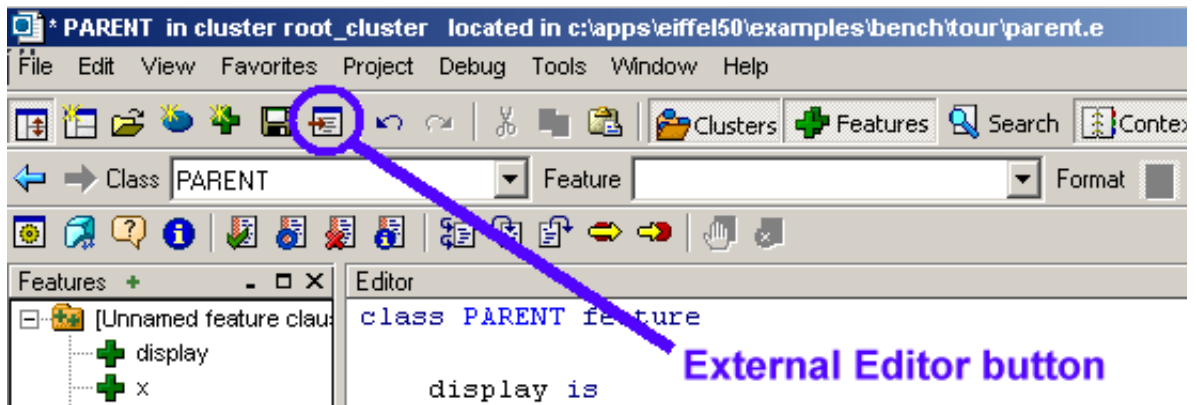
- The mechanism will only work for queries that were present at the time of the last successful compilation. So if you add an attribute, say *attr*, to the current class, and do not recompile, typing *a*-CTRL-SPACE will not display *attr*. To make sure that it's included in completion proposals, save and recompile. (Remember, incremental compilation is fast in EiffelStudio, so there is nothing wrong in compiling early and often.) The same rule holds for features of *other* classes, those that will appear in proposed completions after a period.
- Automatic completion is applicable to features, not local entities or formal arguments.
- The features proposed for automatic completion include all features of the class: those declared in the class itself, or *immediate* features, and those *inherited* from proper ancestors, direct or indirect, with one exception: by default the list will not include features from the universal class *ANY*, which serves as ancestor to all classes and provides many features for comparison, copying, input-output, reflection etc. Including *ANY's* features would clutter all menus with too many features. So for example typing *i* followed by CTRL-SPACE will not suggest *io* among the possible completions. You can change this policy through Preferences. The policy does not apply to remote feature completion for an entity *x* declared of type *ANY*: typing CTRL-SPACE after *x.* will produce the list of *ANY's* features.



## Using your own editor

You may have a favorite editor and prefer to use it at least in some cases. The EiffelStudio incremental compilation mechanism, to be studied shortly, recognizes that files have been modified outside of EiffelStudio (by checking their time stamps) and will without any difficulty take their modified versions into account.

You can also call an outside editor on a class from within EiffelStudio. Just use **File → External Editor** or the corresponding button in the top toolbar:



This will call the editor of your choice. The default is Notepad on Windows and Vi on Unix and Linux. You can easily change this to any editor by entering the desired editor command in **Tools → Preferences → Global Preferences**. In this command text you can use the two special notations **\$target** and **\$line**; when EiffelStudio calls the selected command, it will replace any occurrence of **\$target** by the name of the file where the current class resides, and **\$line** by the line number at which the Editing pane is currently scrolled. If you include one or both of these markers at the appropriate argument positions for the command, this will enable you — assuming the editor supports the appropriate options — to make sure it starts at exactly the right place. For example the default editor command under Unix is

**vi +\$line \$target**

meaning: start the Vi editor on the **\$target** file, initially positioned at line **\$line** (the **+line\_number** command-line option of Vi directs it to start at line **line\_number**).



If you start an external editor on a class, then exit the editor after possibly making changes, EiffelStudio will immediately update the class text in the Editing pane. Also note more generally that EiffelStudio will detect made separately on the same class, and warn you of possible conflicts.

Note that several important text editors have **Eiffel modes**, which support the syntax-directed editing of Eiffel texts. They include:

- *Vim*, for Vi iMproved, an extension of Vi available on both Unix/Linux and Windows — see [www.vim.org](http://www.vim.org)
- *Emacs* — see [www.emacs.org](http://www.emacs.org).
- *Editeur*, a Windows syntax highlighting editor — see [www.studioware.com](http://www.studioware.com).

## 12 HANDLING SYNTAX AND VALIDITY ERRORS

So far we have tried to make sure that everything went smoothly. But in actual software development you may encounter error situations, and it is useful to know what can happen then.

### *Levels of language description*

Let's remind ourselves first of the basics of language specification. The book *Eiffel: The Language*, the language reference, carefully distinguishes between three levels of description: **syntax**, **validity** and **semantics**. Their roles are clearly distinct:

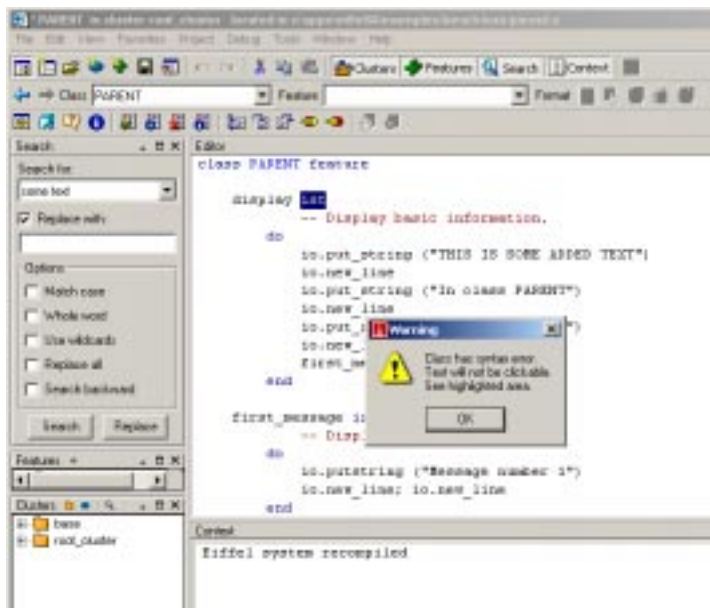
- *Syntax* defines the structure of software texts. A typical syntax rule states that an assignment starts with a **Writable** entity, continues with the symbol **:=**, and ends with an **Expression**. This is a purely structural specification, saying nothing for example about the types of the **Writable** and the **Expression**.
- *Validity*, applicable only to syntactically legal texts, defines required consistency conditions. A typical validity rule states that in an assignment the right-hand-side **Expression** must *conform* — a property of its type, defined rigorously on the basis of inheritance — to the left-hand-side **Writable**. Eiffel has about 75 validity rules; part of the language's originality is that these rules are of the “*if and only if*” form, not only telling you individual error cases (“this is valid *only if*...””) but also reassuring you that your text will in fact be valid *if* it satisfies the conditions listed exhaustively.
- *Semantics*, applicable only to valid texts, defines the software's expected run-time behavior. A typical semantic rule states that an assignment replaces the value of its left-hand-side **Writable** by the value of the right-hand-side **Expression** at the time the assignment is executed, with precise rules on the different possible cases involving references, objects and simple values.

You may make an error at any of these levels. Writing `=` instead of `:=` for the assignment symbol is a syntax error. Writing `your_integer := your_real`, with the types suggested by the names, is a validity error. Calling a feature on a void target, violating a precondition, causing a division by zero, are semantic errors.

Syntax and validity errors will be detected by the compilation process. For semantic errors, you will rely on contract checking and on the debugging tools described later. Let's look at examples of the first two cases.

### *A syntax error*

To see what happens in the case of a syntax error, replace the keyword `is` by `ist` in the first line of routine `display` of class `PARENT` ( click the position immediately after the `s` and type a `t`). Save the file by clicking on the Save button or using CTRL-S. EiffelStudio parses the class as it is saving it, and finds the error:



The position of the error is highlighted in the class text.

To correct the error, just bring the mouse back to its location, remove the spurious `t`, and click Save again; also click Compile to make sure that the project is recompiled up-to-date.

You may wonder why the syntax error messages are not a little more verbose than just **Syntax error**. The reason is merely that Eiffel’s syntax, being simple and regular, does not require sophisticated error messages; syntax errors usually result from trivial oversights. If you make a syntax error and the reason is not immediately clear, check the syntax summary in the appendix of *Eiffel: The Language*.

### Avoiding “Gotcha” panels

This syntax error message also illustrates a user interface principle present throughout EiffelStudio. At ISE, we believe that you have better things to do than spending your days clicking on **OK** buttons. Almost all messages that may appear to signal errors, display warnings, or request confirmation are of one of two kinds:

- Whenever possible, they give you **two** or more possibilities. A panel with just one possibility is not much of a choice.
- If the environment cannot give two possibilities — as in the case of an error message, for which the environment must let the user go on after making sure the warning has been seen — EiffelStudio doesn’t force you, like many of today’s user interfaces, to acknowledge your sins by clicking an **OK** button. You simply correct the error and move on.

In the case of our syntax error there *is* an **OK** button and you may click it, but you don’t have to. It is there mostly to avoid troubling people (almost everyone, that is) who are used to the “Gotcha!” panels of the dominant user interfaces. Once you get used to the EiffelStudio style, you won’t bother clicking **OK** for a syntax error: you will correct the error, and when you click Save again the panel will go away like a bad dream.

### A validity error

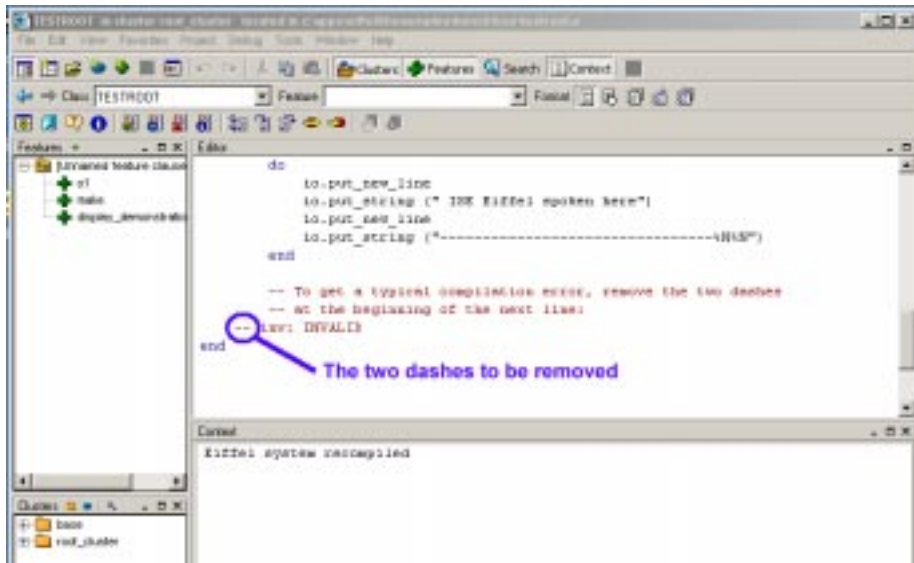
A validity error is a violation of one of the validity constraints given in *Eiffel: The Language*. Every such constraint is identified by a four-letter code of the form **VXXX** (the first letter is always **V**).

A validity error will produce a precise error message, which includes the validity code. Although short, the error message is usually sufficient to find out what the error is. If not, you can get the complete rule, straight from the book.

To see this mechanism at work, let us introduce a validity error. There is in fact one ready for you in class **TESTROOT**. Target a Development Tool to this class; at the end of its text, just before the final **end**, you will find the following comment line:

```
-- inv: INVALID
```

If uncommented, this is a declaration of a feature of type **INVALID**. A class called **INVALID** indeed exists in file **invalid.e** of the root cluster, but it contains a validity error. To see what it is, remove the initial double-dash **--** in the above line from class **TESTROOT** so that it is not a comment any more.



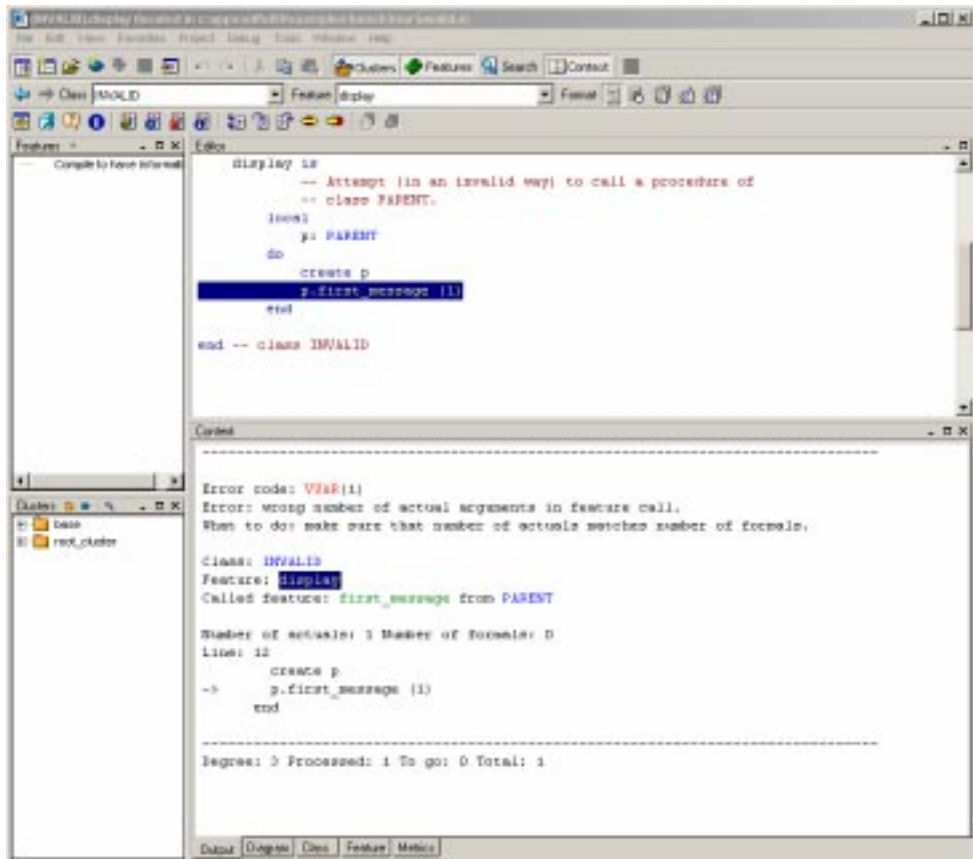
Click **Save**, then **Compile**. Compilation starts but after a few degrees it stops with an error message that appears in the bottom Context pane (you may have to do some resizing to see it in its entirety):



As the error message indicates, you have (shame on you) violated the validity rule **VUAR**, which requires the number and types of actual arguments in a routine call to match the number and types of formal arguments declared in the routine.

One of the interesting properties of the error message is that everything in color is **clickable**: class name, feature name, but also the error code. This means that you can start a Pick-and-Drop on any of these elements to find out more.

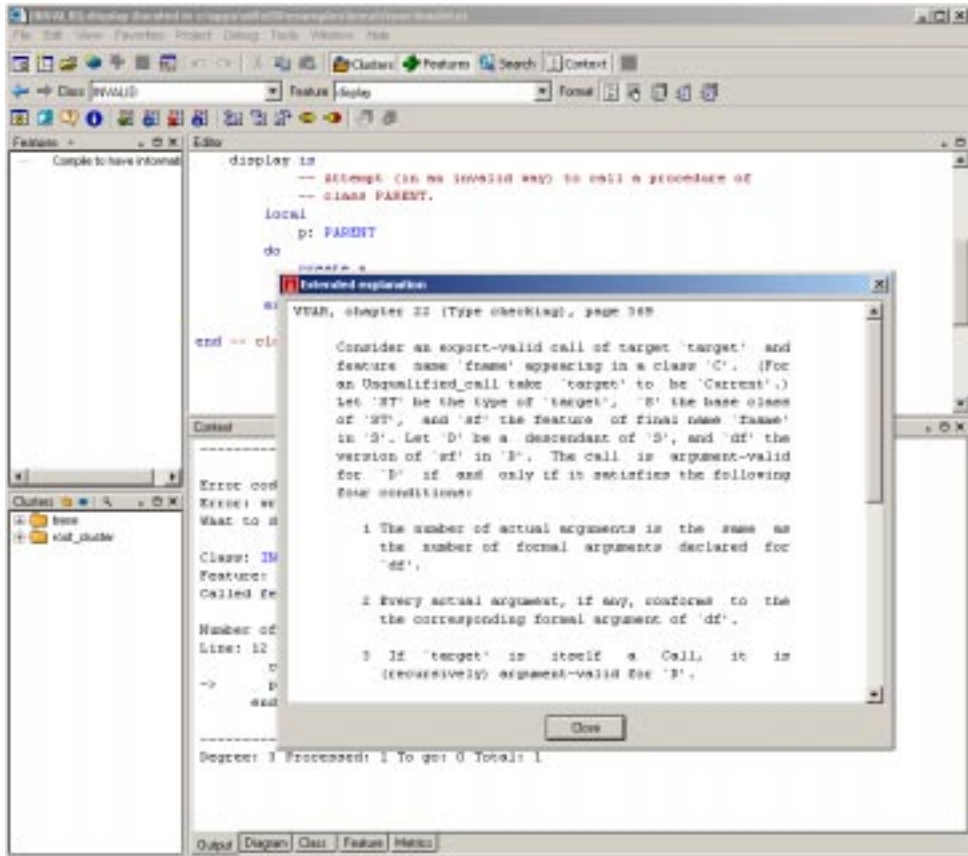
For example, to see the exact context of the error, pick-and-drop the name of the affected feature, *display* — appearing on the fifth non-blank line, after **Feature:** — and pick-and-drop it to the top Text window. (As you remember this means: right-click on it and release; move the mouse to the text window, without clicking any button; right-click again. During the move the cursor shows a cross, the symbol for features). This displays the erroneous feature:



Note on this display a special property of Pick-and-Drop when its source is a feature name appearing in a validity error message: the instruction that causes the error is highlighted.



As is often the case when dropping into a specific hole, you don't need to shoot straight; dropping the pebble anywhere in the Editing pane has the same effect as dropping it into the Explanation hole:



The result is to display the complete text of the violated rule, straight from the pages of *Eiffel: The Language*.

The rule has several clauses, numbered. Since the error message showed the error code as **VUAR(1)**, the violated clause is the first; this convention of showing the clause number in parentheses applies to all multi-clause validity constraints.

To correct the error the easiest is to go back to class **TESTROOT** and reinstate the comment symbol **--** (two consecutive dashes) on the erroneous line. Save and compile to continue with a valid system.



## 13 DEBUGGING AND RUN-TIME MONITORING

The next set of EiffelStudio capabilities enable you to control and monitor the execution of your systems. The obvious immediate application is to debugging; but the more general goal is to let you follow the execution of your systems, explore the object structures, and gain a better understanding of the software.

### *A reminder about debugging in Eiffel*

Before looking at debugging facilities don't forget that debugging in Eiffel is different. The presence of Design by Contract mechanisms gives the debugging process a clear sense of direction. The speed of the recompilation process makes it easy to recompile after a change; after getting rid of syntax and validity errors, you run the system again, and remaining errors are often caught as violations of contract clauses — routine preconditions, routine postconditions, class invariants.

The facilities to be described now are also useful when you find such an error, as they will help you study its execution context. In fact, one of the characteristics of the debugging mechanism is that there is no “debugger” proper, no more than there is a “browser”; you have instead a set of facilities supporting controlled execution and debugging. This means for example that:

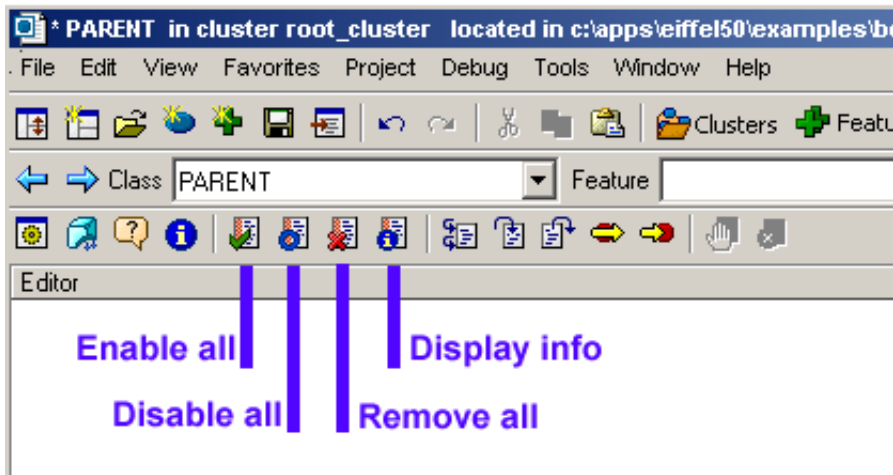
- While debugging, you can access all the browsing capabilities to explore the features and classes surrounding the cause of an error.
- While browsing, you can launch or resume execution, and follow its progress through the debugging facilities.
- If execution stops on an exception — assertion violation, arithmetic overflow, call on a void target ... — you have all the environment's facilities at your disposal to understand what happened.



## Setting breakpoints

To control the execution you will set breakpoints, indicating places where you want to interrupt the execution. You may set a breakpoint on an individual instruction of a routine, on the routine's precondition or postcondition, or on the routine as a whole, meaning its first operation (precondition or instruction).

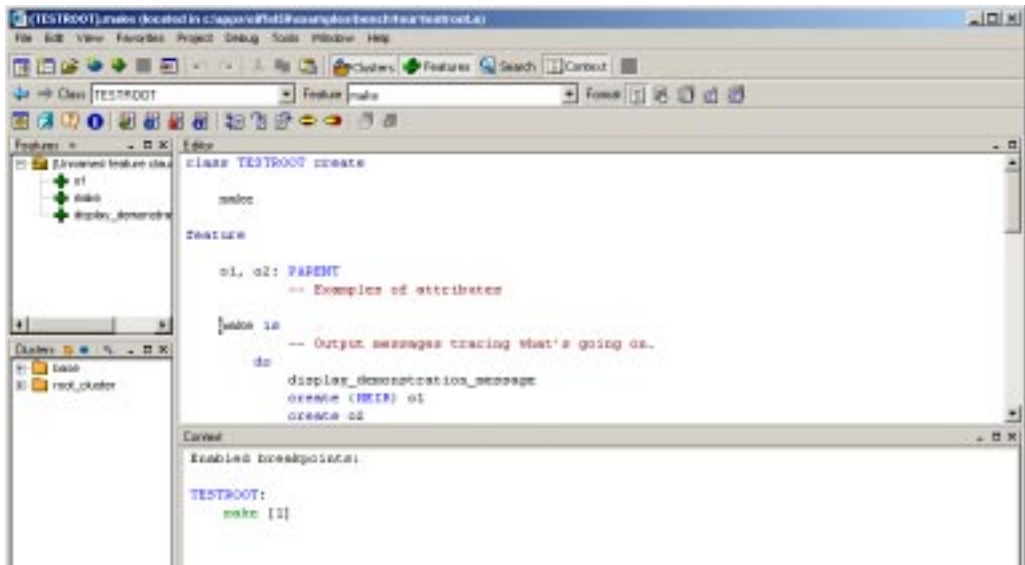
A group of icons on the Project Toolbar help control breakpoints. They are known in EiffelStudio terminology as “*buttonholes*”, meaning that they can serve both as buttons (you can click them to get some functions) and holes (you can pick-and-drop into them to get some other functions).



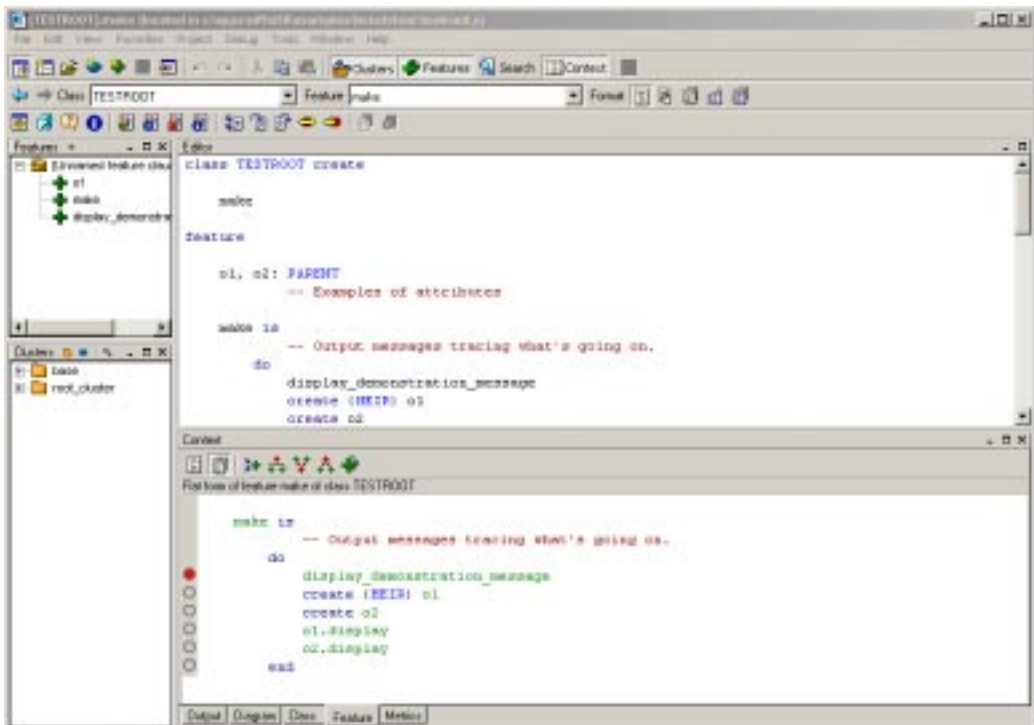
The labels correspond to the icons' use as buttons: enable all set breakpoints, disable them all, clear all, display information on current breakpoints. The difference between “disabling” and “removing” is that disabling turns off breakpoints until further notice but remembers them, so that you can later re-enable them, whereas “removing” clears them for good.

Target a Development Tool to the class **TESTROOT** and pick-and-drop the name of the procedure **make** (the first routine, after the declaration of the two attributes o1 and o2) to the **Enable all** icon, used here as a hole. This sets and enables a breakpoint on the routine. Click the button labeled **Display info** above to get the list of breakpoints, as shown in the the next figure.

If the Context pane is not present, **Display info** will bring it up. You can get the same effect as that of **Display info** by selecting the **Output** tab in the Context pane..

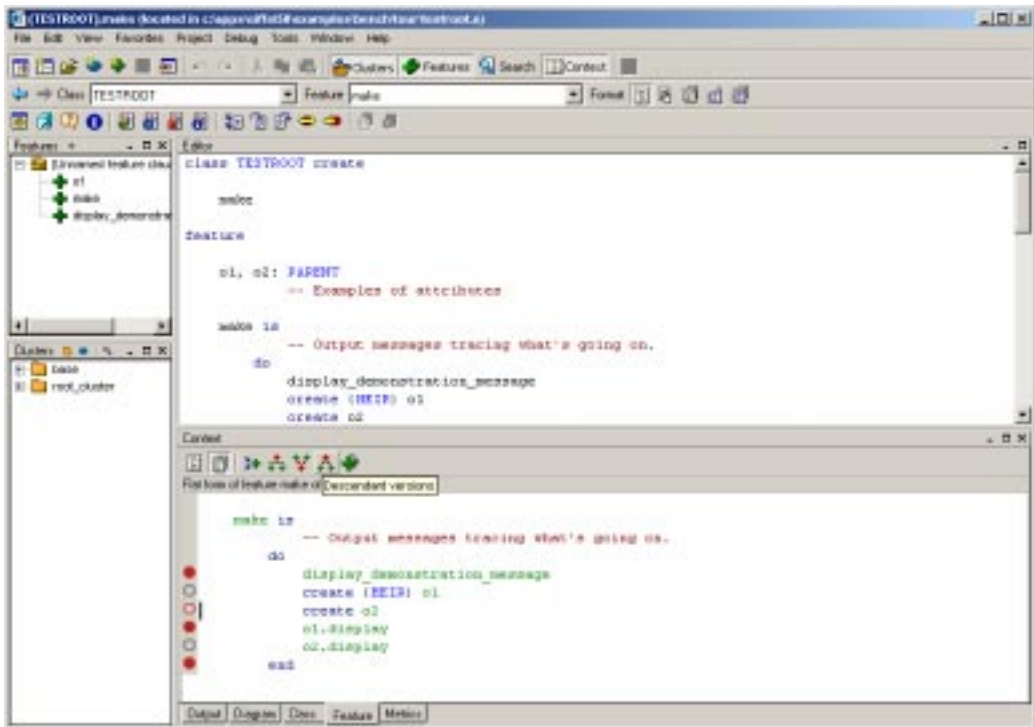


This shows that so far you have enabled only one breakpoint. For a finer degree of control, let's look at the feature's flat form. Pick-and-drop **make** to the Context pane — from the Context pane itself, since its name appears there, or the Editing pane); this sets the Context to the **Feature** tab. Select the **Flat** view if that wasn't the last one used:



The small circles on the left side of the Flat form indicate breakpoint positions. Empty ones are not set; enabled breakpoints are marked by a circle filled with red. At the moment only one is enabled, corresponding to the first instruction of the routine since, as noted, setting a breakpoint on a routine as a whole means setting it on its first operation.

By (left) clicking on a breakpoint mark, you toggle it between enabled and not set. You can also right-click on a mark to get a menu of possibilities. Try enabling and unsetting a few of these marks; you might get something like this:



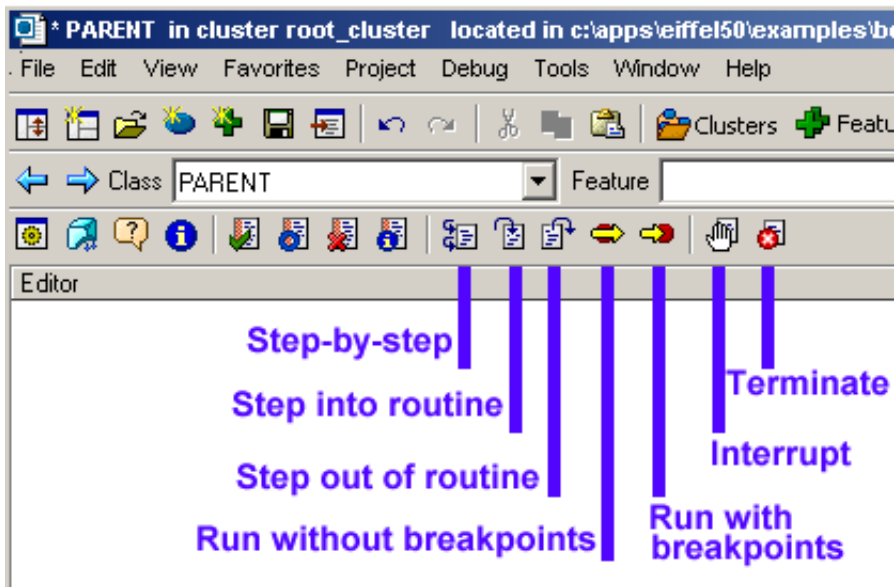
The breakpoint mark for the routine's third instruction, **create o2**, is red but not filled; this means it is set but not enabled. You can obtain this by right-clicking on the mark and choosing **Disable breakpoint** on the menu that comes up. Any potential breakpoint will be in one of three states: not set; enabled; set but disabled.

You can see the list of enabled and disabled breakpoints by clicking the **Output** tab, or the **Display info** button in the Project Tool.

For the continuation of this tour it doesn't matter which exact breakpoints of **make** you've set, as long as the one of its first instruction is set and enabled (red-filled circle) as above. Please make sure this is the case before proceeding.

## Executing with breakpoints

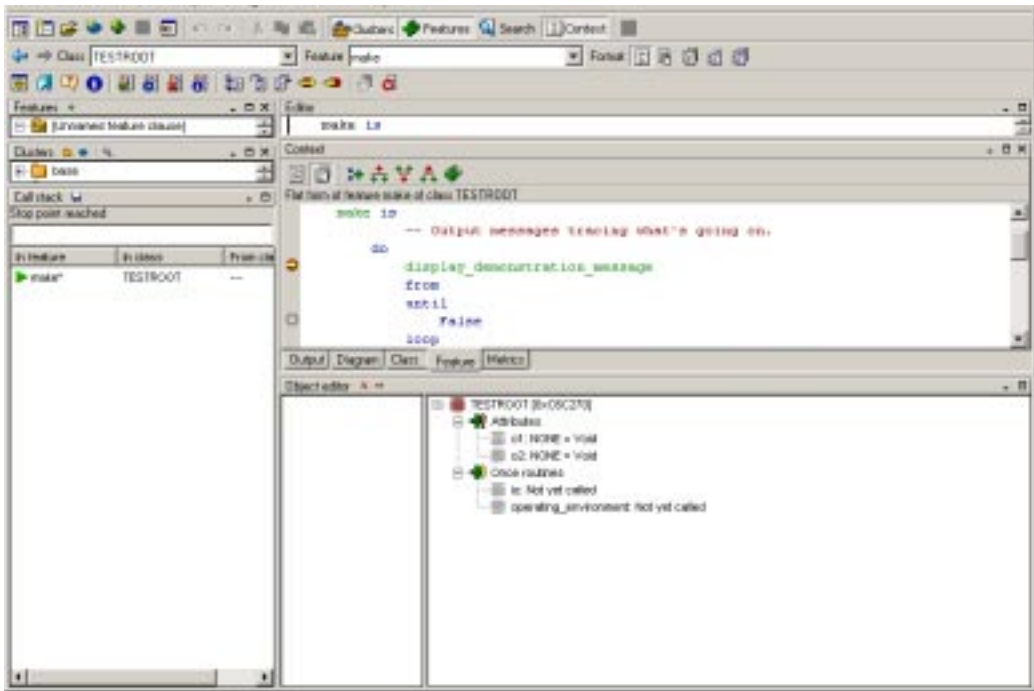
To execute, you will use the following Run buttons in the Project toolbar, or the corresponding entries in the **Debug** menu:



The buttons are shown here in full bloom, but at any times some of them will be grayed out. **Terminate** is only active when execution has started and not terminated; **Interrupt** is only active when the execution is active (not stopped at a breakpoint).

The **Debug** menu entries will also remind you of shortcuts: F10 for **Step-by-step**, F11 for **Step into routine**, Shift-F11 for **Step out of routine**, CTRL-F5 for **Run without breakpoints**, F5 for **Run with breakpoints**, CTRL-Shift-F5 for **Interrupt**, Shift-F5 for **Terminate**.

Start execution of the compiled system by clicking **Run with breakpoints**. The display automatically switches to accommodate supplementary panes providing debugging information. Execution stops on the breakpoint that you have enabled on the first instruction of procedure *make*:



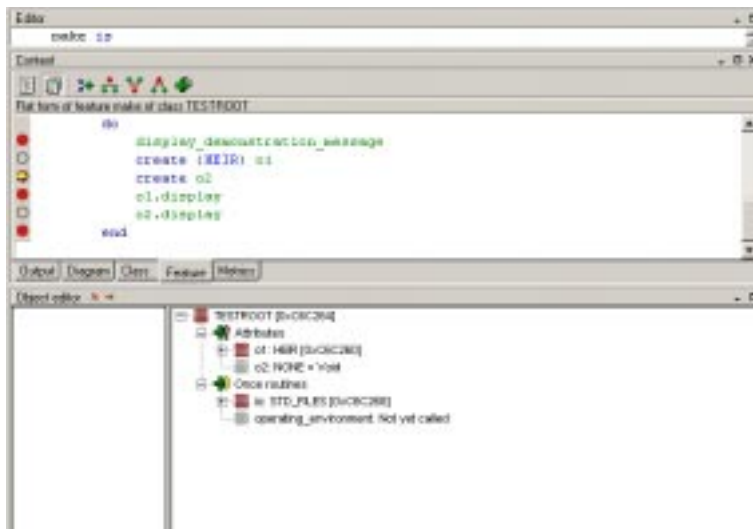
The pane at the bottom left indicates that execution has stopped in *make*. The Context shows the flat form of that routine, with a new icon to indicate the stop point which execution has reached. At the bottom right is a new tool, the **Object Editor**, which shows the content of current object and (later) related objects. At the moment you can see that:

- The current object is an instance of class *TESTROOT*.
- The class (as you can also see from its text in a Development Tool) has two attributes *o1* and *o2*, for which the corresponding fields in the current object are both void; this is as expected since you haven't yet executed the two creation instructions *create {HEIR} o1* and *create o2*, which come after the breakpoint.
- Along with attributes, an Eiffel class may have **once functions**, executed at most once — the first time they are called — in a given session, and from then on always returning the same value. Here the once function *io*, has been called and returned an object of type *STD\_FILES*, but *operating\_environment* (coming, like *io*, from the top-level class *ANY*) has not yet been called.

The execution-time objects that you may display in an Object Editor are our latest kind of EiffelStudio “development object”, along with classes, features, explanations, clusters; notice their distinctive icon, a rectangular mesh suggestion an object's division into fields. It appears colored for actual objects, gray for void references such as *operating\_environment*.

## Monitoring progress

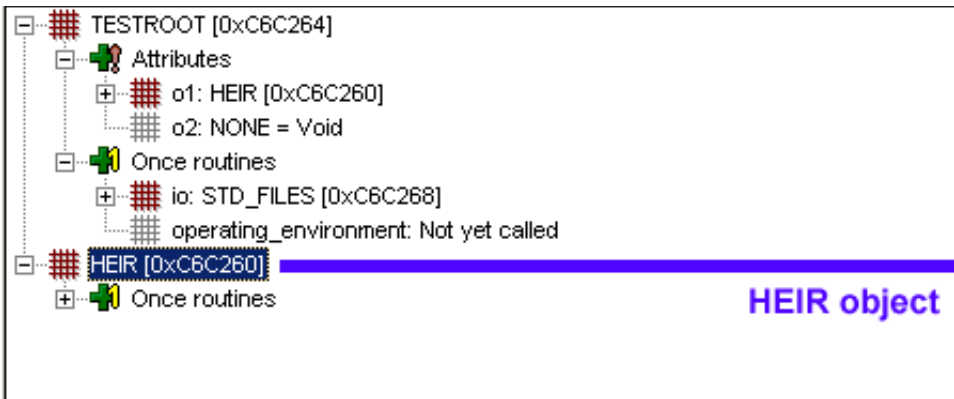
Click twice on **Step-by-step** (or press the function key F10 twice). Monitor, in the flat form of *make*, the marker that shows execution progress; note that the marker always points to the *next* operation to be executed. After the two steps, the Context and Object panes look like this:



The last instruction that you executed is **create {HEIR} o1**, meaning create an object and attach it to *o1*, but instead of using the declared type *PARENT* of *o1* use its proper descendant *HEIR*. As a result, the entry for *o1* in the Object Tool no longer shows void but an object of type *HEIR*. Note that all objects are identified by their address in hexadecimal; such an address is by itself meaningless, but enables you to see quickly whether two object references are attached to the same object. The addresses you see as you run the Guided Tour will — except for some unlikely coincidence — different from the ones appearing here.

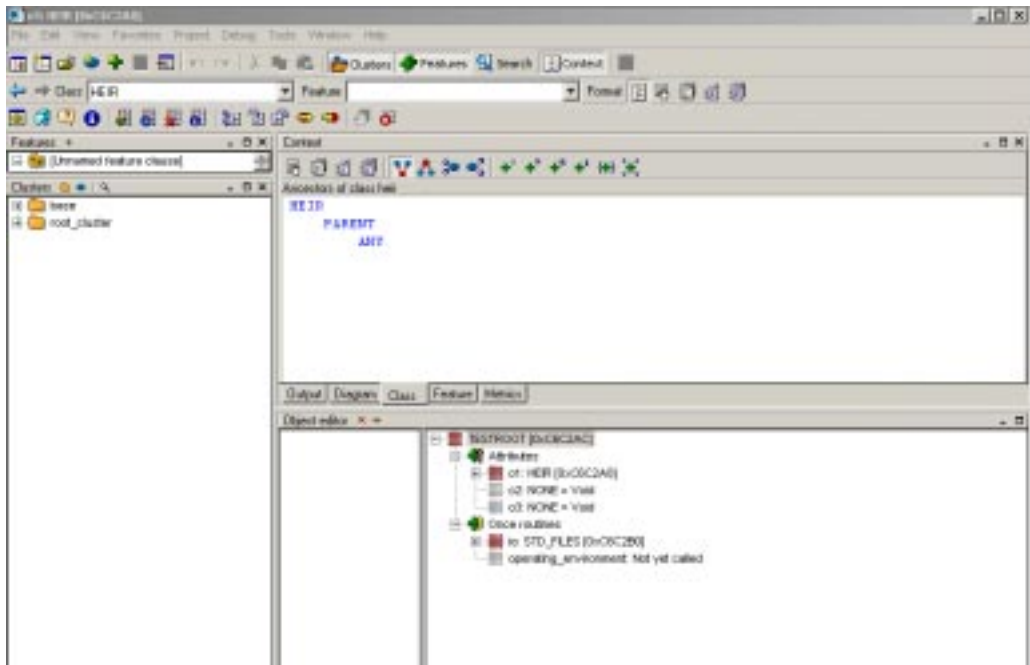
Note that since the ISE Eiffel garbage collector compacts memory and hence may move objects around, the address of a given object is not guaranteed to remain the same throughout a session.

To see the details of the object, pick-and-drop its identifier in place (that is to say, drop it in the Object pane itself). A new object entry appears, showing the object:



### *From the instance to the class*

Now try pick-and-dropping that same object, the instance of *HEIR*, to the Context pane above the Object window. The effect is the same as if you had pick-and-dropped the **class** name *HEIR*: retarget the Context pane to that class.



In the same way that you can drop a *feature* pebble into a tool that expects a class, you can also drop an *object* pebble, which will be understood as denoting the object's generating class.

Because the Context pane is showing a class, it has switched to the default format for classes, **Ancestors**, and is showing the ancestors of **HEIR**. Click the **Feature** tab of the Context pane to set it back to feature information for the continuation of our debugging session. No feature is currently displayed.

### *Stepping into and out of a routine*

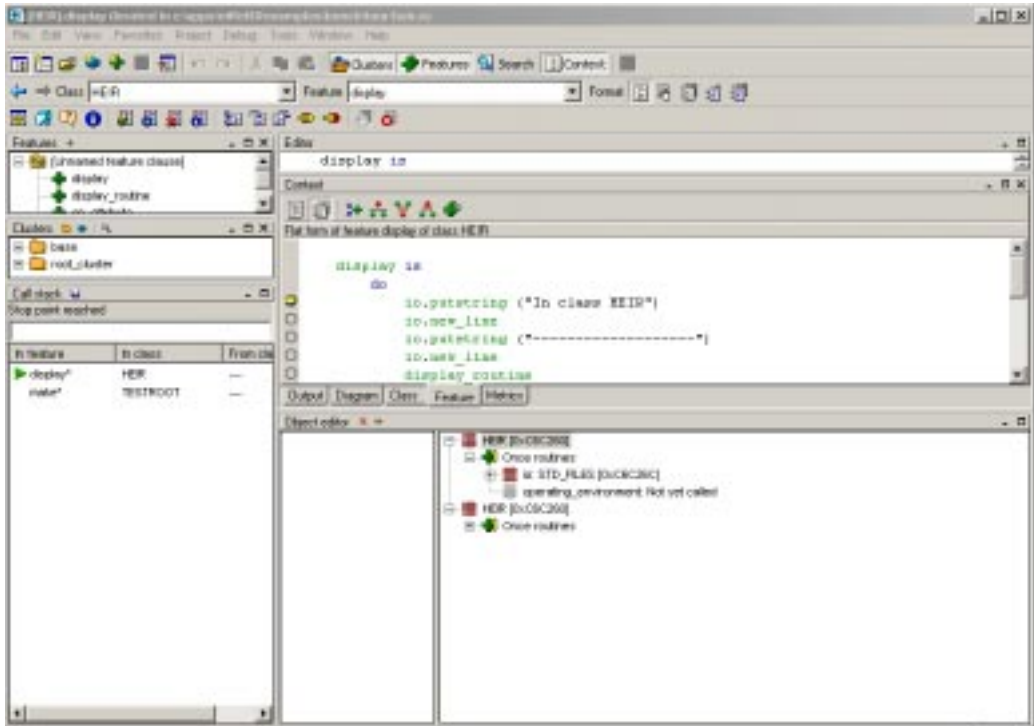
Click **Step-by-step** once more to advance just before the call **o1.display**:



If at this stage you chose **Step-by-step** again this would execute the next step in the current routine, the call **o1.display**, treating the entire execution of **display** from class **HEIR** as a single operation. Assume instead that you want to get into that routine and follow the details of its execution. For one thing, you might not know that it's a routine of class **HEIR**, since **o1** is declared of type **PARENT** and it's only through polymorphism, **o1** being dynamically of type **HEIR** at this point, and through dynamic binding, that the execution ends up calling a routine from **HEIR**. Of course here it's obvious because of the wording of the **create** a few lines up, but in many cases, especially all those for which polymorphism and dynamic binding are *really* interesting, the exact type won't be obvious from the software text.



Click the **Step into routine** button (or press F11). This brings execution to the beginning of the appropriate *display* routine in class *HEIR*:



You can also expand the **Call stack** pane on the left to see the full call stack, consisting here of only two levels. Minimize it to get the above display back.

Now click **Step out of routine** (Shift-F11) to finish the execution of *display*. This brings you back to the next instruction of the calling routine, *make* of *TESTROOT*.

## Terminating

You may now click the **Terminate** button (Shift-F5) to end execution. The execution-specific panes go away and the display returns to what it was before execution.

## Other debugging capabilities

In this little application nothing runs long enough to give you the time to interrupt it. In a longer-running application you may want to interrupt execution, *without* necessarily terminating it, while it's running (not stopped on a breakpoint). This is the purpose of the **Interrupt** button (CTRL-Shift-F5). It will interrupt execution at the closest potential breakpoint position, letting you — as when execution stops because of an exception — take advantage of all the debugging and browsing facilities to see what's going on inside the program. You may then restart execution — with or without breakpoints, single-stepping, out of the current routine, into the next routine — by choosing the appropriate Run button

In debugging sessions for more advanced applications, you will also find the self-explanatory mechanisms enabling you, in addition to what we have seen, to examine all the objects on the “call stack”: arguments and local entities of the current routine, its caller, caller's caller and so on.

The combination of these facilities provides you with a level of *dynamic* information on the execution of your system that matches the *static* information that the browsing mechanisms studied in preceding chapters provide about the system's structure.

## 14 COMPUTING PROJECT METRICS

In earlier sections we saw how EiffelStudio provides extensive documentation on your systems. That information was qualitative. Project managers and developers will also be interested in *quantitative* information about their projects. You can get such information through the **Metrics** tab of the Context pane, which enables you to perform a number of operations, to be detailed next:

- Apply predefined metrics (such as number of classes, number of invariants, number of features, number of compilations so far, and many others) to components of a system at various levels including feature, class, cluster, entire system.
- Define new metrics, through mathematical formulae or boolean selection, and apply them to your project.
- Store measurement results, as well as metric definitions, into an XML archive that can be stored locally or made available on the Web for future reference.
- Compare the measurements on a system to those on record locally or on a Web site. ISE has released on its own site an archive recording the metric properties of its basic libraries, available to any other project for comparison.

## *Methodological observations*

Although the field of software metrics is a rich one with an abundant literature, its methodological basis is sometimes subject to question. One should resist the tendency to believe numbers just because they are numbers (“lies, damn lies, and metrics”).

Software engineers and their managers expect, however, to reap at least some of the benefits that precise quantification has brought to other engineering fields. Such is the purpose of software metrics, defined as **quantitative estimates of product and project properties**. Object-oriented development, with the rich software structures that it induces, is particularly amenable to metric analysis. Even when some of the measures do not seem to bring much by themselves, *comparing* them to those of other projects may point to important properties of the software.

The metrics capabilities of EiffelStudio were designed with these observations in mind. They result from a conservative approach, where no metric is provided without a credible assumption that it reflects some meaningful project or product attribute. For example, you will find a way to define a new metric as a linear combination of existing ones, but not a way to compute arbitrary arithmetic operations, since it isn’t clear that multiplying two metrics, for example, ever makes sense.

## *Metric terminology*

The following terms are used in the presentation of EiffelStudio metric mechanisms.

A **metric** — not to be confused with a measure — is a quantitative property of software products or processes whose possible values are numbers. A **measure** is the value of a metric for a certain product or process.

For example, we can evaluate the metric “number of classes in the system”, called just *Classes*, by counting the classes in our system. This yields a measure.

We may distinguish between *product* metrics, which measure properties of the elements being turned out (code, designs, documentation, bug reports...) and *process* metrics, which measure properties of the process whereby they are being turned out (salaries, expenses, time spent, delays...). The current metric facilities of EiffelStudio are primarily product-oriented, although a few metrics, such as “number of compilations”, are process metrics.

Any metric should be *relevant*: related to some interesting property of the processes or products being measured: cost, estimated number of bugs, ease of maintenance... A **metric theory** is a set of metric definitions accompanied with a set of convincing arguments to show that the metrics are relevant. Neither EiffelStudio nor this document provides a metric theory.

The numbers yielded by measures are meaningless unless we describe what they refer to. Every metric is relative to a certain **unit**, specified as part of its definition. For example the unit for a metric that counts classes, such as *Classes*, is called *CLASS*.

EiffelStudio provides a set of predefined units. Some simply serve to count occurrences of certain construct specimens in the software; examples include *CLASS*, *CLUSTER*, *FEATURE*, *LINE*, ... The metric *RATIO* describes metrics whose values are divisions, for example “average number of classes per cluster”, obtained by dividing the number of classes by the number of clusters.

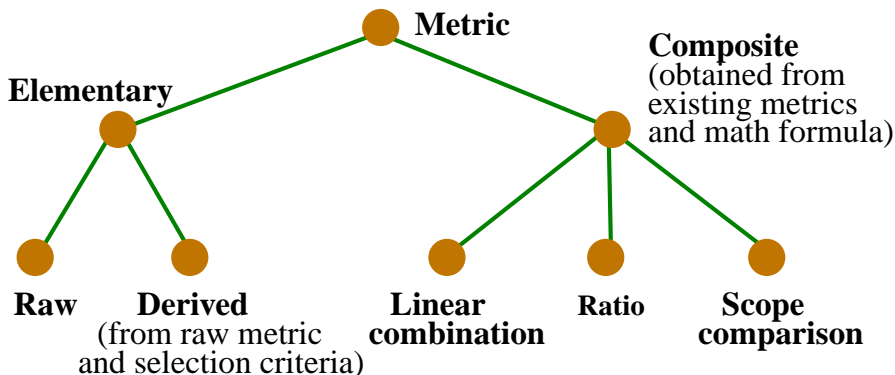
Any metric applies over one or more **scope types**. A scope type is a type of product or process over which the metric is measured; for product metric, examples include *feature* (meaning that we will compute a metric over a single feature), *class*, *cluster*, *system*. They obey an order relation corresponding to containment order: a feature belongs to a class, a class to a cluster and so on.

A **scope** is an instance of a scope type. For example 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. For example we may compute the value of the metric *Classes* over a certain system.

### *Kinds of metrics*

The EiffelStudio metric framework 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. An example is the number of precondition clauses in a class.
- A composite metric, defined by a user of the environment, applies a mathematical or logical formula involving other metrics (elementary or previously defined metrics).

Among elementary metrics, we make a further distinction between raw and derived:

- Raw metrics are simple counts, built-in into the environment, of occurrences of certain basic elements. For example *Classes*, the number of classes, is raw.
- It is often useful to define a new metric by subjecting a raw metric to one or more **selection criteria**. For example a class may be either deferred (abstract) or effective (concrete, i.e. fully implemented). This is a selection criterion. Separately, a class may have an invariant, or not; this is another selection criterion, *Invariant\_equipped*. You might want to know the number of classes that are deferred and have no invariant clause; if so, you may define a derived metric by submitting the raw metric *Classes* to both of these criteria connected by an “and”.

The precise definition of *selection criterion* for a raw metric is: a property with a fixed set of possible values (two or more) characterizing the patterns being counted by the metric. Without the notions of selection criteria and derived metrics, EiffelStudio would need to have predefined (raw) metrics including all possible combinations, such as “deferred and no invariant”. This would quickly grow out of hand.

### Defining a derived metric

Let us define a new derived metric. In the Context pane, choose the **Metrics** tab. Click the **Define new metric** button:



In the dialog that comes up, the first tab, selected by default, is the one we want, **Derived**:



In the **Raw Metric** entry, currently set to *Classes*, bring up the menu and select *Features*. 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:



Assume you want a metric called *Public\_functions\_full\_contracts* that counts the number of features that are functions, are exported, and have both a precondition and a postcondition. Enter its name and check the selected criteria:



Click **OK** to add this metric to the set of available metrics. It's actually been entered as the default metric to be computed, although we'll compute it only in a short while:



Note that the **Unit** field indicates the unit automatically assigned to this derived metric, *Feature*, the same as the unit for the raw metric *Features* from which you have derived it.

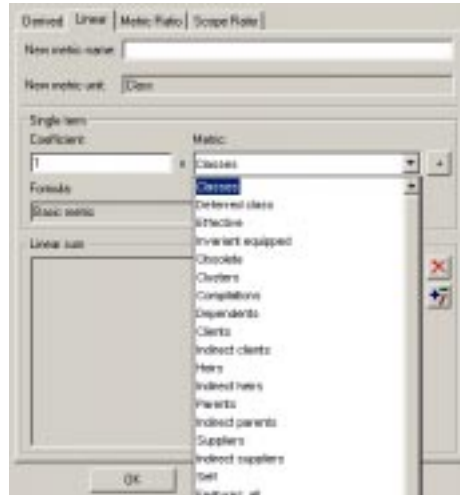
### Composite metrics

A composite metrics applies one or more mathematical operators to a set of metrics, themselves either elementary (raw or derived) or already composite. In line with the conservative approach mentioned earlier, EiffelStudio permits only three kinds of mathematical operator:

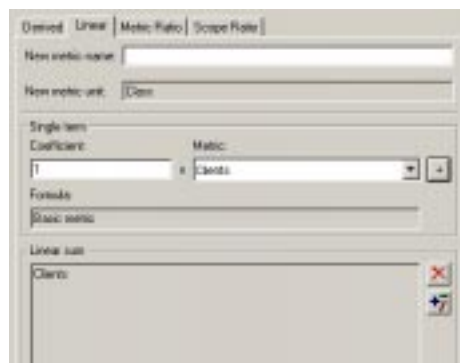
- **Linear** metrics of the form  $\sum k_i \cdot m_i$ , where the  $k_i$  are real values and the  $m_i$  existing metrics (either elementary or basic) with the same unit, other than *RATIO*. (It would be improper to add two *RATIO* since they might be ratios of incompatible things.)

- **Ratio** metrics 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* (again because *RATIO* is a catch-all category for all divisions, so you can't divide further without courting incoherence). The resulting unit is *RATIO*.
- **Scope comparison** metrics that 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 define a linear metric *Direct dependents* that counts heirs (direct descendants) and direct clients of a class. Click **Define new metric** again and choose the tab **Linear**. Bring up the **Metric** menu:



To define the first term of the sum, select *Clients*. Do **not** click the button marked OK (we are not done yet as we need a second term) but instead click the **+** button next to **Metric**:





The field at the bottom reflects the current state of the linear combination, which only includes the term *Clients*, with the default coefficient 1. Now bring up the **Metric** menu again, choose *Heirs* this time, and click **+** again. The expression for the sum appears in the bottom field:

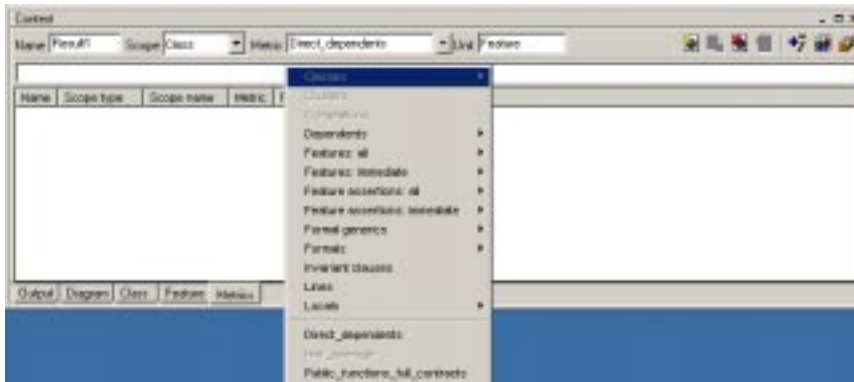
The screenshot shows the 'Metric Ratio' dialog box. The 'New metric name' field is empty. The 'New metric unit' is set to 'Class'. Under 'Single term', the 'Coefficient' is 1 and the 'Metric' is 'Heirs'. The 'Formula' field shows 'Heirs'. The 'Linear sum' field shows 'Clients + Heirs'. The 'OK', 'Save', and 'Cancel' buttons are at the bottom.

The name of our new linear metric will be *Direct dependents*. Enter it into the top field, **New\_metric\_name** (if you forget, you will be reminded), and click **OK**. The name of the new metric and its unit become the defaults in the corresponding fields of the Context pane.

Let's define one more metric, this time a ratio. We want to know the “branching factor”: average number of heirs per class. This will be the ration of metrics *Heirs* and *Classes*. Click **Define new metric** then **Metric ratio**:

The screenshot shows the 'Metric Ratio' dialog box. The 'New metric name' field is empty. The 'New metric unit' is set to 'Ratio'. Under 'Numerator', the 'Metric' is 'Classes'. Under 'Denominator', the 'Metric' is 'Classes'. The 'Formula' field shows 'Classes / Classes'. The 'Percentage' checkbox is checked. The 'OK', 'Save', and 'Cancel' buttons are at the bottom.

Select *Heirs* in the **Numerator** menu on the left, *Classes* in the **Denominator** menu on the right; do *not* check the box “Display as percentage” at the bottom (we prefer to see the branching factor as a number); enter the metric name *Average\_heirs*, more clear than “branching factor”, in the field at the top, and click **OK**. You now have three user-defined metrics available in addition to predefined metrics, as you can check by bringing up the **Metric** menu on the Context pane:



### *Applicable scope types*

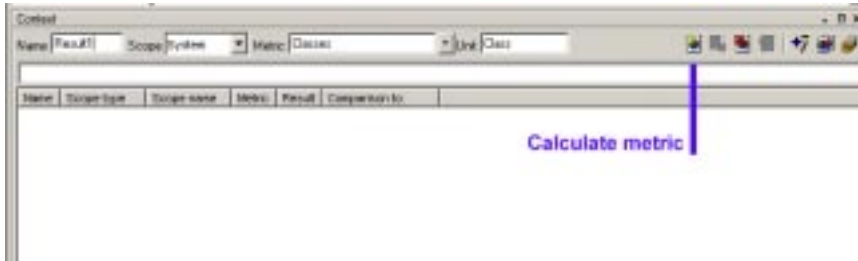
The reason *Average\_heirs* is grayed out on the last figure is that it’s not applicable to the current scope. Each metric indeed has a minimum “applicable scope type”; for example it doesn’t make sense to compute the number of attributes in a feature, since the smallest construct in which attributes appear is a class, bigger than a feature. Similarly, the minimum scope type for *Classes* (the denominator of *Average\_heirs*) is *Cluster*, since we can only start counting classes at that level.

### *Computing measures*

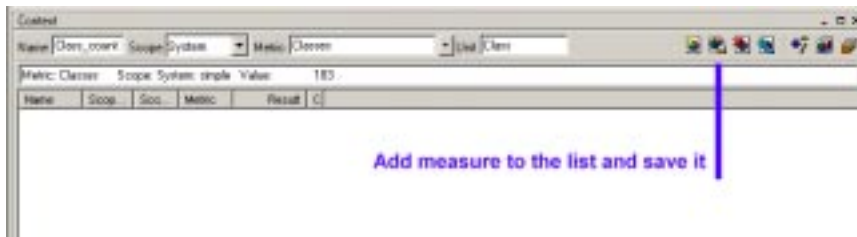
Let’s now compute a few measures. These will all be relative to the entire system, so the first thing to do is to change the **Scope** field of the Context pane (the second field from the left at the top on the preceding figure) to *System*, the next-to-last entry on the corresponding menu.

The measures that you get should be identical to those shown next, or very close. Differences, if any, may result from changes in the EiffelStudio delivery, especially EiffelBase, or operations that you have performed on the Guided Tour system.

First let's use a predefined metric to compute the number of classes in our system. In the **Metrics** field bring up the menu and choose **Classes** → **All**. The **Unit** field is updated to the unit of that metric:



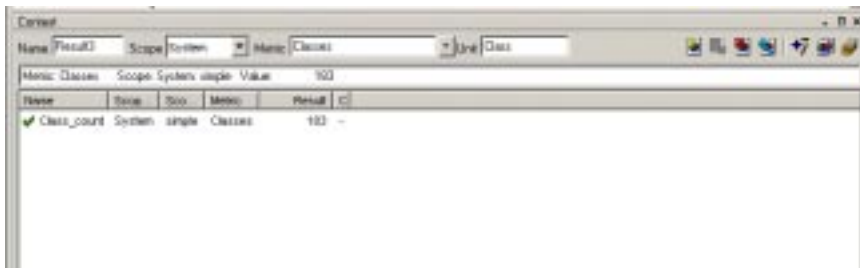
For more clarity, change the name of the next measure, initialized to *Result1* (leftmost top field above) to *Class\_count*. Now compute that measure by clicking the **Calculate metric** button, at the place shown on the figure. This displays the resulting measure in the field just below:



As you see, our system has 183 classes. Refining the measure — for example by restricting the scope to certain clusters — would show that all but a handful belong to the EiffelBase library. But you already know this, so don't perform these measures yet (you can do them later).

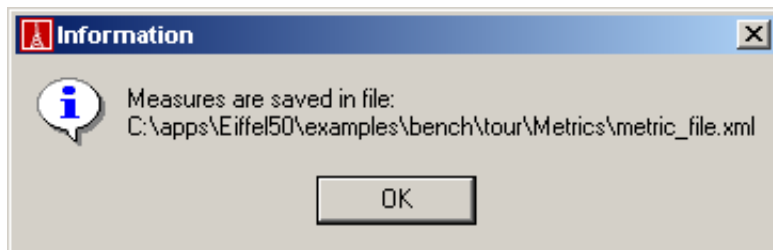
### *Saving measures*

When you compute a measure, it's not kept anywhere, and the next measure will displace it. To keep the one you just made, click the button labeled **Add measure to the list and save it** on the above figure:



The operation has had two effects:

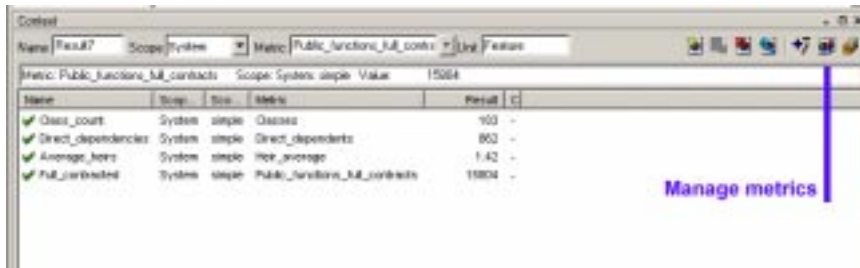
- Add the latest measure to the table of currently displayed measure.
- The first time around, as here, created a metrics file; a messages tells us this:



Metric information will be kept in the subdirectory **Metrics** of the project directory. The generated **metric\_file.xml** file contains both the metrics you have defined, and the measures you have performed. This makes it possible to use it both for:

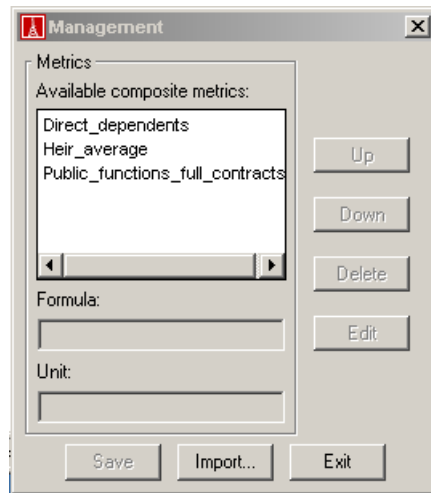
- Applying to a new project the metrics that you have defined for an existing one (so that, as you go, you can accumulate a portfolio of useful metrics).
- Using existing measures as a comparison for new projects.

To obtain more measures, apply the three metrics that you defined earlier — with names *Public\_functions\_full\_contracts*, *Direct\_dependents* and *Average\_heirs* — to the entire system and save the corresponding measures. You can use appropriate names for the results or use the EiffelStudio defaults. The result will be as follows:



### *Operations on metrics*

Click the button labeled **Manage metrics** on the last figure. This brings up the metric management panel:



You can use it to rearrange the order of the metrics you have defined (by selecting one of them and clicking **Up** or **Down**) and delete any of them. The **Formula** field shows the definition — arithmetic, or boolean — of the selected metric. The **Import** button will let you, if you have defined metrics in another project, select the corresponding XML file to add them to those of your current project.

Here (possibly after having tried rearranging the metrics) just click **Exit**.

## Metric archives

What does a measure mean? You don't necessarily know in the absolute, but you might want to compare your results to those of other projects. For example, if you have measured the average number of invariant or other contract clauses in your system, you might be curious to know how this compares to ISE's EiffelBase library.

The notion of metric archive addresses this need. You may:

- As noted above, archive all current measures into an XML file, called a **measurement archive**.
- Make this measurement archive available in a shared directory, or as a **URL on the Internet**.

At any time in a project, select any measurement archive, local or URL, as the **reference archive**; in that case all measures that you perform will be compared to those of the reference archive. You may select various comparison formats: percentage (the default), difference percentage, plain value.

ISE has established a new Web site, <http://metrics.eiffel.com>, as a publicly available reference for metric collections on ISE's own libraries (EiffelBase, EiffelVision, ...) and products. This provides an invaluable source of comparisons for other projects within and without ISE.

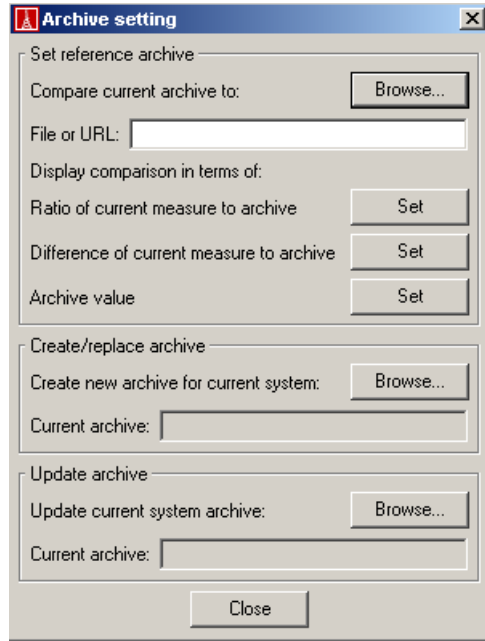
Let's see how this works. We'll compare the number of classes in our system to the number of classes in EiffelVision. Click the last button, **Compare to archive**, in the Context panel:

The screenshot shows the EiffelStudio interface. At the top, there are dropdown menus for 'Scope' (set to 'System'), 'Metric' (set to 'Public\_functions\_full\_contr'), and 'Unit' (set to 'Feature'). Below these, a status bar displays 'Public\_functions\_full\_contractions' Scope: System: simple Value: 15804. A table in the Context panel lists metrics for comparison:

	Scop...	Sco...	Metric	Result	C
	System	simple	Classes	183	-
dependencies	System	simple	Direct_dependents	862	-
s	System	simple	Heir_average	1.42	-
id	System	simple	Public_functions_full_contractions	15804	-

At the bottom right of the Context panel, there is a button labeled **Compare to archive**.

This brings up the archive comparison dialog:



You can **Browse...** to select a local archive (a **.xml** file) for comparison, but if you have access to the Internet enter the following URL in the **File or URL** file:

<http://metrics.eiffel.com/vision.xml>

If you do not have access to the Web, **Browse...** instead to the file **vision.xml** of the documentation directory, which has the same information, although possibly less up to date.

Click the **Set** button next to **Ratio of computed measure to archive** to select this format for displaying the comparisons.

Click **Close**. This updates the display of computed measures as shown by the following figure. 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.

Context					
Name	Result7	Scope	System	Metric	Public_functions_full_contra
				Unit	Feature
Metric: Public_functions_full_contracts		Scope: System: simple		Value: 15804	
Name	Scop...	Sco...	Metric	Result	Comparison ...
✓ Class_count	System	simple	Classes	183	12.67%
✓ Direct_dependencies	System	simple	Direct_dependents	862	-

The option that we saw in the **Compare to archive** dialog enables you to define an existing archive as reference for your current and future measurements.

The other two self-explanatory options, shown in the picture of the dialog on the previous page, enable you to **create** your own archive, and to update an existing one. You can then make this archive available, in a local file or on the Web, for other projects to perform comparisons.

An archive, stored as an XML file, include both the project metrics and its current measures. The metrics will be made available for import to projects selecting the archive for comparison.

## 15 GRAPHICS-BASED DESIGN

So far the project modifications that we have made used the text editor in the Editing pane. We used graphics, but as a way to reflect system structures, not to build them ([“A peek at diagrams”, page 21](#), and subsequent discussions).

In line with the principles of seamlessness and reversibility recalled at the beginning of this Tour, EiffelStudio’s text-diagram interaction is completely both ways. When you make a textual modification, the diagram will be updated after the next incremental recompilation; but you can also work directly from the diagram, and the text will be generated or updated after each graphical operations.

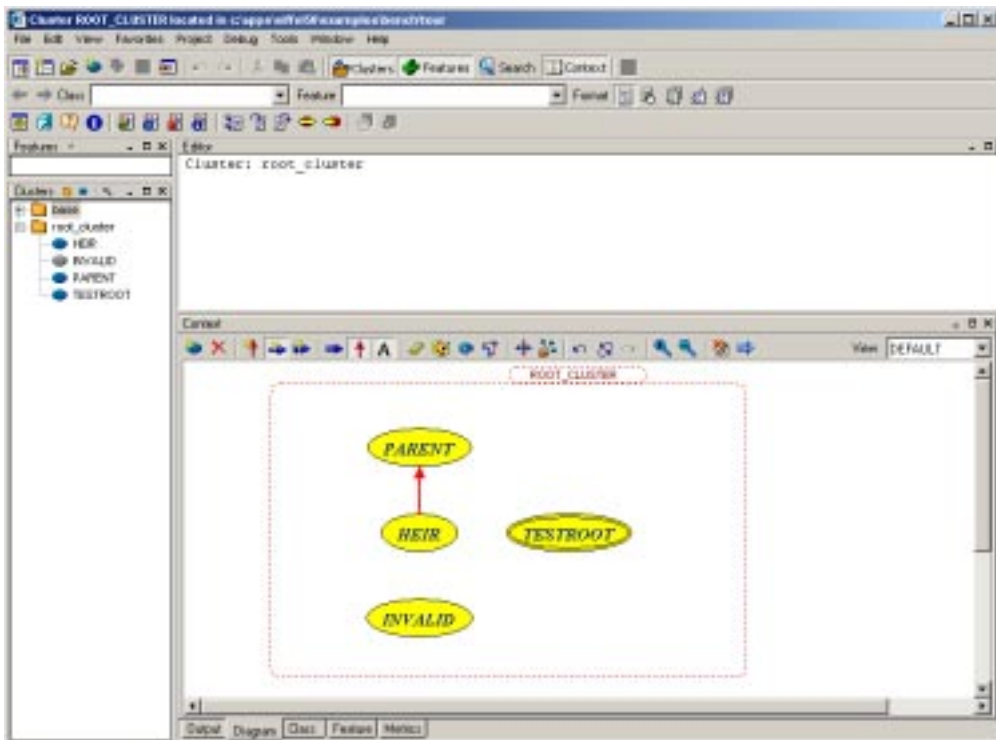
Many people like to use the graphical mechanisms at the beginning of a project, to draft the overall structure of a system in “bubbles-and-arrows” style, then concentrate on text as they get closer to implementation. But there is really no such obligation. At any point in the development, just use the form that is more suited to your taste and to your needs of the moment.



## Displaying a cluster view

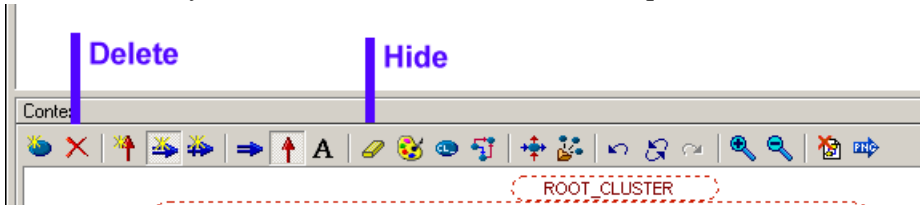
We are going to play with the root cluster. Make sure the Cluster tree and the Context pane are up. Also make sure that the Context tool is in Isolate rather than Merge behavior ([“Isolating the context”, page 68](#)); you can see this by going to the **Edit menu**: if there is an entry **Isolate context tool**, select it; otherwise (the corresponding entry reads “Merge context tool) don’t do anything, you’re fine. (You can also use the Merge/Isolate button added earlier to the Browsing toolbar for this very purpose.)

The earlier diagrams were “class views”, giving a picture of the world around a class. For a change, we are going to work for the moment with **cluster views**, showing the content of a cluster. Select the Diagram tab in the Context pane; from the Cluster Tree, pick-and-drop *root\_cluster* to the Context pane. This displays the graphical view of the root cluster in the Context pane:

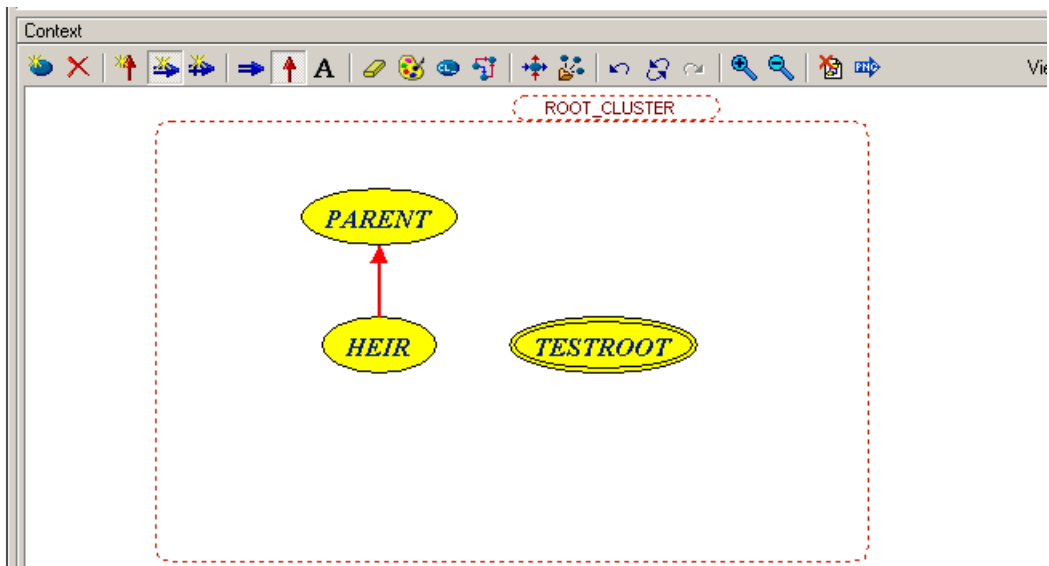


## Hiding a class

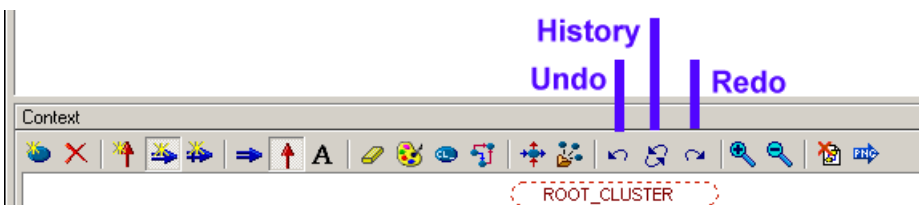
First we decide that we don't want to be bothered with class *INVALID*. We could delete it altogether from the system by pick-and-drop its bubble to the Delete hole. This is not what we want, but try this now to see the confirmation request:



Make sure to answer **No** to that confirmation request (you want to keep the class even though it wouldn't be a catastrophe to lose it) and instead pick-and-drop the *INVALID* bubble into the **Hide** hole. This time there is no confirmation request, since the operation is reversible — it just affects what's displayed in the cluster view — and the class is removed from the display:



You can try undoing this change, then redoing it:



You can also click **History** which, during the rest of the session, will display the list of executed operations, and let you undo or redo many operations at once by clicking the oldest to be kept or the youngest to be redone.

For the rest of this discussion we assume *INVALID* is hidden.

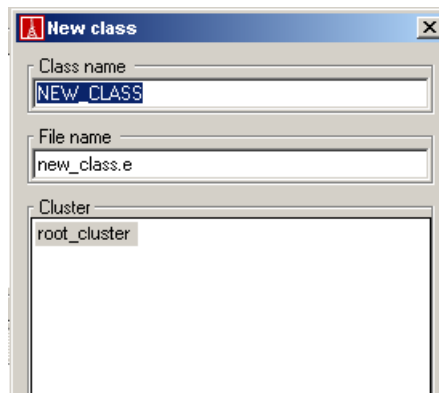
### *Adding a class*

We are now going to add a class graphically to our system. This means you don't have to worry about creating and initializing a file; EiffelStudio will take care of the details.

The useful button here is **New class**:

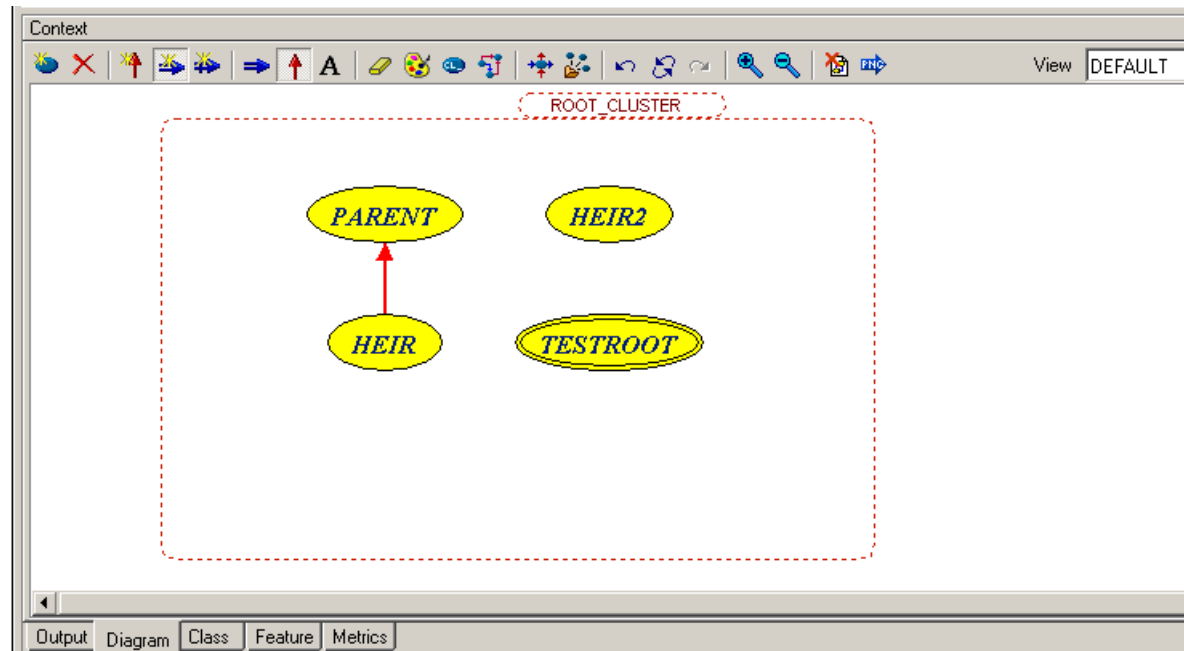


This button is a pebble, meant to be dropped into the diagram. Drop it somewhere above the bubble for *TESTROOT*; the exact place doesn't matter, but it has to be within the aread of the cluster *root\_cluster* because we'll want our class to part of it. You're asked to name the class:

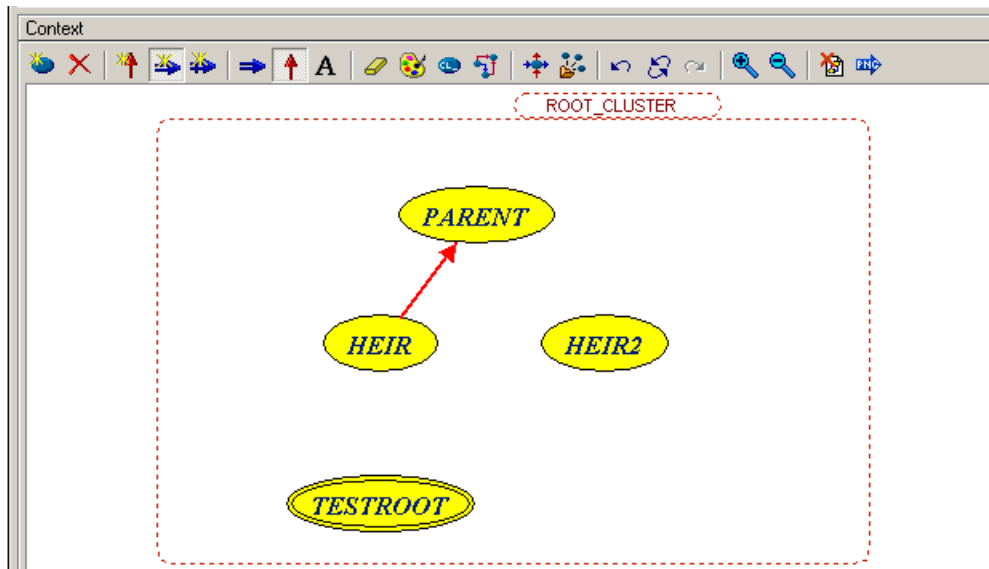


Overwrite the default name being proposed, *NEW\_CLASS*, by the name *HEIR2*, as we are going to create a new heir of *PARENT*. Don't touch the file name in the second field; as you type the class name EiffelStudio automatically sets the file name to *heir2.e*, so you would only set it if you wanted to override the default convention for names of class files.

The new class is now in the diagram, part of *root\_cluster*:

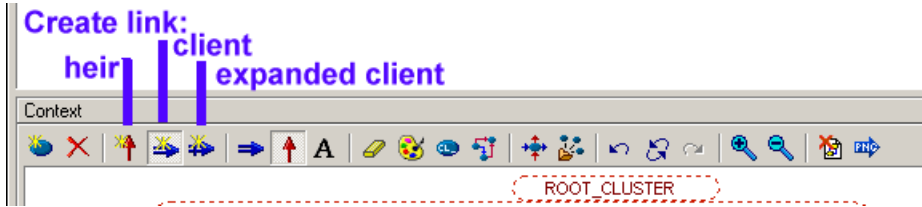


Using conventional drag-and-drop (not pick-and-drop), move the class bubbles for *HEIR2*, *TESTROOT* and *PARENT* so that the display looks approximately like the following. Note that the double circle around *TESTROOT* is the BON convention to identify a system's root class.

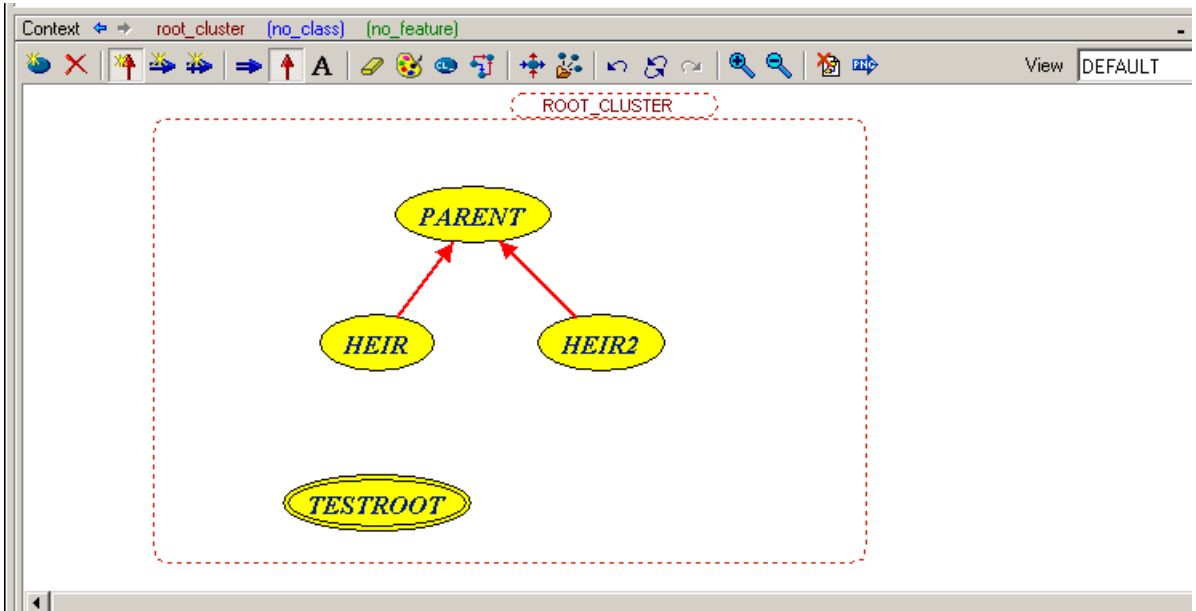


### *Adding an inheritance link*

Now we are going to make *HEIR2* an heir of *PARENT*. To create inter-class relations, you will select a relation by clicking one of the “Create link” buttons, then use pick-and-drop from the source class to the target class. There are three possibilities:



Click the button marked *heir* above. Now pick-and-drop from the *HEIR2* bubble to the *PARENT* bubble. (Now you see why conventional drag-and-drop is used to move bubbles: pick-and-drop on the diagram serves to add links between classes).



To convince yourself that the new class has been made like an heir of *PARENT*, in its text and not just in the diagram, pick-and-drop *HEIR2* bubble to the Editing pane at the top to see its text. (You could also control-right-click to the bubble to bring up a new Development Tool on this class.) You see the result of the initial creation operation, which produced a class template with all the standard style and clauses, and the latest parenting operation, which made *HEIR2* inherit from *PARENT*:

```

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

class
  HEIR2

  -- Replace ANY below by the name of parent class if any lacking more parents
  -- if necessary; otherwise you can remove inheritance clause altogether.
  inherit
    ANY
    runtime
    export
    undefine
    redefine
    select
    end

  -- The following Creation clause can be removed if you need no other
  -- procedure than "default_create":

  PARENT

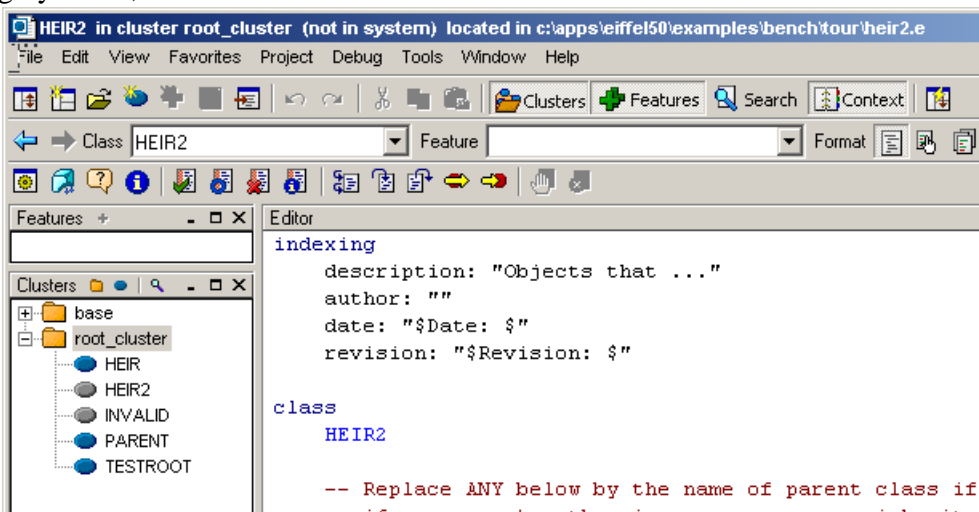
  create
    default_create

  feature -- Initialization
  feature -- Access
  feature -- Measurement
  feature -- Status export
  feature -- Status setting
  feature -- Cursor movement

```

In a moment we'll use this Editing pane to see how, conversely, EiffelStudio will automatically reflect in the diagram a change made to the text. For the moment go back to first the Development Tool.

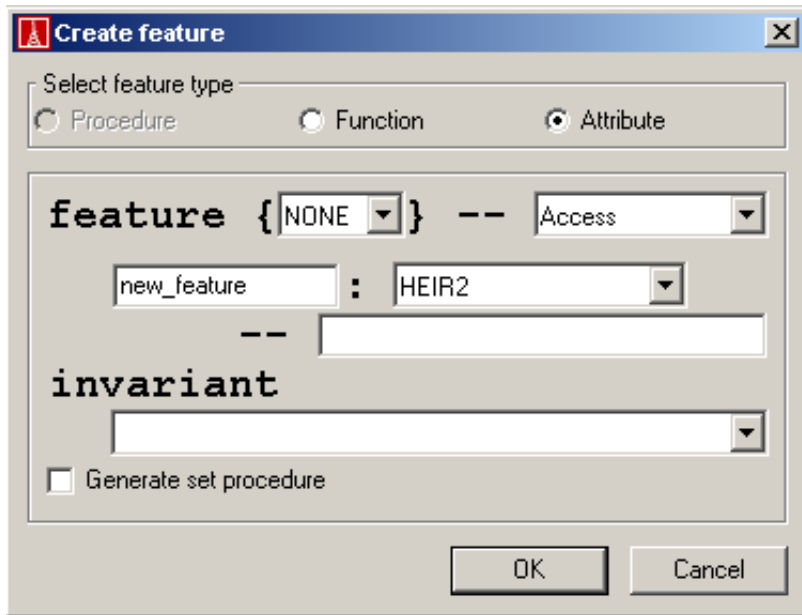
In the Cluster Tree in the top-left pane, you will notice that the name of *HEIR2*, a appears grayed out; so does the name of *INVALID*:



This is EiffelStudio's way of telling us that the two classes are not part of the system, because although they belong to one of its clusters they are not referenced, directly or indirectly, from the root.

### *Adding a client link*

Let's indeed make **TESTROOT** a client of **HEIR2**. Click the button that selects **Client** as the next relation to make links (the link creation buttons were shown on page 117). Pick-and-drop from the **TESTROOT** bubble to the **HEIR2** bubble. This asks you what kind of client link you want:

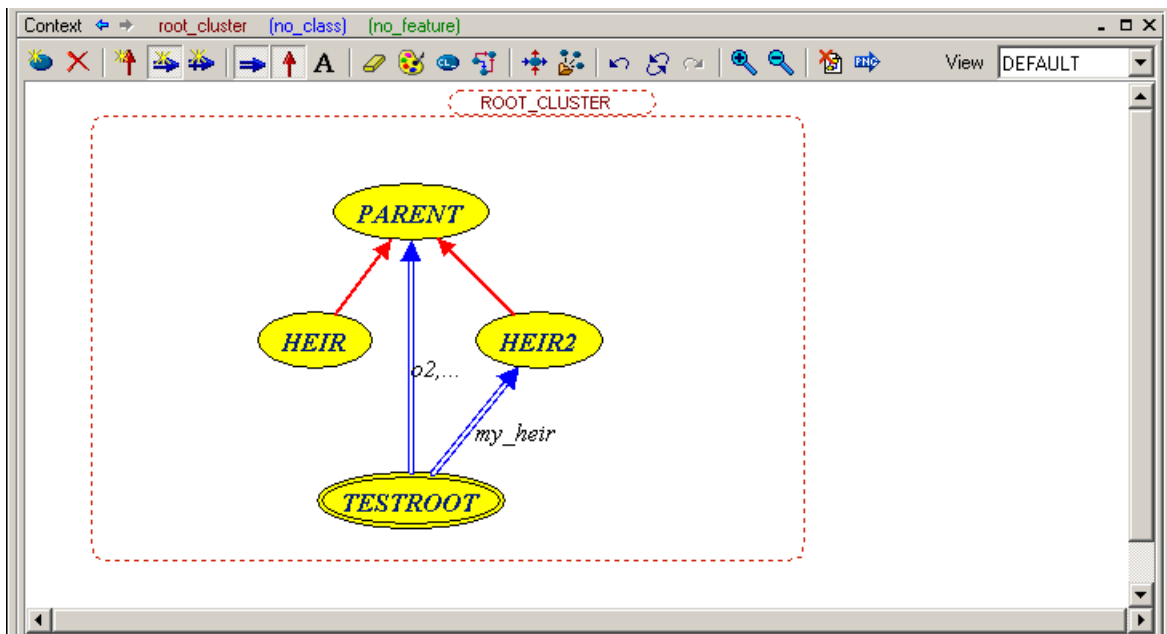


This gives you many options and in fact is a convenient way to build your classes, whether at the analysis, design or implementation level. Here, fill the fields as follows. For the top choice, keep the default, **Attribute**; we'll give class **TESTROOT** an attribute of type **HEIR2**. In the **feature** clause, which will indicate its export status, replace the default choice **NONE** by **ANY**, so that the attribute will be public. For its feature category, keep the choice currently displayed, **Access**. For its name, replace the default, **new\_heir**, by **my\_heir**. In the **invariant** clause, enter

```
my_heir /= Void
```

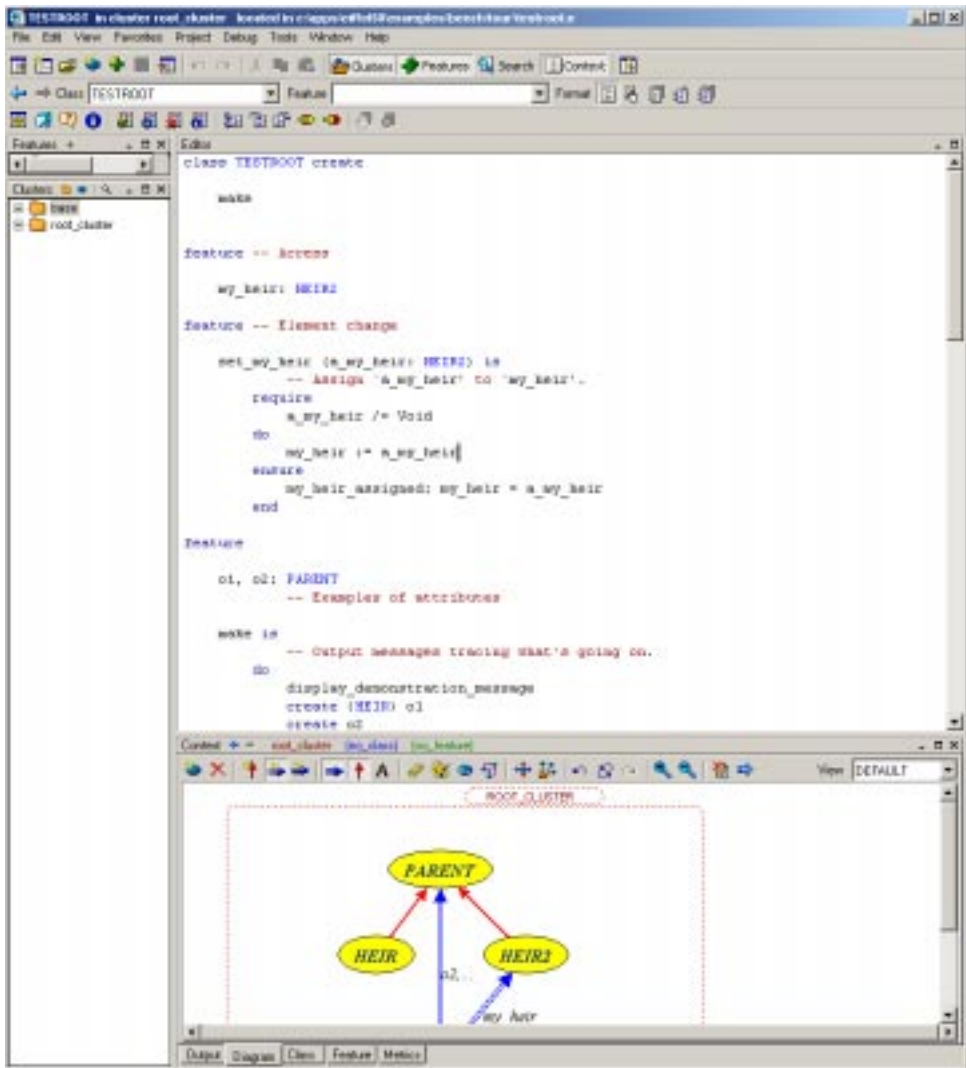
to specify the invariant property that this attribute should never be void. Finally, to see how EiffelStudio can generate the full accompaniment to an attribute, check the box **Generate set procedure**. Click **OK**.

The diagram — shown below after a slight move of the bubble for *TESTROOT* for more readability — shows that *TESTROOT* is now a client of *HEIR2*. By default it only showed inheritance links; now it has switched automatically to a mode that shows client links as well, so that we also see that *TESTROOT* is (and always was) a client of *PARENT* through attributes including *o2*.



Now pick-and-drop the class bubble *TESTROOT* to the top Editing pane to see how the class has been modified. The situation here is different from what we saw earlier with *HEIR2*, which had been generated from scratch by the diagram. Here *TESTROOT* existed before, in text form; so the diagram mechanisms have had to preserve the existing feature and feature clauses, and add the elements corresponding to what you have specified through the diagram mechanisms. Here is the beginning of class *TESTROOT* in its updated form:





Note how EiffelStudio has generated both the attribute and the associated “set” procedure, `set_my_heir`, complete with a precondition — deduced from the invariant you have specified, `my_heir /= Void` — and a postcondition. The unlabeled Feature clause of the existing class has been kept; the new features have been entered into clauses labeled `Access` and `Element change`, observing the Eiffel standard for common feature clauses in libraries.

If you look at the end of the class, you will see an **invariant** clause listing the invariant that you have entered.

## Updating the diagram from the text

In this tour of the diagram facilities we have, so far, worked on the diagram and see the text updated immediately. Of course we want full reversibility. So let's make a change in the text and check the diagram.

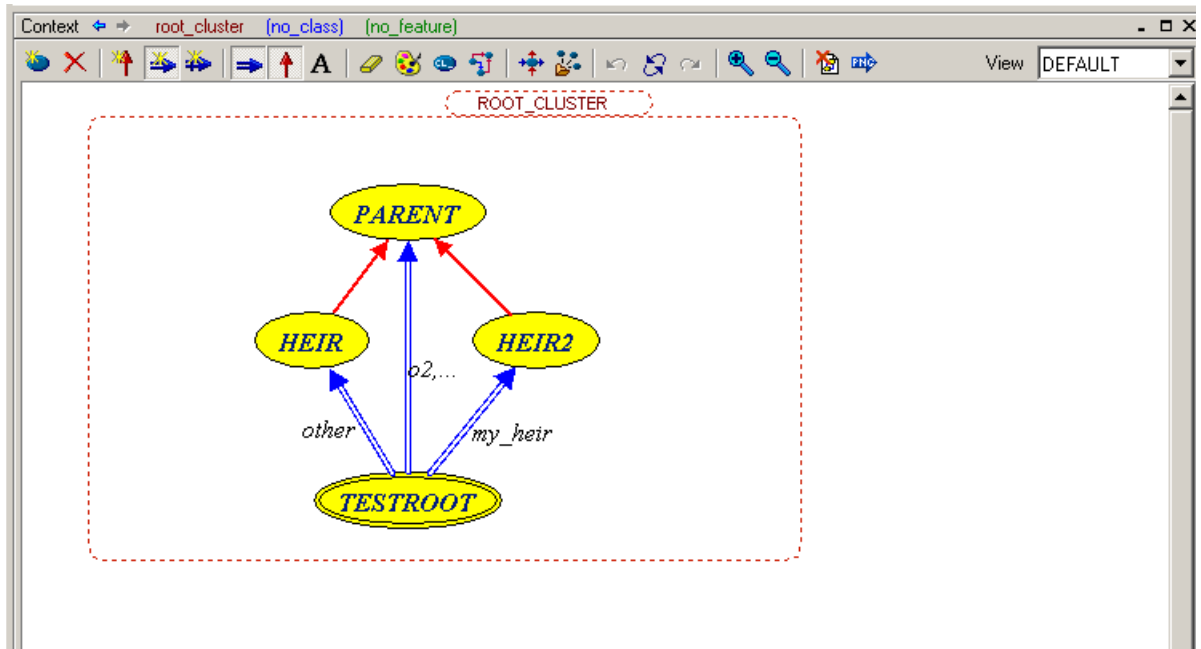
The change will be very simple. We'll make *TESTROOT* a client of *HEIR*. In the top Editing pane, use the editor to add an attribute declaration

```
other: HEIR
```

without further ado, as this is just a simple check.

If you are concerned about the correctness of the class, you may wish to update its creation procedure *make* to add a creation instruction *create my\_heir*. Without it the just added invariant would be violated upon creation.

Nothing happens yet to the diagram. This is normal: EiffelStudio doesn't update the diagram every time you type some text (which, for one thing, might be syntactically incorrect or invalid). You need to recompile first. Click the **Compile** button. Then on the Context pane click **Diagram**; the new relation appears:

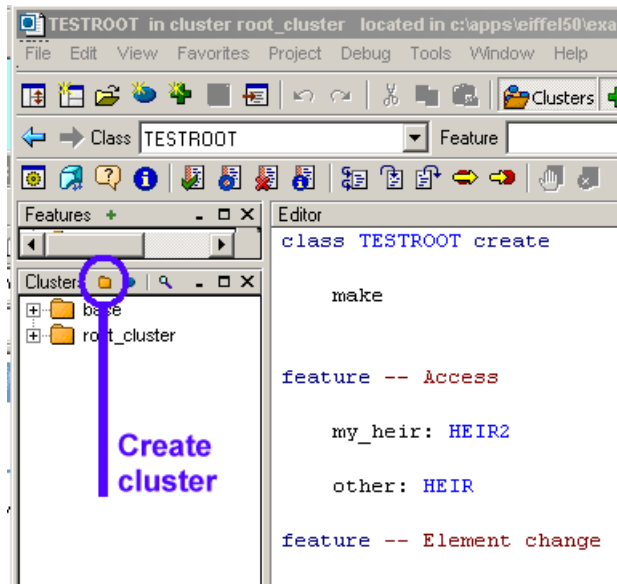


If the label *other* of that relation doesn't appear in the exact place shown here, try moving it using conventional drag-and-drop. You can only move such a link label within a small area on either side of the link.

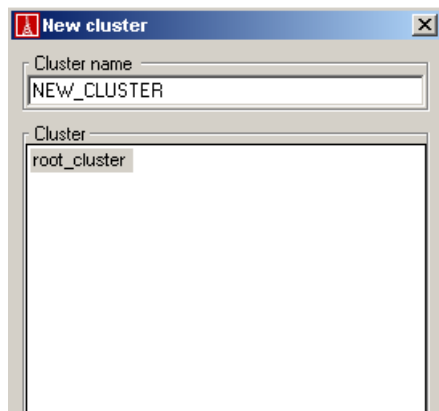
## Creating a cluster

Earlier on, we saw how to create a class from the EiffelStudio diagram, letting EiffelStudio take care of creating and initializing the file. Similarly, you can create a new cluster graphically, and let EiffelStudio create the corresponding directory.

To create a cluster, you can go through **Project → Project settings**, or you can click the little Cluster Creation button at the top of the Cluster Tree:



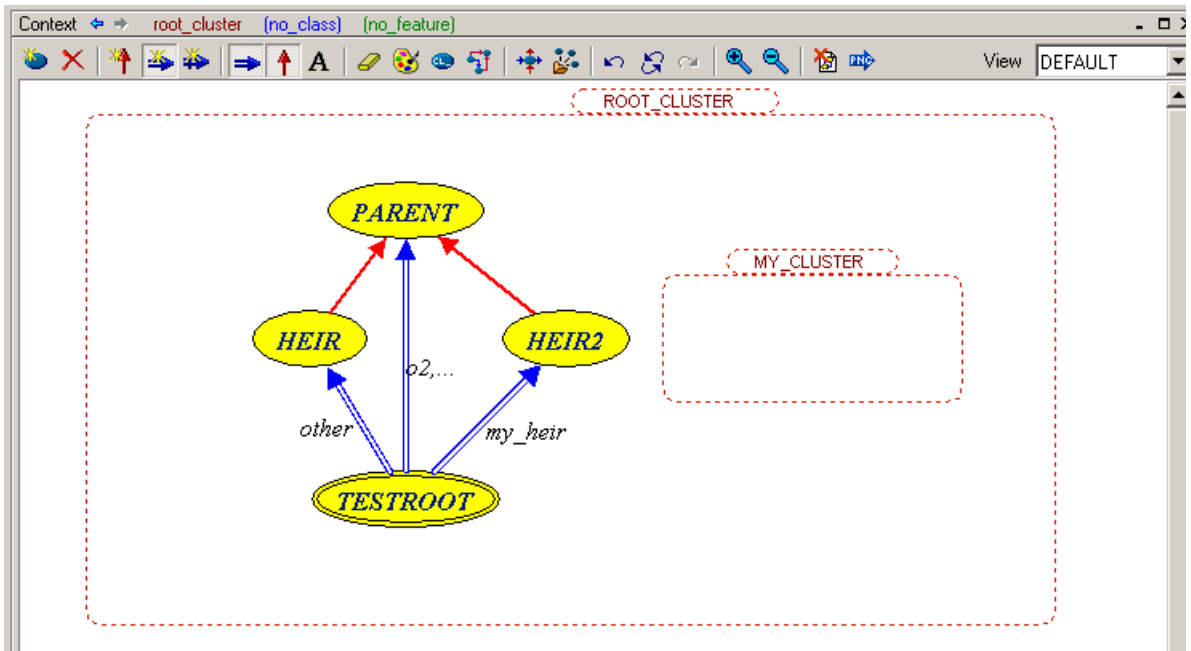
Click this button. The resulting dialog asks you for the cluster name, and the existing cluster (non-compiled) of which you want to make it a subcluster, here leaving only one choice:



Instead of the *NEW\_CLUSTER* default name, type *my\_cluster*; select the only possible supercluster, *root\_cluster*, and click **Create** at the bottom of the dialog.

As you will have noted, this technique only allows you to create a new cluster as a subcluster of an existing one. You can create a top-level cluster by going through **Project** → **Project settings**

Recompile the project and bring up the cluster diagram again. It shows the subcluster:

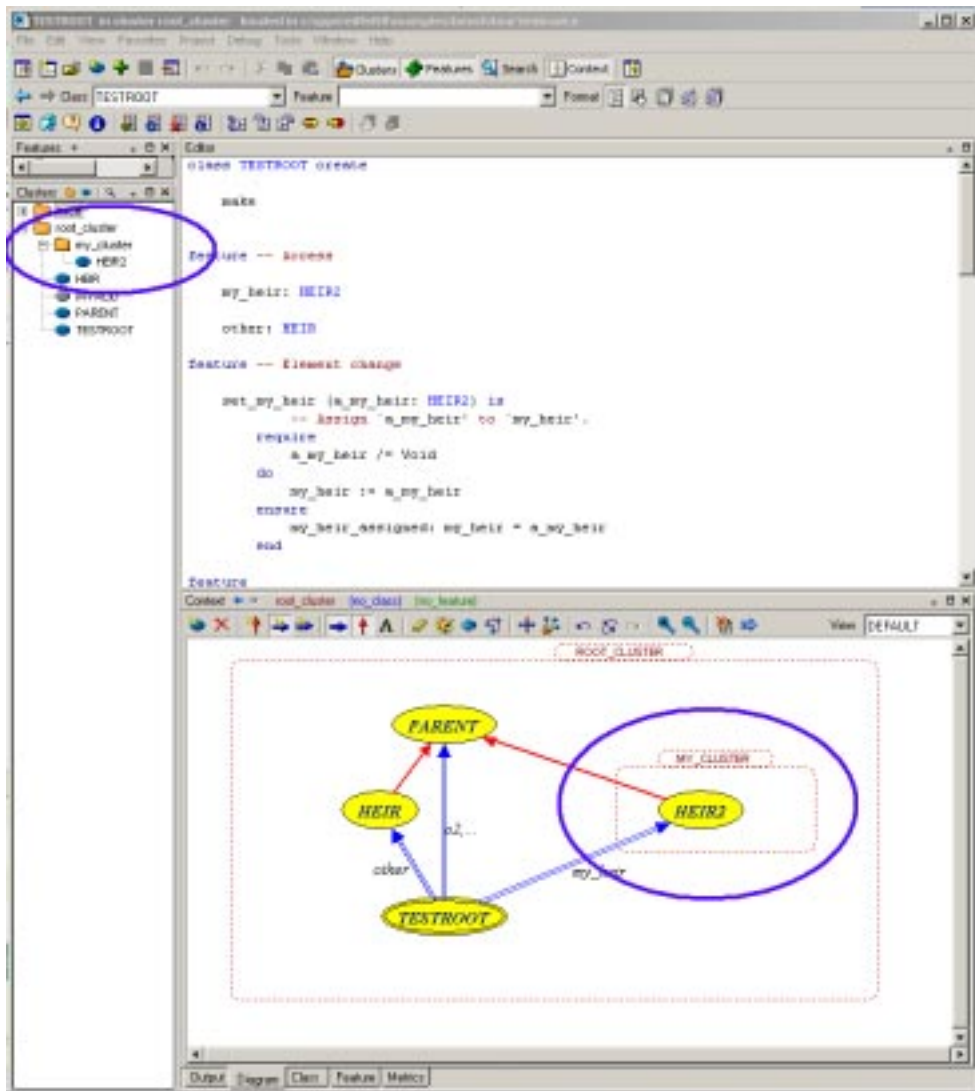


Make sure that the display looks approximately like the above; you may have to resize either or both clusters (drag a corner), and move the small cluster (drag-and-drop).

### *Moving a class to a different cluster*

Among the many operations you can do graphically is to move a class from one cluster to another. Drag-and-drop (again, using conventional drag and drop) the *HEIR2* class bubble to the rounded rectangle for *MY\_CLUSTER*. Make sure the bubble fits entirely (that's why we wanted the cluster rectangle to be big enough).

This graphical manipulation has caused a structural change: class *HEIR2* is now part of *MY\_CLUSTER*. Check this by expanding the Cluster Tree on the left:



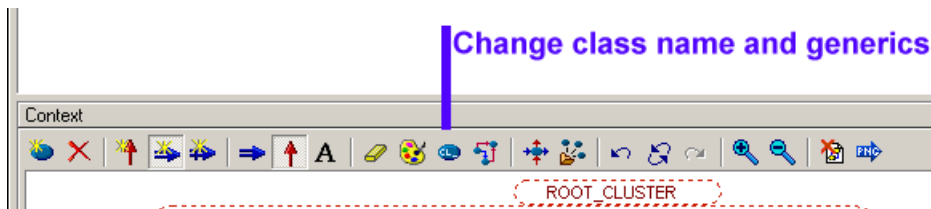
If you like, you can also look into the project directory — using the Windows Explorer, or **cd** and **ls** on Unix/Linux — and check that it now has a subdirectory *my\_cluster* with a file *heir2.e* containing the text of class *HEIR2*.

Clearly, it's much more convenient to use EiffelStudio for such manipulations than to move files around manually.

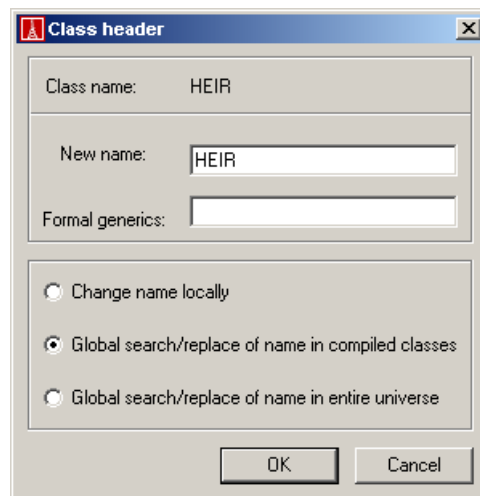
## Changing a class name

Here is another operation that would be even more tedious if you had to perform it manually: changing a class name. You must make sure that *every* reference to the class in the system is updated; but that's difficult to do with a text editor since — assuming we are changing the name of *HEIR* to *HEIR1* — you must check references one by one to avoid, for example, changing an occurrence of the word in a string.

Instead, find the icon whose tooltip reads “Change class name and generics”:

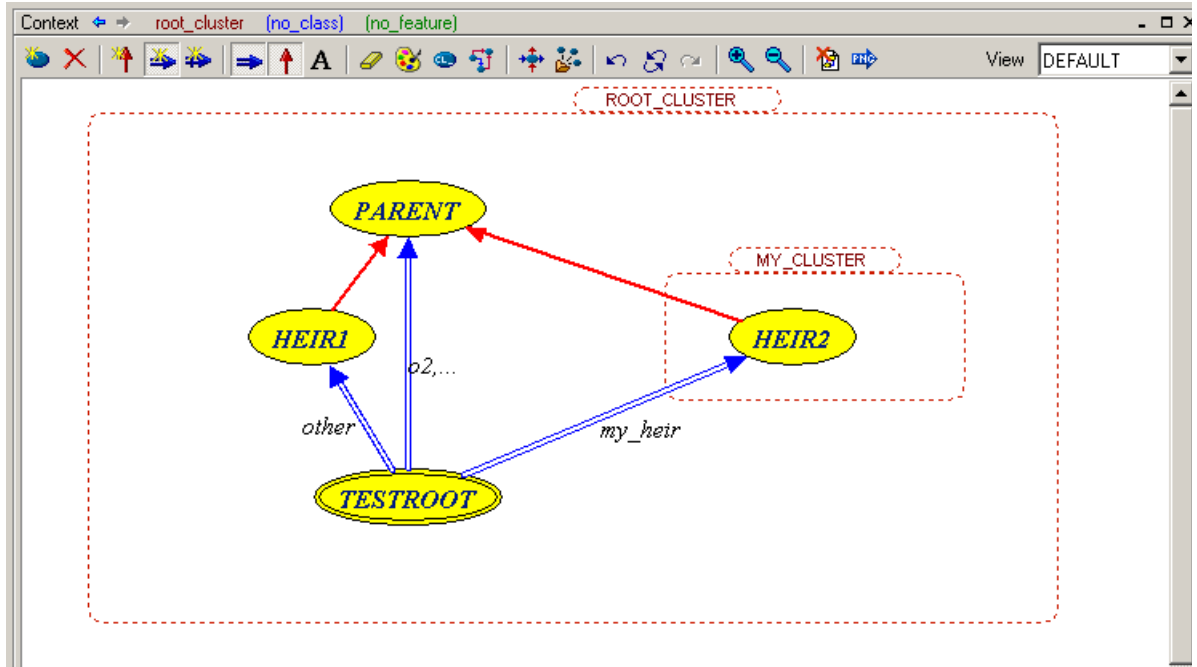


This is not a button but a hole (as you'll be reminded if you try just clicking it). Pick-and-drop *HEIR* to it:



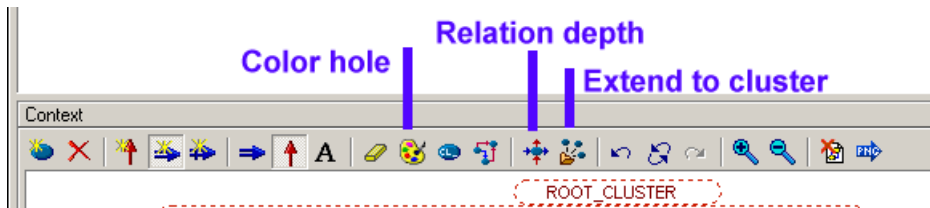
As you can see, this dialog also serves to change the name of formal generic parameters when the selected class is generic. Type *HEIR1* (or *heir1*, EiffelStudio always converts class names to upper case) under **New name** and click **OK**.

As EiffelStudio traverses the system to update all references to *HEIR*, a progress bar appears so that you know what's going on. Everything is updated, including the current cluster diagram, which displays the new name in the class bubble:



### Adjusting the display

A number of buttons enable you to customize the display:



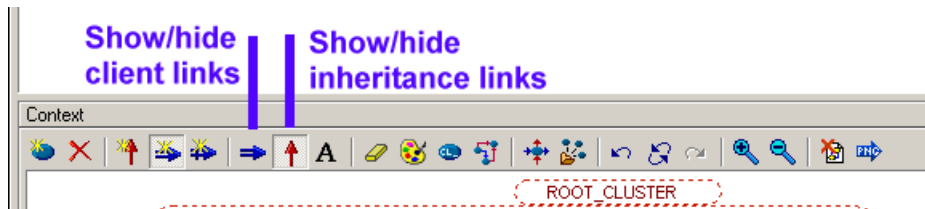
So far all class bubbles had the same default color (yellow). Try pick-and-dropping a bubble into the **Color hole** to get a color palette that enables you to select a different color. This is useful if you want to highlight classes possessing certain properties, for example classes that are part of a certain Design Pattern.

**Relation depth** enables you to select the depth at which inter-class relations will be displayed. (Don't change this setting now.) **Extend to cluster** is more useful for class diagrams than the cluster diagram we have now, which by default included all classes of the cluster; if you click it now it will add the class *INVALID* that you removed earlier. There is no need to do this now.

## Views

So far the top-right **View** field has always shown **DEFAULT**. You can define any number of views in your project, and apply them to various class and cluster diagrams.

For example, using the buttons to show and hide links of various kinds



you can produce diagrams that only show the inheritance links, and others that only show the client links. If you want to keep both, simply define views by typing view names — such as *Inheritance*, *Client*, *All\_links* — into the **View** field.

You can also use views to retain some of the choices seen just before, such as different colors and depths.

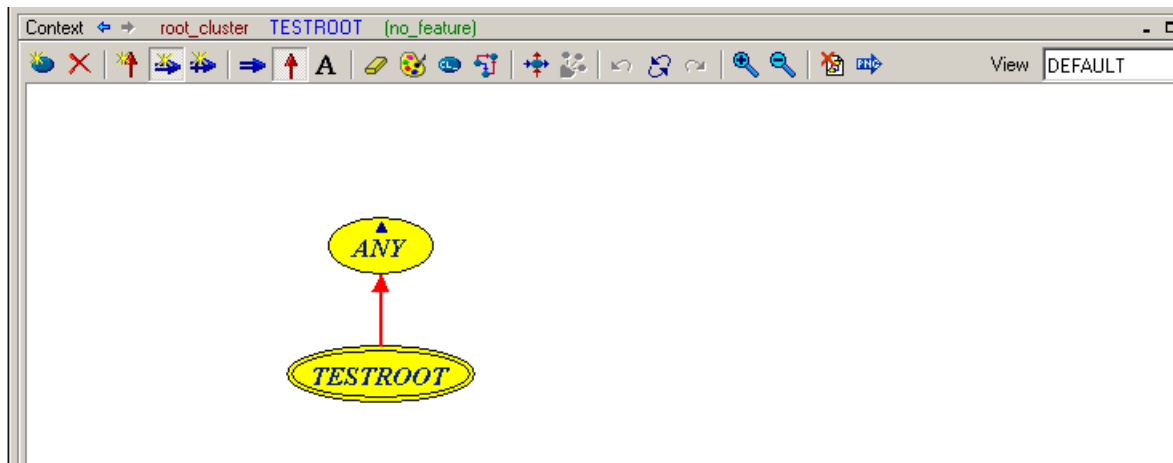
To load a previously defined view, just use the menu associated with the **View** field.

You may remember that when we generated HTML documentation, the dialog (page 46) asked you to select a view among the available ones. You can choose a different view for each cluster.

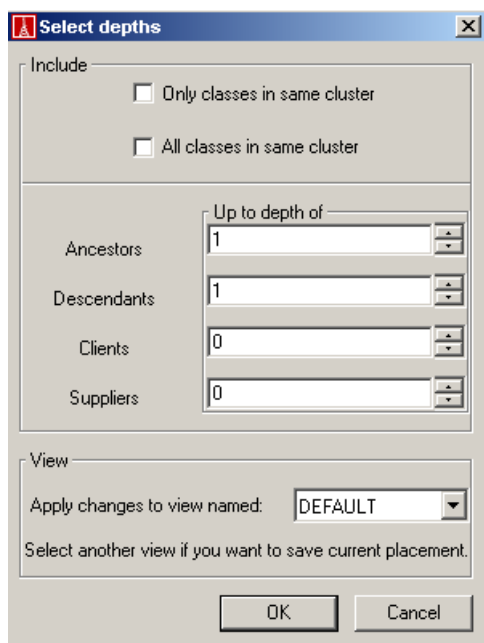
## Class diagrams, cluster diagrams

Whereas our initial encounter with diagrams at the beginning of this Tour used class diagrams, in the present discussion we have used cluster diagrams. Both are interesting. To obtain a class diagram, target a Context pane to a class, and select the **Diagram** tab. By default, this shows the parents of the class. Do this for *TESTROOT*:





It's for class diagrams that the **Relation depth** button is most interesting. It will let you select the exact depth that you wish displayed for every relation:



This will conclude our review of the Diagram facilities of EiffelStudio, although you'll surely discover some further riches by yourself and through the contextual documentation. We hope the complete seamlessness between text and pictures will enable you to increase the effectiveness of your analysis work, or your design work, or your programming — whatever level of system development you need to tackle.

## 16 HOW EIFFELSTUDIO COMPILES

So far we have relied on the compiling capabilities of EiffelStudio without exploring them in any detail. We must now understand the principles behind ISE Eiffel's compiling strategy, in particular how it reconciles fast turnaround, efficient generated code, and strong typing.

### *Compilation is automatic*

Any speed issue aside, the most important property of the compilation process is that it is entirely automatic.

You've seen it from the beginning of this Tour: all the information the compiler has — obtained from an Ace file, as here, or generated automatically by the other options — is the name of the root class and the list of directories holding Eiffel clusters. In fact it only needs these directories for non-precompiled clusters, so here since we are using precompiled EiffelBase the only needed directory is the one containing the root cluster, denoted simply as “.” (current directory) since, to simplify things, we've started EiffelStudio from the Tour's own root cluster directory.

The compiler takes care of the rest, in particular of finding all the classes that must be compiled.

There is never any need, when compiling ISE Eiffel systems, to supply “Make files”, “include files”, or other manual descriptions of inter-module dependencies.

### *Compilation modes*

EiffelStudio offers several forms of compilation, which you can see in the entries of the **Compile** menu (don't trigger any of them right now) as well as keyboard shortcuts and, in some cases, buttons:

- **Melt**: quick incremental recompilation, doesn't optimize code for changed parts.
- **Freeze**: incremental recompilation, not as fast as Melt, but generates more efficient code for changed parts.
- **Finalize**: recompile entire system, generating highly optimized code.
- **Precompile** (available both in the **Project** menu and through **Tools → Precompilation wizard**), to process an entire library, on which many systems can then rely without having to compile it.

You'll quickly learn to use each of these modes to suit your needs.

## Criteria

EiffelStudio's **Melting Ice Technology** reconciles the following goals:

- *Security and efficiency of the generated code:* compiling techniques for the strongly typed Eiffel language ensure that compilers can catch many errors before it is too late, and generate more efficient code. The “validity constraints” of the language, whose violations are caught as compilation errors, are particularly useful here, playing the role of checkable design rules.
- *Quick turnaround:* you should experience an almost immediate transition from the time you write or (more commonly) modify software to the time when you can execute the result of what you just wrote.
- *C code generation:* for portability, it is useful to take advantage of C in its proper role, that of a portable assembly language. C's closeness to machine concepts — one of the very properties making it less suitable for human programming except in the case of short routines to access low-level mechanisms —, its almost universal availability, and good level of standardization, make it an excellent target language for a code generator. This also enables the environment to benefit from the often extensive optimizations performed by good C compilers, and facilitates interfacing new software with the large body of existing C-based systems, tools and libraries. As the final output of Eiffel compilation, you can obtain a complete C package that you can either C-compile on the same machine or port to other platforms, making ISE Eiffel a tool of choice for **cross-platform development**: develop on one platform, deploy on one or more others.

## The Melting Ice Principle

The idea of the melting ice is based on the observation that, for the practicing software developer, the crucial day-to-day compilation problem is not how to process an entire system but how best to process a **changed system**, of which an earlier state had previously been processed.

The change may be big or small; the system may be big or small. (“Small system” here means up to a few tens of thousands of lines.) This gives four possible cases, of which only one is really critical:

	Small system	Large system
Small change		***
Big change		

If the system is small, as in both of the left column entries, speed of recompilation with a good compiler will be acceptable.

In the bottom-right box, the developers have spent days or weeks changing many classes in a large system, so they will not resent having to wait a little to see the results of the recompilation, as long as the time remains reasonable. In EiffelStudio this corresponds, as we’ll see shortly, to *finalization*, which is in fact quite fast, although not as fast as the incremental modes.

In the day-to-day, minute-by-minute practice of building and modifying software, the case that recurs by far the most often — and can cause most frustration — is the one marked \*\*\*: you change only a small part of a big system. Then the result should come quickly enough. More precisely:

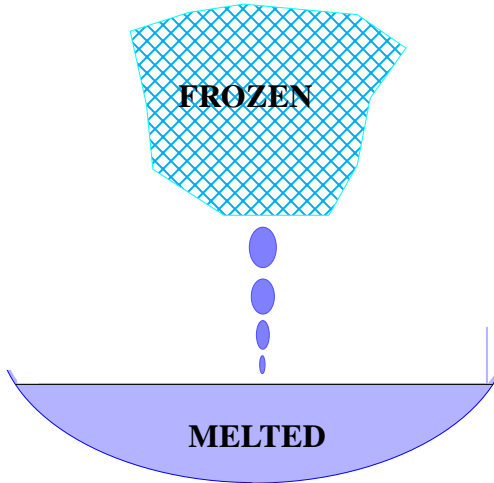
### Melting Ice Principle

The time to re-process a system after a change should be a function of the logical size of the change, not of the size of the system.

The “logical size” of a change may be different from its physical size because a small physical change in a class may have consequences in many others. Imagine for example that you add a feature to class *ANY*, although this is an extreme case and won’t normally happen. Since every class is a descendant of *ANY*, the logical change may affect the entire system.

In practice, however, most small physical changes will also be small logical changes and cause only minimal recompilation. In particular, EiffelStudio will detect that a change does not affect the interface of a class — for example if it’s only a change to non-exported features — so that it doesn’t need to re-process its clients.

Processing such incremental changes, in time proportional to the logical size of the changes, is known in ISE Eiffel as **melting**. The reason for this terminology is the metaphor illustrated on the following figure. Think of a compiled system as a block of ice; it may have taken some time to “freeze” — compile. Now you start working on it again; the changes are like melted drops of water, dripping from the ice as a result of the heat you produce.



The Melting Ice Technology ensures that incremental compilation will only process the “melted” part, usually small, leaving alone the “frozen” part, which may be large. This is crucial to the incrementality of the mechanism.

The roles of the four compilation modes follow from this analysis:

- **Melting** is the fastest mode: it processes the melted part without affecting the frozen part. With EiffelStudio, the melted elements will be *interpreted* while the rest is compiled.
- **Freezing** is the process of putting back the melted parts into the “freezer”: bringing them to the same compiled state as the parts that have not been modified.
- **Finalizing** is the non-incremental process of producing a stand-alone C package and the resulting executable, extensively optimized, from the current system.
- **Precompiling** is the process of compiling an entire set of reusable classes, once and for all, so that it can be shared by many systems and many users without duplicating the code or compiling it again for each project.

### *Properties of the compilation modes*

The following table summarizes the differences between the four compilation modes:

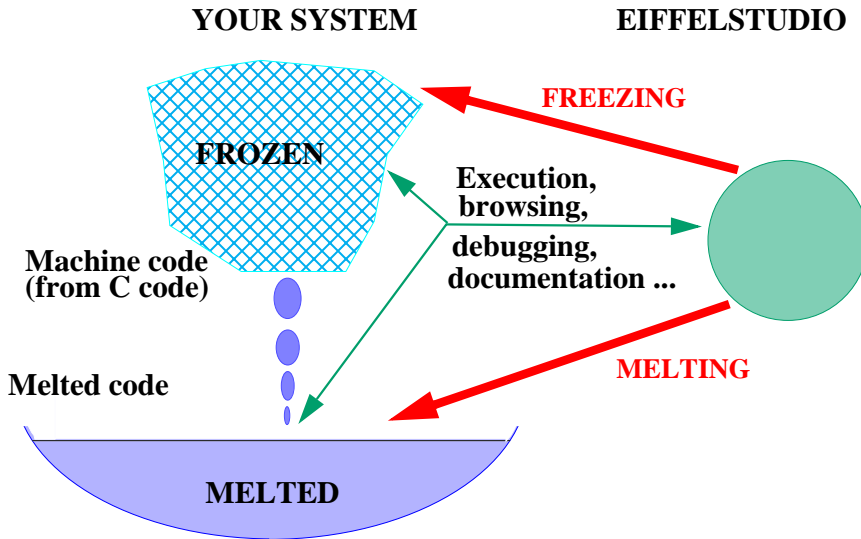
	<b>Regenerate C code?</b>	<b>Incremental?</b>	<b>Compilation result shared between projects?</b>
<b>Melt</b>	No	Yes (fast)	No
<b>Freeze</b>	Yes	Yes (but requires C compilation of changes & linking)	No
<b>Finalize</b>	Yes	No	No
<b>Precompile</b>	Yes	No	Yes

Most of the time, during the production and modification of your software, you will alternate between melting and freezing, since both of these modes are incremental. Most of the time, you will simply **melt**, since melting satisfies the Melting Ice Principle: the time to get back to a working system is very short — proportional to the size of the changes. Note in particular that the unit of melting is the smallest possible one: each feature of a class — attribute or routine — may be melted separately.

The main difference between melting and **freezing** is that freezing implies re-generating C code for the changed elements, and hence relinking the system as well. In contrast, when you melt changes, you do not change any C code: it remains frozen.

As a consequence, melting can only process changes to Eiffel code. If you add new external code (in C, C++ or other languages whose modules will require linking), you must freeze. This is also true if you add new agents. If you ask for a Melt in such cases, the operation will trigger a freeze anyway. More generally, the **Compile** button that you have used a number of times to recompile the system in this Tour triggers a Melt by default, and a Freeze when it has to.

EiffelStudio knows how to hide the differences and present you with a uniform view of the frozen parts (the C code) and the melted parts. Here indeed is the full view of the picture that was previously given in part:



When you examine a component of the system — to edit a class, produce a view such as Contract or Interface, enable a breakpoint on a routine, run the system, inspect a run-time object — EiffelStudio automatically knows where to look for the corresponding information: melted or frozen part. If one of your actions requires melting or freezing more elements, EiffelStudio will also handle this automatically.

As suggested by the lower red arrow, successive melting operations “pour water into the bowl”, corresponding to the elements that you have changed since the last freeze. Freezing, represented by the top red arrow, updates the C code so that it integrates all the latest changes, emptying the bowl in the process.

Because the difference between melted and frozen code is largely invisible to users of the environment, the term **workbench code** will cover both kinds; workbench code is code resulting from a succession of freezing and melting operations. As long as you are working within EiffelStudio, you are using workbench code.

When you are happy with the results of your development, you will normally finalize the system, thereby generating **final code**. Although not strictly required, this step is in most cases appropriate since final code is significantly more efficient than workbench code in both time and space: finalization performs a number of optimizations — dead code removal, replacement of dynamic by static binding — that make little sense in incremental development where, for example, some code element that is “dead” one minute may be resurrected the next moment through the addition of just one line of text. In addition, because finalized code is more efficient than frozen code, it is the natural choice if, using ISE Eiffel for cross-platform development, you wish to port the resulting C-package to other architectures.

If you have a set of reusable classes that may be useful to many applications, you can **precompile them** into a library. This set of classes must be *self-contained* in the sense that all the classes needed by any of them must be either in the library itself or in another library that you will include in the precompilation.

## Bytecode

The result of melting operations — the contents of the “bowl” — is an internal software representation known as melted code or (for no particularly good reason) as *bytecode*. ISE Eiffel bytecode serves two complementary purposes:

- It can be executed directly. This is what happens during melting: while the rest of your system, the frozen part, is executed directly in the form produced by Eiffel compilation generation and C-compilation of the result, the melted part is interpreted “as is” without further translation.
- It can be compiled into C for further processing.

Internally, the melted code is in a file **base.melted** (where **base** refers to the precompilation library used by our example project) in the subdirectory **EIFGENW\_CODE** of the project directory. The file is not human-readable, but as you add elements to your software and melt you watch its size grow. Whenever you freeze, it’s emptied.

On the Microsoft .NET implementation of EiffelStudio, bytecode is replaced by that platform’s own internal code, MSIL.



## *Using EiffelStudio without a C compiler*

Thanks to melting, it is possible to do Eiffel development without a C compiler. You will simply melt the Eiffel elements that you write. This is known as *melt-only* mode and assumes that:

- You do not add new external C functions, which would require re-freezing and linking.
- You rely on precompiled libraries (such as EiffelBase). The precompilation process itself requires a C compiler, but you can obtain precompiled libraries as part of the ISE Eiffel delivery or download them from [eiffel.com](http://eiffel.com).

## *Degrees*

You can now see the reason behind the terminology used to describe compilation steps, called **degrees** on the messages that flash on the screen when you do a compilation. The names are inspired by the international temperature scale — Celsius, also known as centigrade — where water freezes at 0 (and boils at 100, but Eiffel software never reaches that). For EiffelStudio:

- Compilation starts at degree 6, which examines the clusters of your system to determine what classes may have changed. In many cases the compilation can safely skip part of this degree.
- Degree 5 parses modified classes. It's executed not only when you explicitly request a compilation, but also when you save a class from the EiffelStudio editor, or exit from an external editor, so that you can see and fix syntax errors without delay.
- Degrees 4 down to 1 take care of melting.
- Negative degrees only take place when you freeze or finalize.
- After negative degrees comes C-compilation if needed.

## *Using melting and freezing*

When should you melt, freeze, finalize or precompile? The answers are simple and follow directly from the preceding overview; they provide the key to getting the environment to work for you in the most effective way possible.

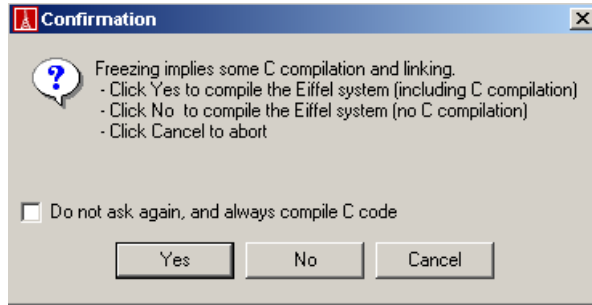
Melting is the bread and butter of the Eiffel developer. As you build your software, either from scratch or by modifying an existing system, you will regularly melt to benefit from the various checks that compilation performs and, of course, to generate executable code that you can test and debug immediately. During this process, there is no need to refreeze, since this operation (although still incremental) takes significantly more time than melting.

Only two operation, noted above, *require* freezing: the addition of external (non-Eiffel) routines, such as C functions or C++ classes, and the addition of agents. The reason is easy to understand: the EiffelStudio compiler knows how to melt Eiffel software, but not software written in C or other languages; agents similarly require special code generation.

For the first compilation of a system that does not use precompiled EiffelBase, a Freeze is needed since class [ANY](#), from which all other classes inherit, uses some external routines. In this case the environment automatically starts a freeze even if you just click Melt. This does not apply if you have access to precompiled EiffelBase.

Except for the addition of external routines or agents, freezing is never strictly necessary. It is indeed possible to use melting throughout a development, never requesting a freeze after the first compilation. But as the melted-to-frozen ratio grows, you may also detect a certain degradation in the performance of the system (determined by how big a share of your system is melted, not how many times you melt it). After a while, then, you may want to refreeze. Like melting, freezing is incremental: only those parts of a system that have been logically changed will be recompiled; as with melting, the determination of what needs to be recompiled is entirely performed by the environment, without any manual intervention on the developer's part. The principal difference is that freezing takes longer than melting.

Because of this you are requested to confirm the first time you freeze. Freeze the example system by choosing the menu entry **Project → Freeze**. You get the following dialog:



Note the **No** option: by default, freezing will start a C compilation, but you can stop after C generation if you wish. This is useful for example if you want to generate a C package for cross-development, C-compiling the result on a different platform.

Click **Yes** to confirm freeze and C-compilation. Once the Eiffel compilation is complete, a message in the Development Tool window (**C compilation launched in background**) tells you when that C-compilation has started. C-compilation does not block EiffelStudio: at this point you can continue working with the environment. Any messages from C compiler will appear:

- On Windows, in a new console that comes up for the occasion (iconify it if you don't want to see the messages).
- On Unix/Linux and VMS, in the window from which you launched EiffelStudio.

You will be able to execute the frozen system as soon as the C compilation finishes.

You will note that freezing, although it takes more time than melting, is actually quite fast, both due to the speed of Eiffel compilation and to the structure of the generated C code, designed to optimize the operation of the C compiler.

## Using finalizing

The main reason for finalizing a system is run-time performance of the generated system. Finalization enables you to generate the high-performance executables that are among the hallmarks of ISE Eiffel. As a consequence, finalized code is the best vehicle for cross-development: you can port the resulting C package to various target platforms and C-compile them on these platforms.

The **optimizations** performed by finalization affect both space and time:

- *Dead code removal* strips the executable module of all the routines in the system that are not actually called, directly or indirectly, by the root's creation procedures. In a large system relying on many general-purpose classes, dead code removal can easily reduce an executable's size by one third or more.
- Finalization also applies *static binding* to non-polymorphic calls and *inlines* some routine calls.

As long as you continue changing, melting and freezing your system the workbench compiling mechanisms cannot perform such optimizations: if a routine is “dead” today you may resurrect it tomorrow by adding a new call to it somewhere; and if a call is non-polymorphic a single additional assignment may require dynamic binding. Compilation can only generate optimal code by working on a full, stable system. This is the task of finalization.

**Cross-development**, the second reason for finalizing, is important if you are taking advantage of the portability of ISE Eiffel to develop your system on a certain platform and then run the result on target computers with possibly different architectures. A target machine may lack an ISE Eiffel compiler (unmistakably signaling its owner's backwardness) but include a C compiler. If the development and target platforms are of different architectures you will need to obtain a copy of the run-time system for the target architecture. The run-time system is also ANSI-C-based, so porting it is usually a straightforward matter.

Note that cross-development does not *require* finalization, since you can cross-compile a frozen version. In practice, however, the finalized version is usually the preferred form for porting a C package because of the performance advantage.

Finalize the example system now by the menu entry **Project → Finalize**. Here too you will be asked to confirm, although the dialog enables you to suppress that confirmation for later attempts, and you may skip C compilation. You will note that finalization is longer than freezing, but still remains quite reasonable thanks to the extensive optimization of the Eiffel compilation process and the structure of the generated C code.

## 17 THE COMMAND-LINE COMPILER

Along with compilation from within EiffelStudio, it is possible to start compilation from a command line (shell). This is useful in particular to recompile your system automatically as part of a script.

To use the command-line compiler — **ec** for Eiffel Compilation — execute

```
ec option ... [class_name] [feature_name]
```

Specify *class\_name* and *feature\_name* only if you wish to produce information about a class or a feature. Otherwise **ec** will compile a system according to each specified *option*. Here is the set of supported options.

OPTION	ARGUMENTS	EFFECT
<b>-ace</b> <i>file_name</i>		Use as Ace the file of name <i>file_name</i> . (Default: file <b>Ace.ace</b> in current directory.)
<b>-ancestors</b>	<i>class_name</i>	Print ancestors of class.
<b>-aversions</b>	<i>class_name</i> , <i>feature_name</i>	Print all versions of feature in ancestors of class.
<b>-callers</b>	<i>class_name</i> , <i>feature_name</i>	Print all routines that call feature.
<b>-clients</b>	<i>class_name</i>	Print clients of class.
<b>-descendants</b>	<i>class_name</i>	Print descendants of class.
<b>-dversions</b>	<i>class_name</i> , <i>feature_name</i>	Print all versions of feature in descendants of class.
<b>-filter</b> <i>filter_name</i>	<i>class_name</i>	Print text of class processed by <i>filter_name</i> .
<b>-finalize</b>		Produce finalized version of system (optimized ANSI C code and executable module).

OPTION	ARGUMENTS	EFFECT
<b>-flat</b>	<i>class_name</i>	Print flat form of class.
<b>-flatshort</b>	<i>class_name</i>	Print interface form of class.
<b>-freeze</b>		Freeze system.
<b>-help</b>		Print short help message listing options of <b>ec</b> .
<b>-implementers</b>	<i>class_name</i> , <i>feature_name</i>	Print all classes that declare or redeclare feature.
<b>-keep</b>		Keep assertions in final mode (useful with <b>-finalize</b> only).
<b>-loop</b>		Enter interactive mode where you may repeatedly request <b>ec</b> operations without having to restart <b>ec</b> .
<b>-precompile</b>		Precompile system, treating it as a library.
<b>-project</b> <i>file.epr</i>		Use the project of which <i>file.epr</i> is the project file.
<b>-project_path</b> <i>directory</i>		Create project directory in <i>directory</i> . (Default: current directory.)
<b>-short</b>	<i>class_name</i>	Print contract form of class.
<b>-stop</b>		Stop on error. (Default: no.)
<b>-suppliers</b>	<i>class_name</i>	Print suppliers of class.
<b>-version</b>		Print EiffelStudio version.

## 18 EXECUTING A SYSTEM WITHIN AND WITHOUT EIFFELSTUDIO

To complete this study of the compilation process let's see a few more properties of how you can *execute* an Eiffel system, both in EiffelStudio and as a compiled system that you deliver to its users, who may need to run it without EiffelStudio.

### *Arguments*

Our example system is very simple and has no need for execution arguments. In more advanced cases you may want to pass values to the execution, such as a numeric parameter or a file name, so that you can have different executions without changing and recompiling the software.

In the Eiffel text, you can access such run-time arguments through the Kernel Library class `ARGUMENTS`. There is another technique — using the arguments to the root creation procedure — but using `ARGUMENTS` is the most general way. Any class of your system can inherit from `ARGUMENTS` and use queries `argument_count` to know the number of arguments passed to the execution, and `argument (i)`, for  $i$  between 1 and `argument_count` to access the  $i$ -th element. Class `ARGUMENTS` has more features; since you have EiffelStudio up, you can check the details if you wish (use the contract form).

To specify execution arguments from within EiffelStudio, enter them, separated by spaces, in the corresponding field in **Project → Debug/settings**.

### *Executing from EiffelStudio*

We have seen how to execute a compiled system from within EiffelStudio: choose one of the appropriate execution buttons, with or without breakpoints.

### *Executing a finalized system outside of EiffelStudio*

A finalized system can be executed on any computer of the appropriate platform; it doesn't need EiffelStudio. The executable version is in the directory

`project_directory/EIFGEN/F_code`

where `project_directory` is the project's directory. The name of the executable file is `system_name.exe`, where `system_name` is the name that you have assigned to your system in the project settings (reflected in the Ace file).

If you run the system from a command line you will give it the appropriate arguments:

```
system_name.exe ... arg ...
```

Because various platforms have different conventions, “relative paths” referenced in your system will mean something different under Unix/Linux, where they start the directory from which the command is launched, and under Windows, where they start from the application directory.

### *Executing a frozen or melted system outside of EiffelStudio*

A system compiled in “Workbench mode” — frozen or melted — is normally meant for execution within EiffelStudio, not for outside delivery, since it is not optimized. If you need to execute it outside of EiffelStudio, make sure that you have access to the **.melted** files in *project\_directory/EIFGEN/W\_code*.

## **19 APPENDIX: WRITING DOCUMENTATION FILTERS WITH EFF, THE EIFFEL FILTER FORMAT**

We saw in the section on documentation ([8, page 39](#)) that you can output documentation about your system in many different formats. A number of predefined formats are available, from Postscript to Microsoft’s Rich Text Format, FrameMaker, T<sub>E</sub>X and others. There’s nothing special about these formats: they just make their conventions known to EiffelStudio through a **filter** expressed in a simple notation called EFF, or Eiffel Filter Format. If you have a favorite format that you’d like EiffelStudio to use for producing documentation, you can define your own filter in EFF. Applications include:

- Producing a variant of an existing format, to support some “house style” that you have defined, such as a different formatting or fonts.
- Producing documentation for a text processing tool that’s not among those supported by default.
- Producing documentation that purposely omit some parts of Eiffel texts, in line with the ideas applied by the Contract and Interface forms.

This appendix describes EFF and its conventions, enabling you to write filters. Note that in practice the best way to write an EFF filter is usually not from scratch, but by copying an existing filter — one that seems closest to your needs — and adapting the copy.



## *Where to put filters*

When you choose to generate documentation, EiffelStudio will ask you to select a filter from a list it obtains by looking up the files of extension **.fil** in the directory

```
$ISE_EIFFEL/examples/bench/filters
```

To make a new filter available to yourself and other users of this installation, just add the corresponding file *name.fil* to this directory. Make sure to choose the appropriate *name*, since this is what the menu of available filters will display.

## *Filter basics*

An EFF filter follows a very simple structure. As with all other Eiffel-related notations (such as Eiffel itself and Lace, the control language for Eiffel systems), any line or part of a line beginning with two consecutive dashes **--** is a comment, except if it immediately follows a percent sign since, as will be seen below, **%--** is used to denote an Eiffel comment in the class text. Blank lines are also permitted. Comments and blank lines carry no semantic value.

Except for comments and blank lines, a filter is a sequence of entries, all of the form

```
Construct | Replacement
```

where: *Construct* is one of a set of possible strings, most of which corresponds to Eiffel constructs, such as **Class\_declaration**, or keywords, such as **class**; and *Replacement* is a string indicating how to format specimens of the *Construct* that appear in a class text.

For readability, there may be any number of blanks or tabs between the *Construct* and the vertical bar **|**, so that you can align all the bars if you wish. On the right of the bar, however, all characters including blanks and tabs are significant, since they are part of the replacement for the *Construct*.

## *The asterisk*

In the *Replacement* part, you may use the symbol \* (asterisk) to denote the construct specimen itself. So for example the entry

```
Feature_clause | %N%N*%N%N
```

specifies the following formatting for any **Feature\_clause**: two successive blank lines (expressed as **%N**, New Line, a convention taken from Eiffel); the feature clause itself; two blank lines.

Similarly, in an HTML format, the entry

```
External |<B>*</B>
```

means that the Eiffel keyword **external** must appear in the filtered form immediately preceded by **<B>**, the HTML code for switching to boldface, and immediately followed by **</B>**, the code for reverting to the previous setup. Here you can also write the right-hand side without the asterisks, as **<B>external</B>**. If, however, all keywords are to use boldface, it is preferable to write a single entry

```
Keyword |<B>*</B>
```

which, thanks to the asterisk, will govern all construct specimens of the **Keyword** category. You can still override this specification for an individual keyword by including a specific entry for it.

## *Constructs*

The following general syntactic constructs may appear as the left-hand side, *Construct*, of an entry:

**Class\_declaration**  
**Class\_end**  
**Class\_header**  
**Class\_name**  
**Comment**  
**Creators**  
**Escape**  
**Feature\_clause**  
**Feature\_declaration**  
**Features**  
**Formal\_generics**  
**Indexing\_clause**  
**Inheritance**  
**Invariant\_clause**  
**Keyword**  
**New\_line**  
**Obsolete\_clause**  
**Suffix**  
**Symbol**  
**Tab**

Most of these denote Eiffel constructs as they appear in the official language reference, the book *Eiffel: The Language*. Since the Eiffel construct names **Feature**, **Invariant** and **Obsolete** are also keywords and EFF, like Eiffel, is case-insensitive, the EFF construct names use the suffix **\_clause**, for example **Feature\_clause**.

The constructs corresponding to syntactic constructs are self-explanatory. The others are:

- **Class\_end**, denoting the final end of a class text.
- **Keyword**, denoting any Eiffel keyword among those listed in boldface in the corresponding appendix in *Eiffel: The Language*
- **New\_line**, denoting any passage to a new line in the class text.
- **Suffix**, used to introduce the file extension for the generated documentation files. If you don't specify this, EiffelStudio will use the filter's name as extension.
- **Symbol**, denoting any of the Eiffel symbols listed in the corresponding appendix of *Eiffel: The Language*.
- **Escape**, to protect special characters of the external tool, as explained below.
- **Tab**, denoting any tab character appearing in the class text.

## Keywords

A *Construct* part may consist of the name of an Eiffel keyword. To see the complete list of possible keywords, look at the *template* filter, file *format.fil-template* in the default filter directory *\$ISE\_EIFFEL/bench/filters*, which includes all of them with a single asterisk *\** as the *Replacement* part.

If entries are present for both the *Keyword* construct and individual keywords, the individual keyword entries will override the general entry for the keywords listed; the general entry will apply to all other keywords. This makes it possible to have both a general convention for keywords and a special convention for some of them.

## Symbols

A *Construct* part may consist of an Eiffel symbol, such as *:=*, *<<* and many others. Again, you may see the complete list by looking at *format.fil-template*. Note the following conventions:

- *%\** represents an asterisk. for example as a multiplication operator; the *%* avoids the confusion with the special meaning of the asterisk for EFF. You can find examples of this convention in the EFF filters for troff and gtroff.
- Similarly, the Eiffel comment symbol appears as *%- -*, since just writing *--* would introduce a comment in the EFF filter itself.

As with keywords, you may specify a general convention for symbols, defined by an entry for the construct *Symbol*, and special conventions for certain individual symbols. Specific symbol entries will override the general *Symbol* convention.

## Escape characters

A text processing system or other external tool may attach a special role to characters that may normally appear in Eiffel texts. For example, the braces *{* and *}*, used in Eiffel's Export clauses, have a special meaning for T<sub>E</sub>X. Including them without precaution in T<sub>E</sub>X input will cause trouble. Similarly, many text processing formats attach a special meaning to the backslash character *\* which, although not special for Eiffel, may appear in an Eiffel string.

In such cases the filter must “*escape*” the special character, that is to say, protect it by other characters. For example troff and other text processing tools treat two successive backslash characters *\\* as denoting a single backslash in the text to be output. The first backslash is the escape character, protecting the second one.

The *Escape* construct addresses such cases. The first character that follows *Escape* (after one or more blanks or tabs) is the character to be escaped. The string after the vertical bar is the replacement for that character.

Here for example is an escape entry for the backslash in tools that need to escape it through another backslash:

Escape \\

### *Special characters and strings*

EFF uses Eiffel-like conventions, based on the percent sign, for control characters appearing in *Replacement* parts of entries. Two of these conventions have just been noted: **%\*** to represent an asterisk and **%-** to represent a dash that does not introduce an Eiffel comment. In addition:

- **%|** denotes a vertical bar. (This is necessary since EFF uses **|** by itself in each entry to separate the *Construct* from the corresponding *Replacement*.)
- **%N** (recommended form) or **%n** denotes a new line.
- **%T** (recommended form) or **%t** denotes a tab.
- **%%** denotes a percent sign.
- **%** (percent followed by a space) denotes a space. This is equivalent to just a space, but more visible.

If **c** is not one of the characters for which special conventions have been listed, **%c** denotes the character **c** itself.

A multi-line entry uses the Eiffel convention for string continuations: **%** at the end of a line to signal that there is a continuation; a continuation line begins with zero or more spaces and tabs followed by a **%**; the characters after the **%** are the continuation of the string.