SNiFF+™

Version 3.2 for Unix and Windows

C++ Tutorial

TakeFive software
An Integrated Systems Company

TakeFive Software, Inc.
Cupertino, CA
E-mail: info@takefive.com

TakeFive Software GmbH
5020 Salzburg, Austria
E-mail: info@takefive.co.at

Copyright

Copyright © 1992–1999 TakeFive Software Inc.

All rights reserved. TakeFive products contain trade secrets and confidential and proprietary information of TakeFive Software Inc. Use of this copyright notice is precautionary and does not imply publication or disclosure.

Parts of SNiFF+:

Copyright 1991, 1992, 1993, 1994 by Stichting Mathematisch Centrum,

Amsterdam, The Netherlands.

Portions copyright 1991-1997 Compuware Corporation.

Trademarks

SNiFF+ is a trademark of TakeFive Software Inc.

Other brand or product names are trademarks or registered trademarks of their respective holders.

Credits

The first version of Sniff was developed at the Informatics Laboratory of the Union Bank of Switzerland. Its development was considerably facilitated by the public domain application framework ET++.

Authors of the first version:

Walter Bischofberger (Sniff)

Erich Gamma (Sniffgdb)

Erich Gamma and André Weinand (ET++)

Table of Contents

Part I Guidelines

Part II Browsing

Part III Version Controlling

Part IV Browsing-Only Project Setup

Part V Edit/Compile/Debug

Part VI Team Setup

Part I

Guidelines

About this Manual 1

What this manual is

This manual is part of the SNiFF+ documentation set, which consists of:

- User’s Guide
- Reference Guide
- C++ Tutorial
- C Tutorial
- Java Tutorial
- Fortran Tutorial
- Quick Reference Guide
- Release Notes, Installation Guide and Application Papers
- Online documentation of the above in HTML, PostScript and PDF formats

Conventions

One basic term

- **Symbol** — any programming language construct such as a class, method, etc.

Two conventions: menu references

For clarity and to avoid undue verbosity, the phrase:
“Choose the MenuCommand from the MenuName” is presented as follows:

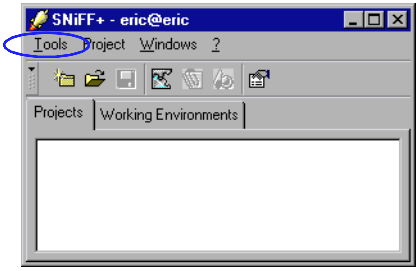
- Choose **MenuName > MenuCommand**.

A context menu that appears when you click the right mouse button is referred to as:
Context menu, and consequently:
“Choose a menu command from the context menu that appears when you click the right mouse button” is presented as follows:

- Choose **Context menu > MenuCommand**

A note on Unix/Windows

The screenshots in this manual are all done on Windows NT. If you are working on Unix, what you see on your screen may look slightly different.

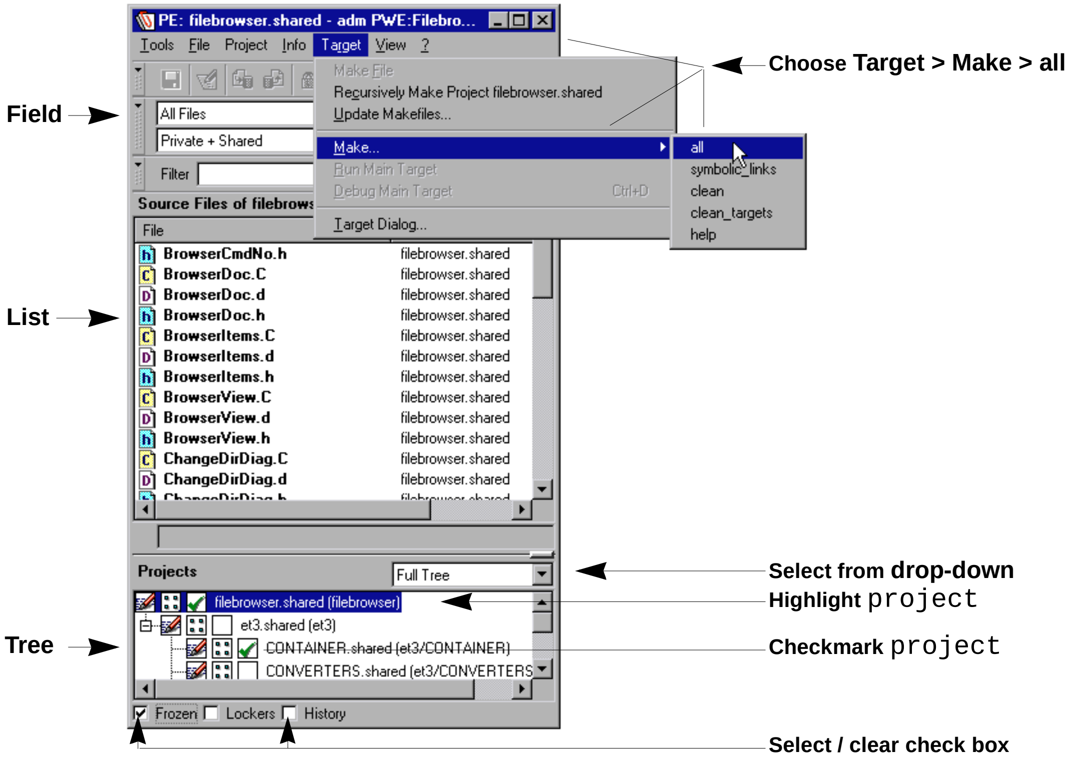


When you start SNiFF+, the first tool that appears is the Launch Pad. In this and other SNiFF+ tools, the first item in the menu bar is for launching tools.

- On **Windows,** it is called **Tools**.
- On **Unix**, it is depicted by an **Icon**.

  When we refer to this menu in order to launch a tool from the Launch Pad, or any other open SNiFF+ tool, we will use the notation:
  Choose **Tools > ToolName**.
- On Unix a “check box” looks like a “button” (Motif Look), and a “drop-down” looks like a “pop-up”.

Tool elements



File	
BrowserCmdNo.h	filebrowser.shared
BrowserDoc.C	filebrowser.shared
BrowserDoc.d	filebrowser.shared
BrowserDoc.h	filebrowser.shared
BrowserItems.C	filebrowser.shared
BrowserItems.d	filebrowser.shared
BrowserItems.h	filebrowser.shared
BrowserView.C	filebrowser.shared
BrowserView.d	filebrowser.shared
BrowserView.h	filebrowser.shared
ChangeDirDiag.C	filebrowser.shared
ChangeDirDiag.d	filebrowser.shared

- ☑ filebrowser.shared (filebrowser)
- □ et3.shared (et3)
  - ☑ CONTAINER.shared (et3/CONTAINER)
  - □ CONVERTERS.shared (et3/CONVERTERS

☑ Frozen □ Lockers □ History

Field

List

Tree

Choose Target > Make > all

Select from drop-down

Highlight `project`

Checkmark `project`

Select / clear check box

Typography

Capitalized Words
Names of tools, windows, dialogs and menus start with capital letters. Examples: Symbol Browser, **Tools** menu, File dialog.

Italics
Names of manuals and newly introduced terms are in italics. Examples: *User's Guide*, the *workspace* concept.

Boldface and ***Bold italics***
Menu, field and button names and menu entries are printed in boldface. Placeholders for symbols, selections or other strings in menus are in bold italics.
Example: From the menu, choose **Show > Symbol(s)** ***selection*****...**

`Monospace`
Code examples and symbol, file and directory names, as well as user entries are printed in monospace type.
Examples: `.login`, `$PATH`, class `VObject`. Type `abc`.

`<Keys>`
Special keys are printed in monospace type with enclosing '`< >`'.
Examples: `<CTRL>`, `<Return>`, `<Meta>`.

Feedback and useful links

Your feedback is always very welcome. Please send feedback to one of our support e-mail addresses.

Europe:

sniff-support@takefive.co.at

USA:

sniff-support@takefive.com

Useful links

SNiFF+ web pages:

- SNiFF+ Users Mailing List

  http://www.takefive.com/support/sniff-list.html
- SNiFF+ Users Mailing List Archive

  http://www.takefive.com/sniff-list
- Frequently Asked Questions

  http://www.takefive.com/support/faq.html
- Customer Newsletter

  http://www.takefive.com/news/customer_newsletter.html

Road Map 2

Introduction

This manual introduces the SNiFF+ solution for C++ development and is centered around 7 tutorials. Each of the tutorials focuses on different SNiFF+ tools, tasks and concepts. Although each consecutive tutorial and chapter is in itself more or less modular, it is assumed that you are familiar with what has gone on before.

You will be using two different C++ example codes in this manual. The C++ example code called `filebrowser`, used in the Browsing and Version Controlling Tutorials, is based on the ET++ public domain class library, whose source is part of the SNiFF+ software distribution. ET++ is an object-oriented application framework developed by the University of Zurich and the UBILAB of the Union Bank of Switzerland. For the other tutorials you will be using the C++ code called `complex`, which is also provided with your SNiFF+ installation.

What this manual is not

This manual is not an exhaustive guide to SNiFF+, nor will it teach you C++.

The SNiFF+ C++ Tutorial

The SNiFF+ C++ Tutorial consists of the following parts:

- Browsing — page 11
- Version Controlling — page 59
- Browsing-Only Project Setup — page 73
- Edit/Compile/Debug — page 77
- Team Setup — page 99
- Developing in a team — page 127
- Team Maintenance — page 137

> **Note**
>
> Please note that a Log Window, displaying SNiFF+ error and control messages, may appear at several stages throughout this tutorial.

Browsing

This tutorial is for you if

- you are a new SNiFF+ user
- you want to quickly learn how to use SNiFF+ for browsing C++ code

Version Controlling

This tutorial is for you if

- you need SNiFF+ and RCS (included in the SNiFF+ package) for configuration management and version controlling (CMVC)

Browsing-Only Project Setup

This tutorial is for you if

- you need to set up your own browsing-only project

Edit/Compile/Debug

This tutorial is for you if

- you want to use SNiFF+ in single-user/single-platform C++ development
- you want to learn about building C++ executables
- you want an introduction to the tools used in the C++ edit/compile/debug cycle

Note that this tutorial introduces concepts and tools used in developing, irrespective of whether you are working alone or as part of a team.

Team Setup

This tutorial is for you if

- you have done the previous tutorials or
- you are familiar with SNiFF+ and
- you need SNiFF+ in a multi-user and/or a multi-platform work situation
- you need SNiFF+ and RCS (included in the SNiFF+ package) for configuration management and version controlling (CMVC)
- you are responsible for setting up projects and working environments in a multi-user/multi-platform work situation (*Working Environments Administrator*).

Developing in a team

This tutorial is for you if

- you have done the previous tutorials or
- you are familiar with SNiFF+ and
- you work in a team and use RCS for version controlling and configuration management. Note that the Edit/Compile/Debug cycle is described in the Edit/Compile/Debug tutorial

Team Project Maintenance

This tutorial is for you if

- you are responsible for maintaining projects and working environments in a multi-user/ multi-platform work situation (*Working Environments Administrator*).

Part II

Browsing

Opening the filebrowser project 1

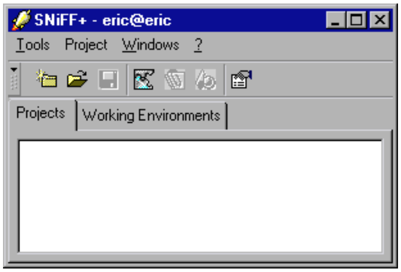
In this chapter, you will

- learn how to open a project in a SNiFF+ Working Environment.

We assume you have successfully installed SNiFF+, and know how to start it. If not, please refer to the *Installation Guide*.

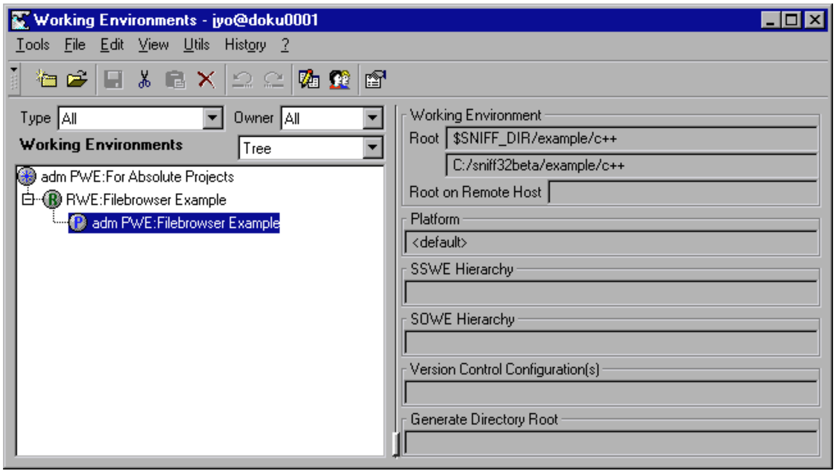
- Start SNiFF+.

  The Launch Pad appears.



1. In the Launch Pad, choose **Tools > Working Environments**.

   The Working Environments tool opens.



Type All

Owner All

Working Environments: Tree

adm PWE:For Absolute Projects
RWE:Filebrowser Example
adm PWE:Filebrowser Example

Root $SNIFF_DIR/example/c++
C:/sniff32beta/example/c++

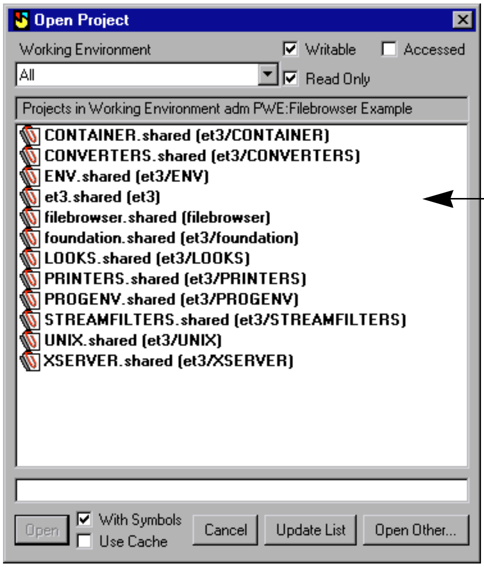
Platform <default>

2. In the Working Environments tool, double-click on:

   `adm PWE: Filebrowser Example`.

   The Open Project dialog opens.

3. In the Open Project dialog, press the **Update List** button to display the projects in the Project List.



Open Project

Working Environment: All

☑ Writable □ Accessed

☑ Read Only

Projects in Working Environment adm PWE:Filebrowser Example

CONTAINER.shared (et3/CONTAINER)
CONVERTERS.shared (et3/CONVERTERS)
ENV.shared (et3/ENV)
et3.shared (et3)
filebrowser.shared (filebrowser)
foundation.shared (et3/foundation)
LOOKS.shared (et3/LOOKS)
PRINTERS.shared (et3/PRINTERS)
PROGENV.shared (et3/PROGENV)
STREAMFILTERS.shared (et3/STREAMFILTERS)
UNIX.shared (et3/UNIX)
XSERVER.shared (et3/XSERVER)

☑ With Symbols

□ Use Cache

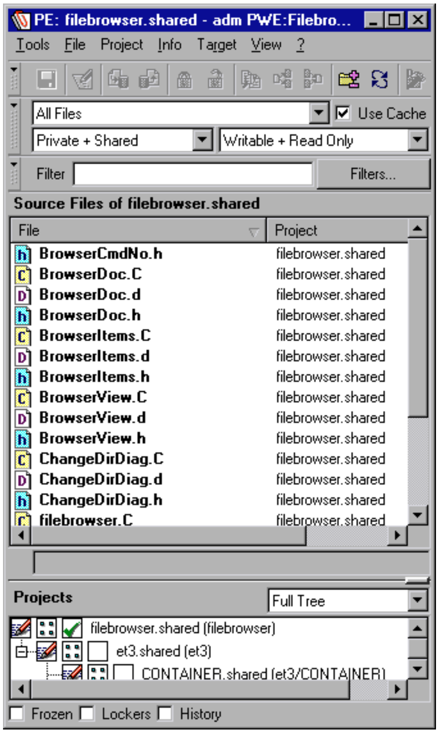
Project List

4. From the Project List, select `filebrowser.shared` and press **Open**.

   SNiFF+ parses all the symbol files in the project and loads the project with symbol information.

   After the project is opened, SNiFF+ displays the project's structure and contents in a Project Editor, which should look like the one illustrated below.

   A project is the main structuring element in SNiFF+ for grouping together files and directories on your file system that logically belong together. The Project Editor will be discussed in more detail in the Edit/Compile/Debug part of this tutorial.



PE: filebrowser.shared - adm PWE:Filebro...

All Files ☑ Use Cache

Private + Shared | Writable + Read Only

File	Project
BrowserCmdNo.h	filebrowser.shared
BrowserDoc.C	filebrowser.shared
BrowserDoc.d	filebrowser.shared
BrowserDoc.h	filebrowser.shared
BrowserItems.C	filebrowser.shared
BrowserItems.d	filebrowser.shared
BrowserItems.h	filebrowser.shared
BrowserView.C	filebrowser.shared
BrowserView.d	filebrowser.shared
BrowserView.h	filebrowser.shared
ChangeDirDiag.C	filebrowser.shared
ChangeDirDiag.d	filebrowser.shared
ChangeDirDiag.h	filebrowser.shared
filebrowser.C	filebrowser.shared

Full Tree

☑ filebrowser.shared (filebrowser)
□ et3.shared (et3)
□ CONTAINER.shared (et3/CONTAINER)

□ Frozen □ Lockers □ History

Browsing Symbols 2

The Symbol Browser displays all the symbols used in the source files of a project. Symbols can be filtered according to symbol type and other criteria. In SNiFF+, a *symbol* is any C++ language construct such as a class, method, function, etc.

In this chapter you will:

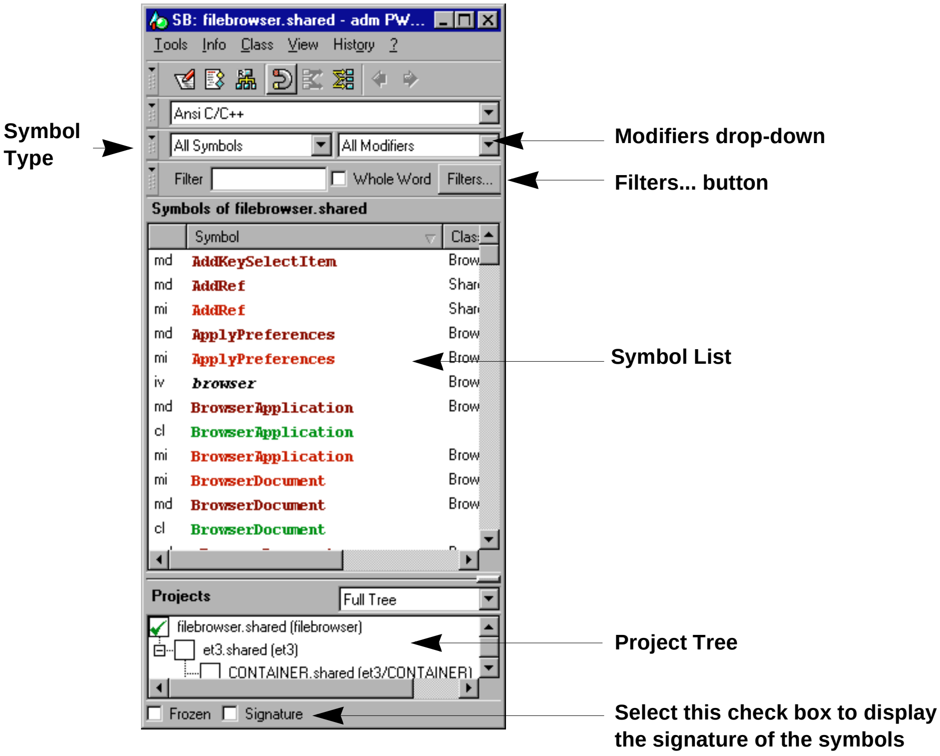
- find out which symbols are defined in the project
- look at which file or project a symbol belongs to
- navigate to a particular symbol
- go to the definition of a symbol

Opening the Symbol Browser

To open the Symbol Browser:

- In the Project Editor, choose **Tools > Symbol Browser.**

A Symbol Browser is opened listing all the classes in the filebrowser project. Note that the content of the Symbol List in the Symbol Browser is determined by the Symbol Type drop down, Modifier drop-down, filters, the Project Tree and a regular expression matching the names of the symbols.



Symbol Type

Modifiers drop-down

Filters... button

Symbol List

Project Tree

Select this check box to display the signature of the symbols

	Symbol	Clas:
md	AddKeySelectItem	Brow
md	AddRef	Shar
mi	AddRef	Shar
md	ApplyPreferences	Brow
mi	ApplyPreferences	Brow
iv	browser	Brow
md	BrowserApplication	Brow
cl	BrowserApplication	
mi	BrowserApplication	Brow
mi	BrowserDocument	Brow
md	BrowserDocument	Brow
cl	BrowserDocument	

☑ filebrowser.shared (filebrowser)
☐ et3.shared (et3)
☐ CONTAINER.shared (et3/CONTAINER)

- In the Project Editor, choose **Tools > Close Tool** to close the Project Editor.

Restricting information in Symbol List

To look at only those classes defined in a given project, e.g. `filebrowser.shared:`

1. In the Project Tree, highlight `filebrowser.shared` by clicking on its name.
2. Right-click anywhere in the Project Tree, and choose

   Context menu > Select from filebrowser.shared Only.
3. In the **Symbol Type** drop-down, scroll up the list and select **class**.

   Now the Symbol List contains all the classes defined in the project `filebrowser.shared`, as you can see there are 11 symbols defined.

To look at which classes are defined in `et3.shared` and its subprojects:

1. In the Project Tree, click on the `'-'` sign to the left of `et3.shared` to collapse the node.

   You can now view and manipulate `et3.shared` and its subprojects as a single project.

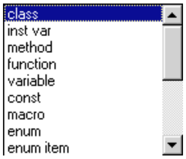
2. Highlight `et3.shared`.
3. Choose **Context menu > Select From et3.shared Only.**

   Because class is displayed in the Symbol Type drop-down, all classes which are defined in `et3.shared` and its subprojects are displayed in the Symbol List.

4. Click on the `'+'` sign to the left of `et3.shared` to expand the node.

Symbol Type drop-down

The Symbol Browser can show symbols of different types and macros.



Besides classes, you can look at functions, friends, variables, typedefs, etc. Methods can also be shown. (The list of methods can get very long because it is a flat view of all methods of all classes if not further constrained.)

- Try to show the functions, macros, typedefs, etc., of `et3.shared` by selecting different types from the **Symbol Type** drop-down .

Displaying signatures of symbols

Currently you see only the names of the various types. By selecting the **Signature** check box in the status line of the Symbol Browser, you can display the complete signature of the listed symbols.

- Choose **class** from the **Symbol Type** drop-down.
- Select the **Signature** check box, and then drag the side of the window outward until you can see the whole text line in the Symbol List.

  The Symbol List now shows the following information:
  `symbolType` **symbolName** `fileName projectName.shared`

  - `fileName.C` is the name of the file where the symbol is implemented
  - `fileName.h` is the name of the file where the symbol is defined
  - `projectName.shared` is the name of the project containing this file

  Notice that class `ActionButton` is defined in file `Buttons.h` and is contained in `et3.shared`.

Keyboard navigation in Lists

In each list of any SNiFF+ tool, you can quickly navigate to entries by clicking into the list, then typing the name of the entry you wish to find. Each consecutive keystroke immediately causes the list to position to the next matched entry.

To find the class `Button` in the Symbol List:

1. In the Project Tree, choose **Context menu > Select From All Projects**.
2. From the **Symbol Type** drop-down, select **class** (by default **All Symbols** is selected).
3. In the **Modifier** drop-down, make sure **All Modifiers** is selected.
4. Click into the Symbol List.
5. Press the `<b>` key.

   The list is positioned to the first entry that starts with an `'b'`.
6. Press the `<u>` key.

   As you can see, the class Button is highlighted because it is matched by the `'bu'` you entered.

Notes on keyboard navigation

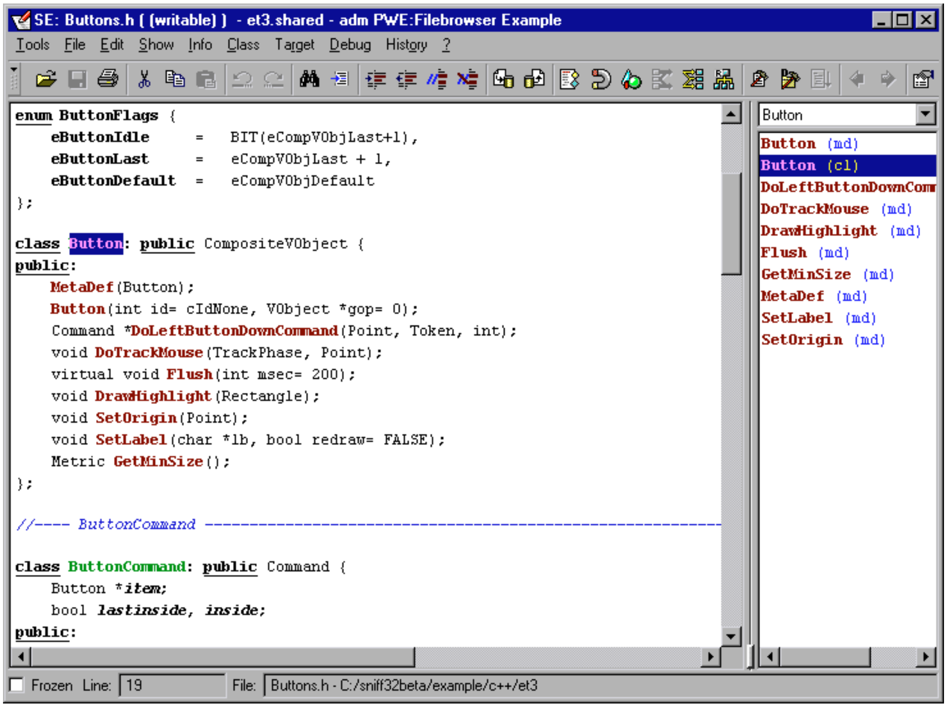
- You can restart searches by pressing `<ESC>`.
- If the pressed key does not match any entry, you will be warned by a beep.
- The cursor keys can also be used for navigating in a list.
- Pressing `<Return>` on a highlighted entry in a list has the same effect as double clicking that entry, i.e., the highlighted symbol definition will be loaded into a Source Editor.

Studying the symbol definition

Each symbol in the Symbol List is defined somewhere in your source code. The quickest way to get to a symbol definition is to double-click on the symbol name in the Symbol Browser.

1. In the Symbol Browser, double-click on `cl Button`.

   A Source Editor opens, the source file `Buttons.h` is loaded and the class `Button` is highlighted.


SE: Buttons.h ((writable)) - et3.shared - adm PWE:Filebrowser Example
Tools File Edit Show Info Class Target Debug History ?

```
enum ButtonFlags {
    eButtonIdle      =   BIT(eCompVObjLast+1),
    eButtonLast      =   eCompVObjLast + 1,
    eButtonDefault   =   eCompVObjDefault
};

class Button: public CompositeVObject {
public:
    MetaDef(Button);
    Button(int id= cIdNone, VObject *gop= 0);
    Command *DoLeftButtonDownCommand(Point, Token, int);
    void DoTrackMouse(TrackPhase, Point);
    virtual void Flush(int msec= 200);
    void DrawHighlight(Rectangle);
    void SetOrigin(Point);
    void SetLabel(char *lb, bool redraw= FALSE);
    Metric GetMinSize();
};

//---- ButtonCommand ------------------------------------------------------------

class ButtonCommand: public Command {
    Button *item;
    bool lastinside, inside;
public:
```

```
Button
Button (md)
Button (cl)
DoLeftButtonDownComm
DoTrackMouse (md)
DrawHighlight (md)
Flush (md)
GetMinSize (md)
MetaDef (md)
SetLabel (md)
SetOrigin (md)
```

□ Frozen Line: 19 File: Buttons.h - C:/sniff32beta/example/c++/et3

2. In the Source Editor, choose **Tools > Close Tool** to close the Source Editor. The Source Editor is discussed onThe SNiFF+ Editor — page 37.

Understanding Class Hierarchies 3

You will now learn more about Top-Down Browsing with SNiFF+. Top-Down browsing is useful when you have a symbol, e.g., a class, and want to learn more about its details — where and in what context it is used. Figuratively speaking, you are coming from a more distant view of your system and are browsing down to the source code (bottom) and greater detail. You will study this type of browsing with the class `Button`. During the following step by step tour, you will start with the Hierarchy Browser and learn more about the Class Browser, Source Editor and Cross Referencer.

The Hierarchy Browser is a tool where you can view the inheritance relationships of all classes in a project or group of projects. Thus, loading all classes into the Hierarchy Browser allows you to get a good overview of the complete class hierarchy. You can also restrict this information by viewing only a class and its relatives.

This chapter is about

- looking at the inheritance relationships of all classes
- viewing the class hierarchy of an individual class

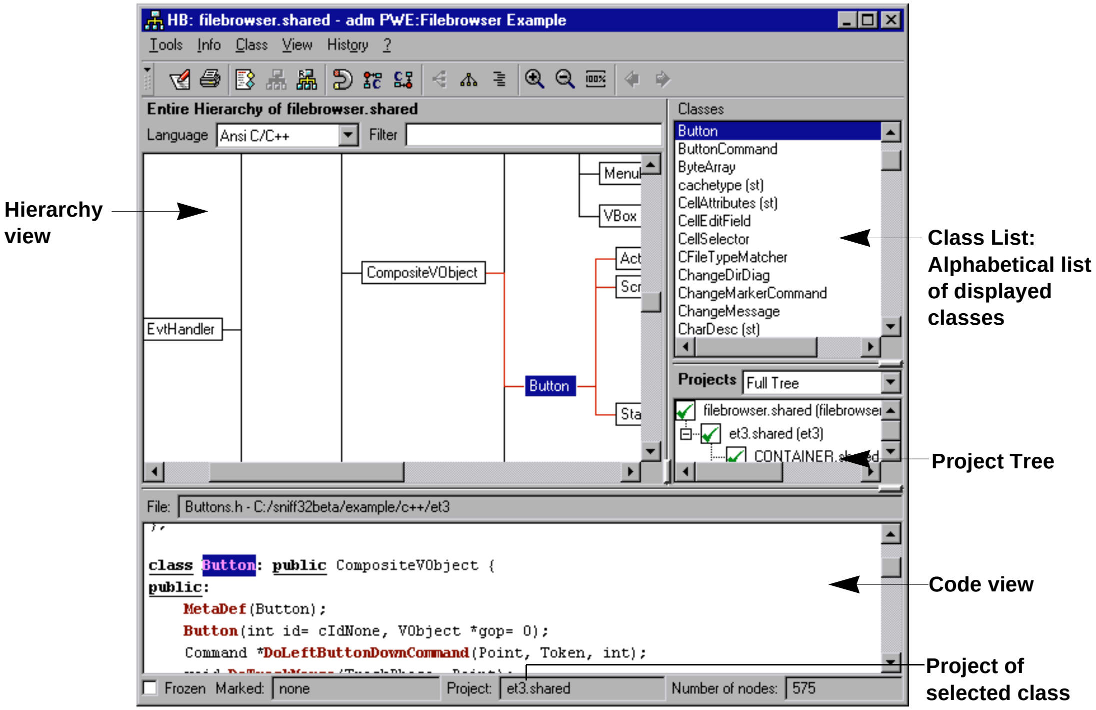
Opening the Hierarchy Browser

Make sure that `Button` is highlighted in the Symbol List of the Symbol Browser.

- Choose **Class > Show Button in Entire Hierarchy**.

  The Hierarchy Browser opens.

- Close the Symbol Browser.


HB: filebrowser.shared - adm PWE:Filebrowser Example
Tools
Info
Class
View
History
?
Entire Hierarchy of filebrowser.shared
Language
Filter
Menu
VBox
Act
Scr
CompositeVObject
EvtHandler
Button
Sta
Classes
Projects
Hierarchy view
Class List: Alphabetical list of displayed classes
Project Tree
Code view
Project of selected class

Language: Ansi C/C++

Classes:
Button
ButtonCommand
ByteArray
cachetype (st)
CellAttributes (st)
CellEditField
CellSelector
CFileTypeMatcher
ChangeDirDiag
ChangeMarkerCommand
ChangeMessage
CharDesc (st)

Projects: Full Tree

☑ filebrowser.shared (filebrowser
☑ et3.shared (et3)
☑ CONTAINER.shared

File: Buttons.h - C:/sniff32beta/example/c++/et3

```
};

class Button: public CompositeVObject {
public:
    MetaDef(Button);
    Button(int id= cIdNone, VObject *gop= 0);
    Command *DoLeftButtonDownCommand(Point, Token, int);
```

☐ Frozen Marked: none Project: et3.shared Number of nodes: 575

Inheritance relationships of all classes

In the illustration above, the complete class graph is displayed in the Hierarchy view, and the class `Button` is highlighted. You can see the definition of `Button` in the Code view.

- Scroll around to get an overview of the class hierarchy and the inheritance path of the class `Button`.

Note

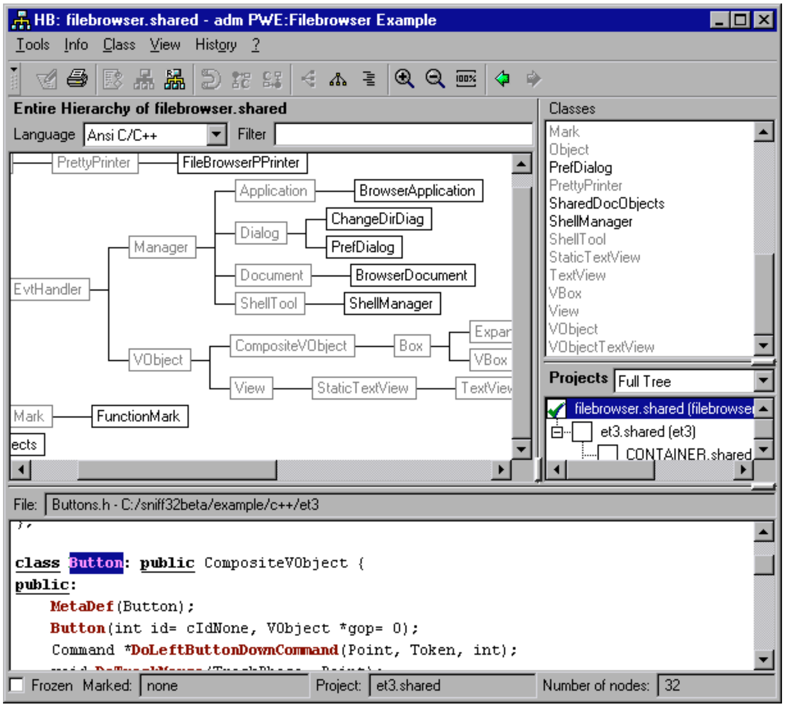
Although we use only single inheritance in our examples, SNiFF+'s tools also support multiple inheritance.

Hiding the classes of a subproject

You may want to look at the minimal graph of the inheritance hierarchy of only one project, hiding those classes that don't belong to the selected project. To do so:

- In the Project Tree, highlight `filebrowser.shared` and choose

  Context menu > Select from filebrowser.shared Only.
  Only the classes of `filebrowser.shared` are displayed. With the exception of those classes that are needed to draw a minimal tree, all other classes are hidden. Classes not part of the checkmarked project are grayed out. This view also gives you a very good overview on how the filebrowser project depends on `et3.shared` in terms of inheritance.



HB: filebrowser.shared - adm PWE:Filebrowser Example

Entire Hierarchy of filebrowser.shared

Language: Ansi C/C++ Filter:

Classes:
Mark
Object
PrefDialog
PrettyPrinter
SharedDocObjects
ShellManager
ShellTool
StaticTextView
TextView
VBox
View
VObject
VObjectTextView

Projects: Full Tree

☑ filebrowser.shared (filebrowse
□ et3.shared (et3)
□ CONTAINER.shared

File: Buttons.h - C:/sniff32beta/example/c++/et3

```
};

class Button: public CompositeVObject {
public:
    MetaDef(Button);
    Button(int id= cIdNone, VObject *gop= 0);
    Command *DoLeftButtonDownCommand(Point, Token, int);
```

□ Frozen Marked: none Project: et3.shared Number of nodes: 32

Class hierarchy of an individual class

So far we've looked at an overview of the complete class hierarchy. Let's now take a look at the superclasses and subclasses of an individual class. To see the superclasses and subclasses (i.e. only the immediate relatives) of class `Button`:

1. Right-click anywhere in the Project Tree.

2. Choose **Context menu > Select from All Projects**.

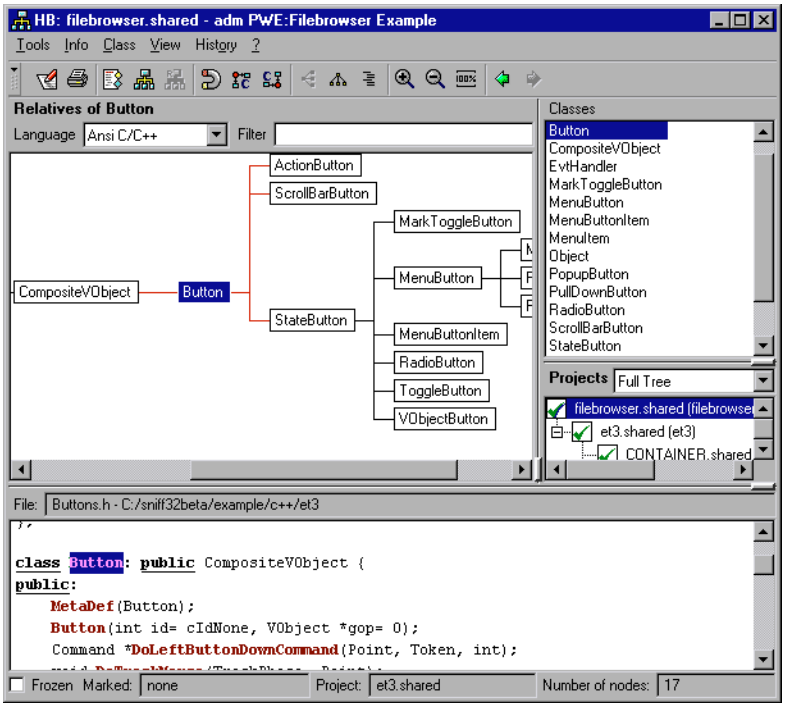
   All the projects in the Project Tree are now checkmarked, and so all the classes of all the projects are now displayed in the Hierarchy view.

3. Make sure that `Button` is selected in the Hierarchy view.

4. Right-click in the Hierarchy view and choose

   Context menu > Show Button Relatives

   The Hierarchy view now displays only the relatives of `Button`.



File: Buttons.h - C:/sniff32beta/example/c++/et3

```
};

class Button: public CompositeVObject {
public:
    MetaDef(Button);
    Button(int id= cIdNone, VObject *gop= 0);
    Command *DoLeftButtonDownCommand(Point, Token, int);
    void DoTrackMouse(TrackPhase, Point);
```

□ Frozen Marked: none Project: et3.shared Number of nodes: 17

As you can see, only the superclasses and subclasses of `Button` are shown. All other classes are hidden. This gives you a better picture of the inheritance of `ActionButton` and related classes.

Browsing class members

4

The Class Browser browses through the locally defined and inherited members of a class. It provides many filtering possibilities based on inheritance, visibility and type of the members. In this chapter you will learn more about

- elements of a class
- interface of a class
- what overrides a certain method

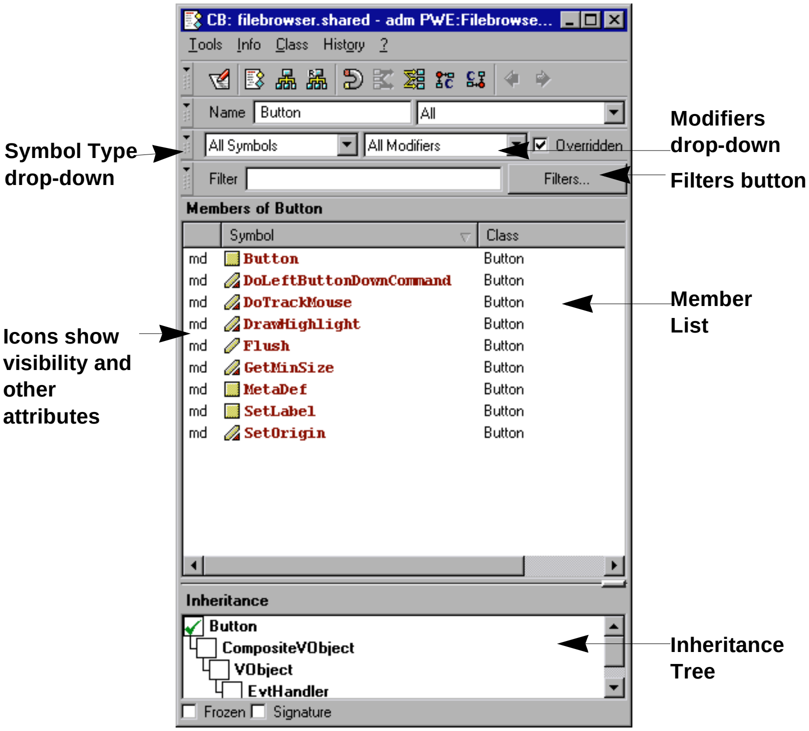
Opening the Class Browser

To open the Class Browser with information on class `Button`:

- Make sure that `Button` is selected in the Hierarchy Browser. If it isn't, click into the Class List and type b. The focus is set to `Button`.

- Choose **Context menu > Browse Button**.

  The Class Browser opens.



	Symbol	Class
md	Button	Button
md	DoLeftButtonDownCommand	Button
md	DoTrackMouse	Button
md	DrawHighlight	Button
md	Flush	Button
md	GetMinSize	Button
md	MetaDef	Button
md	SetLabel	Button
md	SetOrigin	Button

☑ Button
□ CompositeVObject
□ VObject
□ EvtHandler

- Close the Hierarchy Browser.

Elements of a class

As you can see in the above illustration, the symbols of `Button` are listed, because only `Button` is selected.

The Class Browser lists the elements of the current class identified by the element name and the name of the class defining the element. The icons in front of the name show the attributes of the member. To find out what the icons mean, choose **Help(?) > Quick Ref**.

To look at the virtual methods of class `Button`:

1. From the **Symbol Type** drop-down, choose **method**.
2. From the **Modifiers** drop-down, choose **virtual**.

   All virtual methods of class `Button` are displayed in the Member List.

3. Select the **Signature** check box in the status line.

   The signature of each member is also listed.

   The Member List is no longer sorted alphabetically. The members are now sorted in the same order as they appear in the declaration of the class (file order).

Interface of a class

To look at the interface of `Button:`

1. From the **Modifiers** drop-down, choose **All Modifiers**.

   All methods of class `Button` are displayed in the Member List.

2. In the Inheritance view, choose **Context menu > Select from All Classes**.

3. Clear the **Signature** check box to get a less cluttered view of the Member List.

   You now have a completely flat view of the class `Button`, including all overridden methods. Each entry in the list shows the method name and the class defining the method so you can see what overrides what. For example, the method `At` is defined in `VObject` and overridden in `CompositeVObject`.

   A completely flat view of the class is not always useful. Sometimes you want to see just the interface of the current class, hiding all the methods that are overridden.

4. In the Inheritance view, make sure Button is highlighted and choose

   Context menu > Select from Button only.

5. Clear the **Overridden** checkbox.

   Now, only the client interface of the loaded class is visible.

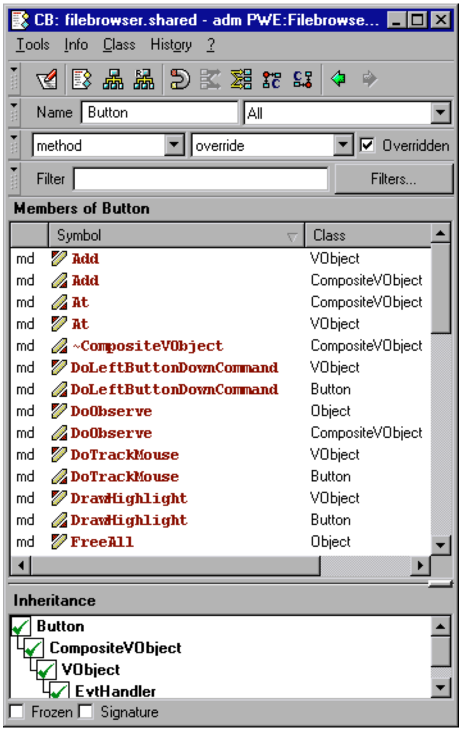
Tracking implementations of methods

Let’s now see what overrides the method `Getminsize`. SNiFF+ makes it possible for you to do this by combining the Class Browser and the Hierarchy Browser.

1. In the Inheritance Tree, choose **Context menu > Select from All Classes**.
2. Set the various attributes of the Class Browser as follows:

   Symbol Type drop-down: **method**
   Modifiers drop-down: **override**
   Visibility: **All**
   Overridden check box: **selected**

   You now have all the members that override methods defined in a superclass, as well as all the methods that are overridden in a subclass. As you can see, method `GetMinSize` is defined in `VObject` and is overridden in `Button`.


CB: filebrowser.shared - adm PWE:Filebrowse...
Tools Info Class History ?
Filters...

Name Button | All

method | override | ☑ Overridden

Filter

Members of Button

	Symbol	Class
md	Add	VObject
md	Add	CompositeVObject
md	At	CompositeVObject
md	At	VObject
md	~CompositeVObject	CompositeVObject
md	DoLeftButtonDownCommand	VObject
md	DoLeftButtonDownCommand	Button
md	DoObserve	Object
md	DoObserve	CompositeVObject
md	DoTrackMouse	VObject
md	DoTrackMouse	Button
md	DrawHighlight	VObject
md	DrawHighlight	Button
md	FreeAll	Object

Inheritance

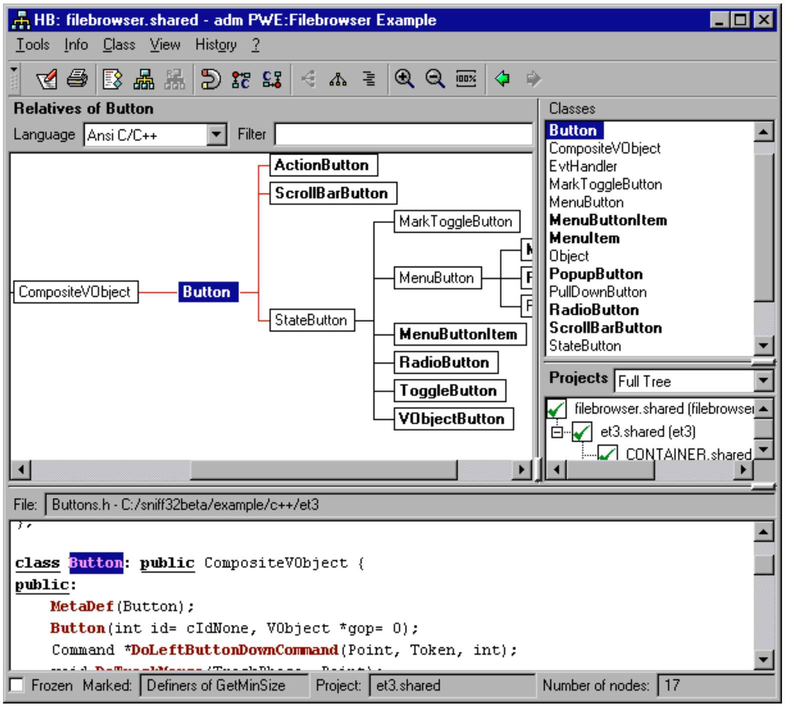
- ✓ Button
  - ✓ CompositeVObject
    - ✓ VObject
      - ✓ EvtHandler

□ Frozen □ Signature

3. In the Member List, highlight the method `GetMinSize` which is overridden in `Button`.

4. From the menu, choose **Class > Mark Relatives Defining GetMinSize**.

   The Hierarchy Browser is opened and all classes related to `Button` are loaded. All classes displayed (marked) in boldface override the method `GetMinSize`.



Language: Ansi C/C++

Filter:

Classes:
- **Button**
- CompositeVObject
- EvtHandler
- MarkToggleButton
- MenuButton
- **MenuButtonItem**
- **MenuItem**
- Object
- **PopupButton**
- PullDownButton
- **RadioButton**
- **ScrollBarButton**
- StateButton

Projects: Full Tree

☑ filebrowser.shared (filebrowser

☑ et3.shared (et3)

☑ CONTAINER.shared

File: Buttons.h - C:/sniff32beta/example/c++/et3

```
};

class Button: public CompositeVObject {
public:
    MetaDef(Button);
    Button(int id= cIdNone, VObject *gop= 0);
    Command *DoLeftButtonDownCommand(Point, Token, int);
```

□ Frozen Marked: Definers of GetMinSize Project: et3.shared Number of nodes: 17

- Close the Class Browser.

Going to the method implementation

By selecting boldfaced classes in the Hierarchy browser, you can view the source code of the overridden methods.

1. Select `Button` if it isn’t already selected.
2. Choose **Class > Edit Button::GetMinSize**.

   A Source Editor is opened, class `Button` is loaded and the Source Editor is positioned at the method implementation.
3. Close the Source Editor.

Component and Interface browsing 5

In SNiFF+, the Cross Referencer provides symbol cross reference information. All different kinds of cross references are displayed. In addition, it provides a component view (has-a hierarchy) of classes and structures.

In this chapter you will learn more about

- symbol types used as components of a given symbol
- all the symbols that refer to a given symbol
- symbol types used in symbol interfaces (parameters, returns)

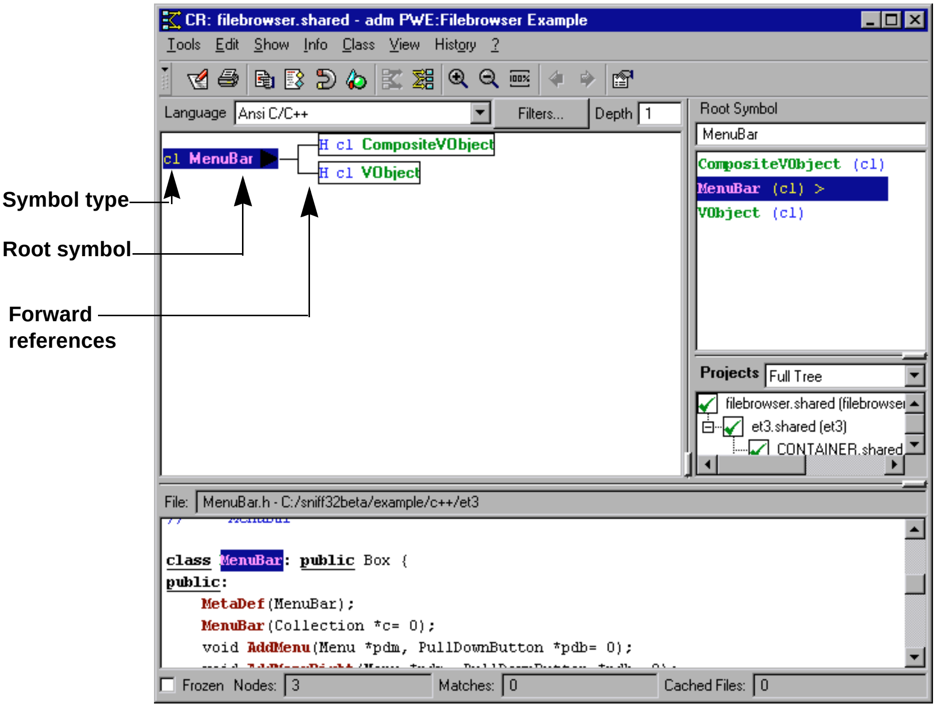
In this chapter, we will browse components and interfaces of class `PullDownButton`.

Opening the Cross Referencer

1. In the Hierarchy Browser, choose **Class > Show Button in Entire Hierarchy**.
2. Click into the Hierarchy Browser's Class List and type '`menub`'.

   Class `MenuBar` is now highlighted in the Hierarchy view.
3. Choose **Context menu > Class MenuBar Refers To Components**.

The Cross Referencer appears.



File: MenuBar.h - C:/sniff32beta/example/c++/et3

```
class MenuBar: public Box {
public:
    MetaDef(MenuBar);
    MenuBar(Collection *c= 0);
    void AddMenu(Menu *pdm, PullDownButton *pdb= 0);
```

- Close the Hierarchy Browser.

Component Browsing (Has-a relationship)

As you can see in the illustration above, the components of the class `MenuBar` are displayed. The second entry in the newly created tree should be the following:

`cl MenuBar > H cl VObject`

This means that: the class MenuBar Has class VObject.

You may also want to know where `VObject` is a component:

1. Select class `VObject` in the created component hierarchy.
2. In the Graph view, choose **Context menu > VObject Referred-By**.

   This may take a little time if it is the first Referred-By query in the project.

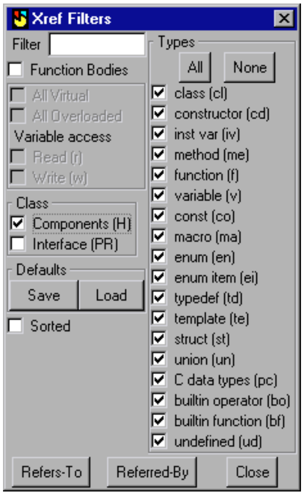
   The subtree of class `VObject` now shows all classes that have `VObject` as a component.
3. Now hide the subnodes of `MenuBar` by selecting it in the Graph view and choosing **Context menu > Hide Subnodes of MenuBar**.

Interface Browsing

You now want to look at the number and type of all symbols `MenuBar` uses as a parameter or return value.

1. Choose **View > Filter...**.

   The Xref Filter dialog appears. Please make sure that only the **Components (H)** checkbox and the **Types** checkboxes are checkmarked.



Filter

□ Function Bodies

□ All Virtual

□ All Overloaded

Variable access

□ Read (r)

□ Write (w)

Class

☑ Components (H)

□ Interface (PR)

Defaults

□ Sorted

Types

☑ class (cl)

☑ constructor (cd)

☑ inst var (iv)

☑ method (me)

☑ function (f)

☑ variable (v)

☑ const (co)

☑ macro (ma)

☑ enum (en)

☑ enum item (ei)

☑ typedef (td)

☑ template (te)

☑ struct (st)

☑ union (un)

☑ C data types (pc)

☑ builtin operator (bo)

☑ builtin function (bf)

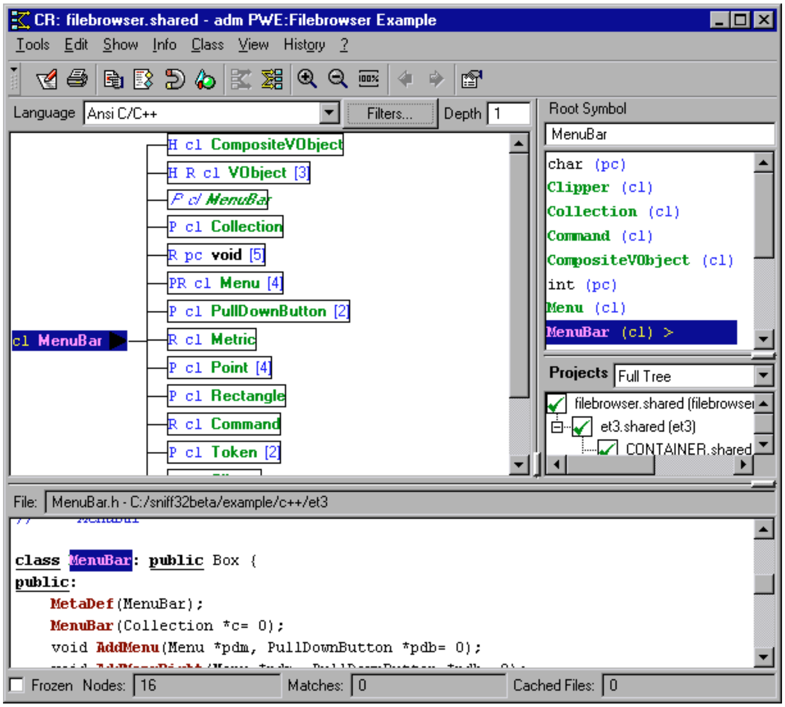
☑ undefined (ud)

2. In the Xref Filter dialog, select the **Interface (PR)** check box.

   P stands for parameter type and R stands for return type. The (H) after Components stands for “Has a”.

3. Press the **Refers-To** button.

   In the illustration below, you can see the type and number of all symbols that `MenuBar` uses as a parameter or return value.



Language: Ansi C/C++ Depth: 1

Root Symbol: MenuBar

Projects: Full Tree

File: MenuBar.h - C:/sniff32beta/example/c++/et3

```
class MenuBar: public Box {
public:
    MetaDef(MenuBar);
    MenuBar(Collection *c= 0);
    void AddMenu(Menu *pdm, PullDownButton *pdb= 0);
```

□ Frozen Nodes: 16 Matches: 0 Cached Files: 0

4. Double-click on the class `MenuBar`.

   A Source Editor is opened and is positioned at the declaration of class `MenuBar`.

5. Close the Cross Referencer.

The SNiFF+ Editor 6

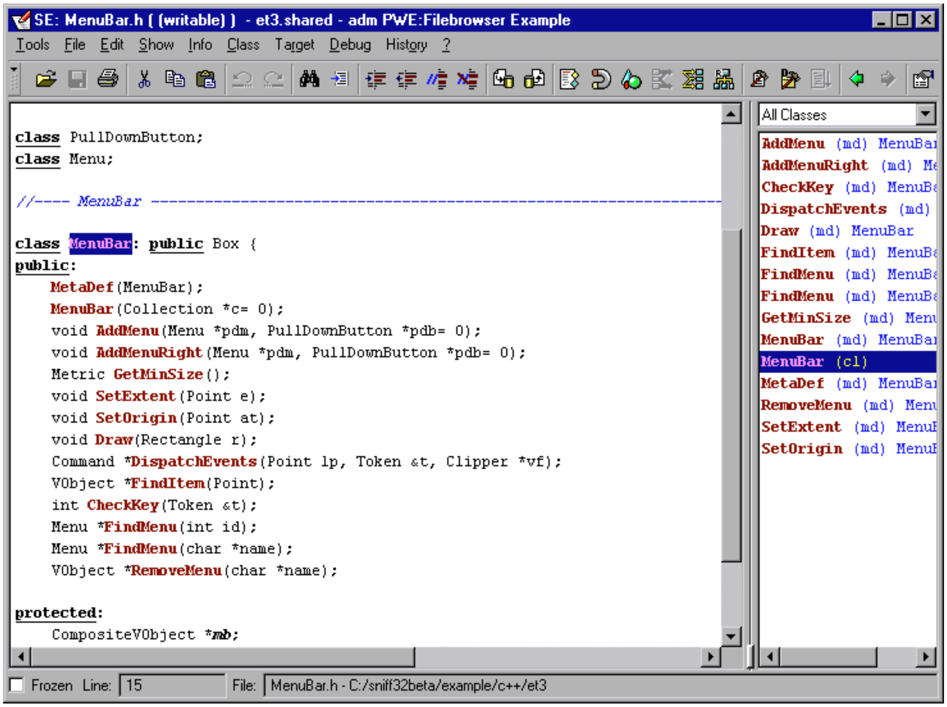
We will now learn more about Bottom-Up Browsing with SNiFF+. This kind of browsing is useful when you start from the source code looking at a symbol, e.g., a variable, and you would like to know more about its declaration and definition. Figuratively speaking, you are coming from a special-usage context (source code, therefore bottom) and are browsing up to its declaration (higher view). In the following step-by-step tour, you'll start with the Source Editor and learn about the Retriever, more about the Cross Referencer and about the Include Browser.

The integrated Source Editor is mouse- and menu-driven. It understands C++ syntax, provides browsing support and automatically highlights structurally important information, such as class names, method names and comments. When a source file is modified and saved, its symbol information is immediately updated.

In this chapter you will:

- go to a variable's definition
- get more information about the class where the variable is defined
- find out if there are more symbols with the same name as the selected symbol

Continuing from the last chapter, the Source Editor is positioned at the definition of class `MenuBar`.


SE: MenuBar.h ((writable)) - et3.shared - adm PWE:Filebrowser Example
Tools File Edit Show Info Class Target Debug History ?
All Classes

```
class PullDownButton;
class Menu;

//---- MenuBar ------------------------------------------------------------------

class MenuBar: public Box {
public:
    MetaDef(MenuBar);
    MenuBar(Collection *c= 0);
    void AddMenu(Menu *pdm, PullDownButton *pdb= 0);
    void AddMenuRight(Menu *pdm, PullDownButton *pdb= 0);
    Metric GetMinSize();
    void SetExtent(Point e);
    void SetOrigin(Point at);
    void Draw(Rectangle r);
    Command *DispatchEvents(Point lp, Token &t, Clipper *vf);
    VObject *FindItem(Point);
    int CheckKey(Token &t);
    Menu *FindMenu(int id);
    Menu *FindMenu(char *name);
    VObject *RemoveMenu(char *name);

protected:
    CompositeVObject *mb;
```

```
AddMenu (md) MenuBar
AddMenuRight (md) Me
CheckKey (md) MenuBa
DispatchEvents (md)
Draw (md) MenuBar
FindItem (md) MenuBa
FindMenu (md) MenuBa
FindMenu (md) MenuBa
GetMinSize (md) Menu
MenuBar (md) MenuBar
MenuBar (cl)
MetaDef (md) MenuBar
RemoveMenu (md) Menu
SetExtent (md) MenuB
SetOrigin (md) MenuB
```

☐ Frozen Line: 15 File: MenuBar.h - C:/sniff32beta/example/c++/et3

We will now use the **History** menu to go to the method `GetMinSize` (implemented in ActionButton).

1. Click on the **History** menu.

   What you see in the menu are all the locations you visited in the source code during the browsing session.

2. Choose **History > Buttons (cl) - et3.shared**.

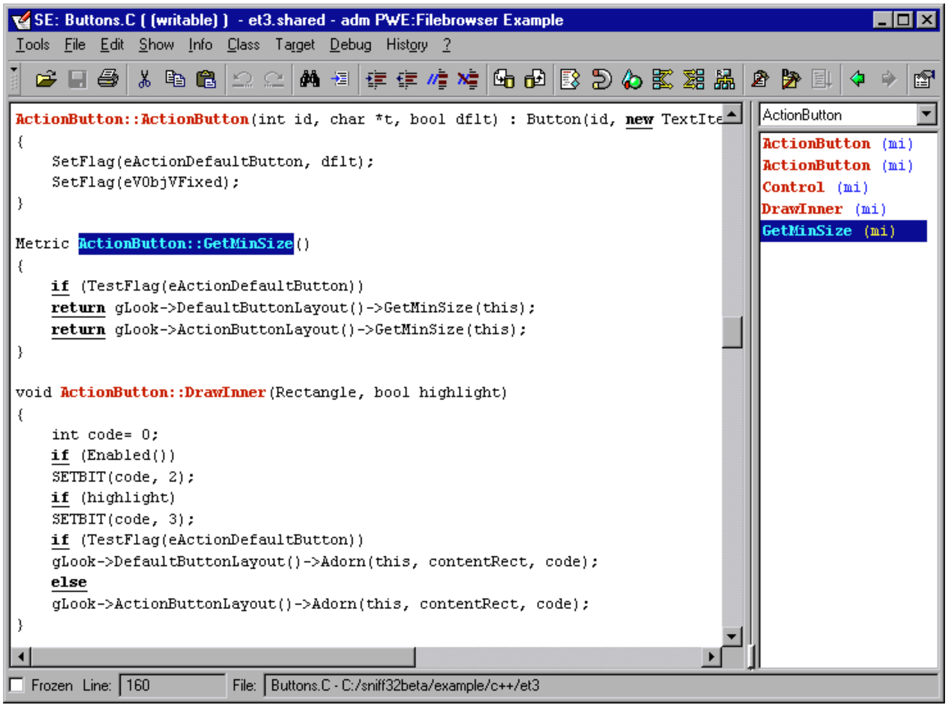
   The `Buttons.h` file is now loaded in the Source Editor.

3. Choose `ActionButton` from the **Class** drop-down.

4. Highlight `GetMinSize` in the Symbol List.

5. Choose **Show > Implementation of GetMinSize**.

   The `Buttons.C` file is loaded into the Source Editor.


SE: Buttons.C ((writable)) - et3.shared - adm PWE:Filebrowser Example
Tools File Edit Show Info Class Target Debug History ?
ActionButton
ActionButton (mi)
ActionButton (mi)
Control (mi)
DrawInner (mi)
GetMinSize (mi)

```
ActionButton::ActionButton(int id, char *t, bool dflt) : Button(id, new TextIte
{
    SetFlag(eActionDefaultButton, dflt);
    SetFlag(eVObjVFixed);
}

Metric ActionButton::GetMinSize()
{
    if (TestFlag(eActionDefaultButton))
    return gLook->DefaultButtonLayout()->GetMinSize(this);
    return gLook->ActionButtonLayout()->GetMinSize(this);
}

void ActionButton::DrawInner(Rectangle, bool highlight)
{
    int code= 0;
    if (Enabled())
    SETBIT(code, 2);
    if (highlight)
    SETBIT(code, 3);
    if (TestFlag(eActionDefaultButton))
    gLook->DefaultButtonLayout()->Adorn(this, contentRect, code);
    else
    gLook->ActionButtonLayout()->Adorn(this, contentRect, code);
}
```

□ Frozen Line: 160 File: Buttons.C - C:/sniff32beta/example/c++/et3

6. Study the method. Note that variable `gLook` is used for a method call.

Going to a variable’s definition

You now want to get information about the declaration and definition of `gLook`. To do so:

1. Double-click on the symbol `gLook` in the Source Editor’s main view.

   `gLook` is now highlighted.

2. Choose **Show > Symbol(s) gLook...**

   SNiFF+ positions the Source Editor to the definition of gLook.
   As you can see the global variable `gLook` is declared to be a pointer of type `Look`. In class `Look` (Look.h) it is referenced as an external variable.

You may now want more information about class `Look`.

1. Double-click on `Look` in the Source Editor’s main view.

   `Look` is now highlighted.

2. Choose **Class > Browse Look**.

   The Class Browser opens.

3. Remember, earlier on you selected **overridden** in the Modifiers drop-down. To see all methods, make sure **All modifiers** is selected in this drop-down.
4. Double-click on method DrawHighlight to load its source code into the Source Editor.
5. Close the Class Browser.

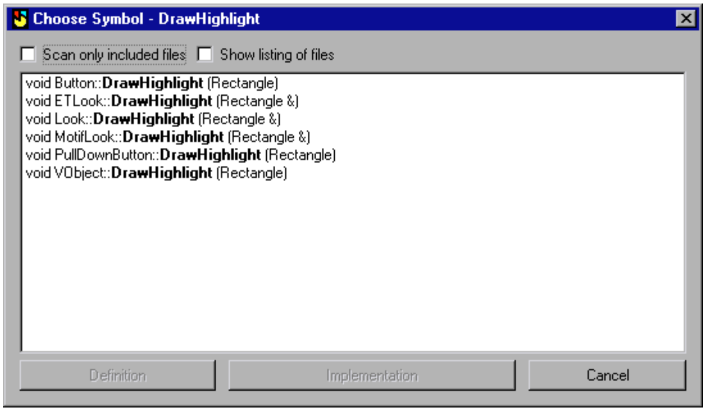
Symbols with the same name

1. Double-click on DrawHighlight in the Source Editor's main view.

   Now only the method DrawHighlight is highlighted.

2. Choose **Show > Symbol(s) DrawHighlight...**.

   SNiFF+ opens the **Choose Symbol - DrawHighlight** dialog presenting the choices. The **Choose Symbol** dialog opens when there is more than one symbol called DrawHighlight. If there is only one symbol called DrawHighlight, the Source Editor is positioned at its implementation.



☐ Scan only included files ☐ Show listing of files

void Button::**DrawHighlight** (Rectangle)
void ETLook::**DrawHighlight** (Rectangle &)
void Look::**DrawHighlight** (Rectangle &)
void MotifLook::**DrawHighlight** (Rectangle &)
void PullDownButton::**DrawHighlight** (Rectangle)
void VObject::**DrawHighlight** (Rectangle)

You have six symbols matching DrawHighlight.

You may now want to see only those symbols that are defined or implemented in the file Look.C and in all header files which are included by Look.C.

- Select the **Scan only included files** check box.

Now only one entry is shown.

- Double-click on void Look::DrawHighlight (Rectangle &).

The symbol is loaded into the Source Editor.

In this chapter you started from ActionButton::GetMinSize and browsed the variable gLook. You then learned more about the class Look, which uses gLook.You then looked at methods which are implemented in Look, e.g. DrawHighlight.

In the next chapter you'll find out where else in your source code gLook is used by using the Retriever, a tool that lets you to query the entire project structure.

Textual search with the Retriever 7

The Retriever is a fast source code retrieval tool with filtering. It can be used to find out where a certain string is used in the source code. It lists all occurrences of strings matching a regular expression in a set of projects. A semantic filter can then be applied to the matches.

The Retriever also allows you to globally find and replace strings in code lines, and to edit code in the integrated Source Editor.

This chapter is about using the Retriever to:

- find every line in your source files containing a given string
- find out where the string is assigned a value
- find out where a string is allocated on a heap

Opening the Retriever

To open the Retriever:

1. Highlight `Look (mi)` in the Symbol List.

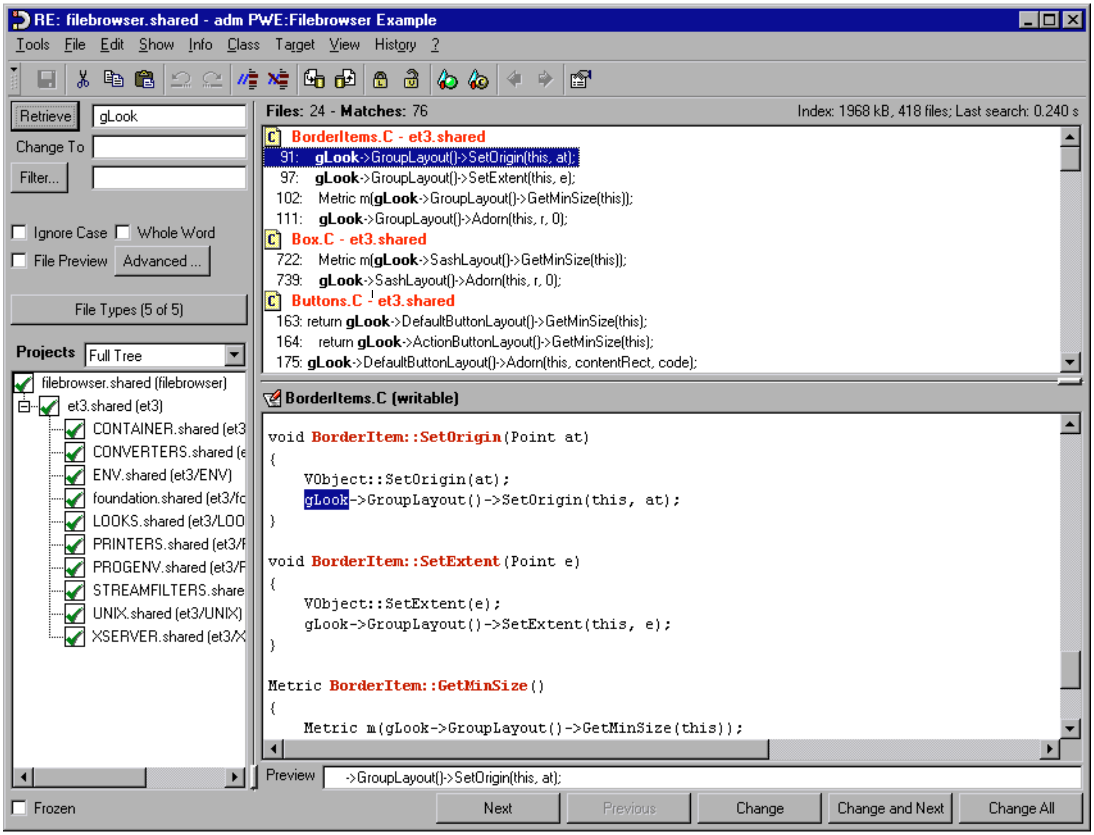
   The Source Editor is positioned at the implementation of `Look`.

2. Scroll up to `Look *gLook` and double-click on `gLook`.

   `gLook` is now highlighted.

3. Choose **Info > Retrieve gLook From All Projects**.

   A progress bar appears indicating that all files are being indexed. The next progress bar shows the progress of the retrieval.
   The Retriever opens.



Retrieve: gLook
Change To:
Filter...:

□ Ignore Case □ Whole Word
□ File Preview

Projects: Full Tree

☑ filebrowser.shared (filebrowser)
☑ et3.shared (et3)
☑ CONTAINER.shared (et3
☑ CONVERTERS.shared (e
☑ ENV.shared (et3/ENV)
☑ foundation.shared (et3/fo
☑ LOOKS.shared (et3/LOO
☑ PRINTERS.shared (et3/P
☑ PROGENV.shared (et3/P
☑ STREAMFILTERS.share
☑ UNIX.shared (et3/UNIX)
☑ XSERVER.shared (et3/X

□ Frozen

Files: 24 - Matches: 76 Index: 1968 kB, 418 files; Last search: 0.240 s

```
BorderItems.C - et3.shared
 91:   gLook->GroupLayout()->SetOrigin(this, at);
 97:   gLook->GroupLayout()->SetExtent(this, e);
102:   Metric m(gLook->GroupLayout()->GetMinSize(this));
111:   gLook->GroupLayout()->Adorn(this, r, 0);
Box.C - et3.shared
722:   Metric m(gLook->SashLayout()->GetMinSize(this));
739:   gLook->SashLayout()->Adorn(this, r, 0);
Buttons.C - et3.shared
163: return gLook->DefaultButtonLayout()->GetMinSize(this);
164:   return gLook->ActionButtonLayout()->GetMinSize(this);
175: gLook->DefaultButtonLayout()->Adorn(this, contentRect, code);
```

BorderItems.C (writable)

```
void BorderItem::SetOrigin(Point at)
{
    VObject::SetOrigin(at);
    gLook->GroupLayout()->SetOrigin(this, at);
}

void BorderItem::SetExtent(Point e)
{
    VObject::SetExtent(e);
    gLook->GroupLayout()->SetExtent(this, e);
}

Metric BorderItem::GetMinSize()
{
    Metric m(gLook->GroupLayout()->GetMinSize(this));
```

Preview: ->GroupLayout()->SetOrigin(this, at);

4. Close the Source Editor.

When the Retriever first appears after being asked to retrieve the string “`gLook`” from all projects, notice that:

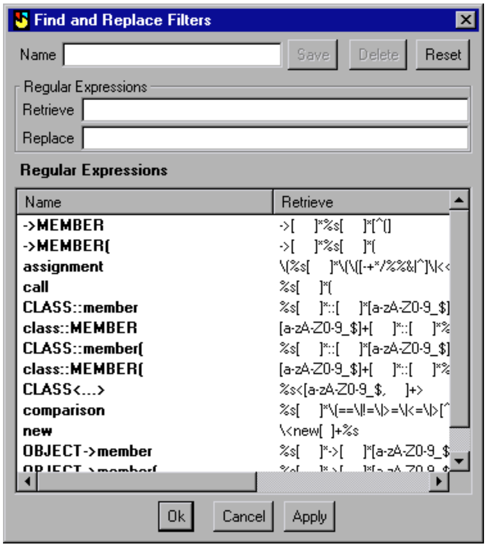
- Above the Files — Matches List, you are informed that 76 matches were found in 24 source files. Each match is listed (in bold print) in its source line context.
- The **Ignore Case** check box is not selected - this means that the search is case-sensitive.
- The **Whole Word** check box is not selected - this means that the compound words with `gLook` as a substring are also retrieved.
- All the projects in the Project Tree are selected. This is because you opened the Retriever with the command: **Info > Retrieve gLook From All Projects**.

Finding out where a symbol is assigned a value

You may now want to restrict the list to places where `gLook` is assigned a value.

1. Press the **Filter...** button.

   The Find and Replace Filters dialog appears.



Name	Retrieve
->MEMBER	->[ ]*%s[ ]*[^(]
->MEMBER(	->[ ]*%s[ ]*(
assignment	\(%s[ ]*\(\([-+*/%%&|^]\\|<<
call	%s[ ]*(
CLASS::member	%s[ ]*::[ ]*[a-zA-Z0-9_$]
class::MEMBER	[a-zA-Z0-9_$]+[ ]*::[ ]*%
CLASS::member(	%s[ ]*::[ ]*[a-zA-Z0-9_$]
class::MEMBER(	[a-zA-Z0-9_$]+[ ]*::[ ]*%
CLASS<...>	%s<[a-zA-Z0-9_$,]+>
comparison	%s[ ]*\(==\\|!=\\|>=\\|<=\\|>[^
new	\<new[ ]+%s
OBJECT->member	%s[ ]*->[ ]*[a-zA-Z0-9_$
OBJECT->member(	%s[ ]*->[ ]*[a-zA-Z0-9_$

2. Select **assignment** from the regular expression list.
3. Press **Ok**.

   As you can see in the Files — Matches List, there are two locations in your project where `gLook` is assigned a value.

The Retriever uses a two-stage filtering process:

- First all lines matching the search string are extracted.
- Then the list is once more restricted using regular expressions (in this case a regular expression representing the syntax of an assignment).

You can look at the source code of the matches in the Code Display:

- Click on a match.

  The Code Display is positioned at the source code of the selected match. If the file that you are browsing is writable, then you can also edit the source code in the Code Display.

Retrieving a string from all projects

The Retriever is a very powerful tool for formulating fuzzy queries.
Let’s try this out by getting information about menu handling. To do so:

1. Select the **Ignore Case** check box.

   The search is now no longer case-sensitive.

2. Delete the filter in the **Filter** field.
3. In the **Retrieve** field type `menu`, then press **Retrieve**.

   As you can see, there are 1358 matches.

> **Note**
>
> For the first query an index is generated for the checkmarked projects, subsequent queries are then much faster.

Let’s further restrict the search using the assignment filter:

1. Press the **Filter...** button.

   The Find and Replace Filters dialog appears.

2. Select **assignment** from the regular expression list.
3. Press **Ok**.

   Notice that there are 72 matches in the project, where a variable called `menu` (or similar) is assigned a value.

Where is a symbol allocated on the heap?

Let’s find out where `menu` (or similar) is allocated on the heap. To do so:

1. Press the **Filter...** button.

   The Find and Replace Filters dialog appears.

2. Select **new** from the regular expression list.
3. Press **Ok**.

   Notice there are 99 matches in the Filebrowser project where `menu` (or similar) has been allocated. Each match is displayed as a separate line in the Retriever, even if two matches are located in the same source line.
   To look at the source code of one of the matches in the Source Editor:

- double-click on the first entry in the Retriever.

  The Source Editor opens and is positioned at the reference of the selected entry.

- Close the Retriever.

  Although the Retriever can help you search for strings in your source code, it can only provide limited cross reference information. In the next chapter you will be working with the Cross Referencer to get more cross reference information about `Menu`.

Code Dependencies and Impact analysis 8

This chapter is about using the Cross Referencer to:

- look at code dependencies
- look at where functions and methods are called

Opening the Cross Referencer

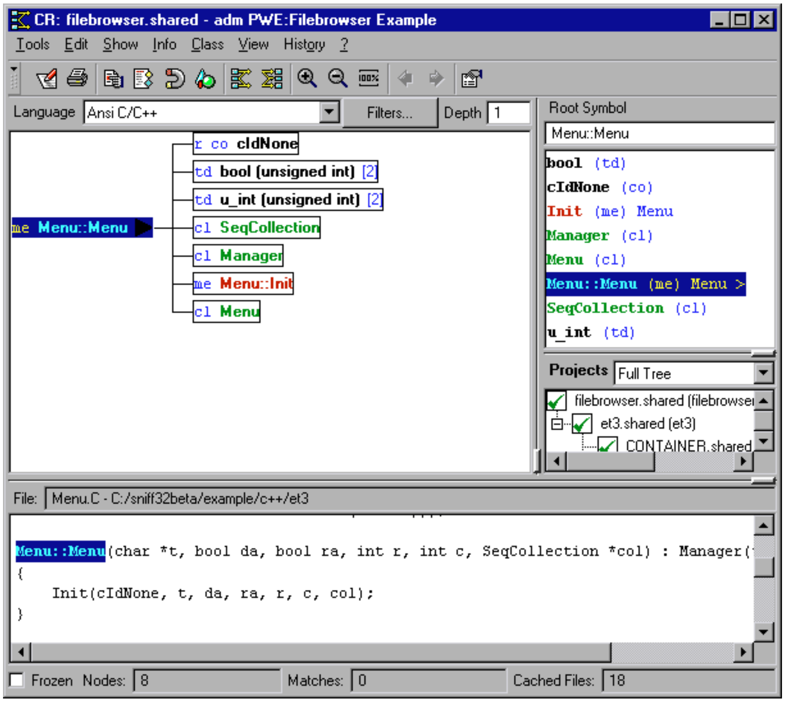
To open the Cross Referencer:

1. In the Source Editor, choose **Info > Menu Refers-To**.

   The Choose Symbol - Menu dialog appears.

2. Double-click on the first entry.

   The Cross Referencer opens.



CR: filebrowser.shared - adm PWE:Filebrowser Example
Tools Edit Show Info Class View History ?
Filters...
me Menu::Menu
r co cldNone
td bool (unsigned int) [2]
td u_int (unsigned int) [2]
cl SeqCollection
cl Manager
me Menu::Init
cl Menu

Language Ansi C/C++ Depth 1

Root Symbol Menu::Menu

bool (td)
cIdNone (co)
Init (me) Menu
Manager (cl)
Menu (cl)
Menu::Menu (me) Menu >
SeqCollection (cl)
u_int (td)

Projects Full Tree

☑ filebrowser.shared (filebrowser
☑ et3.shared (et3)
☑ CONTAINER.shared

File: Menu.C - C:/sniff32beta/example/c++/et3

```
Menu::Menu(char *t, bool da, bool ra, int r, int c, SeqCollection *col) : Manager(
{
    Init(cIdNone, t, da, ra, r, c, col);
}
```

□ Frozen Nodes: 8 Matches: 0 Cached Files: 18

3. Close the Source Editor.

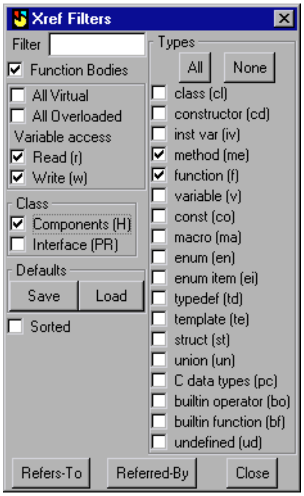
Code Dependencies

As you can see in the Graph view, `Menu::Menu` is taken as the root symbol and symbols it refers to are displayed as its nodes. The Reference view shows all types that are referenced by `Menu::Menu`, even classes.

Let's filter the list to show only functions and methods. To do so:

1. In the **Depth** field enter `<2>`.
2. Press the **Filters...** button.

   The Xref Filter dialog appears.
3. Press the **None** button at the top-right to deselect all types.
4. Select the `method(me)` and `function(f)`checkboxes.

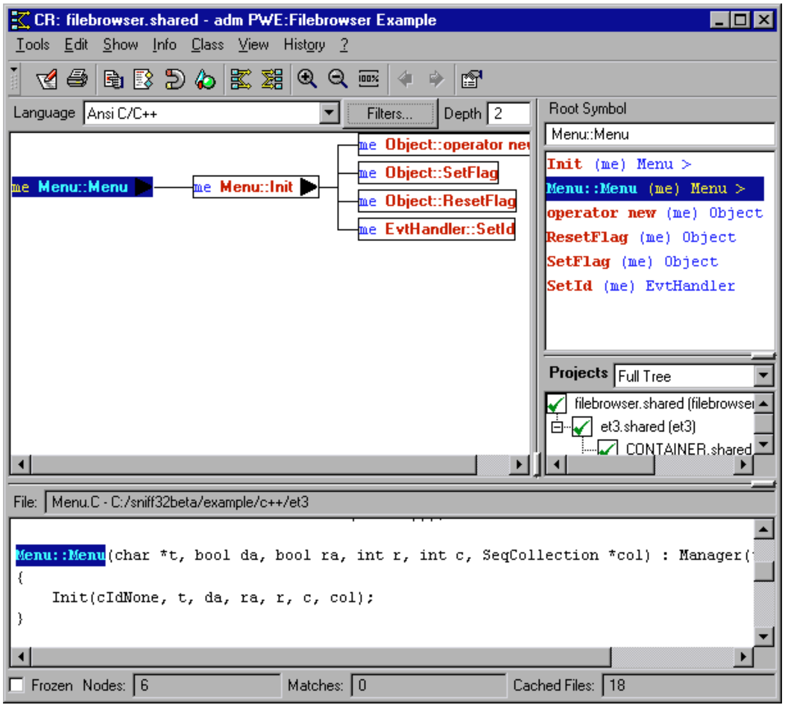


☑ Function Bodies
□ All Virtual
□ All Overloaded
Variable access
☑ Read (r)
☑ Write (w)
Class
☑ Components (H)
□ Interface (PR)
□ Sorted

□ class (cl)
□ constructor (cd)
□ inst var (iv)
☑ method (me)
☑ function (f)
□ variable (v)
□ const (co)
□ macro (ma)
□ enum (en)
□ enum item (ei)
□ typedef (td)
□ template (te)
□ struct (st)
□ union (un)
□ C data types (pc)
□ builtin operator (bo)
□ builtin function (bf)
□ undefined (ud)

5. Press **Refers-To**.

The function-call tree to the depth of 2 is shown. Only forward references to methods and functions (in this case none) are displayed.



Language: Ansi C/C++

Depth: 2

Root Symbol: Menu::Menu

```
Init (me) Menu >
Menu::Menu (me) Menu >
operator new (me) Object
ResetFlag (me) Object
SetFlag (me) Object
SetId (me) EvtHandler
```

Projects: Full Tree

- ✓ filebrowser.shared (filebrowse
- ✓ et3.shared (et3)
- ✓ CONTAINER.shared

File: Menu.C - C:/sniff32beta/example/c++/et3

```
Menu::Menu(char *t, bool da, bool ra, int r, int c, SeqCollection *col) : Manager(
{
    Init(cIdNone, t, da, ra, r, c, col);
}
```

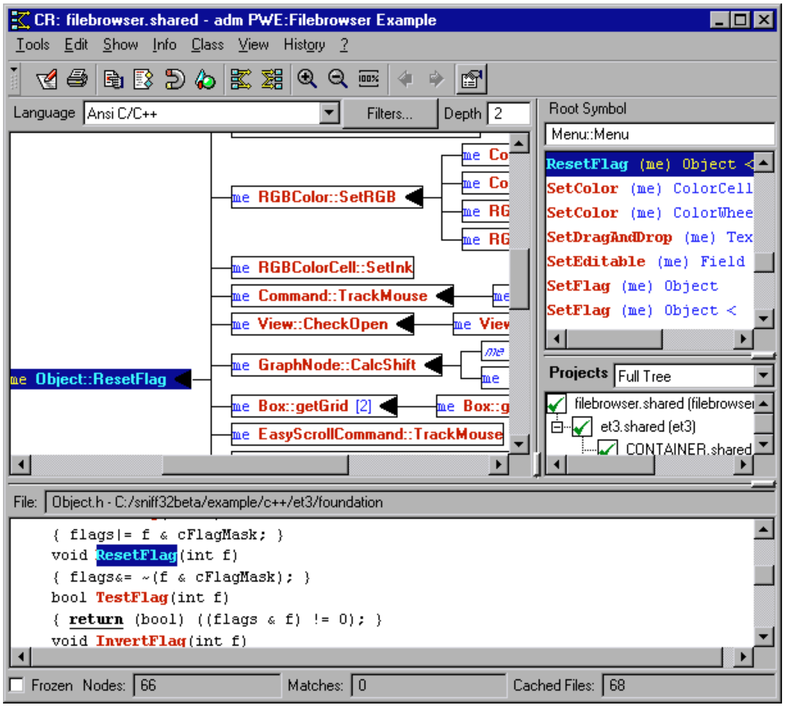
□ Frozen Nodes: 6 Matches: 0 Cached Files: 18

Impact Analysis: Studying function calls

You may want to see where `me Object::ResetFlag` is called:

1. Highlight `me Object::ResetFlag` in the newly created function-call tree.
2. Choose **Context menu > ResetFlag Referred-By**.

   Scroll around. As you can see in the illustration below, the backward references of `me Object::ResetFlag` are added to the graph. All methods that access this method, as well as all methods that call these methods, are shown.



CR: filebrowser.shared - adm PWE:Filebrowser Example

Language: Ansi C/C++ Depth: 2

Root Symbol: Menu::Menu

Projects: Full Tree

File: Object.h - C:/sniff32beta/example/c++/et3/foundation

```
{ flags|= f & cFlagMask; }
void ResetFlag(int f)
{ flags&= ~(f & cFlagMask); }
bool TestFlag(int f)
{ return (bool) ((flags & f) != 0); }
void InvertFlag(int f)
```

□ Frozen Nodes: 66 Matches: 0 Cached Files: 68

Let’s now quickly visit the locations that are displayed in the Cross Referencer. To do so:

1. `<SHIFT>`click on a method.

   The Code view is positioned at the first reference to the method.
2. `<SHIFT>`double-click on a method.

   A Source Editor appears, and the first reference is displayed in it.
3. Position the Source Editor and the Cross Referencer on your screen so that you can see both.
4. In the Source Editor, choose **Show > Next Match**.

   The Source Editor positions to the next reference and the Cross Referencer’s Graph view is positioned at the next match.
5. To see all locations, repeat the previous step until there are no more matches.

Understanding Include Dependencies 9

The Include Browser graphically displays include references made in project source files. It can be used to see which files are included by a particular file and vice-versa, as well as to make sure that there are no redundant includes.

This chapter is about using the Include Browser to:

- find out which header files are included by a particular implementation file
- find out which implementation files include a particular header file

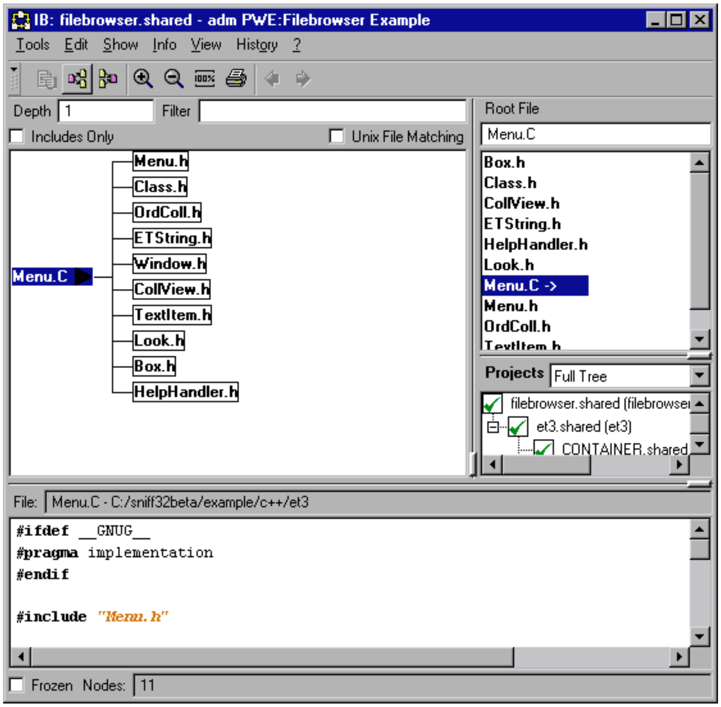
Opening the Include Browser

To open the Include Browser:

1. In the Cross Referencer, highlight the root symbol `me Menu::Menu`.
2. From the menu, choose **Show > Implementation of Menu**.

   The Source Editor is positioned at the implementation of Menu.
3. In the Source Editor, choose **Info > Menu.C Includes**.

The Include Browser opens.


IB: filebrowser.shared - adm PWE:Filebrowser Example
Tools Edit Show Info View History ?
Includes Only
Unix File Matching
Projects
Frozen

Depth 1 Filter

Root File Menu.C

Menu.C → Menu.h, Class.h, OrdColl.h, ETString.h, Window.h, CollView.h, TextItem.h, Look.h, Box.h, HelpHandler.h

Box.h
Class.h
CollView.h
ETString.h
HelpHandler.h
Look.h
Menu.C ->
Menu.h
OrdColl.h
TextItem.h

Projects: Full Tree

☑ filebrowser.shared (filebrowser
☑ et3.shared (et3)
☑ CONTAINER.shared

File: Menu.C - C:/sniff32beta/example/c++/et3

```
#ifdef __GNUG__
#pragma implementation
#endif

#include "Menu.h"
```

□ Frozen Nodes: 11

4. Close the Source Editor and the Cross Referencer.

Files included by a particular file

As you can see in the above illustration, `Menu.C` includes ten header files. Let’s look at the include statement of one of the header files. To do so:

1. Highlight `Menu.h` in the newly created tree.

2. Choose **Context menu > Show Include Statement**.

The Source Editor opens and is positioned at the include statement of `Menu.h`.

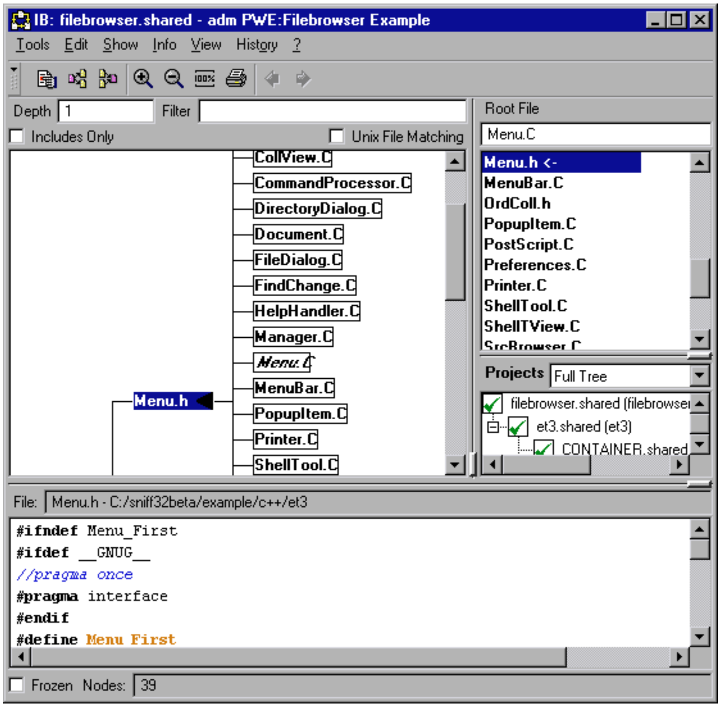
3. Close the Source Editor.

Files that include a particular file

Let’s see which files include the header file `Menu.h`. To get this information:

1. In the Include Browser, make sure that `Menu.h` is highlighted in Graph view.
2. Choose **Context menu > Included-By**.

   In the illustration below, you can see the files which include `Menu.h`.



Depth: 1

Filter:

Root File: Menu.C

Menu.h <-
MenuBar.C
OrdColl.h
PopupItem.C
PostScript.C
Preferences.C
Printer.C
ShellTool.C
ShellTView.C
SrcBrowser.C

Projects: Full Tree

File: Menu.h - C:/sniff32beta/example/c++/et3

```
#ifndef Menu_First
#ifdef __GNUG__
//pragma once
#pragma interface
#endif
#define Menu First
```

□ Frozen Nodes: 39

Let’s now look at which projects these files belong to. To do so:

- Choose **View > Show Project Name**.

  Scroll to the right of the Graph view. You can now see the file names as well as the projects to which they belong.

Browsing Documentation 10

The Documentation Editor supports the iterative and incremental generation, writing and maintenance of source code documentation. You can use its hypertext-like browser to quickly navigate between source and documentation. In addition, you can freely define the structure and contents of the generated documentation.

In this chapter you learn how to:

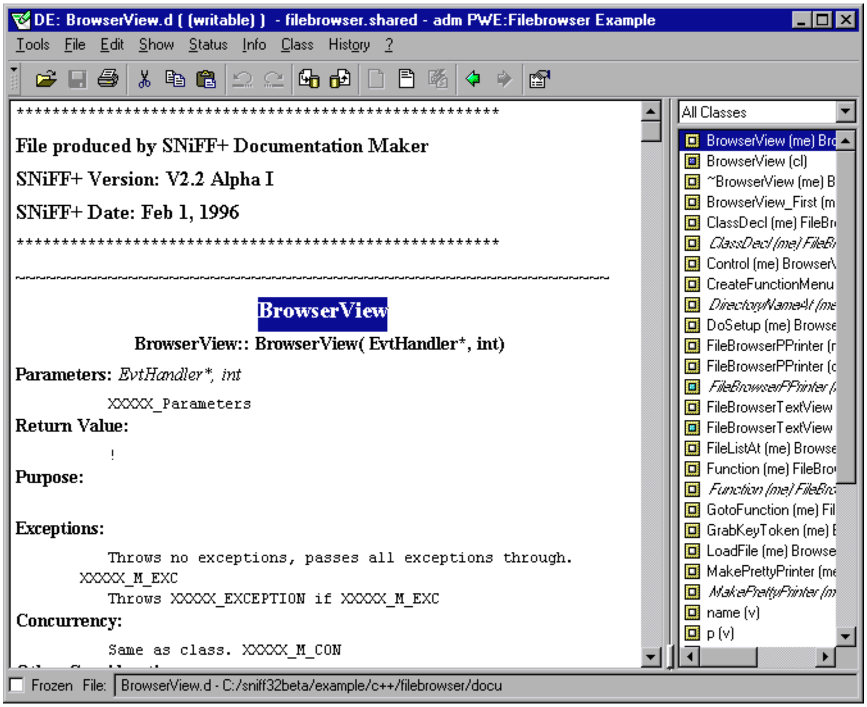
- browse the documentation of a particular file.
- browse the documentation of a particular symbol.

Opening the Documentation Editor

To open the Documentation Editor:

- In any open SNiFF+ tool, choose **Tools > Project Editor**.
- Make sure that `filebrowser.shared` is checkmarked in the Project Tree.
- In the file list, double-click on **BrowserView.d**.

The Documentation Editor opens.



```
******************************************************
File produced by SNiFF+ Documentation Maker
SNiFF+ Version: V2.2 Alpha I
SNiFF+ Date: Feb 1, 1996
******************************************************

                         BrowserView
              BrowserView:: BrowserView( EvtHandler*, int)
Parameters: EvtHandler*, int
          XXXXX_Parameters
Return Value:
          !
Purpose:

Exceptions:
          Throws no exceptions, passes all exceptions through.
       XXXXX_M_EXC
          Throws XXXXX_EXCEPTION if XXXXX_M_EXC
Concurrency:
          Same as class. XXXXX_M_CON
```

□ Frozen File: BrowserView.d - C:/sniff32beta/example/c++/filebrowser/docu

As you can see, the documentation file `BrowserView.d` is loaded in the Documentation Editor and the Documentation Editor is positioned to the beginning of the documentation file. Clicking on the items in the Symbol List positions the Documentation Editor to the respective symbol documentation. Note that the icons that precede the symbols in the Symbol List indicate what their documentation status is.

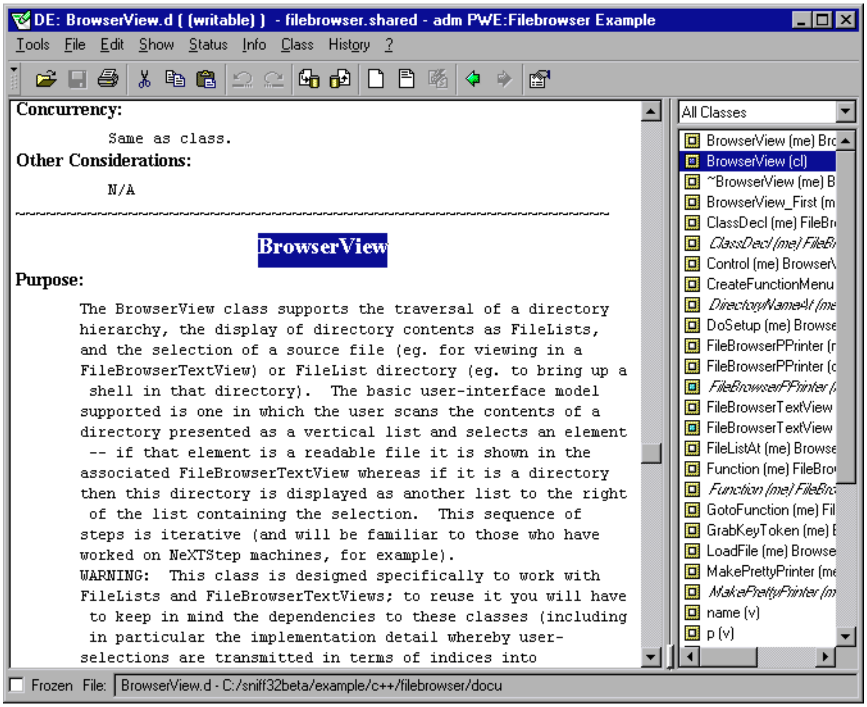
To see what the icons indicate, choose **Help(?) > Quick Ref**.

Viewing documentation of a class

You may now want to look at the documentation of the class `BrowserView`. To do so:

- In the Symbol List, click on `BrowserView (cl)`.

  You now see the documentation frame for class `BrowserView`.



Concurrency:

Same as class.

Other Considerations:

N/A

~~~~~~

BrowserView

Purpose:

The BrowserView class supports the traversal of a directory hierarchy, the display of directory contents as FileLists, and the selection of a source file (eg. for viewing in a FileBrowserTextView) or FileList directory (eg. to bring up a shell in that directory). The basic user-interface model supported is one in which the user scans the contents of a directory presented as a vertical list and selects an element -- if that element is a readable file it is shown in the associated FileBrowserTextView whereas if it is a directory then this directory is displayed as another list to the right of the list containing the selection. This sequence of steps is iterative (and will be familiar to those who have worked on NeXTStep machines, for example).
WARNING: This class is designed specifically to work with FileLists and FileBrowserTextViews; to reuse it you will have to keep in mind the dependencies to these classes (including in particular the implementation detail whereby user-selections are transmitted in terms of indices into

- Close the Documentation Editor.
~~~~~~

Review

This was the last of the SNiFF+ browsing tools to be introduced in this tutorial.
In this part of the C++ Tutorial you were introduced to:

- Opening the project
- Top-Down browsing with the following tools

  Symbol Browser
  Hierarchy Browser
  Class Browser
  Cross Referencer
- Bottom-Up browsing with the following tools

  Source Editor
  Retriever
  Cross Referencer
  Include Browser
  Documentation Editor
- In the tutorial we can only give you a few hints and tips to help you on your way. To get a better understanding of the tools, we recommend experimenting. To really appreciate what SNiFF+ can do for you, you really need to work with it.

How to set up your own Browsing-Only Projects

If you want to browse your own project, please refer to Setting up Browsing-Only Projects — page 75.

What’s next

The only SNiFF+ tool that should now be open is the Launch Pad. We will continue using the Filebrowser project in the version controlling part.

- The next part of this tutorial introduces you to Version Controlling

Part III

Version Controlling

File history and locking information 1

SNiFF+'s configuration management and version control (CMVC) support provides the functionality available in the RCS version control system. This tutorial assumes that you are using RCS. If you are using a different CMVC tool, please refer to the User's Guide.

In this chapter you will look at:

- a file's history information
- locking information

Checking whether RCS is in your path (Unix Only)

The filebrowser example comes with an RCS repository. This tutorial relies on RCS 5.7 which is supplied together with the SNiFF+ package and is installed from there.

To check whether the correct version of RCS is in your path:

1. Open a Unix shell.
2. Enter `% rcs -V1` at the command prompt.

   You will receive **one** of the following outputs:

   - `rcs error: -V1 out of range 3..5`

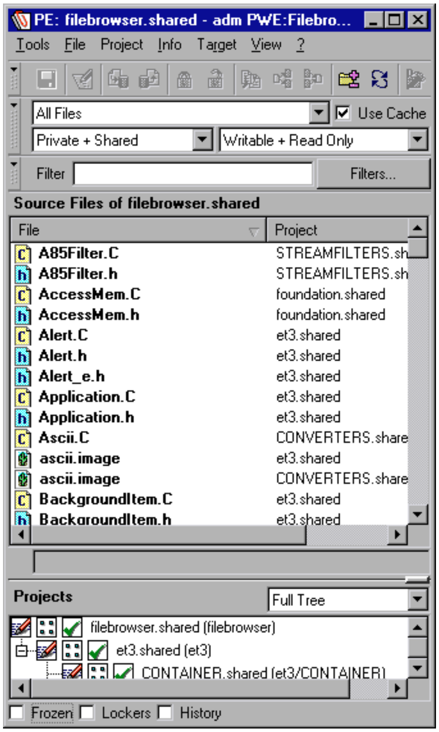
   This shows that the **correct version** of RCS is installed.

   - `rcs error: Unknown option: -V1`

   This shows that an old, **unusable version** of RCS is installed. If you get this output, please install RCS 5.7 to execute the version control steps in the next chapter.

Opening the Project Editor

- In the Launch Pad, highlight `Filebrowser.shared`.
- Choose **Tools > Project Editor**.



All Files ☑ Use Cache

Private + Shared | Writable + Read Only

Filter

Source Files of filebrowser.shared

File	Project
A85Filter.C	STREAMFILTERS.sh
A85Filter.h	STREAMFILTERS.sh
AccessMem.C	foundation.shared
AccessMem.h	foundation.shared
Alert.C	et3.shared
Alert.h	et3.shared
Alert_e.h	et3.shared
Application.C	et3.shared
Application.h	et3.shared
Ascii.C	CONVERTERS.share
ascii.image	et3.shared
ascii.image	CONVERTERS.share
BackgroundItem.C	et3.shared
BackgroundItem.h	et3.shared

Projects Full Tree

filebrowser.shared (filebrowser)
et3.shared (et3)
CONTAINER.shared (et3/CONTAINER)

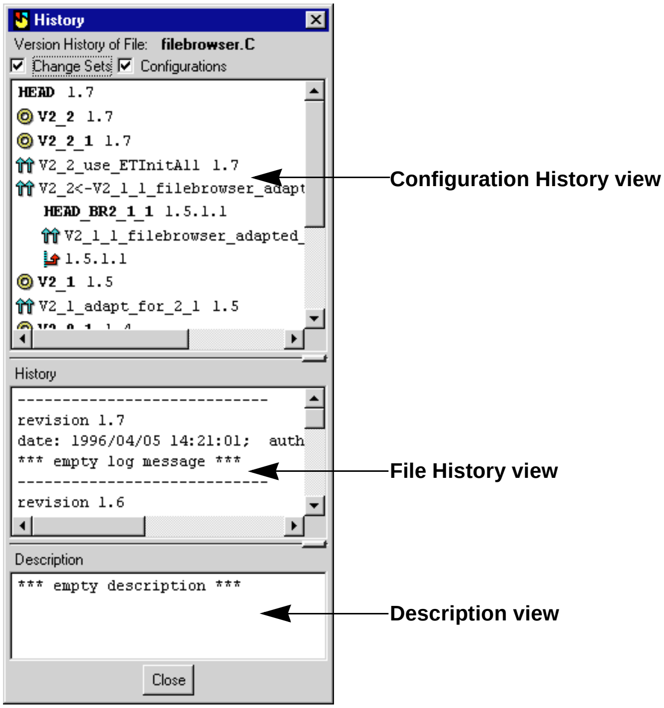
□ Frozen □ Lockers □ History

File’s history information

Lets look at the history information of `filebrowser.C`. To do so:

1. Highlight `filebrowser.C` in the File List of the Project Editor.
2. Select the **History** check box.

A History window appears to the right of the Project Editor. The History window contains three views that show the history information of the selected file. To see a description of the icons used, choose **Help(?) > QuickRef**.



History

Version History of File: **filebrowser.C**

☑ Change Sets ☑ Configurations

```
HEAD  1.7
V2_2  1.7
V2_2_1  1.7
V2_2_use_ETInitAll  1.7
V2_2<-V2_1_1_filebrowser_adapt
   HEAD_BR2_1_1  1.5.1.1
   V2_1_1_filebrowser_adapted_
   1.5.1.1
V2_1  1.5
V2_1_adapt_for_2_1  1.5
```

History

```
----------------------------
revision 1.7
date: 1996/04/05 14:21:01;  auth
*** empty log message ***
----------------------------
revision 1.6
```

Description

```
*** empty description ***
```

Configuration History view

File History view

Description view

All versions of `filebrowser.C` as stored and maintained by your version control tool are displayed. In the **History** view, you can see the complete history of `filebrowser.C`.

To see the history information of a particular version of `filebrowser.C`:

- Highlight `HEAD 1.7` in the **Configuration History** view.

  You can now only see the history record (stored and maintained by your version control tool) of `HEAD 1.7` in the **File History** view.

- Close the History window.

Displaying locking information

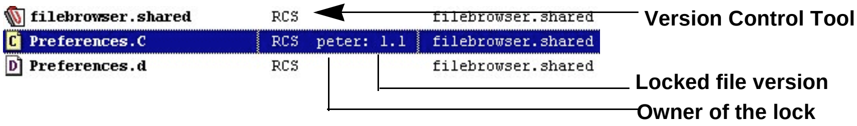
You may now want to see which files are locked. To do so:

1. Highlight `filebrowser.shared` in the Project Tree.
2. Choose **Context menu > Select from filebrowser.shared Only.**
3. In the Project Editor, select the **Lockers** check box.

   In the File List, the Lockers column appears showing locking information.

4. Look at the file Preferences.C in the File List.

   In the illustration below, you can see that the version control tool that is used is RCS, the file is locked by Peter and the locked version number is 1.1



```
filebrowser.shared    RCS                 filebrowser.shared
Preferences.C         RCS   peter: 1.1    filebrowser.shared
Preferences.d         RCS                 filebrowser.shared
```

Version Control Tool

Locked file version

Owner of the lock

Configuration Management 2

The Configuration Manager gives a structural and file-based overview of the changes between two configurations of a software system. Configurations are selected file versions grouped together under the same symbolic name.

In this chapter you will:

- look at the configurations in `filebrowser.shared`
- compare two configurations

Opening the Configuration Manager

1. In the Project Editor, choose **Tools > Configuration Manager**.

   The Configuration Manager opens.

2. Close the Project Editor.

Looking at configurations

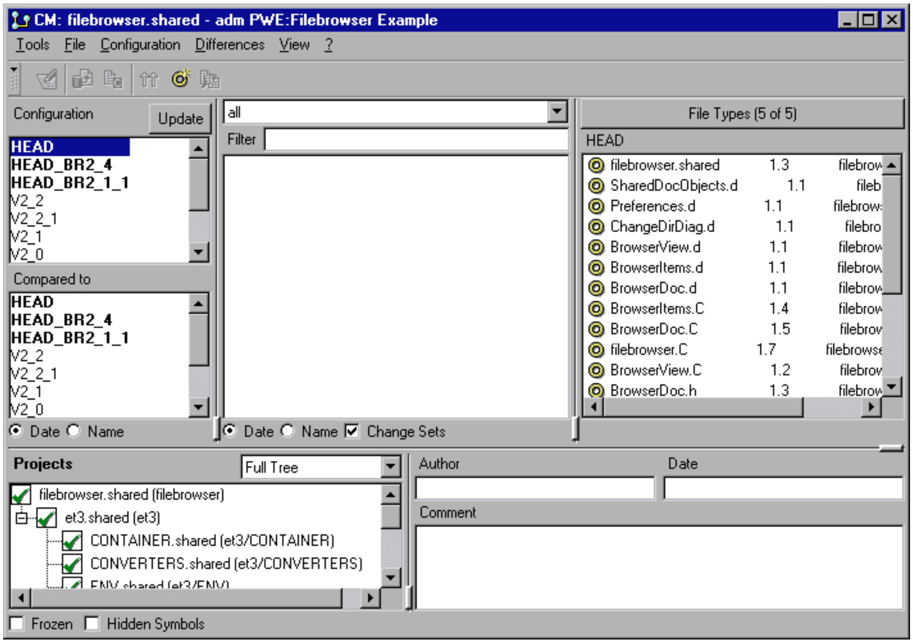
You may now want to look at the configurations of `filebrowser.shared`. To do so:

1. Highlight `filebrowser.shared` in the Project Tree of the Configuration Manager.
2. Choose **Context menu > Select from filebrowser.shared Only.**

   The configuration information for `filebrowser.shared` is shown in the Configuration List.

3. Highlight HEAD in the Configuration List.

HEAD is the symbolic name of the latest version of all files in a particular configuration. The HEAD configuration thus reflects the current state of your software system. In the illustration below, you can see all files that are part of HEAD.


CM: filebrowser.shared - adm PWE:Filebrowser Example
Tools File Configuration Differences View ?
Configuration
Update
all
Filter
File Types (5 of 5)
Compared to
Projects
Full Tree
Author
Date
Comment

Configuration:
HEAD
HEAD_BR2_4
HEAD_BR2_1_1
V2_2
V2_2_1
V2_1
V2_0

Compared to:
HEAD
HEAD_BR2_4
HEAD_BR2_1_1
V2_2
V2_2_1
V2_1
V2_0

● Date ○ Name

● Date ○ Name ☑ Change Sets

HEAD

filebrowser.shared	1.3	filebrow
SharedDocObjects.d	1.1	fileb
Preferences.d	1.1	filebrow:
ChangeDirDiag.d	1.1	filebro
BrowserView.d	1.1	filebrow
BrowserItems.d	1.1	filebrow
BrowserDoc.d	1.1	filebrow
BrowserItems.C	1.4	filebrow
BrowserDoc.C	1.5	filebrov
filebrowser.C	1.7	filebrowse
BrowserView.C	1.2	filebrov
BrowserDoc.h	1.3	filebrow

☑ filebrowser.shared (filebrowser)
☑ et3.shared (et3)
☑ CONTAINER.shared (et3/CONTAINER)
☑ CONVERTERS.shared (et3/CONVERTERS)
☑ ENV.shared (et3/ENV)

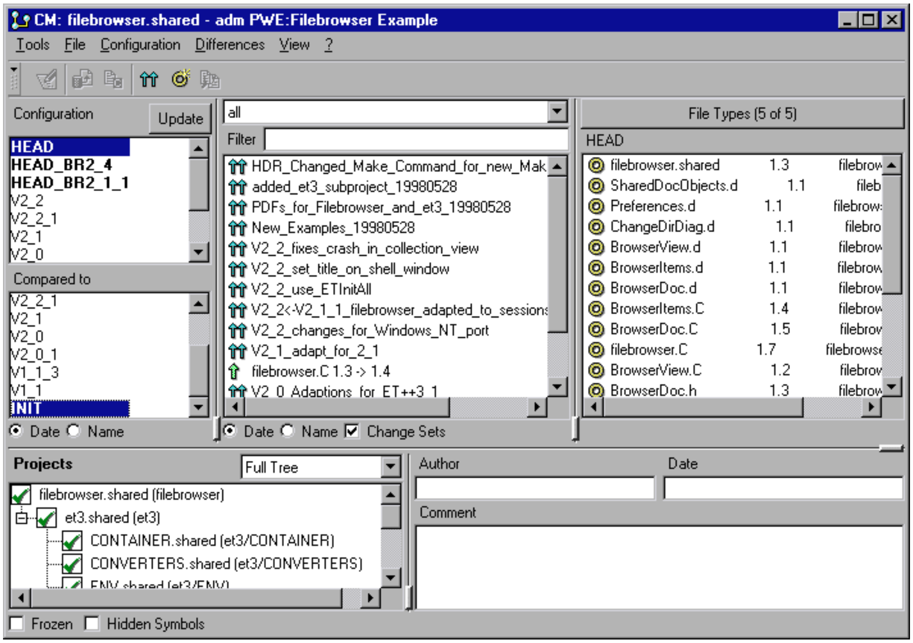
□ Frozen □ Hidden Symbols

Comparing two configurations

Let’s view the changes that occured from the initial configuration (INIT) to the latest configuration (HEAD) of `filebrowser.shared`:

1. Make sure that HEAD is highlighted in the Configuration List.
2. Scroll down the Compared to List and highlight `INIT`.

   As you can see, the differences between the two configurations are displayed in the Change List. Icons in the Change List indicate the nature of the difference.
   To see a description of the icons, choose **Help(?) > QuickRef**.



CM: filebrowser.shared - adm PWE:Filebrowser Example
Tools File Configuration Differences View ?
Configuration
Update
Compared to
● Date ○ Name
all
Filter
● Date ○ Name ☑ Change Sets
File Types (5 of 5)
HEAD
Projects
Full Tree
Author
Date
Comment
☐ Frozen ☐ Hidden Symbols

Configuration:
HEAD
HEAD_BR2_4
HEAD_BR2_1_1
V2_2
V2_2_1
V2_1
V2_0

Compared to:
V2_2_1
V2_1
V2_0
V2_0_1
V1_1_3
V1_1
INIT

HDR_Changed_Make_Command_for_new_Mak
added_et3_subproject_19980528
PDFs_for_Filebrowser_and_et3_19980528
New_Examples_19980528
V2_2_fixes_crash_in_collection_view
V2_2_set_title_on_shell_window
V2_2_use_ETInitAll
V2_2<-V2_1_1_filebrowser_adapted_to_sessions
V2_2_changes_for_Windows_NT_port
V2_1_adapt_for_2_1
filebrowser.C 1.3 -> 1.4
V2_0_Adaptions_for_ET++3_1

```
filebrowser.shared    1.3   filebrow
SharedDocObjects.d      1.1   fileb
Preferences.d         1.1   filebrow:
ChangeDirDiag.d         1.1   filebro
BrowserView.d         1.1   filebrow
BrowserItems.d        1.1   filebrow
BrowserDoc.d          1.1   filebrow
BrowserItems.C        1.4   filebrow
BrowserDoc.C          1.5   filebrov
filebrowser.C       1.7   filebrowse
BrowserView.C         1.2   filebrov
BrowserDoc.h          1.3   filebrow
```

Projects:
☑ filebrowser.shared (filebrowser)
⊟ ☑ et3.shared (et3)
　☑ CONTAINER.shared (et3/CONTAINER)
　☑ CONVERTERS.shared (et3/CONVERTERS)
　☑ ENV.shared (et3/ENV)

You may now want to look more closely at a change set in the Change List. To do so:

1. Choose **change sets** from the drop-down at the top-center of the tool.

   Now only change sets are displayed in the Change List.
2. Highlight `V2_2_changes_for_Windows_NT_port` in the Change List.

   The files of `V2_2_changes_for_Windows_NT_port` are now displayed in the File

Differences between versions of files 3

The Diff/Merge tool shows and merges differences between files and between versions of files. It uses symbol information extracted from source files to highlight the location of the differences. The Diff/Merge tool handles two- or three-way differences.

In this chapter you will:

- view the differences between versions of files

Opening the Diff/Merge Tool

1. In the Configuration Manager, make sure that the following are highlighted:

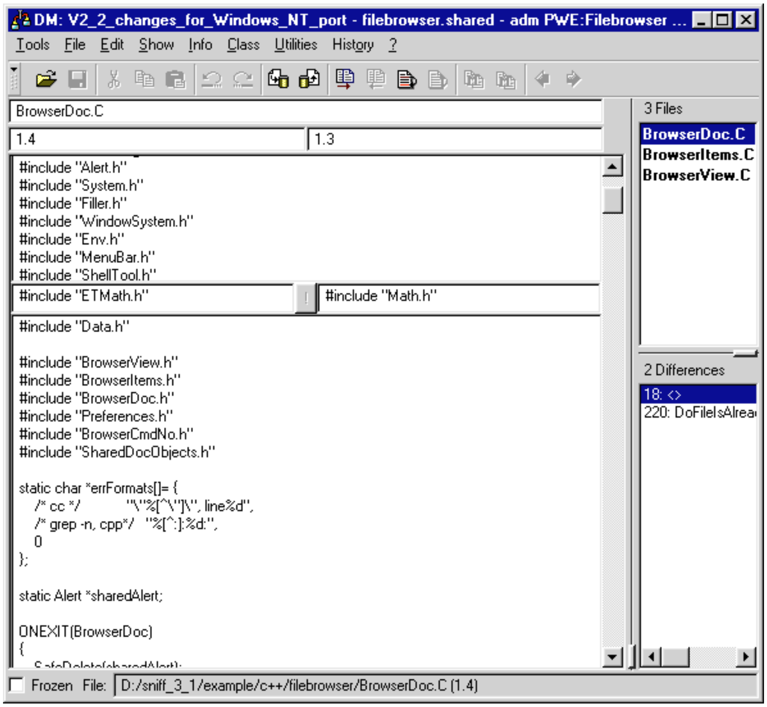
   `HEAD`, `INIT` and `V2_2_changes_for_Windows_NT_port`.

2. Choose **Differences > Show Differences...**

   A Show Differences Dialog appears, in which you can select the type of differences (two-way or three-way) that you want to see.

3. Press the **2-Way** button.

The Diff/Merge tool appears.


DM: V2_2_changes_for_Windows_NT_port - filebrowser.shared - adm PWE:Filebrowser ...
Tools File Edit Show Info Class Utilities History ?
3 Files
2 Differences
Frozen
File:

BrowserDoc.C

1.4 | 1.3

```
#include "Alert.h"
#include "System.h"
#include "Filler.h"
#include "WindowSystem.h"
#include "Env.h"
#include "MenuBar.h"
#include "ShellTool.h"
#include "ETMath.h"          |  #include "Math.h"
#include "Data.h"

#include "BrowserView.h"
#include "BrowserItems.h"
#include "BrowserDoc.h"
#include "Preferences.h"
#include "BrowserCmdNo.h"
#include "SharedDocObjects.h"

static char *errFormats[]= {
   /* cc */          "\"%[^\"]\", line%d",
   /* grep -n, cpp*/   "%[^:]:%d:",
   0
};

static Alert *sharedAlert;

ONEXIT(BrowserDoc)
{
   SafeDelete(sharedAlert);
```

BrowserDoc.C
BrowserItems.C
BrowserView.C

18: <>
220: DoFileIsAlrea

D:/sniff_3_1/example/c++/filebrowser/BrowserDoc.C (1.4)

4. Close the Configuration Manager.

Differences between two file versions

In the above illustration, you can see the differences between `BrowserDoc.C 1.4` (the file in the change set `V2_2_changes_for_Windows_NT_port`) and its previous version `BrowserDoc.C 1.3`.

1. Drag the layout handle to the left to increase the width of the File List.
2. Highlight `220: DoFileIsAlreadyOpen (mi) BrowserDocument` in the Differences List.

   In the two file versions, the difference in the method `DoFileIsAlreadyOpen` is shown.

To look at the differences between the next file in the change set `BrowserItems.C 1.3` and its previous version `BrowserItems.C 1.2`:

- Highlight `BrowserItems.C` in the File List.

  In the Differences List, you can see that there is one difference between the two versions of `BrowserItems.C`.

- Close the Diff/Merge tool.

Review

This was the last of the SNiFF+ version controlling and configuration management tools to be introduced in this tutorial.

In this part of the C++ Tutorial you were introduced to:

- file history and locking information
- Configuration Manager
- Diff/Merge tool

Please note that this was only an introduction to these tools. To learn more about the version control tools in SNiFF+, please refer to the *User's Guide*.

What's next

The only SNiFF+ tool that should now be open is the Launch Pad.

- The next part of this tutorial introduces you to Setting up Browsing-Only projects.

Part IV

Browsing-Only Project Setup

Setting up Browsing-Only Projects 1

The Project Setup Wizard guides you through the process of setting up a browsing-only project. SNiFF+'s Make Support is not available for browsing-only projects. In a real-world situation you would use browsing-only projects, e.g., to browse libraries.

We assume that all your source code directories are under a single root directory (recommended). We will use `<your_source_root_directory>` to refer to this directory.

In this chapter you will:

- set up your own project for browsing only

The Project Setup Wizard for browsing only

- To start the Project Setup Wizard, choose **Project > New Project > with Wizard...** in the Launch Pad.

In the Project Setup Wizard

The Wizard starts by asking you to select how you intend to use the new SNiFF+ Project.

- Select **Browsing-only Setup**, and press **Next**.

  The "Select file types" page appears.

In the "Select file types" page

- Select **C/C++** and press **Next**.

  Note that, after project setup, you can add new standard file types (like the ones in the "Additional File Types Column"), or create and add your own.

In the "Specify project location and name" page

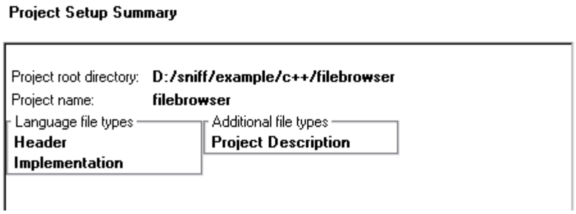
1. Press the **Browse** button next to the **Source code root directory** field and navigate to `<your_source_root_directory>`.
2. Double-click on `<your_source_root_directory>` and press **Select**.

   SNiFF+ sets the path and gives the project the same name as the value of the **Source code root directory** field. By default, the **Create Subprojects** check box is selected in the Wizard. This means that SNiFF+ will automatically create subprojects for all the subdirectories of your project. Which is fine.
3. Press **Next**.

In the "Project Setup Summary" page

This page summarizes your specifications for the new SNiFF+ C++ Project.

1. Make sure that your Project Setup Summary page is similar to the following. Except for the Project root directory and the project name, the rest should be the same. If it isn't, please go back to the beginning of the Wizard and start again.



Project Setup Summary

Project root directory: D:/sniff/example/c++/filebrowser

Project name: filebrowser

Language file types: Header, Implementation

Additional file types: Project Description

If the information on the Project Setup Summary page is similar to that in the illustration:

2. Press **Finish**.

   SNiFF+ will now create the project and all its subprojects.

3. In the dialog that appears asking if you want to generate cross reference information, press **No**.

   Cross Reference information will be automatically generated when you open the Cross Referencer.

   When SNiFF+ is finished, it opens the new project and displays its structure and contents in a Project Editor. You can now browse your own project, following the steps in the browsing part of the C++ tutorial.

4. Close the new project that you have created.

What's next

You won't be using the filebrowser project in the tutorials to follow. You can either delete the project, or, if you want to keep the project for later reference, just close it. Note that "deleting a project" means that only the files generated by SNiFF+ are deleted, source code files remain untouched.

- To **delete** the project:

  In the Launch Pad, make sure that `filebrowser.shared` is highlighted and choose **Project > Delete Project filebrowser.shared**.

- To **close** the project:

  In the Launch Pad, make sure that `filebrowser.shared` is highlighted and press **Close Project filebrowser.shared**.

The next tutorial introduces you to

- setting up a single user/single platform project for C++ development
- setting up the SNiFF+ build system
- tools used for editing, compiling and debugging C++ code with SNiFF

  Please note that in the tutorials to come, you will be using the Complex code example.

Part V

Edit/Compile/Debug

Single-User Project Setup 1

This chapter is about

- using the Project Setup Wizard for setting up a SNiFF+ single-user/single-platform project for development.

The Project Setup Wizard guides you through the process of setting up a single-user/single-platform project without version controlling. Multi-user/multi-platform projects using RCS for configuration management and version control (CMVC) are described in the next tutorial, “Working in Teams”.

If you aren’t continuing on from the “Version Controlling” tutorial

We assume you have successfully installed SNiFF+, and know how to start it. If not, please refer to the *Installation Guide*.

Preparing the Environment

- Copy the directory

  `<your_sniff_installation_dir>/example/c++/complex_dir`

  including subdirectories, to a place where you have write permissions.
  In the rest of this tutorial, we will use `<complex_dir>` to refer to the complete path to this directory.

- Start SNiFF+.

  The Launch Pad appears.

Single-user Project Setup Wizard

- To start the Project Setup Wizard, choose **Project > New Project > with Wizard...** in the Launch Pad. The Project Setup Wizard appears.

In the Project Setup Wizard

The Wizard starts by asking you to select how you intend to use the new SNiFF+ Project.

- Accept the default selection, **Standard Setup**, and press **Next**.

  The “Select developmental task” page appears.
  In the remaining steps, we will refer to the names of Wizard pages. You can find a page’s name in the title bar of the Wizard.

In the “Select Developmental task” page

- Select **Create a new SNiFF+ Project from scratch** and press **Next**.

In the “Your development organization” page

This tutorial is for single-user / single platform development without CMVC, so:

- accept the defaults (**No/No/None**) and press **Next**.

In the “Select file types” page

- Select **C/C++** and press **Next**.

  Note that, after project setup, you can add new standard file types (like the ones in the “Additional File Types Column”), or create and add your own.

In the “Specify Private Working Environment” page

You are asked to specify your *Private Working Environment* (PWE) root directory. A Private Working Environment is simply the directory in which you work and where SNiFF+ administers your project.

1. Press **Browse**, and in the Directory dialog, navigate to `<complex_dir>/user`, double-click on it and then press **Select**.
2. In the **PWE name** field, enter a name for the PWE, e.g., `complex private`.

   Notice that your user name is entered next to the selected **Owner** check box. SNiFF+ needs your user name to correctly handle permissions. Being the owner of the PWE means that you are the only one who is allowed to modify its attributes.
3. Press **Next**.

In the “Create New SNiFF+ Project” page

You are asked to specify the root directory of the new project and to specify two other setup attributes (described below).
SNiFF+ automatically enters the root of your PWE in the Project root directory field.

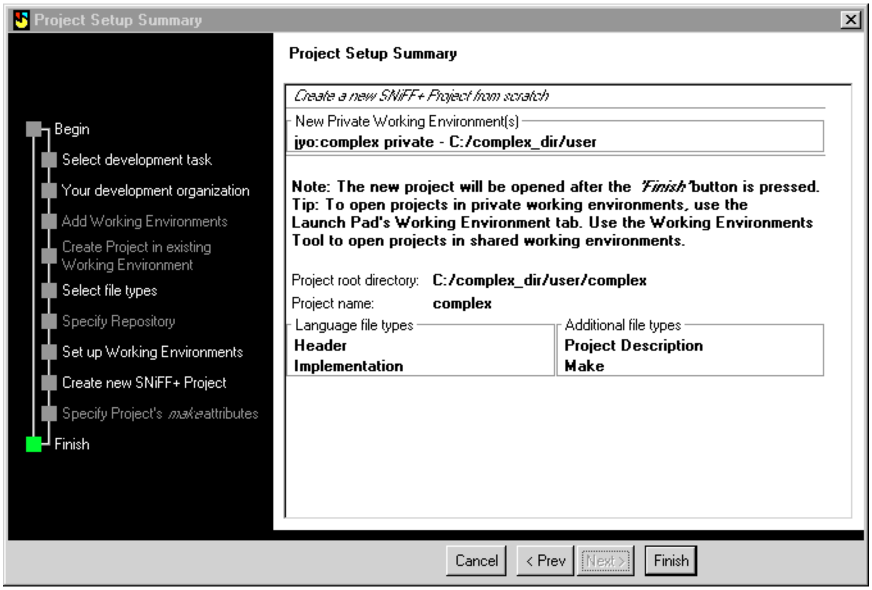
1. Modify the entry in the **Project root directory** field to specify the root directory of the new project’s source code. This should be:

   ```
   <complex_dir>/user/complex
   ```
2. Notice that the new project’s name has changed to `complex`. We suggest that you accept this name. Also by default, **Create Subprojects** is enabled.
3. Select the **Use SNiFF+’s Makefiles** checkbox.
4. Press **Next**.

In the “Project Setup Summary” page

This page summarizes your specifications for the new SNiFF+ project and required Working Environments.

- Make sure that your Project Setup Summary page is similar to the following. If it isn’t, please go back to the beginning of the Wizard and start again.



Project Setup Summary
Begin
Select development task
Your development organization
Add Working Environments
Create Project in existing Working Environment
Select file types
Specify Repository
Set up Working Environments
Create new SNiFF+ Project
Specify Project's *make* attributes
Finish
Cancel
< Prev
Next >
Finish

Project Setup Summary

Create a new SNiFF+ Project from scratch

New Private Working Environment(s)

iyo:complex private - C:/complex_dir/user

Note: The new project will be opened after the *'Finish'* button is pressed.
Tip: To open projects in private working environments, use the Launch Pad's Working Environment tab. Use the Working Environments Tool to open projects in shared working environments.

Project root directory: **C:/complex_dir/user/complex**
Project name: **complex**

Language file types	Additional file types
Header	**Project Description**
Implementation	**Make**

- Press **Finish**.

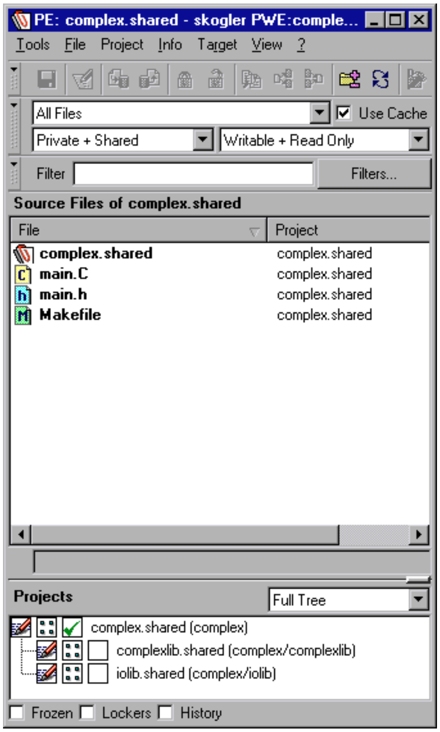
  SNiFF+ will now create the new `complex` project and all its subprojects.

- In the dialog that appears asking if you want to generate cross reference information, press **No**.

  When SNiFF+ is finished, it opens the new project and displays its structure and contents in a Project Editor.

Viewing the results

The Project Editor on your screen should look this.



PE: complex.shared - skogler PWE:comple...

All Files ☑ Use Cache

Private + Shared | Writable + Read Only

Filter

Source Files of complex.shared

File	Project
complex.shared	complex.shared
main.C	complex.shared
main.h	complex.shared
Makefile	complex.shared

Projects | Full Tree

✓ complex.shared (complex)

□ complexlib.shared (complex/complexlib)

□ iolib.shared (complex/iolib)

□ Frozen □ Lockers □ History

SNiFF+ Build Support 2

This chapter is about:

- setting up the build system for a single-user project
- building the project’s executable

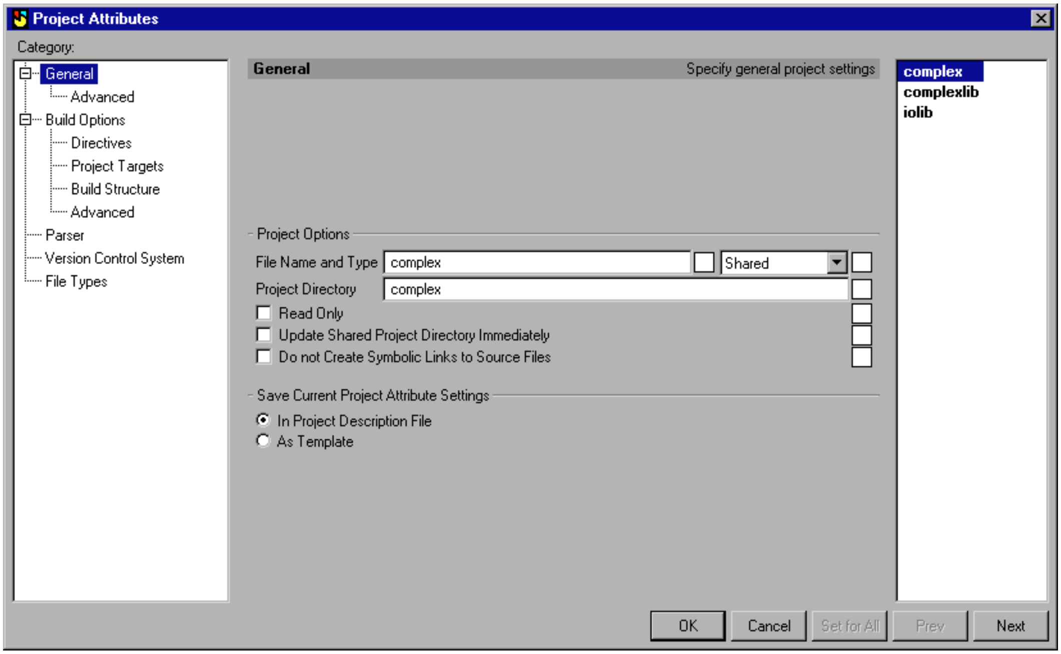
Setting up the build system

In the Project Editor

- In the Project Tree of the Project Editor, choose **Context menu > Select From All Projects** to checkmark all projects.
- Choose **Project > Attributes of Checkmarked Projects...**.

  The Group Project Attributes dialog appears. In this dialog, you can look at and modify the project attributes of multiple projects. For a description of the dialog, please see Reference Guide — Project Attributes.

In the Group Project Attributes dialog



complex
complexlib
iolib

Project Options

File Name and Type: complex □ Shared □

Project Directory: complex □

□ Read Only □

□ Update Shared Project Directory Immediately □

□ Do not Create Symbolic Links to Source Files □

Save Current Project Attribute Settings

● In Project Description File

○ As Template

Setting up Make Support for complexlib.shared

1. Highlight **complexlib** in the Project List.
2. Under the **Build Options** node, select **Project Targets**.
3. In the **Library** field of the of the Ansi C/C++ tab, enter `complexlib.a`. This will be the name of the library built in this project.
4. Under the **Build Options** node, select **Build Structure**.
5. In the Build Structure view, choose **Passed to Superproject** drop-down **> Library**.

   The project's library is exported to `complex.shared` and is used to build the Complex executable.

Setting up Make Support for iolib.shared

1. Highlight **iolib** in the Project List.
2. In the Build Structure view, choose **Passed to Superproject** drop-down **> Object Files + Received**.

   The project's object file (`iolib.o`) is exported to `complex.shared`.

Setting up Make Support for complex.shared

1. Highlight **complex** in the Project List.
2. Under the **Build Options** node, select **Project Targets**.
3. In the **Executable** field of the Ansi C/C++ tab, enter `complex` (on Windows `complex.exe`). This will be the name of the project's executable.
4. **On Unix only**, enter **-lstdc++** in the **+Libraries Linked** field (below the **Executable** field).
5. Under the **Build Options** node, select **Build Structure**.
6. In the Build Structure view, press the **Generate** button next to the **Recursive Make Dir(s)** field.

   The executable is built using recursive Make rules. By pressing the **Generate** button, SNiFF+ generates the order of subprojects in which Make is executed.

Generating the include paths for all projects

1. Under the **Build Options** node, select **Directives**.
2. Select the checkbox to the right of the **Generate** button.
3. Press the **Set for All** button to generate the include paths for all projects in the Project List.
4. Press **Ok** to apply the changes to the project attributes.

   The icons in the Project Tree of the Project Editor warn you that the projects have been modified.
5. A dialog appears asking you to update Makefiles. We will do this later so press **No**.

In the Launch Pad

To Save the changes made to `complex.shared` and its subprojects:

1. Select `complex.shared` in the Project List.
2. Choose **Project > Save Project complex.shared**.
3. In the Alert dialog that appears, press the **Save All** button.

Building the executable

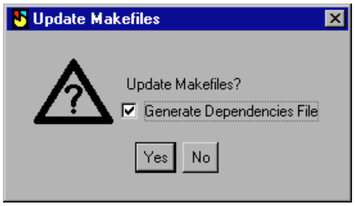
> **Note**
>
> SNiFF+ doesn't have its own compiler therefore you must have a compiler installed on your computer to compile SNiFF+ projects. By default, the gnu compiler is specified on Unix, and Microsoft Developer is specified on Windows. If you are using another compiler, it must be specified in your Platform Makefile. For more information, see *User's Guide — Build and Make Support*.

In the Project Editor

Before building, make sure that the projects' Make Support information is up-to-date. Makefiles should be updated whenever structural changes are made to the projects, or when projects are first opened.

1. Choose **Target > Update Makefiles...** to generate the Make Support Files for all the projects.

   A dialog appears asking you whether the dependencies information should also be updated.



Update Makefiles

Update Makefiles?

☑ Generate Dependencies File

2. Press **Yes**.

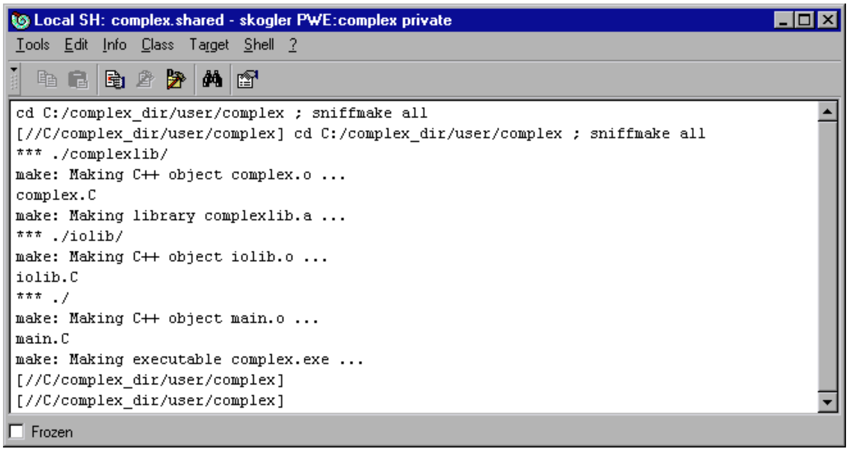
   SNiFF+ generates the Make Support Files and stores them in the `.sniffdir` subdirectory of each project directory.

Make execution

SNiFF+ needs to know where to start the Make execution. You tell SNiFF+ this by selecting the appropriate project. In the example project, Make execution starts in `complex.shared`.

1. In the Project Tree, highlight `complex.shared` by clicking on its name.
2. Choose **Target > Make > all** to recursively build the executable.

   A Shell Tool appears, in which the `make all` command is executed in each project directory. At the end of the build it should look like this:



Local SH: complex.shared - skogler PWE:complex private
Tools Edit Info Class Target Shell ?
Frozen

```
cd C:/complex_dir/user/complex ; sniffmake all
[//C/complex_dir/user/complex] cd C:/complex_dir/user/complex ; sniffmake all
*** ./complexlib/
make: Making C++ object complex.o ...
complex.C
make: Making library complexlib.a ...
*** ./iolib/
make: Making C++ object iolib.o ...
iolib.C
*** ./
make: Making C++ object main.o ...
main.C
make: Making executable complex.exe ...
[//C/complex_dir/user/complex]
[//C/complex_dir/user/complex]
```

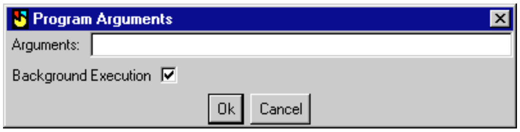
If compiler errors are reported in the shell at this stage, something went wrong with the setup of the project’s Make attributes. We recommend that you go through the steps in this tutorial again, carefully check them, compare screenshots, and try compiling again.

Running the executable

In the Project Editor

1. Make sure that `complex.shared` is highlighted in the Project Tree.
2. Choose **Target > Run complex.exe**.

   The Program Arguments dialog appears:



Program Arguments

Arguments:

Background Execution ☑

3. Clear the **Background Execution** check box.
4. Press **Ok**.

   The Shell Tool appears.

In the Shell Tool

The complex example outputs the numbers that you enter in the Shell Tool.

1. Enter some values and test to see if the output is correct. You'll notice that it isn't.

   Obviously, there is an implementation error in one of the files in the project. To fix this error:

   - **On Windows**, you will modify the source file that contains the error. Ignore the next chapter, Debugging (Unix only), and continue with Opening and editing the file (Windows only) below.
   - **On Unix**, you will use the Debugger to find the error. Skip the rest of this chapter and go to the chapter Debugging (Unix only) — page 89.
2. Close the Shell tool.

Opening and editing the file (Windows only)

After using the Project Editor to open a file in the Source Editor, you will edit the file by correcting the implementation error that occurs in it.

In the Project Editor

1. In the Project Tree, make sure that `complexlib.shared` is checkmarked, so that you see the files in the project.
2. In the File List, double-click on `complex.C` to open the file in the Source Editor.

In the Source Editor

1. Choose **Edit > Go To Line...**
2. In the Goto dialog that appears, type `26` and press the **Go To** button.

   You are now positioned at the line in which the variable `im` is assigned the incorrect value.
3. Fix the error by changing `r` to `i`.
4. For changes to take effect, choose **File > Save**.

Testing the changes (Windows only)

1. In the Source Editor, choose **Target > Make File complex.o** to compile the file.

   A Shell tool opens and SNiFF+ compiles the file. After successful compilation, let’s build and run the executable to see the effects of your modifications.
2. In the Source Editor, choose **Target > Update Makefiles** to generate the Make Support Files for all the projects.

   A dialog appears asking you whether the dependencies information should also be updated.
3. Press **Yes**.
4. In the Project Editor, choose **Target > Make > all** to build the shared project’s executable.

   The project’s Make command is recursively executed in the Shell tool. Upon completion, you should have an executable named `complex.exe` in:
   `<complex_dir>/user/complex`
5. Run the executable to assure yourself that it executes properly. To find out how to do so, see Running the executable — page 87.
6. Close all open tools except the Source Editor and the Launch Pad.

Debugging (Unix only) 3

This chapter is about:

- using the SNiFF+ Debugger

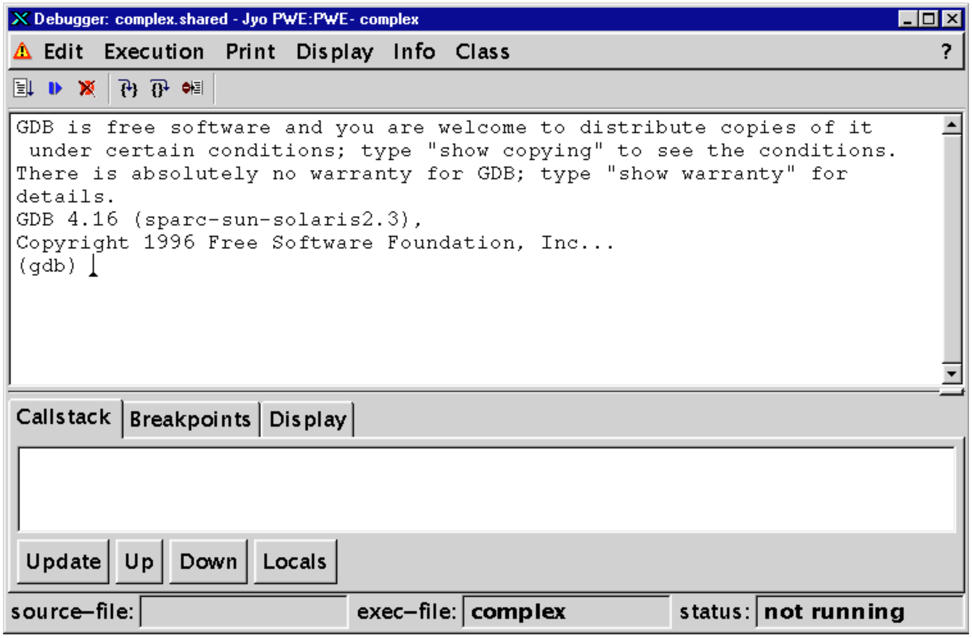
SNiFF+ provides a graphical front-end to widely used debuggers such as gdb and dbx. It provides a menu interface to most of the commands and interprets the debugger output messages. The Source Editor is used to show the current stack frame. After starting the Debugger, you will set a breakpoint and try and find out exactly where the error occurs.

> **On Windows NT/95**
>
> The project’s target is automatically loaded into Microsoft Developer Studio. For a description of the debug commands in Microsoft Developer Studio, please refer to the Microsoft product documentation.

In the Project Editor

- To start the Debugger, choose **Target > Debug complex**.

  The Debugger command line shell opens.



```
GDB is free software and you are welcome to distribute copies of it
 under certain conditions; type "show copying" to see the conditions.
There is absolutely no warranty for GDB; type "show warranty" for
details.
GDB 4.16 (sparc-sun-solaris2.3),
Copyright 1996 Free Software Foundation, Inc...
(gdb)
```

source–file: exec–file: **complex** status: **not running**

Note

If the Debugger doesn't come up correctly, either gdb is not installed on your system or the back-end has not been specified in the Debugger preferences (for details, please refer to the Reference Guide).

In the Debugger Command Line Shell

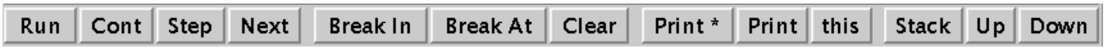
- For a summary of debug commands, type `help` at the command line prompt.

  Note that many of these commands can also be posted from the Source Editor.

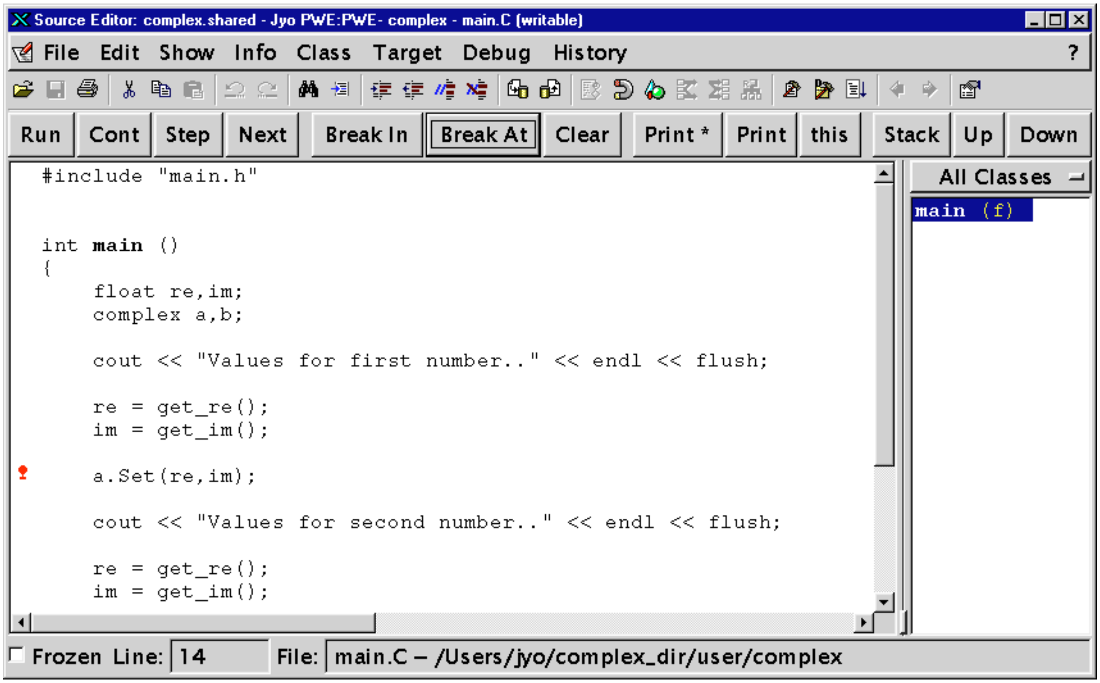
In the Project Editor

- You will be setting a breakpoint in the source file, `main.C`. To open the file, double click on it in the File List.

  The file is opened in the Source Editor and, because you are in debug mode, a row of buttons for the most commonly needed debug commands has been added below the tool bar.



Setting Breakpoints



Source Editor: complex.shared - Jyo PWE:PWE- complex - main.C (writable)

```
#include "main.h"


int main ()
{
    float re,im;
    complex a,b;

    cout << "Values for first number.." << endl << flush;

    re = get_re();
    im = get_im();

    a.Set(re,im);

    cout << "Values for second number.." << endl << flush;

    re = get_re();
    im = get_im();
```

□ Frozen Line: 14 File: main.C – /Users/jyo/complex_dir/user/complex

In the above illustration the breakpoint has already been set.
To set this breakpoint:

1. Choose **Edit > Go To line...**

   The Goto dialog appears.

2. To go to the line where you will set the breakpoint, type 14 in the Goto dialog and press the **Go to** button.

   You are now positioned at line 14 in the file.

3. To set the breakpoint, press **Break At**.

   A small stop sign at the beginning of the line indicates the breakpoint.

Running the executable

1. Press **Run**.

   A Program Arguments dialog appears.

2. Press **Ok**.

In the Debugger

- Enter some values for the real part and the imaginary part of the first number.

  Since you set a breakpoint, the Debugger stops now and the Source Editor shows a small right arrow to indicate the actual program counter.

Single-stepping

There are two possibilities for single-stepping:

- **Next** steps over functions and methods.
- **Step** steps into functions and methods.

In the Source Editor, press **Next**. You'll notice that the program counter moves down to the next source line.

Displaying values

In the Source Editor

1. Highlight `a` in line 14 by double-clicking on it.
2. Press **Print**.

   The Debugger displays the value you entered for the real part (`re`) and the imaginary part (`im`). Notice that the value for the imaginary part is incorrect.
3. In the Debugger, choose **Tools > Close Tool**.

   The Debugger quits and the button bar with the debug commands is removed from the Source Editor.

Editing the file

We'll now edit the file where the error occurs.

1. Click on the method `Set` in line 14.
2. Choose **Show > Symbol(s) Set...**

   The `complex.C` file is loaded and you are now positioned at the method `complex::Set`. As you can see the variable `im` is assigned the incorrect value.
3. Fix the error by changing `r` to `i`.
4. For the changes to take effect, choose **File > Save**.

Testing the changes

- In the Source Editor, choose **Target > Make File complex.o** to compile the file.

  A Shell tool opens and SNiFF+ compiles the file. After successful compilation, let’s build and run the executable to see the effects of your modifications.

In the Project Editor

1. Choose **Target > Make > all** to build the shared project’s executable.

   The project’s Make command is recursively executed in the Shell tool. Upon completion, you should have an executable named `complex` in:
   `<complex_dir>/user/complex`
2. Run the executable to assure yourself that it executes properly.
3. Close all open tools except the Source Editor and the Launch Pad.

Global Editing with the Retriever 4

This chapter is about using the Retriever to:

- find every line in your source files containing a particular string
- re-filter search results to conform more closely to your needs
- effectively edit text in combination with the Source Editor

Opening the Retriever

This complex example is not as complex as the name suggests therefore we will change all internal references from complex to simplex.

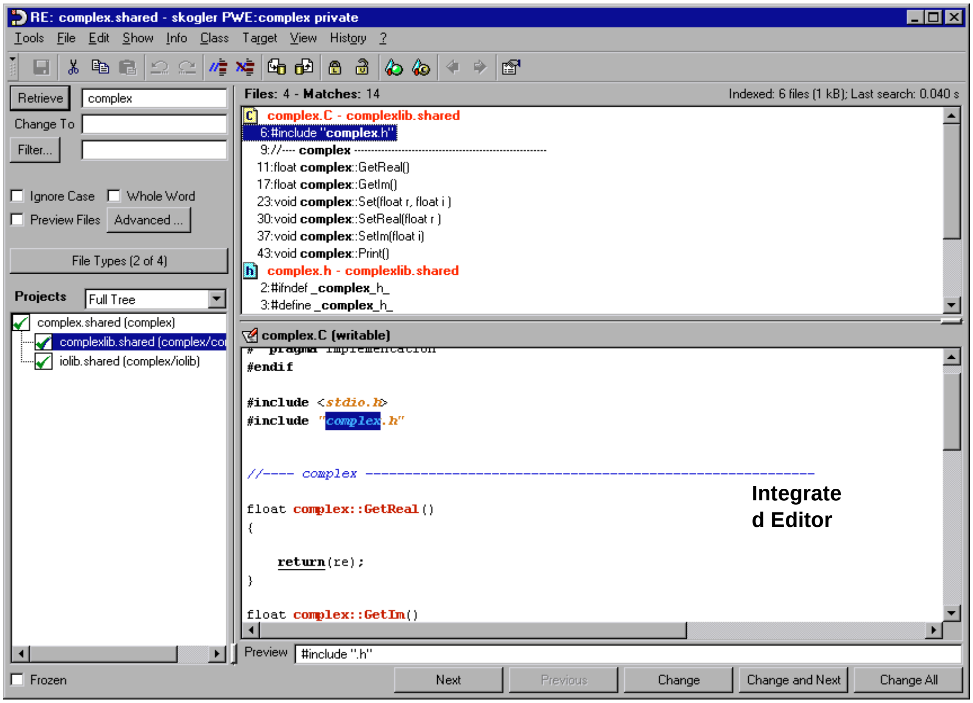
In the Source Editor

- Double-click on string `complex`.

  `complex` is now highlighted.
- Choose **Info > Retrieve complex From All Projects**.

The Retriever opens and displays all occurrences of the string `complex`. The search string is in bold. As you can see, there are 14 matches in 4 files.

You could also open the Retriever from any tool by choosing **Tools > Retriever**


RE: complex.shared - skogler PWE:complex private
Tools File Edit Show Info Class Target View History ?
Retrieve
Change To
Filter...
Advanced ...
File Types (2 of 4)
Next
Previous
Change
Change and Next
Change All

Retrieve: complex

□ Ignore Case □ Whole Word

□ Preview Files

Projects: Full Tree

- ☑ complex.shared (complex)
  - ☑ complexlib.shared (complex/co
  - ☑ iolib.shared (complex/iolib)

□ Frozen

Files: 4 - Matches: 14 Indexed: 6 files (1 kB); Last search: 0.040 s

```
complex.C - complexlib.shared
 6:#include "complex.h"
 9://---- complex -----------------------------------------------
 11:float complex::GetReal()
 17:float complex::GetIm()
 23:void complex::Set(float r, float i )
 30:void complex::SetReal(float r )
 37:void complex::SetIm(float i)
 43:void complex::Print()
complex.h - complexlib.shared
 2:#ifndef _complex_h_
 3:#define _complex_h_
```

complex.C (writable)

```
#  pragma implementation
#endif

#include <stdio.h>
#include "complex.h"


//---- complex ----------------------------------------------------------

float complex::GetReal()
{

    return(re);
}

float complex::GetIm()
```

Preview: #include ".h"

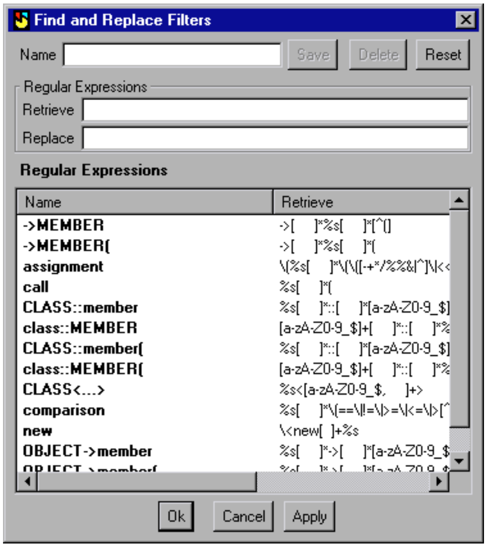
Integrate d Editor

Filtering

You will notice that there are include statements in the above illustration and if you change these, you won’t be able to compile the project. Lets now exclude all include references. By doing so we won’t have to change any file names and can filter for symbols only. We will do so by using regular expressions. Regular expressions (regex) are a powerful means to specify patterns for filters and search strings in the various SNiFF+ tools, especially in the Retriever. Basically, regular expressions are a system of matching character patterns. For more information, please see *Reference Guide — Regular Expressions*.

- In the Retriever, press the **Filter...** button to open the **Find and Replace Filters** dialog.

In the Find and Replace Filters dialog



Name	Retrieve
->MEMBER	->[ ]*%s[ ]*[^(]
->MEMBER(	->[ ]*%s[ ]*(
assignment	\(%s[ ]*\(\([-+*/%%&|^]|<<
call	%s[ ]*(
CLASS::member	%s[ ]*::[ ]*[a-zA-Z0-9_$]
class::MEMBER	[a-zA-Z0-9_$]+[ ]*::[ ]*%
CLASS::member(	%s[ ]*::[ ]*[a-zA-Z0-9_$]
class::MEMBER(	[a-zA-Z0-9_$]+[ ]*::[ ]*%
CLASS<...>	%s<[a-zA-Z0-9_$,]+>
comparison	%s[ ]*\(==\\|!=\\|>=\\|<=\\|>[^
new	\<new[ ]+%s
OBJECT->member	%s[ ]*->[ ]*[a-zA-Z0-9_$
OBJECT->member(	%s[ ]*->[ ]*[a-zA-Z0-9_$

1. In the **Name** field, enter a name for the regular expression e.g., No Includes.
2. In the **Retrieve** field, enter `!#include`. An exclamation point at the beginning of a regular expression means “match everything except the following regex”.

   Note that this is a SNiFF+ specific implementation and not usually part of the regular expression syntax.
3. Press **Ok** to add the new regular expression to the Regular Expression List, to apply the regular expression to the list of matches and to close the dialog.

   As you can see, there are now only 12 matches in 3 files.

Global Editing

Now we will change all occurrences of `complex` (excluding include references) to `simplex`.

1. In the **Change To** field, type `simplex`.

   Take a look at the **Preview** field below the integrated Source Editor, the code line (highlighted in the Files — Matches List) is shown as it would appear after modification. You can use the **Next** button at the bottom of the tool to look at each line as it would appear after being changed.
2. To change all occurrences of complex to simplex, press the **Change All** button (in the lower right corner of the Retriever).
3. In the dialog that appears, press **Yes**.

   All occurrences of complex are changed to simplex. You can verify this by pressing the **Retrieve** button again to requery.

Undoing global changes

If you want to undo changes:

- If you want to undo changes, choose **Edit > Undo Change All**.

  All changes that you've made are discarded. You can verify this by pressing the **Retrieve** button again to requery.

What's next

- Close the project. To do so:

In the Launch Pad

1. Select `complex.shared - <`*Username*`> PWE:complex private`.
2. Choose **Project > Close Project complex.shared**.

- The next part of this tutorial introduces you to Team Setup.

Part VI

Team Setup

Key Concepts 1

Note

This tutorial assumes that you will use RCS (included in the SNiFF+ package) for version controlling. Most other CMVC tools are also supported by SNiFF+. Please refer to the *Release Notes* for details.

This chapter introduces 2 key concepts

- *shared projects*
- *working environments*

Although these concepts are not in themselves difficult, what follows in the hands-on tutorial chapters may tend to get a little confusing if you don't have a reasonable understanding of what these things are and how they work.

For detailed information going beyond this very brief introduction, please refer to the *User's Guide*.

Shared projects

A shared project is, as the name suggests, suitable for team development. However it is equally recommended for single-user work situations.

Shared projects offer a great deal of flexibility. Because all references to files and subprojects are relative to a root directory, you can easily move a shared project to another location on a file system.

Each team member can access a shared project and make changes to its files and/or structure, regardless of what other team members are doing.

This means that the integrity of the project system as a whole needs to be maintained in some way, which is why shared projects are always used in conjunction with working environments and a configuration management and version control (CMVC) tool.

It is strongly recommended that one person be appointed to administer this "maintenance system". In SNiFF+ this person is called the *Working Environments Administrator*. This tutorial mainly covers the tasks performed by a Working Environments Administrator.

- From now on, shared projects are simply referred to as *projects*.

Working environments

SNiFF+ uses 4 different kinds of working environments:

- Repository Working Environment (RWE)
- Shared Source Working Environment (SSWE)
- Shared Object Working Environment (SOWE)
- Private Working Environment (PWE)

The RWE (Repository Working Environment)

Your team members access and modify a permanent shared data Repository using commands provided by your underlying configuration management and version-control (CMVC) tool.

SNiFF+ provides an interface to your CMVC tool. This interface needs to know the location of your Repository.

You provide this information by defining a *Repository Working Environment* (RWE), which specifies the root directory of your Repository.

In this tutorial, we will be using RCS, the CMVC tool provided with the SNiFF+ package.

The SSWE (Shared Source Working Environment)

SNiFF+ requires you to specify the root directory under which your team's shared source code is located. The files and directories under this root directory access your team's Repository. At regular intervals, all these files and directories are updated to reflect the most current state of your team's software system.

When creating software systems from scratch, your team's (Working Environments Administrator's) first job is to populate this root directory with source code. For existing software systems, your team will already have such a central location.

In either case, once you have such a root directory, you have to tell SNiFF+ where it is. You do this by defining a *Shared Source Working Environment* (SSWE).

All team members see, or *share*, all the source files in the SSWE. When browsing the source files, this view is read-only. When editing source files, team members work on *local* copies of the shared source files they want to modify—they never directly modify the shared source files in the SSWE. The view to all other source files remains read-only.

The SOWE (Shared Object Working Environment)

Just like with shared source code, SNiFF+ also requires you to specify a central location for your team's shared object files. In SNiFF+, you define one *Shared Object Working Environment* (SOWE), which specifies the root directory containing these files, for each target platform.

SOWEs serve as shared repositories for your team's most current and stable object code. During an update of an SOWE, source files in the SSWE are compiled and the resulting object code is stored in the SOWE.

An essential aspect of SOWEs is avoiding unnecessary builds in Private Working Environments (see below) that access them.

The PWE (Private Working Environment)

Developers must be able to work in isolation from other team members. They need their own workspaces in which they can edit, compile and debug projects without interfering with the work of other team members. Furthermore, they continually need to have access to their software system’s most current source code and object code base.

SNiFF+ supports this by allowing each member of a team to work in an isolated workspace. In SNiFF+, you define a *Private Working Environment* (PWE) to specify the root directory of each team member’s workspace.

You can go through the entire edit/compile/debug cycle in your PWE. In your PWE, you have a read-only view to the shared source files located in your team’s SSWE. When you need to modify shared source files, you check out the necessary files from your team’s Repository. When you’re satisfied that the changes you’ve made are error-free, you check the modified files back into your team’s Repository. The next time your team’s SSWE is updated, these changes are incorporated, and the shared source files in the SSWE once again reflect the most current state of your software system.

Multi-User Project Setup 2

This chapter is about

- using the Project Setup Wizard for setting up a SNiFF+ multi-user/multi-platform project for development.

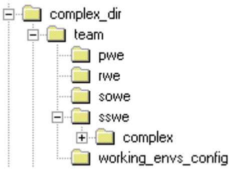
  The Project Setup Wizard guides you through the process of setting up a multi-user/multi-platform project with version controlling.

Preparing the Environment

- In the Edit / Compile / Debug tutorial, you copied the directory

  `<your_sniff_installation_dir>/example/c++/complex_dir,`

  including subdirectories, to a place where you have write permissions. If you haven’t done so, please do so now. You should have the following directory structure:



```
complex_dir
  team
    pwe
    rwe
    sowe
    sswe
      complex
    working_envs_config
```

In the rest of this tutorial, we will use `<complex_dir>` to refer to the complete path to this directory.

Working environment information

`pwe` — This directory holds your own workspace, i.e., your **P**rivate **W**orking **E**nvironment.

`rwe` — This directory holds your team’s shared data repository, i.e., your **R**epository **W**orking **E**nvironment.

`sowe` — This directory holds your team’s shared object code, i.e., your **S**hared **O**bject **W**orking **E**nvironment.

`sswe` — This directory holds the source code your team shares, i.e., your **S**hared **S**ource **W**orking **E**nvironment.

`working_envs_config` — This directory will hold the working environment files generated and maintained by SNiFF+.

Setting your Preferences

To set the `working_envs_config` directory as your preferred maintenance directory:

- In the Launch Pad, choose **Tools > Preferences...** to open the Preferences dialog.

In the Preferences dialog

1. Under the **Tools** node, select **Working Environments**.
2. In the Working Environments view, press **Dir...** next to the **Working Environment Config. Directory** field.
3. Navigate to the `<complex_dir>/team/working_envs_config` directory.
4. Double-click on the directory name and press **Select**.
5. Press **Ok** to apply the changes.

Multi-user Project Setup Wizard

- To start the Project Setup Wizard, choose **Project > New Project > with Wizard...** in the Launch Pad.

In the Project Setup Wizard

The Wizard starts by asking you to select how you intend to use the new SNiFF+ Project.

- Accept the default selection, **Standard Setup**, and press **Next**.

  The “Select developmental task” page appears.

  In the remaining steps, we will refer to the names of Wizard pages. You can find a page’s name in the title bar of the Wizard.

In the “Select developmental task” page

- Select **Create a new SNiFF+ Project from scratch** and press **Next**.

In the “Your development organization” page

This tutorial is for multi-user/multi platform development with configuration management and version control (CMVC), so:

1. Select **Yes** for both Yes/No questions.
2. Choose **RCS** as the version control tool.
3. Press **Next**.

> **Note**
>
> This tutorial assumes that you will use RCS for version controlling. Most other CMVC tools are also supported by SNiFF+. Please refer to the *Release Notes* for details.

In the “Select file types” page

- Select **C/C++** and press **Next**.

  SNiFF+ will automatically include all necessary file types needed for working with C/C++ source code in the new project. Note that, after project setup, you can add new standard file types (like the ones in the “Additional File Types Column”), or create and add your own.

In the “Specify Repository” page

You are asked to specify your *Repository Working Environment* (RWE). SNiFF+ uses the RWE for version control administration. To specify the `rwe` directory:

1. Press **Browse** and, in the Directory dialog, navigate to:

   `<complex_dir>/teams/rwe`

   Double-click on the rwe directory, and then press **Select**.
2. In the **RWE name** field, type a name for the RWE, e.g., `Complex Repository`.
3. Press **Next**.

In the “Specify team source code location” page

You are asked for your *Shared Source Working Environment* (SSWE).

1. Press **Browse** and, in the Directory dialog, navigate to:

   `<complex_dir>/teams/sswe`

   Double-click on the sswe directory, and then press **Select**.
2. In the **SSWE name** field, type a name for the SSWE, e.g., `Complex SSWE`.
3. Press **Next**.

In the “Specify team object code location” page

You are asked to specify your *Shared Object Working Environment* (SOWE) root directory.

1. Press **Browse** and, in the Directory dialog, navigate to:

   `<complex_dir>/teams/sowe`

   Double-click on the sowe directory, and then press **Select**.
2. In the **SOWE name** field, type a name for the SOWE, e.g., `Complex SOWE`.
3. Press **Next**.

In the “Specify Private Working Environment” page

You are asked to specify your *Private Working Environment* (PWE) root directory.

1. Press **Browse** and, in the Directory dialog, navigate to:

   `<complex_dir>/teams/pwe`

   Double-click on the pwe directory, and then press **Select**.

2. In the **PWE name** field, type a name for the PWE, e.g., `Complex PWE`.
3. Notice that your user name is entered next to the selected **Owner** check box. SNiFF+ needs your user name to correctly handle permissions.

   Being the owner of the PWE means that you are the only one who is allowed to modify the working environment's attributes.
4. Press **Next**.

In the "Additional team members?" page

You are asked whether any additional PWEs are needed. Since you are working alone through this Tutorial, you don't need to specify additional PWEs.

- Accept the default value and press **Next**.

In the "Additional target platforms?" page

You are asked whether any additional SOWEs are needed. Since the code in this Tutorial will only be compiled for one platform, you don't need to specify additional SOWEs.

- Accept the default value and press **Next**.

In the "Create new SNiFF+ Project" page

You are asked to specify the root directory of the new project. SNiFF+ automatically enters the root of your SSWE in the **Project root directory** field, since your team's shared source code is located in it. Our project root directory is `complex`, so:

1. Press **Browse** and, in the Directory dialog, navigate to:

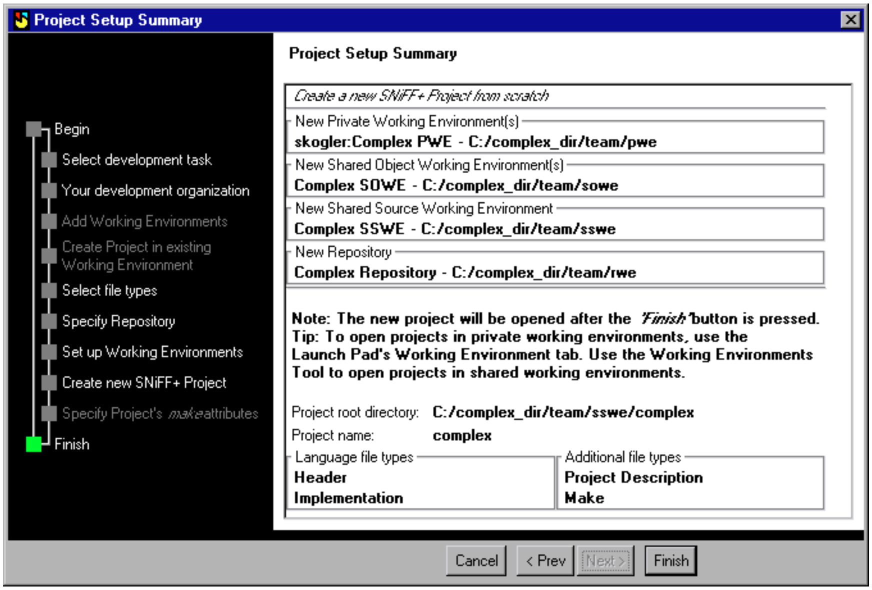
   ```
   <complex_dir>/teams/sswe/complex
   ```

   Double-click on the `complex` directory, and then press **Select**.
   Notice that the new project's name has changed to `complex`, which is the name we will use throughout the tutorial. Also by default, **Create Subprojects** is enabled.
2. Select the **Use SNiFF+'s Makefiles** checkbox.
3. Press **Next**.

In the “Project Setup Summary” page

This page summarizes your specifications for the new SNiFF+ project and required working environments.

1. Make sure that your Project Setup Summary page is similar to the following. If it isn’t, please go back to the beginning of the Wizard and start again:


Project Setup Summary
Begin
Select development task
Your development organization
Add Working Environments
Create Project in existing Working Environment
Select file types
Specify Repository
Set up Working Environments
Create new SNiFF+ Project
Specify Project's *make* attributes
Finish
Cancel
< Prev
Next >
Finish

Project Setup Summary

Create a new SNiFF+ Project from scratch

New Private Working Environment(s)
skogler:Complex PWE - C:/complex_dir/team/pwe

New Shared Object Working Environment(s)
Complex SOWE - C:/complex_dir/team/sowe

New Shared Source Working Environment
Complex SSWE - C:/complex_dir/team/sswe

New Repository
Complex Repository - C:/complex_dir/team/rwe

**Note: The new project will be opened after the *'Finish'* button is pressed.
Tip: To open projects in private working environments, use the Launch Pad's Working Environment tab. Use the Working Environments Tool to open projects in shared working environments.**

Project root directory: **C:/complex_dir/team/sswe/complex**
Project name: **complex**

Language file types
Header
Implementation

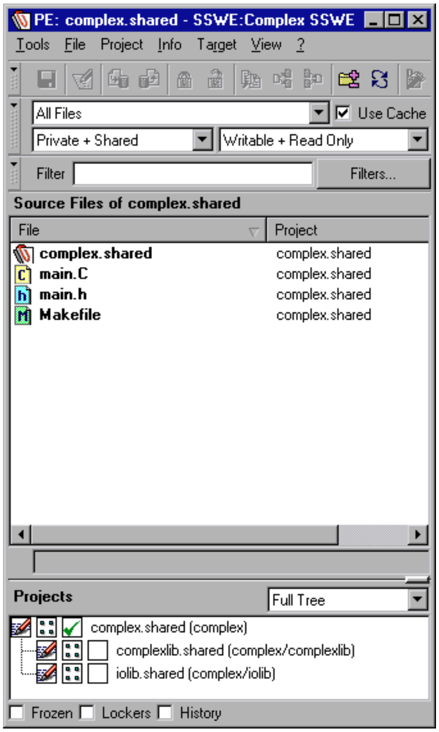
Additional file types
Project Description
Make

2. Press **Finish**.
3. In the dialog that appears asking if you want to generate cross reference information, press **No**.

   SNiFF+ will now create the new `complex` project and all its subprojects. When SNiFF+ is finished, it opens the new project in the SSWE (where you set up the project) and displays its structure and contents in the Project Editor.

Viewing the results

The Project Editor on your screen should look this.



PE: complex.shared - SSWE:Complex SSWE

All Files ☑ Use Cache

Private + Shared

Writable + Read Only

Filter

Source Files of complex.shared

File	Project
complex.shared	complex.shared
main.C	complex.shared
main.h	complex.shared
Makefile	complex.shared

Projects Full Tree

✓ complex.shared (complex)

□ complexlib.shared (complex/complexlib)

□ iolib.shared (complex/iolib)

□ Frozen □ Lockers □ History

Setting up the build system in the SSWE 3

This chapter is about

- setting up the build system in the Shared Source Working Environment (SSWE).

Although you don't build your targets in the SSWE, you set the Make attributes here. Then, when you later open the project in the SOWE and PWEs, you can build targets without first having to modify the project's Make attributes.

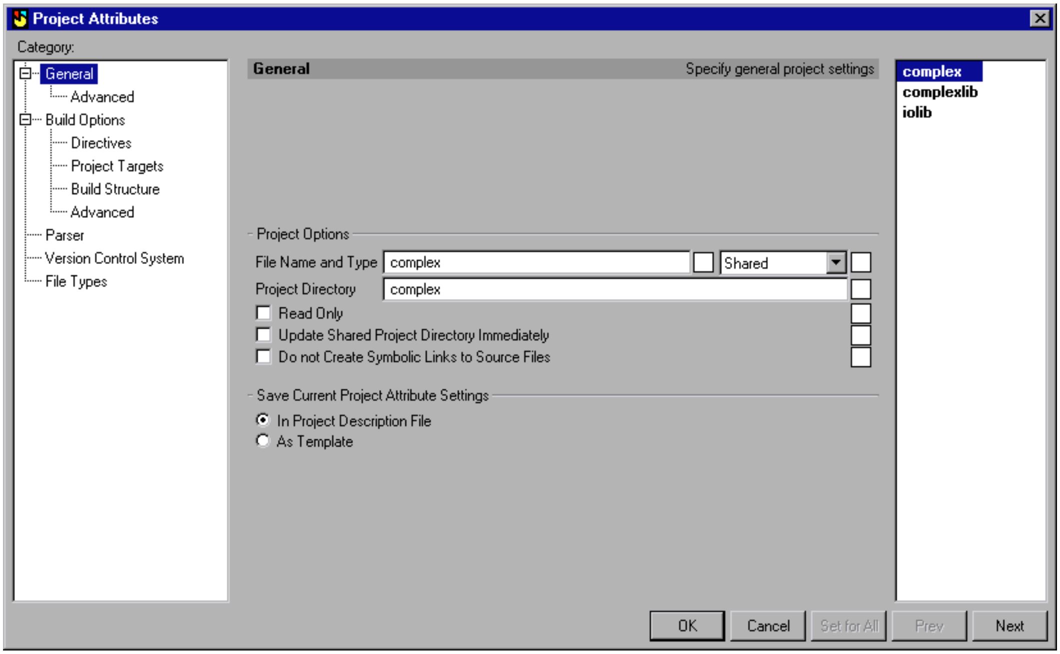
Setting up the build system

In the Project Editor

- In the Project Tree of the Project Editor, choose **Context menu > Select From All Projects** to checkmark all projects.
- Choose **Project > Attributes of Checkmarked Projects...**.

  The Group Project Attributes dialog appears. In this dialog, you can look at and modify the project attributes of multiple projects. For a description of the dialog, please see the *Reference Guide*.

In the Group Project Attributes dialog



File Name and Type: complex □ Shared □

Project Directory: complex □

□ Read Only □

□ Update Shared Project Directory Immediately □

□ Do not Create Symbolic Links to Source Files □

● In Project Description File

○ As Template

Setting up Make Support for complexlib.shared

1. Highlight **complexlib** in the Project List.
2. Under the **Build Options** node, select **Project Targets**.
3. In the **Library** field of the of the Ansi C/C++ tab, enter `complexlib.a`. This will be the name of the library built in this project.
4. Under the **Build Options** node, select **Build Structure**.
5. In the Build Structure view, choose **Passed to Superproject** drop-down **> Library**.

   The project's library is exported to `complex.shared` and is used to build the Complex executable.

Setting up Make Support for iolib.shared

1. Highlight **iolib** in the Project List.
2. In the Build Structure view, choose **Passed to Superproject** drop-down **> Object Files + Received**.

   The project's object file (`iolib.o`) is exported to `complex.shared`.

Setting up Make Support for complex.shared

1. Highlight **complex** in the Project List.
2. Under the **Build Options** node, select **Project Targets**.
3. In the **Executable** field of the Ansi C/C++ tab, enter `complex` (on Windows `complex.exe`). This will be the name of the project's executable.
4. **On Unix only**, enter **-lstdc++** in the **+Libraries Linked** field (below the **Executable** field).
5. Under the **Build Options** node, select **Build Structure**.
6. In the Build Structure view, press the **Generate** button next to the **Recursive Make Dir(s)** field.

   The executable is built using recursive Make rules. By pressing the **Generate** button, SNiFF+ generates the order of subprojects in which Make is executed.

Generating the include paths for all projects

1. Under the **Build Options** node, select **Directives**.
2. Select the checkbox to the right of the **Generate** button.
3. Press the **Set for All** button to generate the include paths for all projects in the Project List.
4. Press **Ok** to apply the changes to the project attributes.

   The icons in the Project Tree of the Project Editor warn you that the projects have been modified.
5. A dialog appears asking you to update Makefiles. We will do this later so press **No**.

In the Launch Pad

To save the changes made to `complex.shared` and its subprojects:

1. Select `complex.shared` in the Project List.
2. Choose **Project > Save Project complex.shared**.
3. In the Alert dialog that appears, press the **Save All** button.

What's next

You may think that the next step is to build the project's executable in the SSWE. It isn't. In SNiFF+'s working environments concept, SSWEs contain only shared source code, and SOWEs contain the objects and targets based on this code.

During project setup, you created a SNiFF+ project in the directory that contains your team's shared source code, i.e., in the SSWE. Once the project has been created, the only time you open it in the SSWE is to update it. For any real development work, open the project in a PWE.

So, the next step is to check in the project (its Project Description File) and its source files into the Repository. When the process is over, all the files in the SSWE will be read-only.

Then, you can open the project in the SOWE and build the executable in it. For details, see First Build in the SOWE — page 121.

Checking In the project from the SSWE 4

This chapter is about

- checking in project files from the SSWE

Checking in project files for the first time is the first step in version-controlling your SNiFF+ projects. We recommend that you version control at least the following types of files:

- Project Description Files (PDFs), i.e., `*.shared` files in our case.
- source files
- Makefiles (only if you don't use SNiFF+'s Make Support)

Once files have been checked in, you can see their history and version tree information.

In a real world situation, it may not matter to you whether your team's shared source code is initially compilable. However, when creating new SNiFF+ team projects from scratch, we recommend that you verify that your source files are compilable before checking them in for the first time. Do not, however, perform builds in the SSWE. The SSWE should only contain source files.

Checking in the project

To check the project in, complete the following steps:

In the Project Editor

1. In the Project Tree, checkmark all projects by right-clicking anywhere in the Project Tree, and then choosing **Context menu > Select From All Projects**.

   You now see all the files in all the projects.

2. Press the **Filters...** button.

   The **Filters...** dialog appears.

3. In the File Types tab, clear the **Make** check box to filter out SNiFF+'s Makefiles from the Project Editor's File List (we assume you are using SNiFF+'s Make Support).

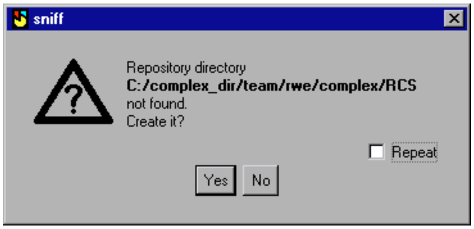
   SNiFF+'s Make Support files are generated and maintained by SNiFF+, so there's no reason to version control them.

4. Press the **OK** button to apply changes and to close the **Filters...** dialog.
5. Choose **File > Select All**.

6. Choose **File > Check In...**.

SNiFF+ informs you that it cannot find the directories of the shared project in the RWE root directory (they haven't been created yet). You will now have SNiFF+ initialize your RWE by copying the SSWE project directory structure into the RWE.

This dialog will reappear for each new Repository directory, unless you select the **Repeat** check box.



Repository directory
C:/complex_dir/team/rwe/complex/RCS
not found.
Create it?

7. Select the **Repeat** checkbox and press **Yes** to create the necessary Repository directories for the project.

When SNiFF+ has finished initializing your RWE, the Check In dialog appears.

In the Check In dialog

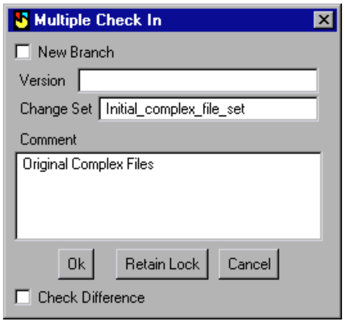
You can use this dialog to check in versions of single or multiple files.When you have made changes to multiple files, you can check in all the files at the same time and associate them with a *change set*. By doing so, you can perform a variety of version-control operations on all the files in a change set at the same time.

At this point, although we haven't made any changes, we will make use of the **Change Set** field to indicate that we are checking in the initial versions of all the files in the project.

1. Leave the **Version** field blank. SNiFF+ will automatically assign a version number (1.1) and later increment it automatically.

2. In the **Change Set** field, enter a name for the change set, e.g., `Initial_complex_file_set.`

3. In the **Comment** field, enter a descriptive text, e.g, `Original Complex Files`.



Version

Change Set Initial_complex_file_set

Comment
Original Complex Files

4. Press **Ok**.

In the Project Editor

When the check-in process is over, take a look at your Project Editor. You should notice the following changes:

- The files in the File List are no longer in bold typeface. This means they are now read-only.
- The icons in the Project Tree have also changed to indicate that the projects, too, are read-only. You can verify this by comparing your screen to the illustration, see Viewing the results — page 110.

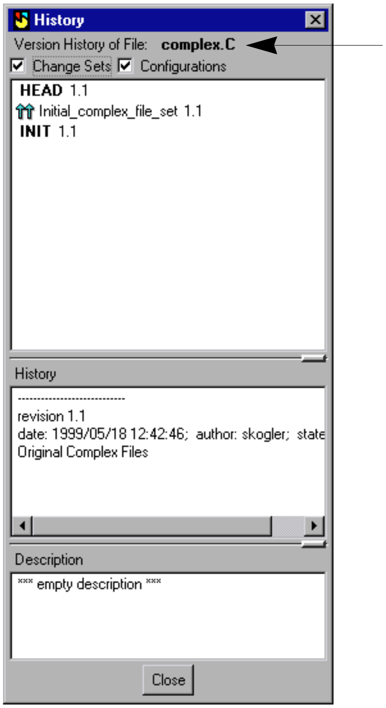
Looking at the history of a file

You can check to see whether the files were checked in properly by looking at their history. Let’s look at the history of a file.

In the Project Editor

1. In the File List, highlight the file `complex.C`.
2. Select the **History** check box (at the bottom of the tool).

   A new History window opens. For a discription of icons used, choose **Help(?) > Quick Ref**.

   History

   Version History of File: **complex.C**

   ☑ Change Sets ☑ Configurations

   HEAD 1.1
   Initial_complex_file_set 1.1
   INIT 1.1

   History

   ----
   revision 1.1
   date: 1999/05/18 12:42:46; author: skogler; state
   Original Complex Files

   Description

   *** empty description ***

   Selected file

   Since only one version of the project files has been checked in so far, the Version Tree only displays this version (`1.1`).

   `INIT` is used by SNiFF+ to refer to the initial version of a file in the Repository. The version number of the `INIT` version of a file is always `1.1`. The latest version on the main trunk or branch of a file’s version tree is called `HEAD`. In this example, the `HEAD` and `INIT` versions of the file are naturally the same.

3. In the Project Editor, clear the **History** check box to close the History window.

What’s next

The next step is to open the project in your SOWE.

Although you can open projects in more than one working environment at a time, this tends to get confusing. We therefore suggest that you first close the project in the SSWE.

In the Launch Pad

To close the project in the SSWE:

1. Highlight `complex.shared - SSWE:Complex SSWE`.

   You can see the name of the project and the working environment in which you opened it by increasing the size of the Launch Pad.

2. Choose **Project > Close Project complex.shared**.

First Build in the SOWE 5

This chapter is about

- **Opening the shared project in the SOWE** — When you first open the shared project in the SOWE, SNiFF+ automatically will initialize the environment by copying the directory structure found in the SSWE.
- **Building and running the complex executable** — A successful build verifies that you have set the project's Make attributes correctly. After the initial build you will have the targets for a stable running version of the project in your SOWE.

During the edit/compile/debug cycle, each developer should only build targets in his/her own Private Working Environment (PWE). Builds in the SOWE should only take place during regular updates of your team's working environments (described in the "Team Maintenance" tutorial). The initial build in the SOWE is in fact your first update of this working environment.

Opening the shared project in the SOWE

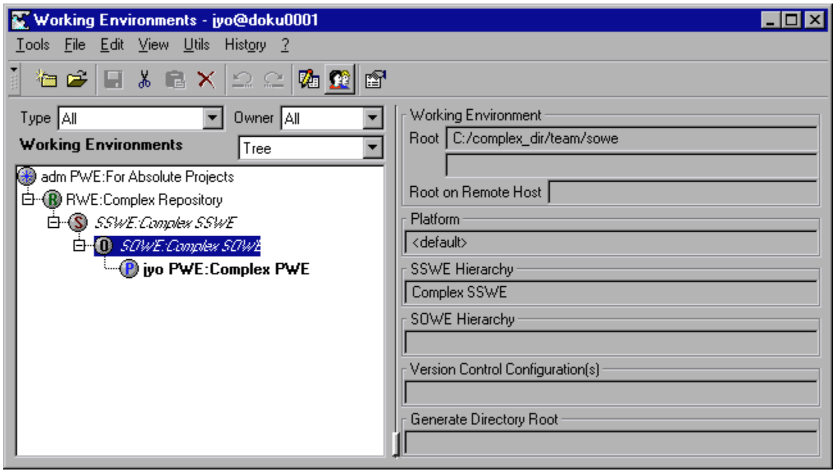
First, you need to tell SNiFF+ that you intend to work in the SOWE. You do this in the Working Environments tool.

- In the Launch Pad, choose **Tools > Working Environments**.

In the Working Environments tool

1. To open a project in the SOWE, double-click on the SOWE entry in the Working Environments Tree. If you used the same names as we did, the full designation of the SOWE is:

   `SOWE:Complex SOWE`



Type All

Owner All

Working Environments Tree

- adm PWE:For Absolute Projects
  - RWE:Complex Repository
    - SSWE:Complex SSWE
      - SOWE:Complex SOWE
        - jyo PWE:Complex PWE

Working Environment

Root C:/complex_dir/team/sowe

Root on Remote Host

Platform

<default>

SSWE Hierarchy

Complex SSWE

SOWE Hierarchy

Version Control Configuration(s)

Generate Directory Root

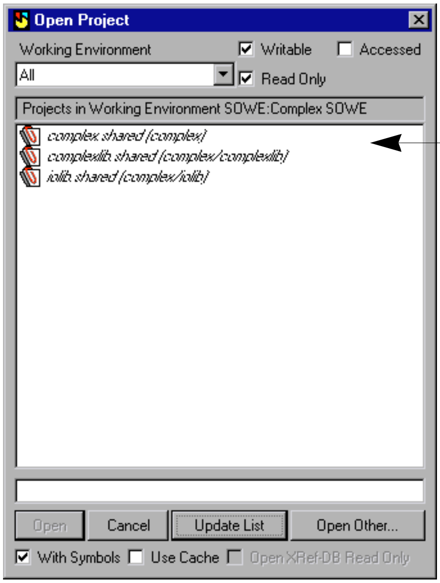
2. In the Open Project dialog that appears, press the **Update List** button to display all the projects that can be opened in the SOWE.

   A dialog appears asking you whether SNiFF+ should also look in any accessed working environments for projects that can be opened in the SOWE. Here, the SOWE accesses the SSWE, so pressing **Yes** will also display the projects in the SSWE.

3. Press **Yes**.

The Open Project dialog on your screen should now look like this.

For a description of the the dialog, move the mouse pointer over the dialog, and press the <F1> key.



Working Environment: All

☑ Writable □ Accessed ☑ Read Only

Projects in Working Environment SOWE:Complex SOWE

complex.shared (complex)
complexlib.shared (complex/complexlib)
iolib.shared (complex/iolib)

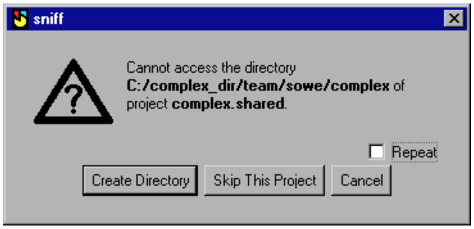
☑ With Symbols □ Use Cache □ Open XRef-DB Read Only

Project List. Projects listed in italics are located in the SSWE

4. To open the root project and all its subprojects, double-click on `complex.shared`.

5. In the dialog that appears, press **Yes**.

SNiFF+ informs you that it cannot find the directories of the shared project in the SOWE root directory (they haven't been created yet). You will now have SNiFF+ initialize your SOWE by copying the SSWE project directory structure into the SOWE.



Cannot access the directory **C:/complex_dir/team/sowe/complex** of project **complex.shared**.

□ Repeat

6. Select the **Repeat** check box and then press **Create Directory**.

Selecting **Repeat** saves you from having to press **Create Directory** for each new project directory.

When SNiFF+ has finished initializing your SOWE, the project is automatically opened in it and displayed in the Project Editor.

7. Close the Working Environments tool.

Building the executable

> **Note**
>
> SNiFF+ doesn't have its own compiler therefore you must have a compiler installed on your computer to compile SNiFF+ projects. By default, the gnu compiler is specified on Unix and Microsoft Developer is specified on Windows. If you are using another compiler, it must be specified in your Platform Makefile. For more information, see *User's Guide — Build and Make Support*.

In the Project Editor

Before building, make sure that the projects' Make Support information is up-to-date. Makefiles should be updated whenever structural changes are made to the projects, or when projects are first opened in a new working environment.

1. Make sure that all the projects in the Project Tree are checkmarked. If they are not, right-click anywhere in the Project Tree and choose **Context menu > Select From All Projects**. This command allows you to checkmark all projects in one step.
2. Choose **Target > Update Makefiles...** to generate the Make Support Files for all the projects.

   A dialog appears asking you whether the dependencies information should also be updated.
3. Press **Yes**.

   SNiFF+ generates the Make Support Files and stores them in the `.sniffdir` subdirectory of each project directory.
4. Highlight `complex.shared` by clicking on its name.

   SNiFF+ needs to know where to start Make execution. You tell SNiFF+ this by selecting the appropriate project. In the example project, Make execution starts in `complex.shared`.
5. Choose **Target > Make > all** to recursively build the executable.

   A Shell opens, in which the `make all` command is recursively executed. Upon completion, you should have an executable named `complex` (on Windows: `complex.exe`) in:

   `<sniff_complex>/sowe/complex`
6. Run the executable to assure yourself that it executes properly. To find out how to do so, see Running the executable — page 87.

   You'll notice that the output is incorrect. This is due to an implementation error; however this error can be safely ignored. (How to fix this error is shown in the Edit/Compile/Debug tutorial).

What's next

- Close the project in the SOWE. To do so:

In the Launch Pad

1. Select `complex.shared - SOWE:Complex SOWE`.
2. Choose **Project > Close Project complex.shared**.

- The next tutorial introduces you to Developing in a team.

Part VII

Developing in a team

Working in the PWE 1

In this chapter, you will go through the basic tasks when working in a PWE in a team context:

- opening the shared project in the PWE for the first time and letting SNiFF+ initialize it for you
- checking out a shared source file and making a minor modification to it
- checking the modified file back in
- creating a file, adding it to, and removing it from a project

This tutorial does not cover the day to day development work or the various browsing tools. For an introduction to the tools used in daily development work, see Edit/Compile/Debug — page 77. For an introduction to the tools used for browsing, see Browsing — page 11.

Opening the shared project in the PWE

First, you need to tell SNiFF+ that you intend to work in the PWE. You do this in the Working Environments tool.

- In the Launch Pad, choose **Tools > Working Environments**.

In the Working Environments Tool

1. To open a project in the PWE, double-click on the PWE entry in the Working Environments Tree. If you used the same names as we did, the full designation of the PWE is:

   Username `PWE:Complex PWE`

2. In the Open Project dialog that appears, press the **Update List** button to display all the projects that can be opened in the PWE.
3. In the dialog that appears, press **Yes** to confirm that shared workspace information should also be used.
4. To open the root project and all its subprojects, double-click on `complex.shared`.

   SNiFF+ informs you that it cannot access the directories of the shared project in the PWE root directory (they haven't been created yet). You will now have SNiFF+ copy the SSWE project directory structure into the PWE.

5. Select the **Repeat** check box and then press **Create Directory**.

   Selecting **Repeat** saves you from having to press **Create Directory** for each new project directory.
   When SNiFF+ has finished initializing your PWE, the project is automatically opened in it and displayed in the Project Editor.

   > **On Windows**
   >
   > Warnings, stating that symbolic links are not supported and that a copy is made instead, appear in the Log window. These can be safely ignored.

6. Close the Working Environments tool.

In the Project Editor

Makefiles should be updated whenever structural changes are made to the projects, or when projects are first opened in a new working environment.
Note that SNiFF+ needs to know where to start Make execution. You tell SNiFF+ this by highlighting the appropriate project. In the example project, Make execution starts in `complex.shared`. So:

1. In the Project Tree, highlight `complex.shared` by clicking on its name.
2. Choose **Target > Update Makefiles**.

   A dialog appears asking you whether dependencies information should also be updated.
3. Press **Yes**.
4. To see all the files in the project structure, checkmark all projects by right-clicking anywhere in the Project Tree and choosing **Context menu > Select From All Projects**.

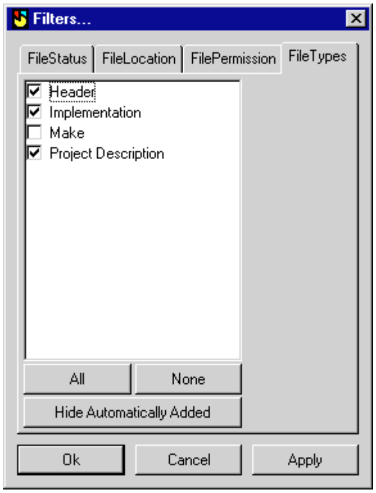
   Notice that all the file names, except for Makefiles, are now *italicized*. This means that you now have a read-only view to these files, which are physically still stored in the SSWE.

   > **On Windows**
   >
   > The read-only files are not italicized because they are local copies in the PWE

5. To filter out SNiFF+’s Makefiles from the File List, press the **Filters...** button.

   The Filters dialog opens.

6. In the File Types tab, clear the **Make** checkbox.



7. Press **Ok**.

   Generally, your Working Environments Administrator is responsible for setting up SNiFF+'s Make Support. Therefore, you don't need to see project Makefiles in your day-to-day work. Also, SNiFF+'s Makefiles are generated and maintained by SNiFF+, so there's no reason to version control them.

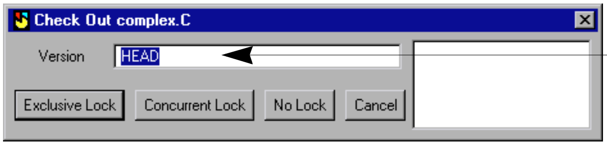
Check out and check in

Remember that you checked in the project to the Repository from the SSWE, see Checking In the project from the SSWE — page 115, so the view to the project files is read only. To modify a file, you first need to check it out.

In the Project Editor

The file which you will modify is `complex.C`, which belongs to the `complexlib.shared` project. To check out `complex.C`:

1. In the Project Tree, make sure that the `complexlib.shared` project is checkmarked, so that you can see its files.
2. In the File List, highlight `complex.C` by clicking on it once.
3. Choose **File > Check Out...**.



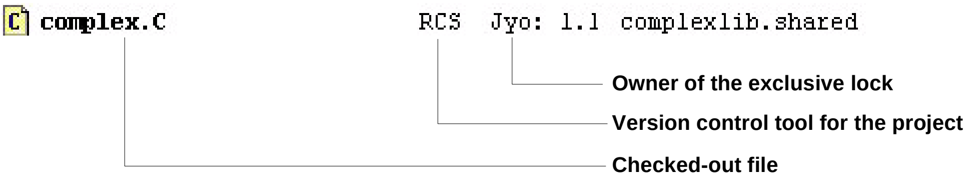
HEAD is the latest version of the file in the Repository

4. In the Check Out dialog, press **Exclusive Lock**.

In the File List, notice that `complex.C` is now displayed in **bold** typeface, which indicates that it is writable.

5. Select the **Lockers** check box at the bottom of the Project Editor. This check box allows you to see which users have locked which files.

If you scroll to the right of the File List, you will notice that the entry for `complex.C` now looks similar to this:



```
C complex.C                    RCS  Jyo: 1.1  complexlib.shared
```

Owner of the exclusive lock
Version control tool for the project
Checked-out file

6. Clear the **Lockers** check box.

7. To load the, now writable, `complex.C` file into the Source Editor, double-click on it in the File List.

In the Source Editor

All we want to do here is to make a modification, so that a newer version of the file exists.

1. Enter a comment in the file.

Notice that the Source Editor now indicates that the file has been modified.

- On **Unix,** the icon in the upper-left corner of the Source Editor indicates that the file has been modified.
- On **Windows NT/95**, the write permissions of the loaded file and its status are indicated in the title bar of the Source Editor.

2. Save `complex.C` by choosing **File > Save**.

3. Close the Source Editor.

Checking in the file

Once you are satisfied with the changes you have made to a file, you check it back in. The rest of the team then has access to the modified file (after the shared working environments have been updated - see next chapter).

- Note that “being satisfied with changes”, above, means that, at the very least, your code is compilable. Do NOT check in untested, possibly uncompilable, code!

  Since you only added a comment to the checked out file, it is safe to check it back in.

In the Project Editor

You can check in files either from the Project Editor or the Source Editor. Here, you will use the Project Editor (the menu command is the same in both tools).

1. In the File List, make sure `complex.C` is highlighted. This is the file you checked out, as you can see by the **bold** typeface.
2. Choose **File > Check In...**.
3. In the Check In dialog, enter a comment in the **Comment** field and press **Ok**.

   You can leave the **Version** field blank, because SNiFF+ will automatically increment the version number to 1.2. Also leave the **Change Set** field blank; this is usually only used for multiple files.

   > **Note**
   >
   > Notice that the file name no longer appears in bold typeface because it is now read-only.

4. To look at the history of `complex.C`, highlight the file in the File List and select the **History** check box at the bottom of the Project Editor.

   Notice that the HEAD version of the file is now version `1.2`.
   Take a look at the history of some of the other project files. Since you have not modified those files since checking them in, their HEAD version is still `1.1`.
5. Clear the **History** check box.

Adding a new file to a project

In the course of your day-to-day development work you will often create new source files. These files must be included in the project they logically belong to. SNiFF+ will do this for you by modifying the Project Description File (PDF) accordingly. But first you have to check out the PDF to make it writable.

In the Project Editor

You will add a new file to the `complex.shared` project. To first get an uncluttered view of the project, and then check out its PDF:

1. In the Project tree, highlight the `complex.shared` project, right-click and choose **Context menu > Select from complex.shared only**.
2. In the File List, highlight the `complex.shared` file (the PDF).
3. Choose **File > Check Out...**.
4. In the Check Out dialog, press **Exclusive Lock**.
5. In the dialog that appears to warn you about project structure changes (you are checking out a PDF), press **Yes**.

   In the File List, notice that `complex.shared` is now displayed in **bold** typeface, which indicates that it is writable.

Creating and adding the new file

Once you have checked out the PDF:

1. In the Project Tree, make sure that `complex.shared` is highlighted.

   You highlight the project so that SNiFF+ knows which project you intend to modify.

2. Choose **Project > Add New File to complex.shared...**

3. In the New File dialog that appears, enter the name of the file you want to create, e.g. `Test.C.`

4. Press **Ok**.

   The new file is added to the File List, and the icon next to `complex.shared` in the Project Tree changes to warn you that the project structure has changed.

> **Note**
>
> SNiFF+ will only allow you to add file types that you have specified as being part of the project. During project setup, see In the “Select file types” page — page 107, you specified the **C/C++** file types. To find out how to add new file types to a project, please refer to the *User’s Guide*.

5. Choose **Project > Save complex.shared**.

   The icon in the Project Tree has changed again; it indicates that the project is writable (the PDF is still checked out). The project information has now been saved, and will be used in your PWE only. As soon as you want to make the file you added available to the rest of the team, check in the PDF again (don’t check it in yet).

Removing files from a project

When you remove a file from a project, the file is not physically deleted; SNiFF+ simply edits the PDF, which means it must be writable (checked out).

In the Project Editor

The `complex.shared` PDF should still be checked out (**bold** typeface).

1. In the Project Tree, make sure the `complex.shared` project is highlighted so that SNiFF+ knows which project you intend to modify.

2. Choose **Project > Add/Remove Files to/from complex.shared...**

In the Add/Remove Files dialog

1. Double-click on `Test.C.`

   The file is removed from the project, but is still physically stored in the directory. Files in the project directory can later be easily added to the project again using this dialog.

2. Press **Ok**.

In the Project Editor

The `Test.C` file no longer appears in the File List, and the icon in the Project Tree warns you that the project has been modified.

1. To save the changes you made, choose **Project > Save complex.shared**.

   The icon in the Project Tree and the **bold** typeface in the File List indicate that the PDF is writable - you haven't checked it in yet.

2. In the File List, make sure the `complex.shared` PDF is highlighted, and choose

   File > Check In...

3. In the Check In dialog that appears, press **Ok**.

   The latest version of the PDF is now in the Repository. After your next working environments update (described in the following chapter), any changes to the project structure will be visible to all team members.

- In the Launch Pad, close `complex.shared -` *Username* `PWE:Complex PWE`.

Part VIII

Team Maintenance

Updating Working Environments 1

When a developer checks out a file, the checked-out version is locked in the Repository, and a local copy is made in the developer's PWE. When a developer is satisfied with changes he/she has made to a checked-out file (compilable!), he/she checks it back in. This means that the new (checked-in) version replaces the older (checked-out) version in the Repository. However, the SSWE still has the older version of the file, and the objects in the SOWE are also based on this version.

Clearly, the working environments are no longer consistent with each other, and they need to be updated so that all PWEs (i.e., their owners) can access the most current state of the project.

Updates should be done on a regular (daily) basis, especially if you have a large development team. The shared working environments (SSWE and SOWE) should only be updated by the Working Environments Administrator. Although it is relatively natural for individual developers to update their PWEs when they start work, this can also be done by the Working Environments Administrator.

You update your working environments in the following order:

- First, update the SSWE - the latest information is taken from the Repository.
- Then, update the SOWE (which accesses the SSWE) and build the targets with the latest file versions. The PWEs access the SOWE, so you can use the up-to-date object code in the SOWE for builds in PWEs.
- Finally, update your PWE so that you have a view to the latest configuration.

Here, only the most basic update requirements are described. For information on more advanced options and unattended updates, please refer to the *User's Guide*.

Updating the SSWE

`complex.shared` is the root project of all the other projects. When you update a root project in a particular working environment, SNiFF+ will automatically update all its subprojects.

First, you need to tell SNiFF+ that you intend to work in the SSWE.

- In the Launch Pad, choose **Tools > Working Environments**.

In the Working Environments tool

1. To open a project in the SSWE, double-click on the SSWE entry in the Working Environments Tree. If you used the same names as we did, the full designation of the SSWE is:

   `SSWE:Complex SSWE`

2. In the Open Project dialog that appears, press the **Update List** button to display all the projects that can be opened in the SSWE.

3. To open the root project and all its subprojects, double-click on `complex.shared`.
4. In the dialog that appears, press **Yes**.

   The project is opened in the SSWE and displayed in the Project Editor.
5. Close the Working Environments tool.

In the Project Editor

1. Checkmark all projects by right-clicking anywhere in the Project Tree, and then choosing **Context menu > Select from All Projects**.
2. Choose **Project > Synchronize Checkmarked Projects...**.

   The Files Compared To dialog appears. All files in the SSWE will be updated to the version that appears in the dialog's **Version** field (HEAD by default).
3. Press **Ok**.

   A progress bar appears, and SNiFF+ updates all the files in the SSWE.
4. In the dialog that appears asking you to reload the project structure, press **Yes**.

In the Launch Pad

- Close `complex.shared - SSWE:Complex SSWE`.

Updating the SOWE

After you have updated the SOWE files, you should compile them. Subsequent builds in PWEs are then quicker, because the compiler uses the up-to-date object code in the SOWE.

First, you need to tell SNiFF+ that you intend to work in the SOWE.

- Use the Working Environments tool to open `complex.shared` in your SOWE (how to do so was described in Opening the shared project in the SOWE — page 121).

In the Project Editor

1. Checkmark all projects by right-clicking anywhere in the Project Tree, and then choosing **Context menu > Select From All Projects**.
2. Choose **Project > Synchronize Checkmarked Projects...**.

   The Files Compared To dialog appears. All the files in the SOWE will be updated to the version that appears in the dialog's **Version** field (HEAD by default).
3. Press **Ok**.

   SNiFF+ now updates all the files in the SOWE.
4. In the dialog that appears asking you to reload the project structure, press **Yes**.
5. Choose **Target > Update Makefiles...** and press **Yes** in the dialog that appears.

   Make Support Files are regenerated for all projects in the working environment.
6. In the Project Tree, highlight `complex.shared`.

7. Choose **Target > Make > all** to build the project's targets.

   A Shell tool opens. The project's Make command is recursively executed in each of the projects in the Project Editor's Project Tree. Upon completion, you should have an executable named `complex` (on Windows: `complex.exe`) in:

   `<sniff_complex>/sowe/complex`

In the Launch Pad

- Close `complex.shared - SOWE:Complex SOWE`.

Updating the PWE

First, you need to tell SNiFF+ that you intend to work in the PWE.

- Use the Working Environments tool to open `complex.shared` in your PWE (how to do so was described in Opening the shared project in the PWE — page 129).

In the Project Editor

1. Checkmark all projects by right-clicking anywhere in the Project Tree, and then choosing **Context menu > Select from All Projects**.
2. Choose **Project > Synchronize Checkmarked Projects...**.
3. In the dialog that appears, press **Ok**.

   SNiFF+ now updates all the files in the PWE.
4. In the dialog that appears asking you to reload the project structure, press **Yes**.
5. Choose **Target > Update Makefiles...** and press **Yes** in the dialog that appears.

   Make support files are regenerated for all projects in the working environment.
6. In the Project Tree, select `complex.shared`.
7. Choose **Target > Make > symbolic_links** to build the `symbolic_links` help target.

   A Shell tool opens. Symbolic links are made in the PWE to all the objects and targets in the SOWE. On Windows NT/95, local copies are made instead of symbolic links.
   This completes the update of the PWE. When you next build targets in your PWE, the results will reflect the latest status of the team project.

In the Launch Pad

- Close `complex.shared - <`*Username*`> PWE:Complex PWE`.

Freezing the Project in the SSWE

2

Goals of this chapter

All your working environments are now up-to-date, your source files are compilable, and the project's executable functions properly. In this chapter, you will learn how to create a "virtual snapshot" of the project (or, to be exact, of its source files). You do this in SNiFF+ by associating the current state (configuration) of all project source files with a single symbolic name. The process of creating a single configuration and associating it with a symbolic name is called "freezing a configuration".

You can freeze configurations in the Configuration Manager. You can also use this tool to view the lists of configurations of your projects and to compare configurations. To learn more about the Configuration Manager, please refer to the *User's Guide* and the *Reference Guide*.

Freezing the project

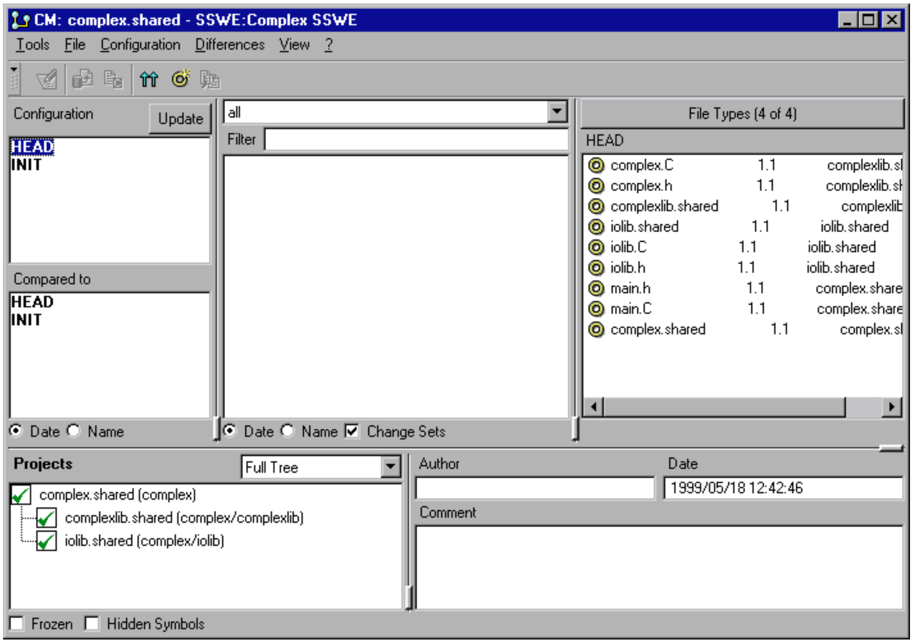
To freeze the project:

1. Open the `complex.shared` project in the SSWE.
2. In any open SNiFF+ tool, choose **Tools > Configuration Manager**.

In the Configuration Manager

1. Select the **HEAD** configuration in the Configuration List (see the following screen shot).

   The project’s configuration information is loaded into the Configuration Manager. Your Configuration Manager should look similar to the following:



Configuration: HEAD, INIT

Compared to: HEAD, INIT

● Date ○ Name

all

● Date ○ Name ☑ Change Sets

HEAD

complex.C	1.1	complexlib.sl
complex.h	1.1	complexlib.sl
complexlib.shared	1.1	complexlit
iolib.shared	1.1	iolib.shared
iolib.C	1.1	iolib.shared
iolib.h	1.1	iolib.shared
main.h	1.1	complex.share
main.C	1.1	complex.share
complex.shared	1.1	complex.sl

Projects: Full Tree

☑ complex.shared (complex)
☑ complexlib.shared (complex/complexlib)
☑ iolib.shared (complex/iolib)

Date: 1999/05/18 12:42:46

□ Frozen □ Hidden Symbols

2. Choose **Configuration > Freeze Head...**.

   The Freeze Head dialog appears.

3. Enter a name for the new configuration in the **Configuration** field of the dialog (e.g. `complex_configuration`) and press **Ok**.

   The Configuration List is now updated to include the newly created configuration.

4. In the Project Editor, take a look at the history of any of the project files. A circle next to one of the file’s versions in the Version Tree indicates that the version is part of a configuration. The configuration name comes after the circle, followed by the version number.

5. Close the `complex.shared` project in the SSWE.

Concluding remarks

This concludes this tutorial on working with C/C++ team projects.
In this tutorial, you:

- set up working environments for a team project
- created the project in the Shared Source Working Environment
- set up the build system for the project in the Shared Source Working Environment
- checked in all project files to the Repository from the Shared Source Working Environment
- built the project's executable in the Shared Object Working Environment
- worked with files (check out, check in, create, add, remove) in your Private Working Environment
- updated the working environments of the project
- froze a stable version of the project in the Shared Source Working Environment

This concludes the C++ tutorial. For detailed explanations of any of the concepts in this tutorial, please refer to the *User's Guide*. To learn more about any of the SNiFF+ tools, see the *Reference Guide*. To learn how you can use SNiFF+ for your C, Java, or Fortran projects, please refer to the respective tutorials.

Colophon

This manual was produced with FrameMaker.

We at TakeFive have tried to make the information contained in this manual as accurate as possible. We cannot, however, guarantee that it is error-free.

© 1992-1999 TakeFive Software GmbH.
All rights reserved.

sniff \'snif\ *vb* -ED/-ING/-S
[ME *sniffen*; prob. akin to ME *snivelen* to snivel]
vt (14c)
3: to recognize or detect by or as if by smelling
<German shepherd dogs are parachuted in the Austrian Alps to *sniff* out survivors of avalanches — P.T.White>

Webster's Unabridged Third New International Dictionary