

XOTcIIDE User Guide

Artur Trzewik
Edited by Bill Paulson

XOTclIDE User Guide

by Artur Trzewik and Bill Paulson
Copyright © 2006 Artur Trzewik

This document contains the user documentation and tutorials for XOTclIDE

XOTclIDE is an Integrated Development Environment for XOTcl and Tcl. XOTcl is an object oriented extension for Tcl. XOTcl can also manage old Tcl code (procs).

XOTclIDE was suggested and inspired by such great Smalltalk graphical environment systems as Squeak and Envy.

XOTclIDE is licensed under the GNU Public License

Copyright Artur Trzewik. License GNU Free Documentation License (GFDL)

Table of Contents

1. XOTclIDE Overview	1
About this document	1
Main Features	1
Benefits	1
Ancestry	2
2. Getting Started Tutorial	4
Developing Your First Application - Tutorial	4
Starting XOTclIDE	4
Creating new Components	5
Adding Tcl procedures	6
Interactive work with procedures	7
Saving Component in File System	8
Using Components without XOTclIDE	9
Loading a Package or Component from File System	9
Creating Configuration Maps and Distributing Programs	10
Evaluating Short Tcl Scripts	11
Advanced Usage: Overview	12
Object-Orientation with XOTcl Components and Object Introspection - Tutorial	12
Load Sample Application	12
Create an Instance of a Class	13
Object Inspector	14
3. Programming with XOTclIDE	17
System Requirements and Installation	17
XOTclIDE Programs and Start Options	17
Starting XOTclIDE from tclkit	17
Starting XOTclIDE without Version Control System	17
Starting XOTclIDE with Version Control System	17
Starting XOTclIDE with Version Control System by loading from Version Control	18
Options synopsis	18
Building an Application	18
XOTclIDE Components	20
Component lifecycle	20
Browsers and Inspectors	21
Writing Source	21
Refactoring	22
Debugging running Systems	22
Version Control	23
Configuration Management and Deploying	23
Special Browsers	23
Source Editing	23
Basic Editor Function	23
Code Completion	24
Navigation in Sources	25
Syntax highlighting	25
Checking parentheses	25
Automatic Indenting	26
Evaluate Tcl Scripts in Editor	26
4. Extended Features	29
Version Control System	29
Benefits of Version Control	29
Base Characteristics	29
Principle	29
Database Schema of Version System	30

Definitions (Editions, Versions)	31
Using Version System, Main Functions	31
Changes Browser	32
Component Loader	33
Installing Version Control System	33
Syntax Checking	34
Reason for syntax checking in Tcl/XOTcl	34
Syntax checker implementation	34
Example Tcl procedures	35
Example XOTcl methods	35
Syntax Checking while editing	35
Syntax Checker Browser	36
Tcl/XOTcl Parser	37
How to extend syntax interpretation	37
Problems	38
Magic strings for checker	38
Checking Referenced Object Calls	38
Configurations Management	39
Main Features	39
Configuration Map - Without Version Control System	39
Using Configurations Maps	40
Deploying Application	41
Configuration Browser - with Version Control System	41
Debugging	43
Debugger Browser	43
Stack Error Browser	44
Tracker Browser	45
Variable access tracking and watching	46
Importing Tcl Projects into XOTclIDE Components	47
Importing by definition tracking	47
Importing by System Introspection	49
Importing Tcl comments	50
Plug-ins Architecture	51
5. Additional Information	53
Author and License of XOTclIDE	53
XOTclIDE WWW Resources	53

List of Figures

1.1. Ancestry of XOTclIDE	2
2.1. Component Browser	4
2.2. Create Component	6
2.3. Invoking Procedures	7
2.4. Saving Components	8
2.5. Configurations Map Browser	10
2.6. Workspace	11
2.7. Sample Component in Component Browser	13
2.8. Create Instance Dialog	14
2.9. Sample Railway application	14
2.10. Object Inspector	15
2.11. Methods in Object Inspector	15
3.1. UML Structure of Components	18
3.2. Code Completion	24
3.3. Syntax Highlighting	25
3.4. Checking Parenthesis	25
3.5. Evaluate Scripts	26
3.6. Substitute Scripts	27
3.7. Inspect Script Evaluation	27
4.1. Version System Principle	29
4.2. Version System ER Diagram	30
4.3. Changes Browser	32
4.4. Syntax Checker Dialog	36
4.5. Syntax Checker Tool	36
4.6. Configuration Map Browser	40
4.7. Configuration Browser	41
4.8. Debugger	43
4.9. Error Stack Browser	44
4.10. Method Call Tracker	45
4.11. Variable Tracker and Variable Watch	46
4.12. Load Package Dialog	49
4.13. Comment Scanner Tool	50

Chapter 1. XOTclIDE Overview

XOTclIDE is an Integrated Development Environment for the XOTcl [<http://www.xotcl.org>] programming language. XOTcl is an object oriented extension for Tcl. XOTclIDE can also be used to program Tcl code that does not use XOTcl. XOTclIDE was suggested and inspired by such great Smalltalk graphical development environment systems as Squeak and Envy.

About this document

This document is the user guide for XOTclIDE. The first chapter is a short tutorial that demonstrates the specifics of programming with XOTclIDE. The next chapter describes the use of XOTclIDE. Many functions and characteristics are not described in this document because they are assumed to be quasi standard in similar applications. Therefore component development and object-orientation will be not explained in this document.

This documentation does not contain the programmer guide for programming in Tcl and XOTcl. Suitable documentation can be found on the Internet (see the section called “XOTclIDE WWW Resources”).

Main Features

- Many source editing features for both Tcl and XOTcl, such as syntax highlighting and code completion
- Debugging
- System Introspection
- Source Documentation
- Source Syntax Checking for Tcl and XOTcl
- Version Control System
- Team Programming Support
- Testing Support (Unit Test Framework)
- Configuration Management

Benefits

- XOTclIDE can manage large projects with thousands of lines of Tcl or XOTcl program code.
- XOTclIDE supports component based development that helps to structure and reuse code in different applications.
- XOTclIDE enables interactive XOTcl development, using the introspection functions of XOTcl. There is no difference between using, developing, debugging and browsing (introspection) the system. You work on a live XOTcl system.

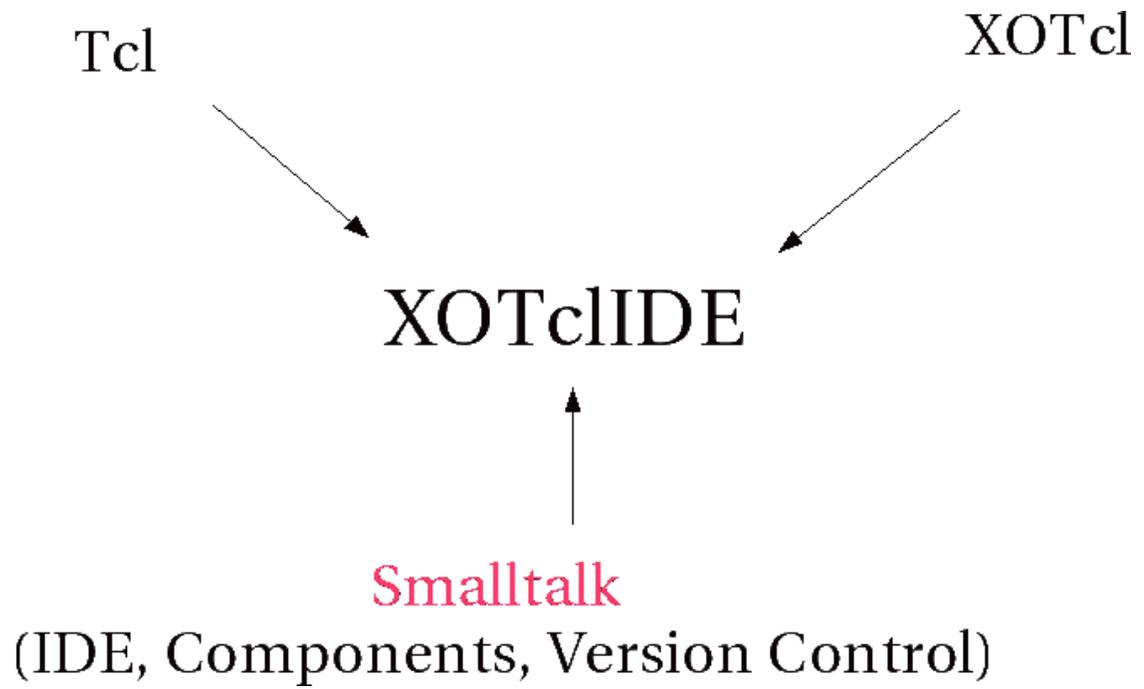
- XOTclIDE is a set of structure browsers. They can be used to browse class and object structure and add and modify the methods or meta-data.
- The XOTclIDE Object Inspector lets you browse and manipulate all XOTcl objects. You can navigate through sub-object structures, inspect and modify variables, or invoke objects method with a GUI interface.
- Includes a powerful syntax checking tool to find all typos while editing.
- XOTclIDE manages normal Tcl procedures, allowing reuse of old non-Xotcl code or mixing of Tcl and XOTcl in one project.
- Supports programming process. Version Control, Unit Tests Framework
- XOTclIDE is easily adapted and extended by the user
- XOTclIDE can be extended with dynamic loadable modules (plugins)
- XOTclIDE enables team development.

Ancestry

XOTclIDE does not try to invent “yet another way” to write computer programs. Rather it is a synthesis of three streams - Tcl, XOTcl and Smalltalk - in the computer world, along with the author's experience. Indeed XOTclIDE is a Smalltalk like IDE programmed in Tcl/XOTcl. What are the main ideas taken from these streams?

- TCL - is the base programming language and platform for this system. Flexible string oriented interpreter allows implementing many ideas in a short time. TCL is widely used, has many additional libraries, offers a GUI Toolkit (TK) and supports many platforms.
- XOTcl - enables flexible object-oriented language support for Tcl. Allows programing and structuring large application, and allows reuse of code in an object oriented manner. The implementation of XOTcl in C brings reasonable performance.
- Smalltalk - the model for a big IDE. How to work with it and use all advantages of interpreted language and dynamic/interactive programming. It was also the reference system for some basic programming practices and tools (Debugger, Version Control, Unit Tests, Object inspector, Browsers). Although Smalltalk lost popularity several years ago it has influenced many programming languages (C++, Java, C#) and IDEs so the concepts will be familiar to many programmers.

Figure 1.1. Ancestry of XOTclIDE



Chapter 2. Getting Started Tutorial

This chapter describes the first steps in using XOTclIDE. You will learn how to write your first XOTclIDE application, and how to save and reload it.

Developing Your First Application - Tutorial

In this section you will learn how to manage the basic tasks of programming with the XOTclIDE: how to create a new project (component), how to add new source, how to save the project and how to reload it. This section assumes a basic knowledge of Tcl and programming tools and is designed for programmers with some experience with Tcl or XOTcl.

Starting XOTclIDE

Change to your chosen working directory and start XOTclIDE.tcl in it. The command line to start XOTclIDE depends on your installation. For example, using a Linux RPM installation:

```
[artur@rybnik xotclIDE]$ tclsh /usr/local/lib/xotclIDE/XotclIDE.tcl  
or  
[artur@rybnik xotclIDE]$ /usr/local/lib/xotclIDE/XotclIDE.tcl
```

On Windows systems you can just click on XOTclIDE.tcl.

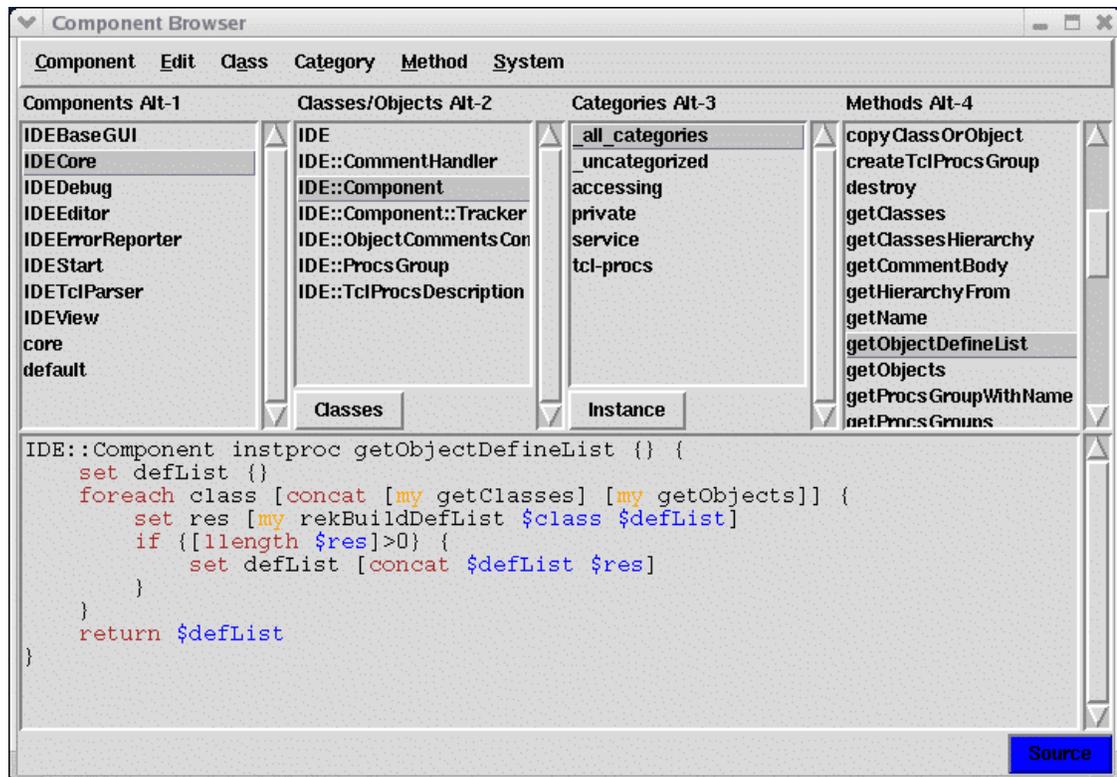
After loading two windows appear. One window is the **Transcript** window with a welcome message. This is used to display system messages and evaluate short scripts. The second window is the **Component Browser**, the main browser used to explore and modify your system structure and program code.

The Component Browser is divided into five panes: Component, Classes/Object/Tcl Procs Groups, Categories, Methods and Text Editor. The four upper list-views correspond to source code structures within XOTclIDE. Each of the four has an associated menu. The highest level structure is a Component. A Component has a specific name and is a container for Classes, Objects and Tcl-Procedures-Groups. Each Class can have instance methods or class methods, corresponding respectively to XOTcl's instprocs and procs. These methods can be grouped together in Categories. Standard non object oriented Tcl procedures can be part of a Component when grouped in Tcl-Procedures-Groups. Individual Tcl procedures may not belong to Categories. The Methods pane lists the names of methods (procs and instprocs) in the selected Class and Category. The three levels (Component, Class/Object/Tcl-Procedures-Group, and Category) may be used to practice component oriented programming (More Information the section called "Building an Application").

Exercise

Select component IDECore in the Components list view. In the Class/Objects list view all classes contained in this component will be listed. Select class IDE::Component in the Class/Objects list view. Select category _all_categories in Categories list view. Select method getObjectDefineList in the Methods list view. In the editor area the body (source code) of this method will appear. The browser will look like screen-shot Figure 2.1, "Component Browser". The blue background of the Source button in the right lower corner indicates that this method has a programmer comment. Press it to see this comment. The editor changes to comment view. Press the button again to change back to source view. Press button Objects in the Class/Objects list view to see another type of element of this component.

Figure 2.1. Component Browser



Creating new Components

Programming in XOTclIDE is component oriented. Assume that you want to write simple program as follows.

```
# This procedure dump elements of global array Tcl_platform
# to file with name specified by parameter "file"
proc generateTclPlatformProtocol {file} {
    global tcl_platform
    set fhandler [open $file w]
    foreach key [array names tcl_platform] {
        puts $fhandler "$key = $tcl_platform($key)"
    }
    close $fhandler
}
generateTclPlatformProtocol tclPlatform.log
```

A regular Tcl-Programmer would open his favorite editor, type the text and save it as a Tcl script. Perhaps he would not create the procedure generateTclPlatformProtocol but would program it directly in the global context as follows:

```
set fhandler [open tclPlatform.log w]
foreach key [array names tcl_platform] {
    puts $fhandler "$key = $tcl_platform($key)"
}
close $fhandler
```

This can be good for small scripts. To program larger systems or to create reusable code a good programmer would prefer to write a Tcl-package like this:

```
package provide PlatformLogDumper 0.1
# This procedure dump elements of global array tcl_platform
# to file with name specified by parameter "file"
proc generateTclPlatformProtocol {file} {
    global tcl_platform
    set fhandler [open $file w]
    foreach key [array names tcl_platform] {
        puts $fhandler "$key = $tcl_platform($key)"
    }
    close $fhandler
}
```

You would also need to create a file `pkgIndex.tcl` and add its directory to the Tcl `auto_path` global variable. You can use the package as follows

```
package require PlatformLogDumper
generateTclPlatformProtocol tclPlatform.log
```

The Tcl package mechanism allows the function library definition to be separated from the function call.

Important

XOTclIDE Components are normal Tcl packages that contain additional meta information that is handled by XOTclIDE.

Let's make make a first Component. Choose menu **Component->New** from **Component Browser**. In the dialog enter the name of the new component "PlatformLogDumper" and click the apply button. Your new component will appear in the list of components. In the next section you will learn how to add procedures to this component.

Adding Tcl procedures

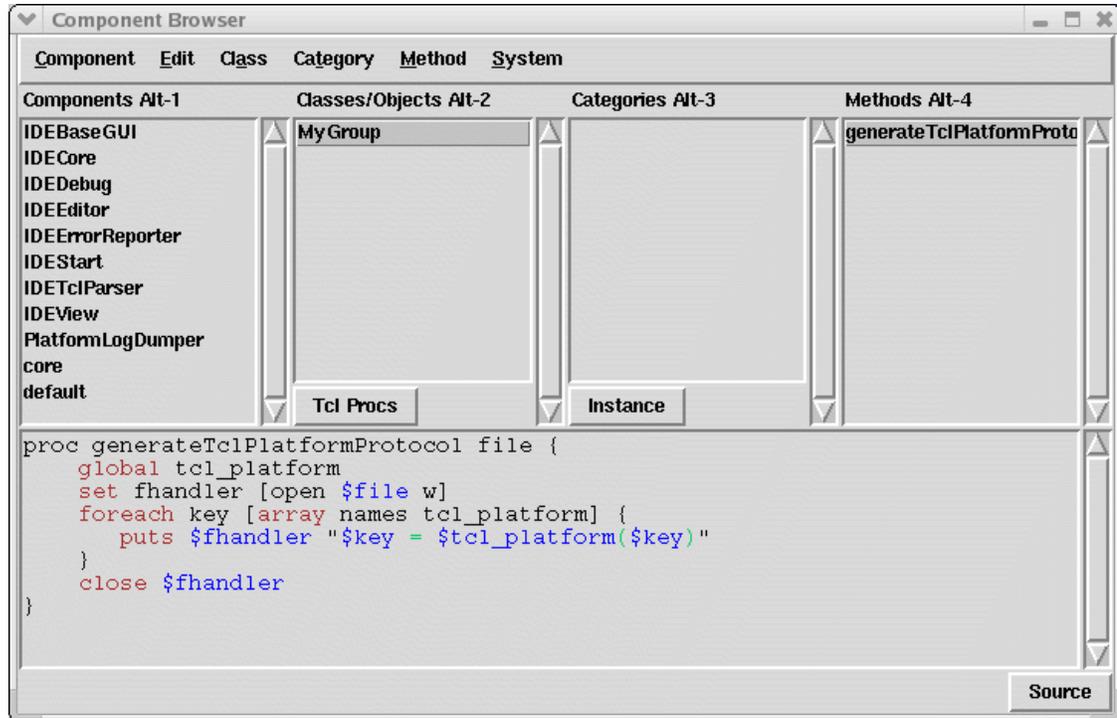
Your new created component appears in the Components list view. Select it. Now create a new Tcl-Procedures-Group using the menu **Class->Tcl Procs Group->New Group**. In the dialog type the name of the group.

The name of the newly created Tcl-Procedures-Group will appear in the **Class/Objects** list view. Select it. Now you can create your procedure. Select menu **Method->New Method Template** In the editor window the template for a Tcl procedure will appear as follows:

```
proc procName {args} {
    # enter the body here
}
```

Type your new procedure. When you are ready you must add the procedure to the interpreter. Use menu or key accelerator **Edit->Save/Apply (Control-s)** Your **Component Browser** will appear as follows: Figure 2.2, "Create Component"

Figure 2.2. Create Component



If you want to create a comment for this procedure press the Source button in the right lower edge of the edit area. The editor pane will change to comment view. Type a comment and apply it with Edit->Save/Apply (**Control-s**)

Interactive work with procedures

The main advantage of XOTclIDE is that it is not only an editor for Tcl scripts but it “sits” directly on a Tcl-Interpreter. Tcl for XOTclIDE is the same as Lisp for Emacs. You can try (invoke, call) a procedure immediately after you define it. Select the procedure and choose the menu Method->Invoke You will be ask to specify the parameters for the procedure call. In Tcl everything is a string so it is no problem just to type the parameters in the dialog as Tcl words (e.g “2 {2 3}” are two parameters)

Figure 2.3. Invoking Procedures

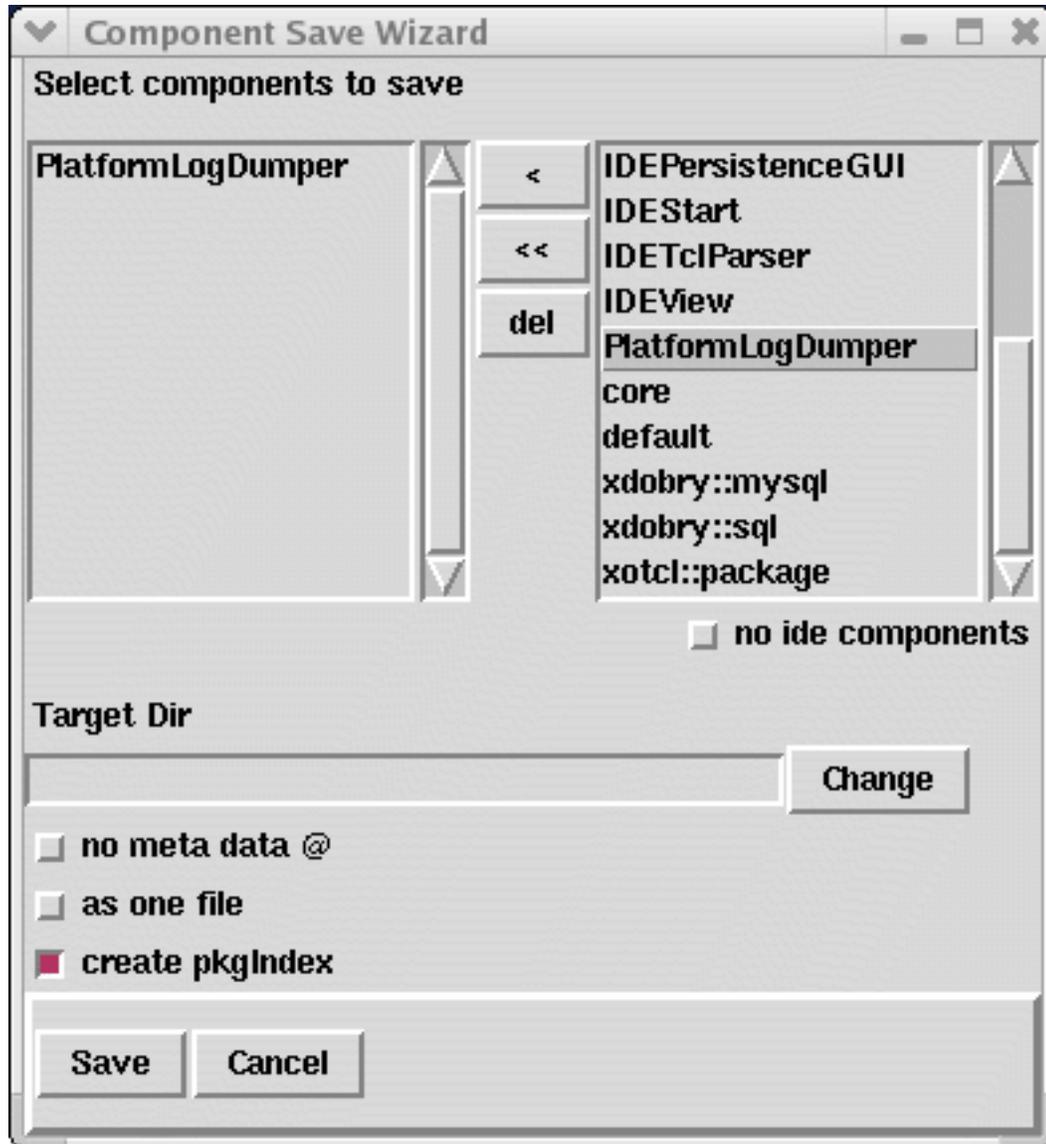


The result is returned as result text. If the result is an XOTcl object or list of XOTcl objects the object inspector will be displayed. If the result is an empty string then a special message box is shown.

Saving Component in File System

To save your new component in the file system use the menu **Component Browser** Component->Save Components. In the dialog box select your component and check the button create pkgIndex (see Figure 2.4, "Saving Components").

Figure 2.4. Saving Components



After accepting dialog a package file PlatformLogDumper.xotcl will be saved in a filesystem directory. The file should look as follows.

```
# automatically generated from XOTclIDE
package provide PlatformLogDumper 0.1
```

```
@ tclproc generateTclPlatformProtocol idemeta struct \  
PlatformLogDumper MyGroup  
proc generateTclPlatformProtocol file {  
    global tcl_platform  
    set fhandler [open $file w]  
    foreach key [array names tcl_platform] {  
        puts $fhandler "$key = $tcl_platform($key)"  
    }  
    close $fhandler  
}
```

Note that the component was saved as a “regular” Tcl package. The one difference is the line beginning with @. This is XOTcl notation for meta-data and is used in XOTclIDE to code additional structure information. To use this package from pure Tcl you can check the button no meta data @ before saving the component or define a dummy proc to ignore the meta-data as follows

```
proc @ args {}
```

The checked option create pkgIndex caused the generation of the file pkgIndex.tcl in the same directory as the package.

```
# Tcl package index file, version 1.1  
# This file is generated by the "pkg_mkIndex -direct" command  
# and sourced either when an application starts up or  
# by a "package unknown" script.  
It invokes the  
# "package ifneeded" command to set up package-related  
# information so that packages will be loaded automatically  
# in response to "package require" commands.  
When this  
# script is sourced, the variable $dir must contain the  
# full path name of this file's directory.  
package ifneeded PlatformLogDumper 0.1 \  
[list source [file join $dir PlatformLogDumper.xotcl]]
```

In this index file appear all packages from the directory with names that match the pattern *.xotcl.

Using Components without XOTclIDE

As mentioned above, XOTclIDE components are normal Tcl packages. To use them from Tcl scripts you need first to register the package in Tcl package system by setting the auto_path global variable. If you do not use XOTcl, define a dummy procedure to ignore @ lines. Here is a sample usage from the tcl shell.

```
[artur@localhost own_oxtcl]$ tcl  
tcl>proc @ args {}  
tcl>lappend auto_path .  
/usr/share/tcl8.3 /usr/share /usr/lib /usr/share/tclX8.3 .  
tcl>package require PlatformLogDumper  
0.1  
tcl>generateTclPlatformProtocol out.log
```

Loading a Package or Component from File System

To load a package or component into XOTclIDE use the menu Component->Load Package. If you do not see your component in the package list, this means Tcl is unable find the package. The package list

is generated using the Tcl **package names** command which uses the `auto_path` variable as a list of directories to search for packages. XOTclIDE adds the current working directory to the `auto_path` list at start time. One way to load your package is to change into the directory with your packages before starting XOTclIDE.

Normally you do not need to worry about how the components are saved or loaded in the filesystem or how a component is represented in text. You can install your component in Tcl distribution subdirectory. Read the Tcl **package** command manual for more about the package mechanism in Tcl.

Warning

It can be quite tricky to force Tcl to find and load packages you need. Some people use the phrase “Package Hell”. The package list are built only once and cannot be rebuilt if you change packages dynamically. You will need to restart XOTclIDE to load packages you have just created if your components are stored as files. You would not have this problem if you use XOTclIDE's Version Control.

Creating Configuration Maps and Distributing Programs

Components in XOTclIDE are like function- or class libraries in other systems. You can build your application as one or more components. You also need an application starting script that contains a line like this

```
generateTclPlatformProtocol out.log
```

You can of course write you own start script as follows and save it as a file

```
lappend auto_path $pathToComponentFile
proc @ args {}
package require PlatformLogDumper
generateTclPlatformProtocol out.log
```

Alternatively, XOTclIDE supports configuration maps that group components together and specify a start script. A configuration map specifies

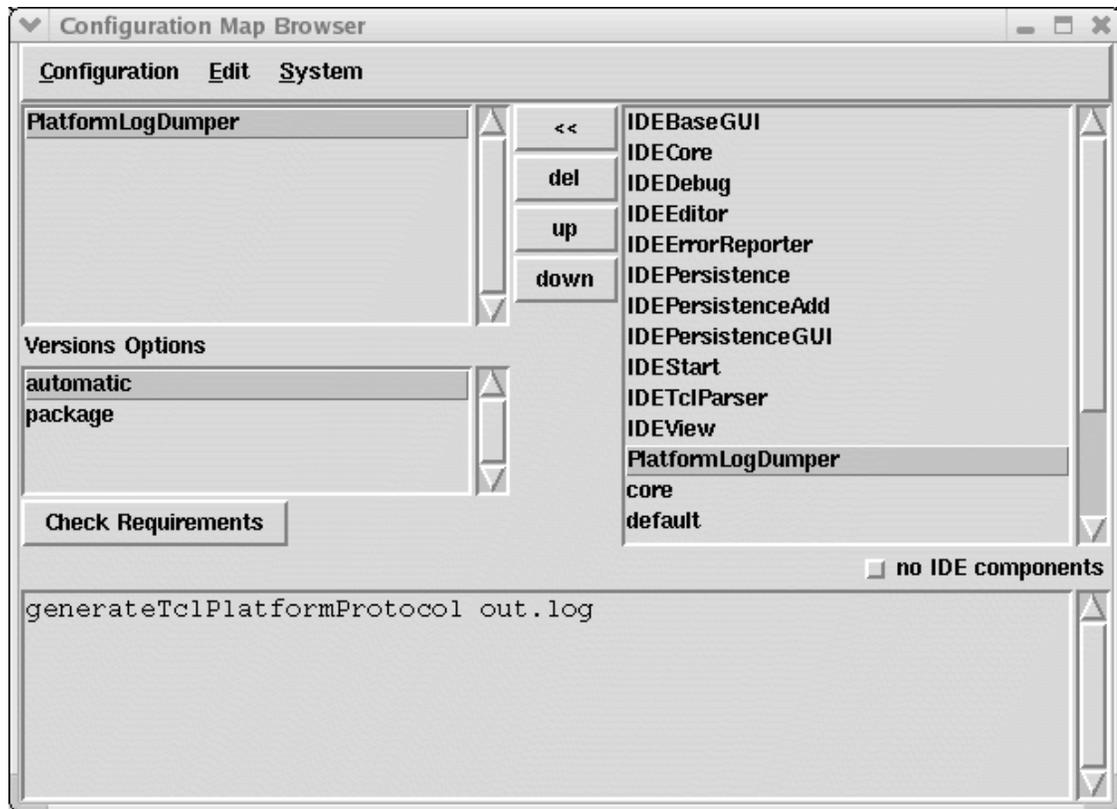
- Components to load (the order to load and how they should be loaded from file-system or version control system)
- `preStartScript` - a script that will be evaluated before loading components
- `startScript` - a script that starts the application after components are loaded

Configuration maps may be also used to deploy applications.

Tip

“Configuration maps” correspond to project or “solution maps” in other IDEs. To specify a configuration map you use the **Configuration MapBrowser** that can be started from the menu `System->Configuration Map Browser`.

Figure 2.5. Configurations Map Browser



Do not forget to apply the start script from the menu Edit->Save/Apply (**Control-s**). After defining a configuration map you can save it as a configuration map file (extension `.cfmap`).

You can set this configmap file as a start parameter of XOTclIDE so all components from this configuration map are loaded at starting time.

```
[artur@localhost own_oxtcl]$ XotclIDE.tcl -- -configmap platformLogDumper.cfmap
```

If you develop a big application it's a good idea to define a configuration map and use it as a start parameter. (see the section called "Configuration Management and Deploying" for more informations)

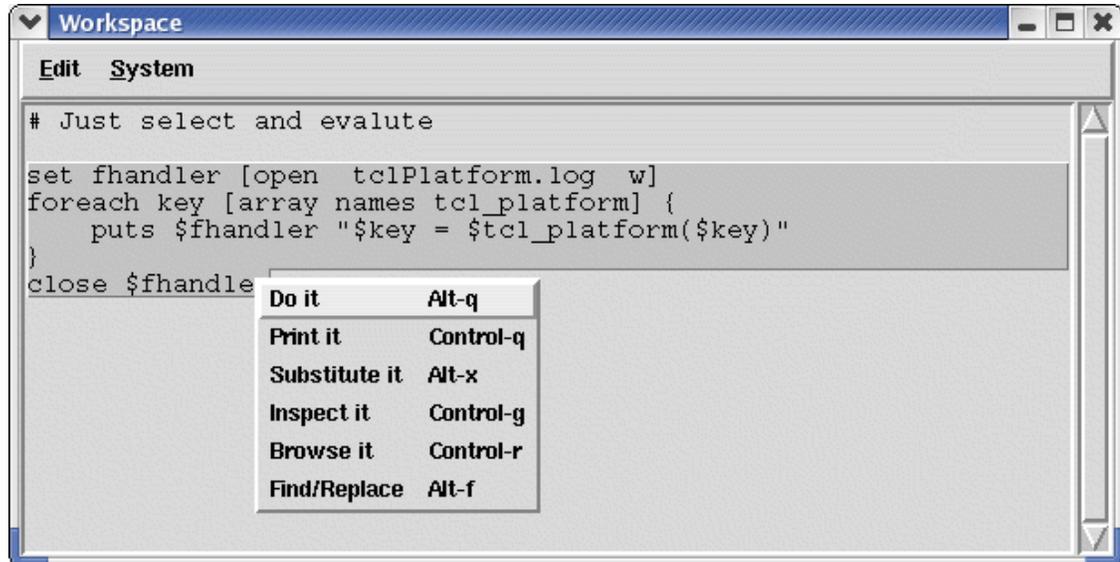
Evaluating Short Tcl Scripts

Tcl is often used to write short ad hoc programs called scripts (Scripting Language). Scripts are often sequences of simple Tcl commands that are evaluated in the global context. Often no procedures are defined in such scripts.

Every Editor Text Area in XOTclIDE has the ability to evaluate short scripts. To run (evaluate) the script just select the script text and invoke it from the pop-down menu (Right mouse-button) or the menu Editor.

You can load and manage your scripts by using **Workspace** windows. To open a **Workspace** use the menu System->Workspace A **Workspace** is simply a text area (simple Editor) that can be used to type scripts, save or load them from the file system. It can be also used as a temporary "scratch pad", to enter and evaluate expressions during the process of testing new method definitions.

Figure 2.6. Workspace



Advanced Usage: Overview

The tutorial above describes how to get your first results with XOTclIDE. The main advantages of this system can be seen by using the advanced features of XOTclIDE:

Version Control System	integrated object oriented version control system lets you track all changes in a project and restore old versions (the section called "Version Control System").
Object orientation with XOTcl	XOTclIDE was developed primarily to support object oriented development with XOTcl. Many features - e.g. the object inspector - are XOTcl specific.
System Introspection	XOTclIDE lets you inspect and change all objects and variables in the system
Debugging	With the extended debugger extension you can debug your program with a professional system. Conditional breakpoints, program stepping, program stack introspection are possible (the section called "Debugging").

Object-Orientation with XOTcl Components and Object Introspection - Tutorial

In this section you will learn about the dynamic and object-oriented aspects of XOTcl programming in XOTclIDE.

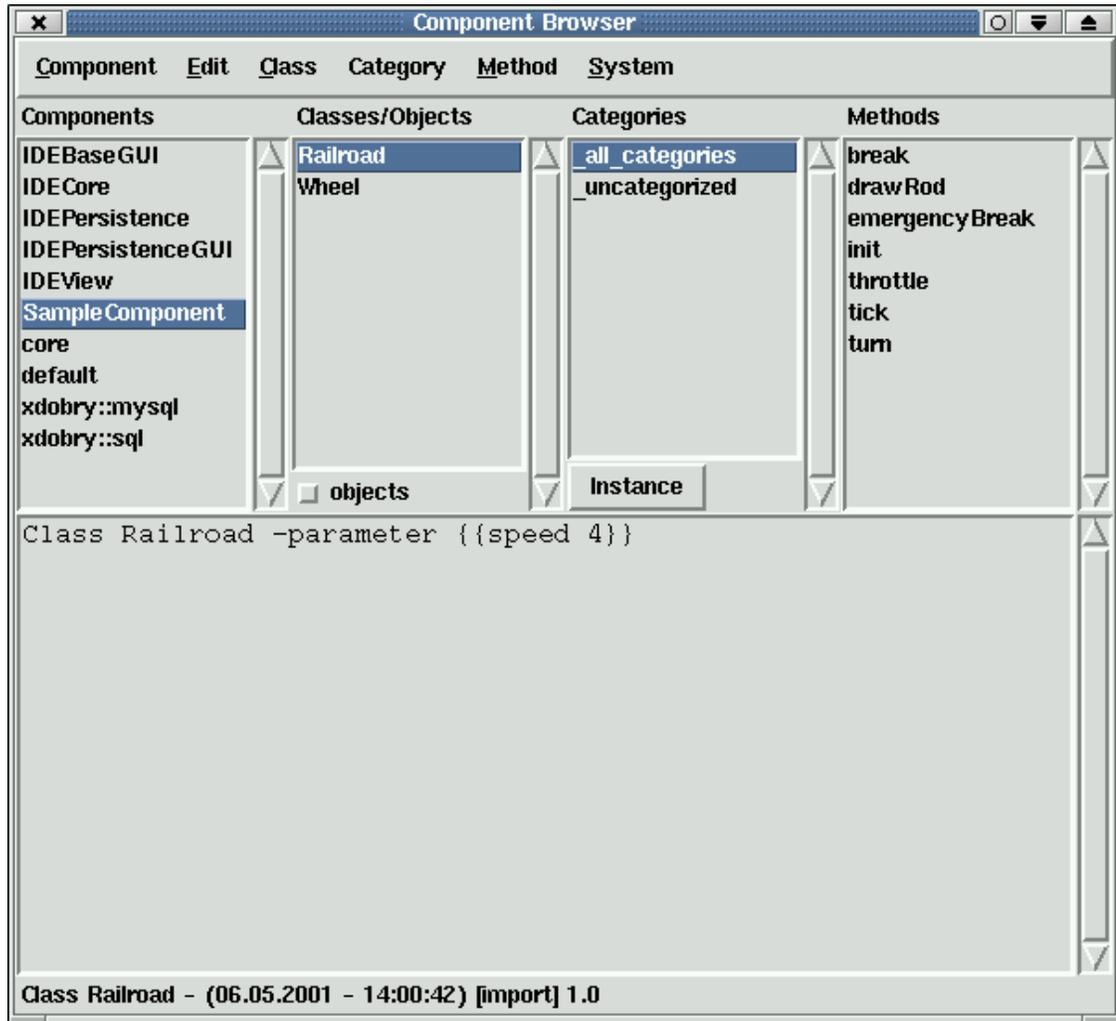
Load Sample Application

Start XotclIDE:

```
[artur@rybnik xotclIDE]$ ./XotclIDE
```

Now load the component *SampleComponent* 1 that is included in the XOTclIDE distribution. Choose the menu Component->Load Package in Component Browser. From the dialog-box select the “SampleComponent” package to load. Now you should see the following in your Component Browser window:

Figure 2.7. Sample Component in Component Browser



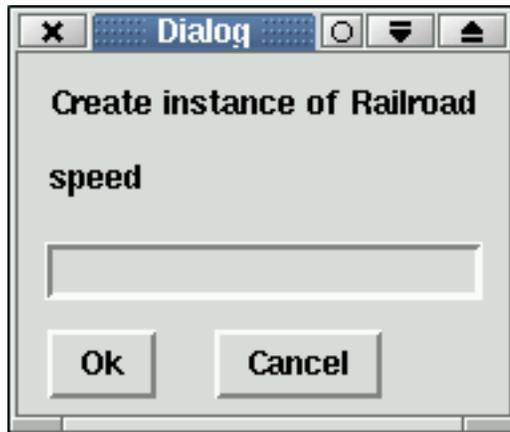
The Component has two classes (Railroad and Wheel). You can browse the methods of these classes to see how the component works.

Create an Instance of a Class

Now try creating some instances of the *Railroad* class. Select the *Railroad* class in Class view and choose the menu Class->Create Instance. You will see the following dialog

¹ Example by Richard Suchenwirth translated from Tcl to XOTcl by Gustaf Neumann and modified by me. First published on tcl-wiki.

Figure 2.8. Create Instance Dialog



In this dialog you can specify the arguments (or additional arguments) for the init method (passed to init method or parameters). The class definition of *Railroad* is

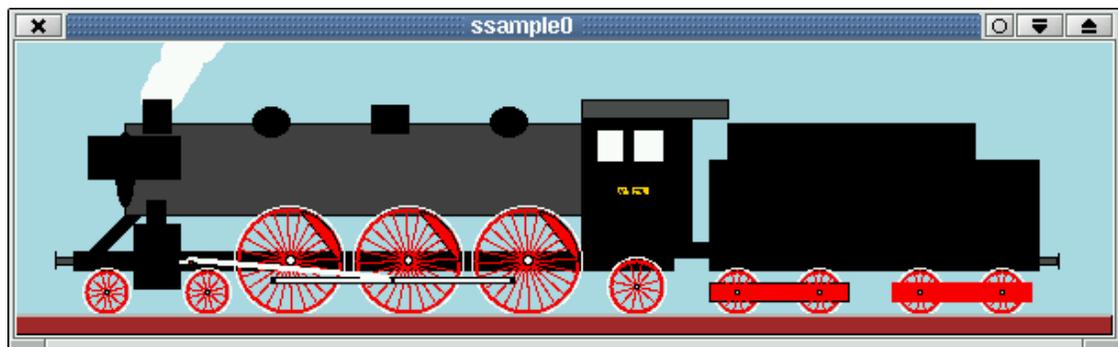
```
Class Railroad -parameter {{speed 4}}
```

You could modify the start speed by specifying a different speed value (try "-speed 10"). For now you can simply push the OK button and use default speed 4.

Object Inspector

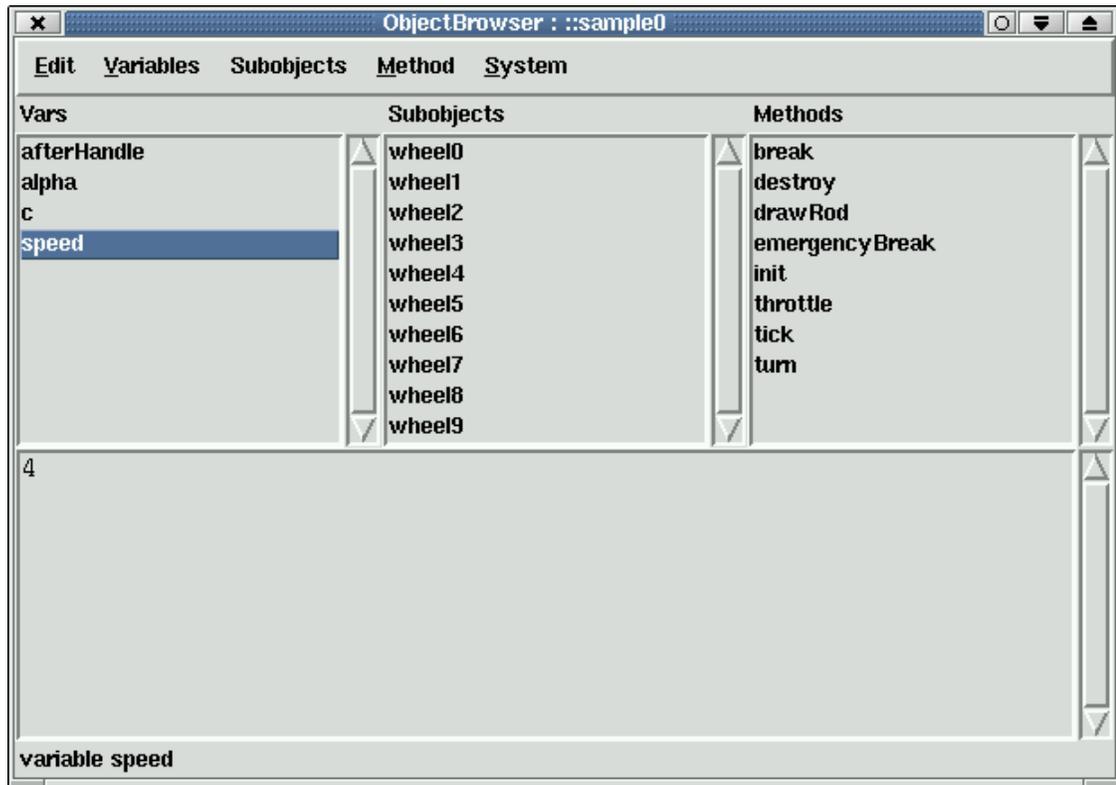
The new created object of class Railroad builds a window and starts to run.

Figure 2.9. Sample Railway application



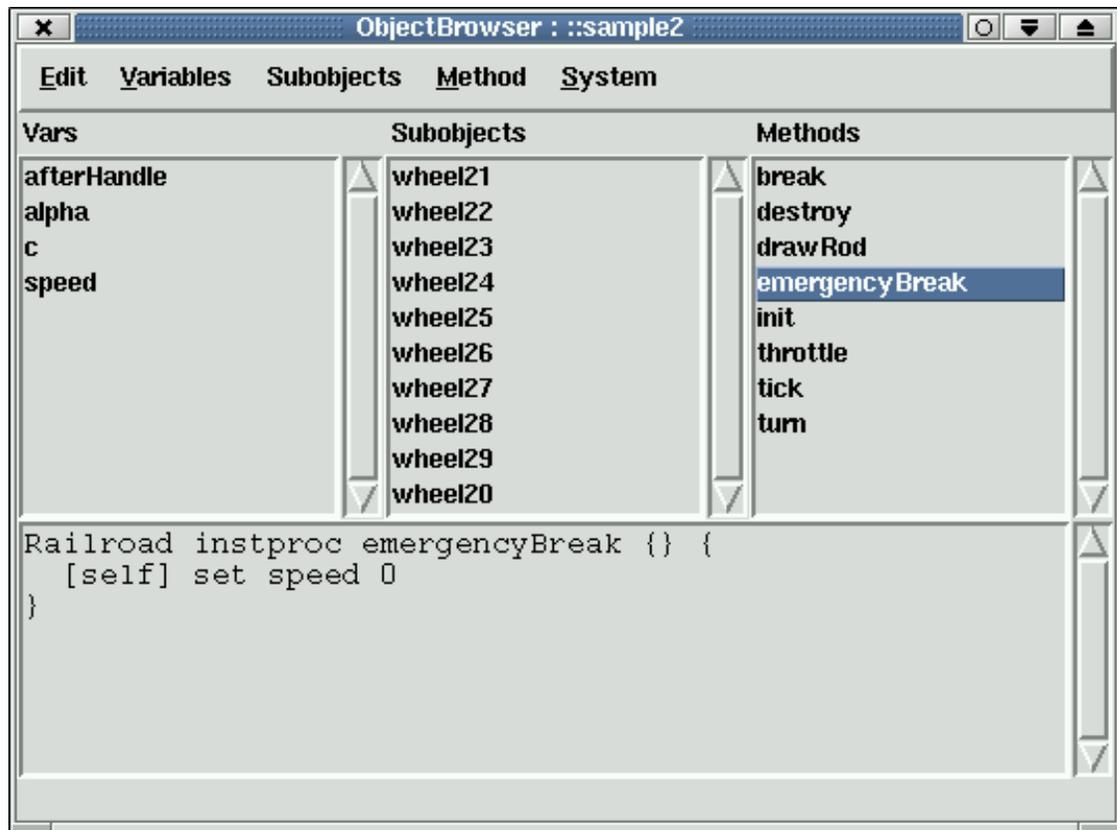
In addition XOTclIDE can show another window called an object inspector browser. This window shows the internal structure of XOTcl Objects. You can see an Object's variables (attributes), subobjects (aggregated objects) and methods.

You can modify an Object's state by changing a variable in the browser. Choose the *speed* variable. You should see the value 4 in your edit-area view. Type 20 in this area and press Control-S or choose menu Edit->Save You should see the locomotive speed up.

Figure 2.10. Object Inspector

You can also invoke methods on an object directly in the Object Inspector. Select the *emergencyBreak* method and choose menu Method->Invoke.

Figure 2.11. Methods in Object Inspector



You can also specify which methods you should see - only those from *Railroad* or also those inherited from other Objects or Superclasses. You can even see the mixin methods available on the object.

Chapter 3. Programming with XOTclIDE

System Requirements and Installation

XOTclIDE is supplied in three versions, either as a set of Tcl/XOTcl scripts (packages), or as a Starpack or Tclkit [<http://www.equi4.com/tclkit>] that needs no further installation. The Starpack version is available only for Windows. The Tclkit version runs on both Windows and Intel Linux.

To run the script version you need an installed Tcl with the Tk and XOTcl extensions. XOTclIDE can run anywhere Tcl/XOTcl runs but it has been tested only for Linux (RedHat) and Windows. Tcl(Tk) and XOTcl can be obtained free from the Internet - see Chapter 5, *Additional Information*. Almost all Linux distributions offer Tcl and Tk as base packages, but often in the older 8.3 version. For Linux and Windows, XOTcl must be installed separately. To use Version Control you need in addition a relational database manager (SQL-Database) and the proper Tcl interface to it (see the section called “Version Control System”). I suggest the following infrastructure.

- Linux as platform
- Tcl and Tk (required)
- XOTcl Extension (required)
- MySQL Database (required for version control)
- mysqltcl - MySQL Tcl interface (required for version control)

XOTclIDE Programs and Start Options

There are two main options for starting XOTclIDE: with or without Version Control. To use the Version Control System you must install a database with the program `installVC.tcl` included in XOTclIDE (see the section called “Installing Version Control System”).

Starting XOTclIDE from tclkit

```
XotclIDE.exe -- [-help] [-startMode ideOnly|ideDB|ideFromDB|installVC] [-ignoreprefs]
[-nodialog] [-preferences preferencesList]
[-configmap configmapFile] [-dumpcompid Id]
[-dumpconfid Id] [-startconfid Id]
[-configmap configmapFile] [-configmapdb configmap_name] [-preloadcomponents conpon-
ents_list] [-preexec script]
```

Starting XOTclIDE without Version Control System

```
XotclIDE.tcl -- [-help] [-configmap configmapFile] [-preloadcomponents conpon-
ents_list] [-preexec script]
```

Starting XOTclIDE with Version Control System

```
XotclIDEDB.tcl -- [-ignoreprefs] [-nodialog] [-preferences preferencesList]
[-configmap configmapFile] [-dumpcompid Id]
[-dumpconfid Id] [-startconfid Id]
```

```
[-configmapdb configmap_name] [-preloadcomponents components_list] [-preexec script]
```

Starting XOTclIDE with Version Control System by loading from Version Control

Version Control is important if you want to develop and change XOTclIDE itself. **XotclIDEEDBFromDB.tcl** loads the IDEStart, SQL interface and the rest of XOTclIDE from a Version Control database. Before calling it, you must install the version control database and import the XOTclIDE sources into version control.

```
XotclIDEEDBFromDB.tcl -- [-ignoreprefs] [-nodialog]
[-preferences preferencesList] [-configmap configmapFile]
[-dumpcompid Id] [-dumconfid Id]
```

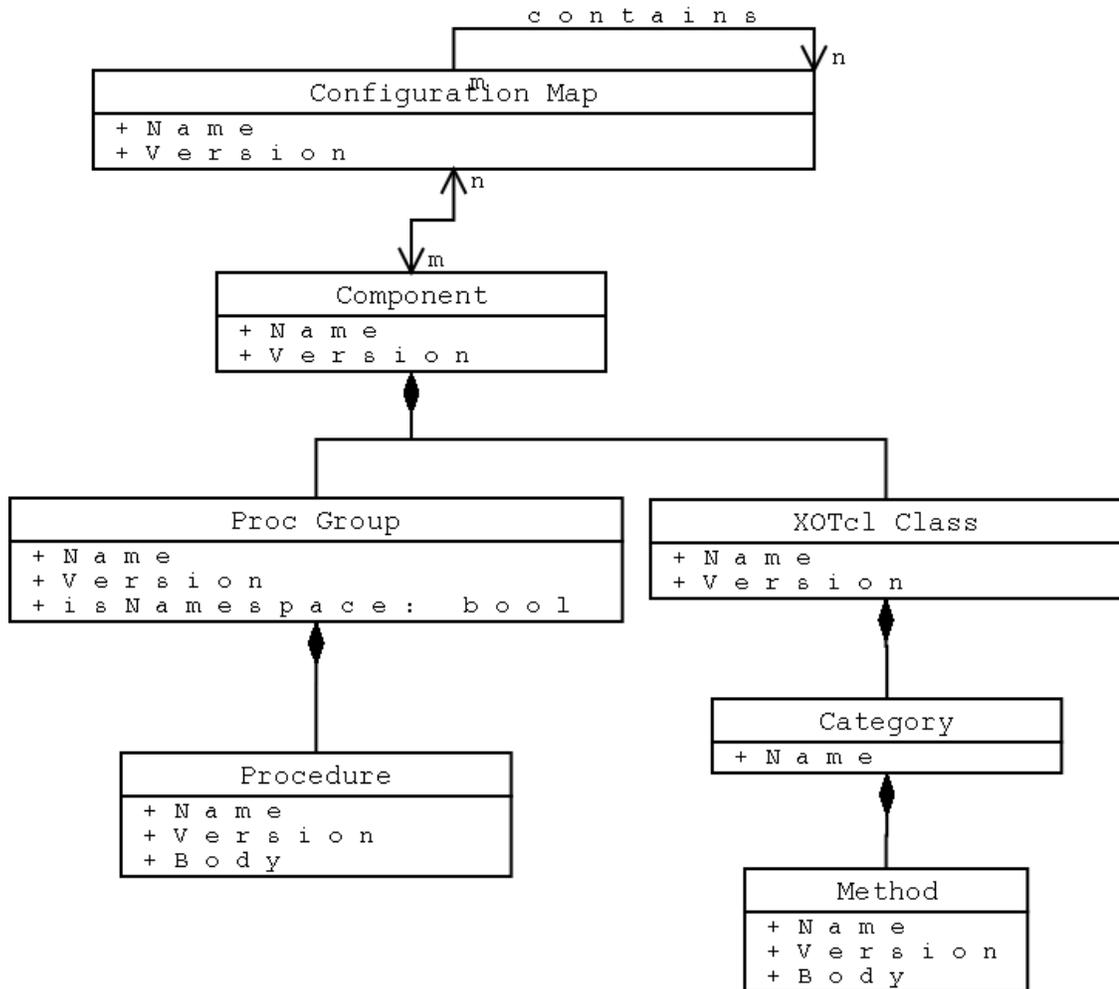
Options synopsis

-help	print all available options and exit
-preferences	You can specify the preferences for database connection. The parameter is a Tcl keyed list, for example {{interface mysqltcl} {connection {user root dbank xotcllib}}}
-configmap file	Load config map and tell XOTclIDE to ignore its own components
-configmapdb configmap_name	Load config map from version control
-preloadcomponents components_list	Load components after startup
-preexec script	Evaluate script after startup
-ignoreprefs	do not read ~/.xotclide preferences file
-nodialog	connect to version control database without showing a connection dialog. All parameters will be read from preferences file or preferences list
-dumpcompid	Read component with id from version control database, print it and exit
-dumconfid id	Read configuration map with id from version control database print it and exit

Building an Application

XOTclIDE supports so called component oriented development. Components are big, reusable and complete pieces of code. An application can be built from one or more components that can be also used by another application. The structure of a component is as shown in the UML diagram Figure 3.1, “UML Structure of Components”.

Figure 3.1. UML Structure of Components



The component is the container for XOTcl classes, objects and Tcl procedures. Configuration maps organize the components into different applications. To allow more detailed and structured organization of code XOTclIDE also provides “procedure groups” and “method categories”. Method categories and procedure groups have no influence on the semantics of a method. Method categories can be effectively used to group methods into different kinds. Method categories can be very useful to keep track of a large number of methods defined in one class. In Smalltalk these categories are often used:

- access
- initialize
- actions
- private
- persistence
- release

Procedure groups provide the ability to define the namespace of procedures. If you want to create a collection of procedures in one namespace, use the dialog for creating the Procedure Group. The name of this procedure group is also the name of a Tcl namespace. All procedures in this procedure group will be defined in this namespace. One procedure group can only correspond to one namespace (and vice versa).

XOTclIDE Components

An XOTclIDE Component is a named persistent set of Classes, Objects, and Tcl Procedure Groups administered by XOTclIDE. When you program in XOTclIDE, a Component is the thing that stores your program. It serves the same purpose as a library in compiled languages or a package in Tcl, providing a name that can be used to locate and load a set of items. A Component only has meaning when an application or program is being loaded - once the program is running, the name of the Component where a proc or Class was originally stored is no longer useful.

XOTclIDE Components are built on top of Tcl packages - each Component has a package name, calls package provide, and has an entry in a pkgIndex.tcl file. A Component can be used in any Tcl or XOTcl program by calling

```
<code>package require</code>
```

with the Component's name. Component are a slightly restricted form of package that contains only proc definitions, Class definitions, Object definitions, and special metadata generated by XOTclIDE. No other Tcl commands are stored in a Component.

Each Component is stored either in one text file named after the Component, or in a Version Control database.

Component lifecycle

It may be useful to consider how XOTclIDE loads and stores Components. When a Component is loaded, all its proc and instproc names, arguments, and bodies are loaded into the running Tcl/XOTcl interpreter. Classes, Objects and their relationships are also loaded. Imagine that you need to edit the body of a proc found in a Component. When you call up an edit area containing the body, XOTclIDE copies the current body from the Tcl interpreter into an editable window. When you save the edit (Control-s), XOTclIDE copies the body back into the Tcl interpreter (by doing a proc command.) At this point, the new body is available in the Tcl interpreter, and you can test whether it works to your satisfaction. Once you decide that it's good enough, you save the Component that contains the proc. XOTclIDE gets the current proc names, args, bodies, Classes and Objects in the Component from the Tcl interpreter, constructs XOTcl commands that will regenerate the current state of each thing, and stores the commands either in a file or a Version Control database.

One thing to note about this process is that anything other than Classes, Objects, procs, and metadata that was in the Component file will disappear, since the Component stored form is completely rebuilt when XOTclIDE stores the Component. There's no guarantee that a Component file will preserve previous order. The text of a Component file may be edited, and any changes to proc bodies will be available the next time the Component is loaded. The next time XOTclIDE saves the Component, the text file may not look much like it previously did. For example, any comments added to the file that stores a Component will disappear the next time XOTclIDE saves the Component.

Another thing to note is that changes to the heritage of Classes or Objects (is-a relationships) and changes to the bodies of procs that occur while your Component is running under XOTclIDE will be stored persistently. Suppose your Component has a Class apple - if running your component under XOTclIDE adds a new superclass Class familyRose to Class apple, saving the Component will preserve the new superclass relationship. The next time your Component is loaded, apples will be in the rose family.

Nested Classes and Object aggregations (has-a relations) are stored in Components only if they have been defined in a Classes/Objects view. To define a nested class that will be persistently stored, define the class from the menu and give it a name that has the name of the class it's nested in followed by two colons (::) and the name of the new class. Similarly, Objects can have persistent subobjects defined by the programmer. Other subobjects that an Object acquires while running will not be persistent. For example, suppose that your component defines a Class vehicle, and an Object orientExpress of Class vehicle. The Object orientExpress might have many permanent subobjects of Class Wheel. These would appear in the Classes/Objects view as orientExpress::leftFrontWheel, orientExpress::rightFrontWheel, etc. While your program is running, the orientExpress Object might get some passengers and save them

as subobjects of Class person. These passengers would not be visible in the Component Browser, although they would be visible in an Object Browser. When XOTclIDE saves your Component, the orientExpress will be saved with Wheel subobjects, but without any passengers.

Browsers and Inspectors

XOTclIDE was designed as a set of browsers that let you investigate your system in the way that you need for your task. Other IDEs offer one all purpose single window to access all functions which tend to be overloaded and complex. In software development you have many different tasks that need special views into your system. You need different views and functions for programming, debugging, version management, deployment, quality management, code review. These views should be similar enough to reuse user knowledge and offer consistent handling. Of course some browsers (for example the **Component Browser**) will be used more often and should have more functionality. The right balance must be found between the two extremes - either one view with overloaded function or many views with only a few functions. This section presents all XOTclIDE browsers sorted by main programming task.

In XOTclIDE we distinguish between browsers and inspectors. A Browser offers a view on code definition (classes, methods and other). Inspectors allow the user to navigate through Tcl Interpreter data space. They can show the state of XOTcl Objects and global variables.

All browsers in XOTclIDE are built on the same principles. All browsers are divided (composed) into areas (panes). There can be many list areas (list-views) and one or more edit areas. Each view area has a corresponding pop-down menu in the main menu. Pop-up menus with the most used menu items let you reach needed functions in the place you need them. Many browser subcomponents (sub views) are reused in different browser windows with the same functionality.

Almost all component windows have a special System menu that offers access for launching other browsers and reaching main IDE functions, like settings.

In XOTclIDE you can open one browser type several times. They work independently. Normally you will use more than one component browser to see different code at the same time. You can write a method call in one browser and open another browser to see the called method body.

Warning

XOTclIDE Browsers use a passive “model view controller”. If two browsers show the same code and in one of them the code is changed the changes are not updated automatically in the other view (browser). You can force the view to be updated by double-clicking on the list-view item.

Writing Source

Component Browser	Probably the most used browser. It is suitable for viewing and changing components, classes (or object, Tcl-Proc-Groups), categories, procs or methods. With this browser you have access to all the source code in your system (definition space). The browser also has functions to search after a specific class in your system.
Heritage Browser	This browser is suitable for examining and developing class hierarchies. The XOTcl language provides multiple inheritance - one class can have many parent classes. To visualize this relationship the parent classes are represented as leaves of a tree in this view. To see the inheritance of a class select it in the <i>Component Browser</i> and use menu Class->Heritage Browser. You can also launch this browser with the System->Heritage

	Browser In this case you must specify (by Name or choice-list) the class name you want to view.
Children Browser	This browser is suitable for examining and developing class hierarchies. Unlike the <i>Heritage Browser</i> in this view the children of a class are shown. To see the descendants of a class select the <i>Component Browser</i> and use menu Class->Children Browser.
Method List Browser	This Browser is launched as a result of searching on method bodies. Menu Method->Search

Refactoring

Refactoring is a normal part of developing. We learn every day we develop. Refactoring allows transferring new knowledge to old code, and adapting old code for new purposes.

The main condition for successful refactoring is quick understanding of old code that was probably written by another people. Code reading skills are important in this point. Since methods in object oriented system are relatively short, the main question is to know and find the context of their usage. To view the definition of a method that you see in a method body select the method's name in the editor and use menu Method->Search Implementors. If you want to know where a method is called from select it in **Methods** view and use menu Method->Search Senders. Also the function "Browse it" in Menu Method->Search Implementors (**Control-r**) is very useful in finding symbol definitions. Just select a symbol (word) in any editor area and invoke it from the pop-up menu. The function first searches for a class or object name equal to the selected text, then for a method or procedure name.

The following refactoring functions are possible in XOTclIDE

Rename Component	Menu Component->Rename
Moving Classes to Another Component Coping Classes/Objects	Menu Class->Move to Component Menu Class->Copy Class/Object This function can be also used to rename a class. Just copy the class to a new name and delete the old class.
Extract method body parts	Menu Method->Selection to new method This task is normally done if the method becomes too long and you want to divide it into smaller methods. This function cannot resolve the correct variable definitions and functions parameters.
Rename method name	There is no direct function for it. Select the method, then in the editor type a new name in method body, and then accept the change in the editor. The new method will be added. The old one must be removed with the delete function.

Debugging running Systems

Debugger	This is a pure Tcl debugger. Call ":::xotcl::Object halt" in the method body.
Object Inspector	In the Object Inspector you can view and change variables of an XOTcl object. It also has method lists that can be used to invoke methods on objects. To launch the object inspector Class->Inspect All Instances You can also call

method *inspect* on every XOTcl object.

Version Control

If you start XOTclIDE with version control the browsers offer additional functions for managing version control. The component, class and method areas (views) have an additional sub menu *Version Control*

Configuration Management and Deploying

Configuration Map Browser	Launch from the menu System->Configuration Map Browser
Configuration Browser	Available only if you started XOTclIDE with Version Control System->Version Control->Configuration Browser

Special Browsers

Transcript	This is a main singleton edit area that is either used to display system messages (per command: Transcript message "Hallo World") or to evaluate short Tcl scripts.
Workspace	It is a simple edit area that can be used to evaluate Tcl scripts. You can load text files into a workspace or save the contents into a text file.
Global Variable Inspector	Access from the menu System->Global Vars Inspector It can be used to inspect all global variables, or namespace variables.

Source Editing

Basic Editor Function

The XOTclIDE editor is based on a Tk Text widget. The following functions are available.

Cut Text	Key Accelerator Control-x
Copy Text	Key Accelerator Control-c
Paste Text	Key Accelerator Control-v
Undo	Key Accelerator Control-z available only with Tcl8.4

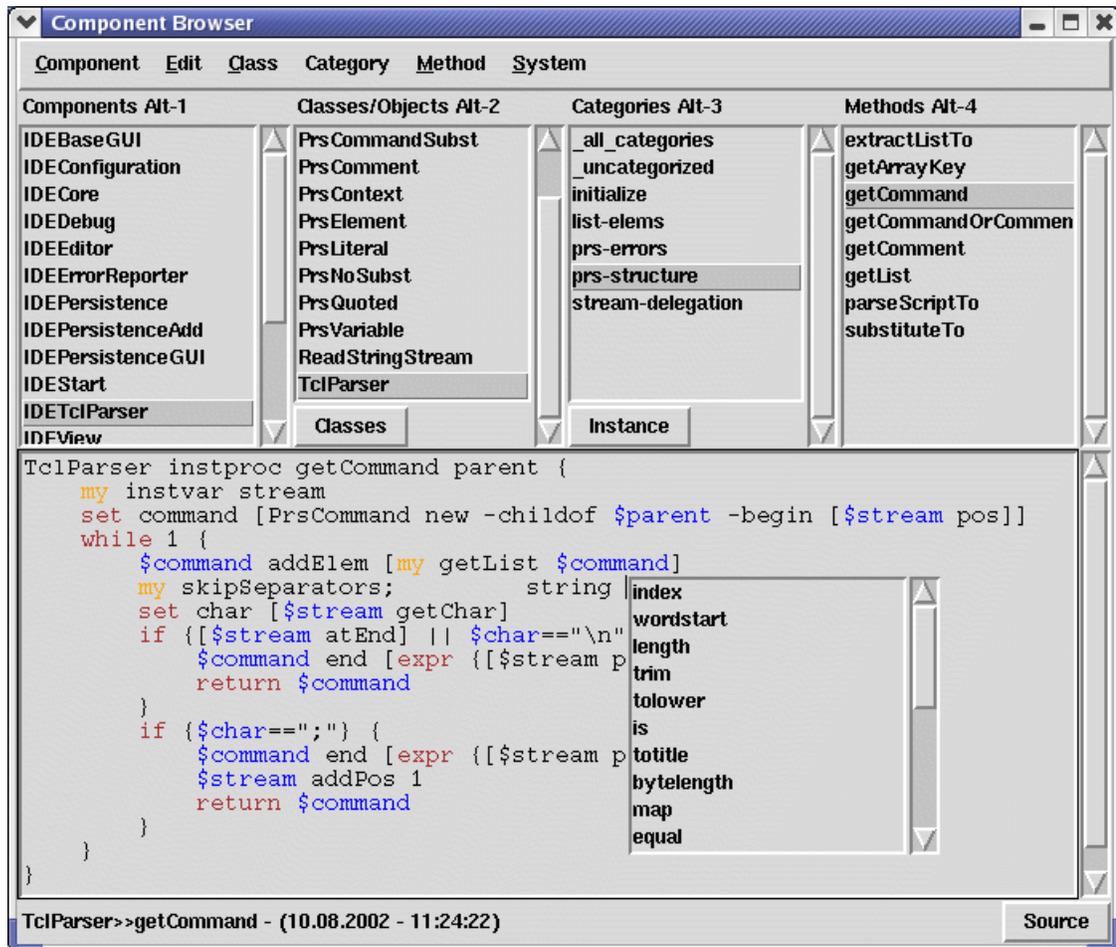
To see other accelerators see the Tk Text Widget documentation. text widget manual [<http://purl.org/tcl/home/man/tcl8.4/TkCmd/text.htm>].

Standard text find and replace dialogs are available in menu Edit. To search for text in many methods use the menu in Browser **Component Browser** Method->Search Text (**F4**).

Code Completion

It works like file name completion in bash (Unix Shell) or code completion in typed program language IDEs. The user can type the first character of token than press Tab or Control-Space. XOTclIDE tries to complete the token depending on its context. If many alternative completions are available a pop-down window list will be displayed.

Figure 3.2. Code Completion



The following language tokens can be completed

Tcl Core-Commands	commands like: lappend list foreach
Tcl Core-Commands parameters	for example string length match range
Defined Tcl Procedures	all procedures known from info procs
Defined XOTcl Classes and Object	all XOTcl Classes and Object in every namespace.
local class methods	all methods (including methods derived from other classes) found by \$Instance info instprocs

visible variables

variables defined per **set append instvar foreach ...** or method parameters. Just type \$ and press the **Tab** key

Navigation in Sources

The Method menu in the Component Browser offers some additional help functions for browsing XOTcl/Tcl methods. You can return to the last shown method by Functions “Back Method (**Alt-Left**)” or “Forward Method (**Alt-Right**)” just like from your HTML browser.

Spawn View opens a new editor window with the same contents as the current edit area.

Syntax highlighting

XOTclIDE implements two kinds of syntax highlighting:

Simple syntax highlighting

is based on regular expression patterns. It highlights only "", substitution and # comments.

Parser-based syntax highlighting

This syntax highlighting gives a truer result. It highlights Tcl core-commands, variables, comments, "", substitution and XOTcl key-words.

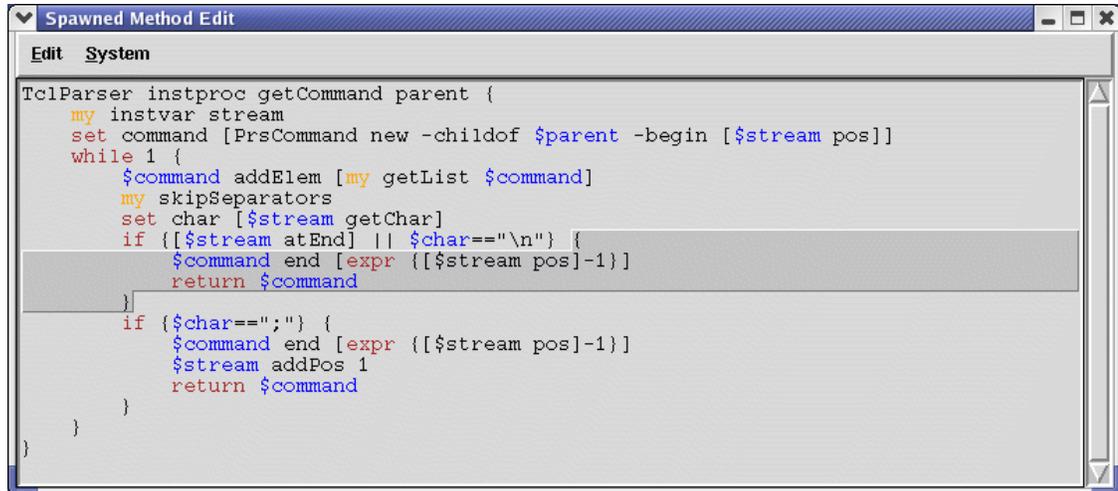
Figure 3.3. Syntax Highlighting

```
TclParser instproc searchCloseBrace {obrace cbrace} {
  my instvar stream
  set open 0
  # search closing parenthesis until and of stream
  while {![ $stream atEnd]} {
    set c [my getCharWithMasking]
    if {$c==$obrace} {
      incr open
    } elseif {$c==$cbrace} {
      incr open -1
      if {$open==0} {
        return 1
      }
    }
    $stream addPos 1
  }
  my parseError "Parse Error: can not find closing $cbrace"
}
```

Checking parentheses

Finding opening or closing parentheses can be hard work in understanding some Tcl methods. Double-click on [{" or }]" and XOTclIDE will find the corresponding opening or closing parenthesis for you and select it.

Figure 3.4. Checking Parenthesis



Automatic Indenting

XOTclIDE tries to set the same indent in a newly inserted line as in the last line. If the last character before a new line is an opening parenthesis { then in the new line the indenting will be increased by four and a matching closing parenthesis will be inserted.

```
foreach a $list {here cursor
```

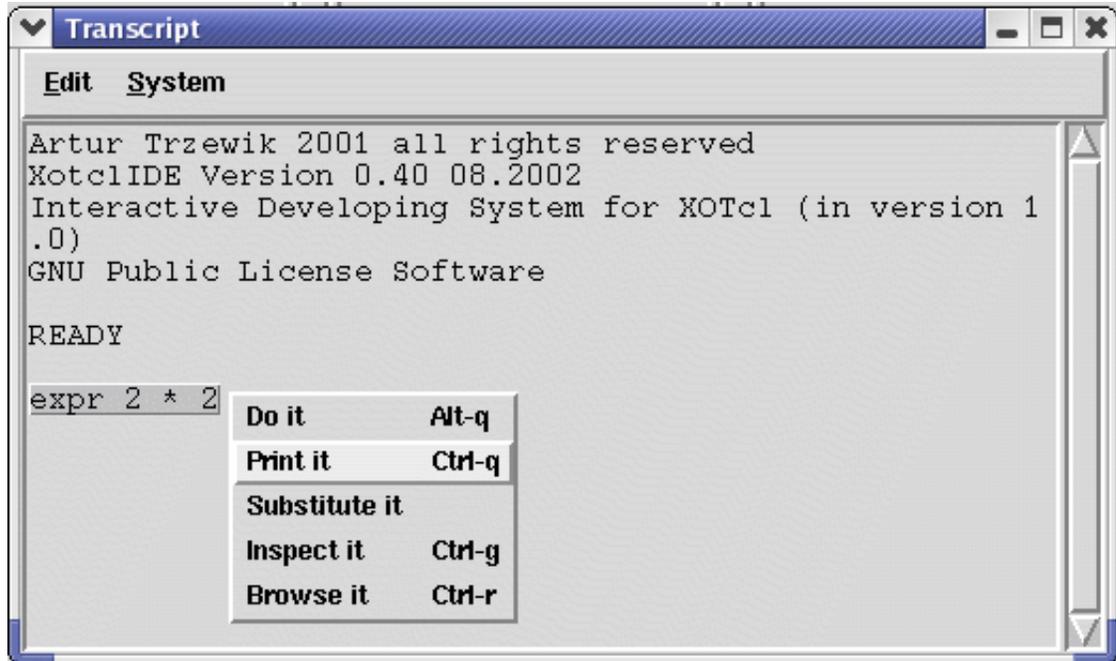
After a new line you will see this code

```
foreach a $list {
here cursor
}
```

Evaluate Tcl Scripts in Editor

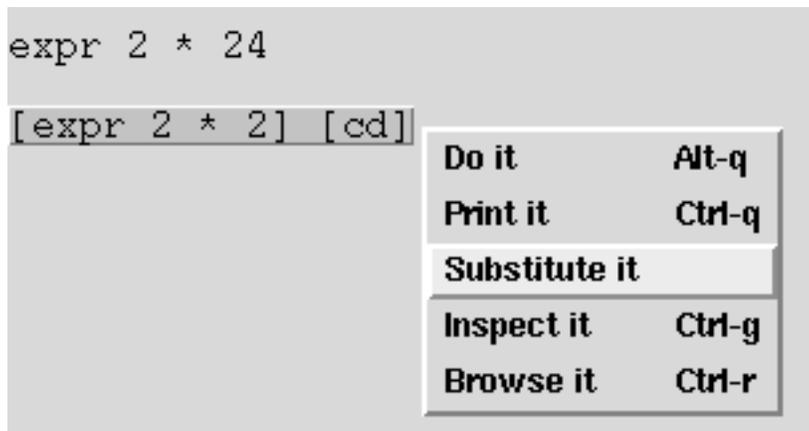
This is a very powerful function. It lets you evaluate Tcl scripts as in the Tclshell console but is more flexible. Simply select a string and evaluate it with the pop-down menu or key-accelerator. Now you will notice that in Tcl everything is a string. It is also the easiest and fastest way to program in XOTcl/Tcl.

Figure 3.5. Evaluate Scripts



The Menu "Print it" corresponds to the Tcl `eval` command. You can also use Control-q.

Figure 3.6. Substitute Scripts



Menu "Substitute it" corresponds to the `subst` Tcl command.

Figure 3.7. Inspect Script Evaluation

```
[expr 2 * 2] [cd]4  
Class MyClass  
set a [MyClass new]  
$a set myVar "my Value"  
set a
```

Do it	Alt-q
Print it	Ctrl-q
Substitute it	
Inspect it	Ctrl-g
Browse it	Ctrl-r

Menu “Inspect it” evaluates the selected text and launches an inspector on the returned value if it is a XOTcl object. You can also use Control-g. If the result of the last operation is an XOTcl Object then it will be displayed in an Object inspector.

You can use Workspaces (see menu System->Workspace) to evaluate short Tcl scripts.

Chapter 4. Extended Features

Version Control System

As soon as you develop more than “Hallo World” programs you need a version control system. You need it to save and archive your code and follow the code changes. Version Control gives you a guarantee of returning to a former code state so you can experiment with your code. Version control is an implicit requirement for many extreme programming practices. In XOTclIDE all code changes are updated in version control immediately so you need not worry about saving your source. The Version Control System is also a code repository that can be centralized for many developers. Components can be loaded and executed directly from the version control repository without the need to save them as files in a file system.

Benefits of Version Control

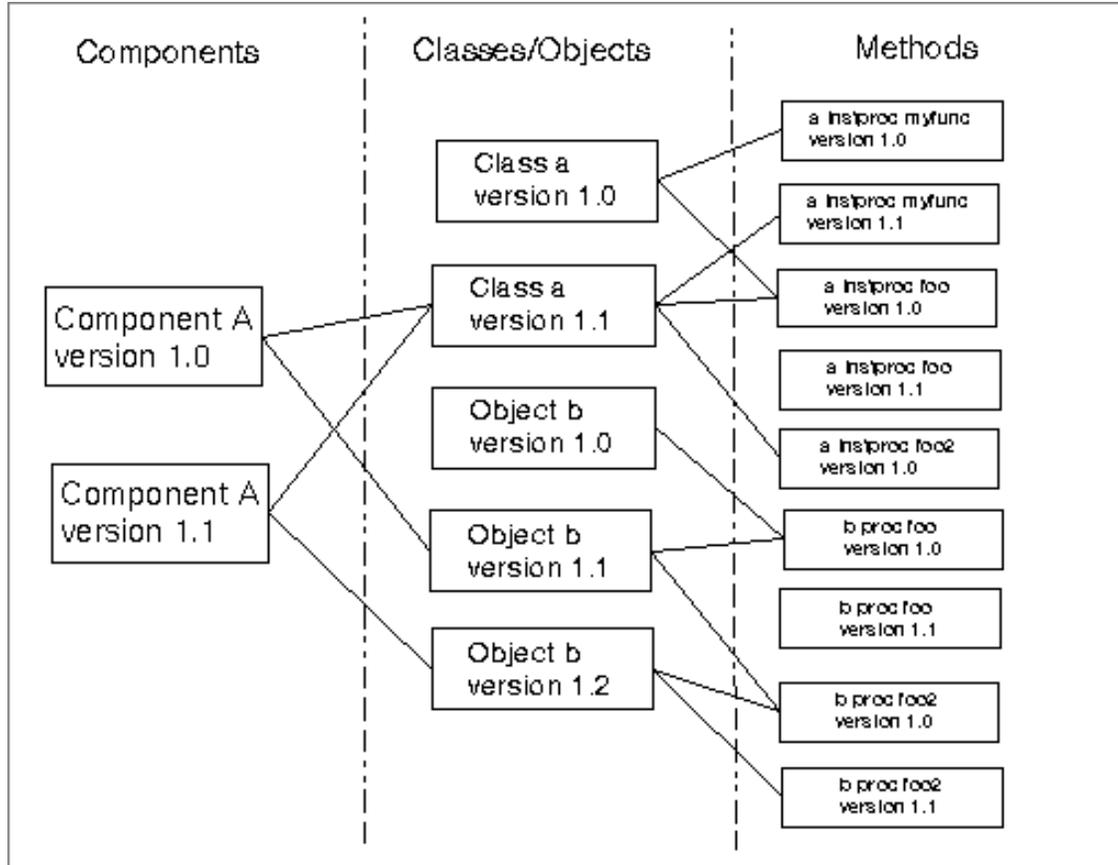
- database oriented - all information is stored in database
- central code repository - no file copying
- suitable for big projects - able to browse in thousands of lines of code
- multiuser capable
- flexible as XOTcl - no locks, no commit, developed specially for object oriented languages like XOTcl.

Base Characteristics

- based on relational database (currently mysql, postgres, sqlite or odbc)
- XOTcl programs are stored and managed corresponding to their structure. (Components, Classes/Objects, Methods).
- Every change in your system is stored on the fly. You need not ask the system to update the structures.
- If you add or modify any method, a new row is inserted into the database. The old method version is normally not deleted. You can always return to every state in the past (no comment and uncomment of code pieces).
- Stores not only sources but also additional data as documentation, comments, meta data.

Principle

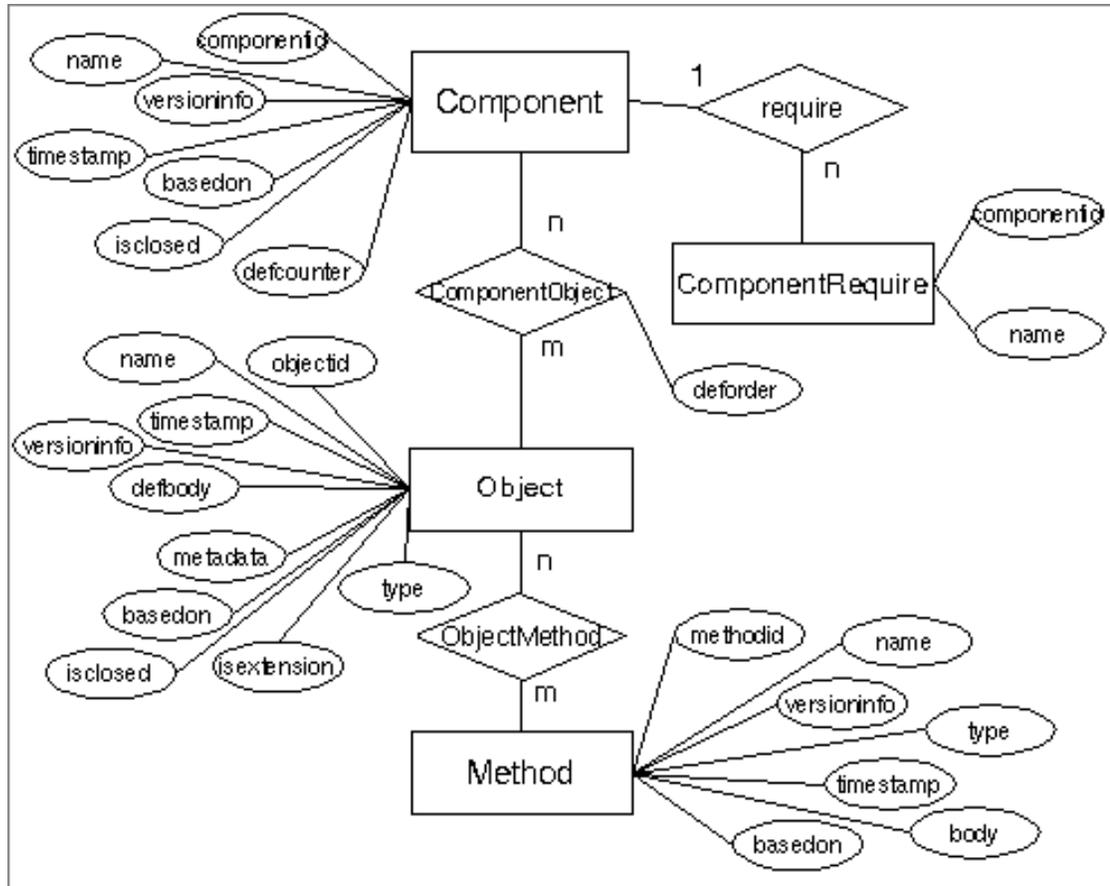
Figure 4.1. Version System Principle



XOTcLIDE Programs are structured in Components (a concept specific to XOTcLIDE), Classes/Objects and Methods (see the section called “Building an Application”). All these structure elements are stored and managed separately in the database. A class can have many versions without having to store a complete class definition in a separate text file (as is usual in cvs if a class is one file) with many redundant copies even though only one method differs. Normally the programmer doesn’t care about version control. He just develops. The version control uses so called optimistic locking. If the programmer thinks the state of a component or class is stable and ready to be marked he can “version” this component or class. After this he can return to this state or recognize changes among different versions.

Database Schema of Version System

Figure 4.2. Version System ER Diagram



There are m:n relationship between Components, Objects/Classes and Methods, meaning one method version can belong to m object-versions. This diagram does not include configuration maps. If you know this schema you can formulate queries on the repository directly in SQL. It is quite easy to find the newest methods or statistics about code development with simple SQL queries.

The object attribute “deforder” specifies the order of loading Objects/Classes into an interpreter.

The ComponentRequire entity maps the requirements for a single component version. The information that a special component version needs a special version of another component is not stored.

Version info attributes are intended to store short text information for developer about the version and are not used to identify entities.

Definitions (Editions, Versions)

An Edition is a piece you can work on. There are Component, Object and Method editions. Editions can be changed. Methods can only have editions.

Versions are frozen editions which means they cannot be modified. A single class version always has the same methods. To freeze an edition you have to version it. To modify a versioned class you must open a new edition of the class. This new class edition is based on the last version of the class.

Using Version System, Main Functions

The following functions are available on the menu for most structure items (Components, Objects, Methods).

- Available - Show all items in database. All Components, all objects, and all methods that belong to the selected object. You can choose a edition and load it into the system so you can find and reload the methods you have deleted.
- Editions - show all editions of currently selected item (Component, Object, Method). You can load your chosen version into the system.
- Changes - start the Changes Browser to show changes compared to another edition in the database.
- Load Previous - you can return to a previous edition. For methods you can simply discard the latest changes.
- Version - Freeze current selected edition. (cvs tag or labels) You can version only components and classes/objects. (It makes no sense for methods). You can always return to this state of the class or component.
- New Edition - to develop a versioned edition you must make a new edition. The system makes a working copy of a version.
- Version Info - Show a small info dialog. You can specify the version string which can be 30 characters long. The first number will be automatically incremented by opening a new edition. This number is used as version number for Tcl packages.

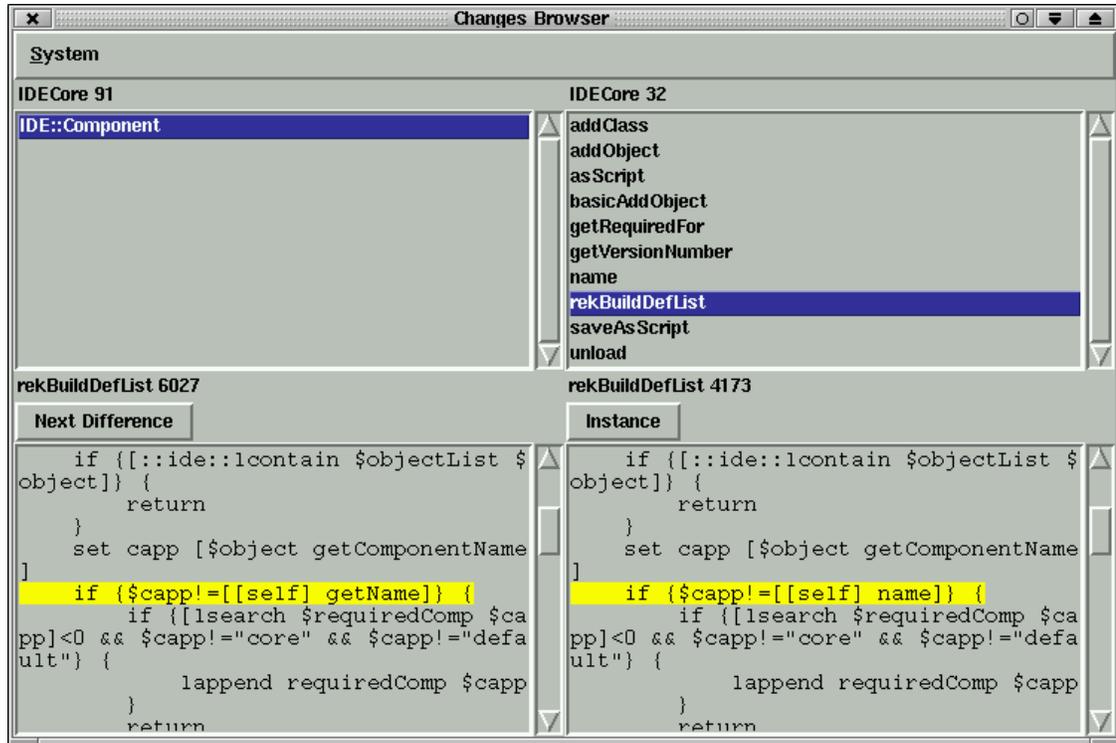
The functions below are available only for components:

- Import - Install a package into repository. Probably loaded per load package by package require into database.
- Requirements - Show and edit the required components for a selected component. The requirements are set automatically. Normally you do not need to change them.

Changes Browser

You can see the differences between this version and another version. It is **diff** command for XOTclIDE. In figure Figure 4.3, “Changes Browser” we can see the changes between class versions of class IDE-Core. In the method list are shown all instance methods with differences. Since the method `rek-BuildDefList` is selected in the text area we can see (yellow selection) the exact differences.

Figure 4.3. Changes Browser



Component Loader

You can execute programs directly by loading them from the Version Control System. It is not necessary to deploy your application as a set of files. This makes the management of many client systems very easy but the loading time can increase. **CompLoader.tcl** is a small independent script that can connect to repository, load the components specified by a configuration map and execute them.

Usage: CompLoader.tcl [-nodialog] {-ignoreprefs} {-nosynchronize} [-help] [-preferences list]

-ignoreprefs	do not read preferences
-nosynchronize	no IDE self developing mode
-nodialog	no user interaction while connecting to db
-preferences string	overwrite preferences (use keyed list)
-help	show all options

Installing Version Control System

XOTclIDE supports the following databases as repository

- mysql
- postgres
- odbc

- `sqlite`

The Tclkit distribution (Starpack) comes with `sqlite`. I recommend using `mysql` because it is my primary developing system and the best tested.

To get version control on a Linux system you must:

- Install `mysql` database. I recommend using RPM packages for your distribution.
- Install `mysqltcl` extension to access `mysql` from `tcl`. To compile it yourself you need header files for `mysql` database (for redhat the package `mysql-devel`). There are also RPMs for Linux available on the `mysqltcl` site.
- Run the installer tool `installVC.tcl`. It can check the database connection, specify the connect parameter, install tables and copy components to repository. Then you can start `XotclIDE` with `XotclIDEDB` or `XotclIDEFromDB`. The first connects to the database after loading itself using package require. `XotclIDEFromDB` tries to load the entire application from DB.

Syntax Checking

Reason for syntax checking in Tcl/XOTcl

Tcl has no types and this is good. One disadvantage is that ugly syntax errors (typically typos) crash your program the first time you run it. Therefore you must care about running every piece of your code through interpreter by writing special test procedures. Syntax checking with `XOTclIDE` can find most of these errors (syntax errors) at editing time by simulating the interpreter. The syntax checker can parse and interpret code without running it.

Syntax checker implementation

`XOTclIDE` implements a static syntax checker. It parses `Tcl/XOTcl` procedures and finds errors that normally appear only at run time. Because `XOTclIDE` does not manage source code but manages a `Tcl/XOTcl` interpreter it always checks a method in the actual interpreter context. You cannot check your `Tcl` files without loading (source `$your_file`) the procedures into the interpreter. The following syntax checks are processed.

- check all called procedures
- check the count of arguments
- simulate quote and command substitution
- simulate evaluation of control structure commands (`if`, `for`, `foreach`, ...)
- check variable visibility
- check `XOTcl` self method calls. (`my callMe`; `[self] callMe`)

Syntax Checking can be used in two ways

- Syntax check while editing or accepting code.
- Run the syntax checker on projects, so you can check existing `Tcl/XOTcl` sources.

Example Tcl procedures

```
procs example {a} {
    set b [lindex $a 0]
    puts "$a $c"
    set e [lindex $a end dummy]
    foreach d $a {
        if {$d==a} {
            putd $d
        }
    }
}
```

The syntax checker will find the following errors:

```
procs example {a} {
    set b [lindex $a 0]
    puts "$a $c"
    # unknown variable c
    set e [lindex $a end dummy]
    # await (2,2) arguments
    foreach d $a {
        if {$d==a} {
            putd $d
            # unknown proc
        }
    }
}
```

Example XOTcl methods

```
Class Test -parameter {par1}
Test instproc fool {a {b 1}} {
    puts "[self] $a $b"
}
# Method to check
Test instproc foo2 {b} {
    my fool test
    my fool test 1 2
    # await (1,2) arguments
    foreach elem $b {
        puts "[my par1] $elem"
        my par1 $elem
        my foo3
        # unknown instproc
    }
    set c $d
    # unknown variable d
}
```

Syntax Checking while editing

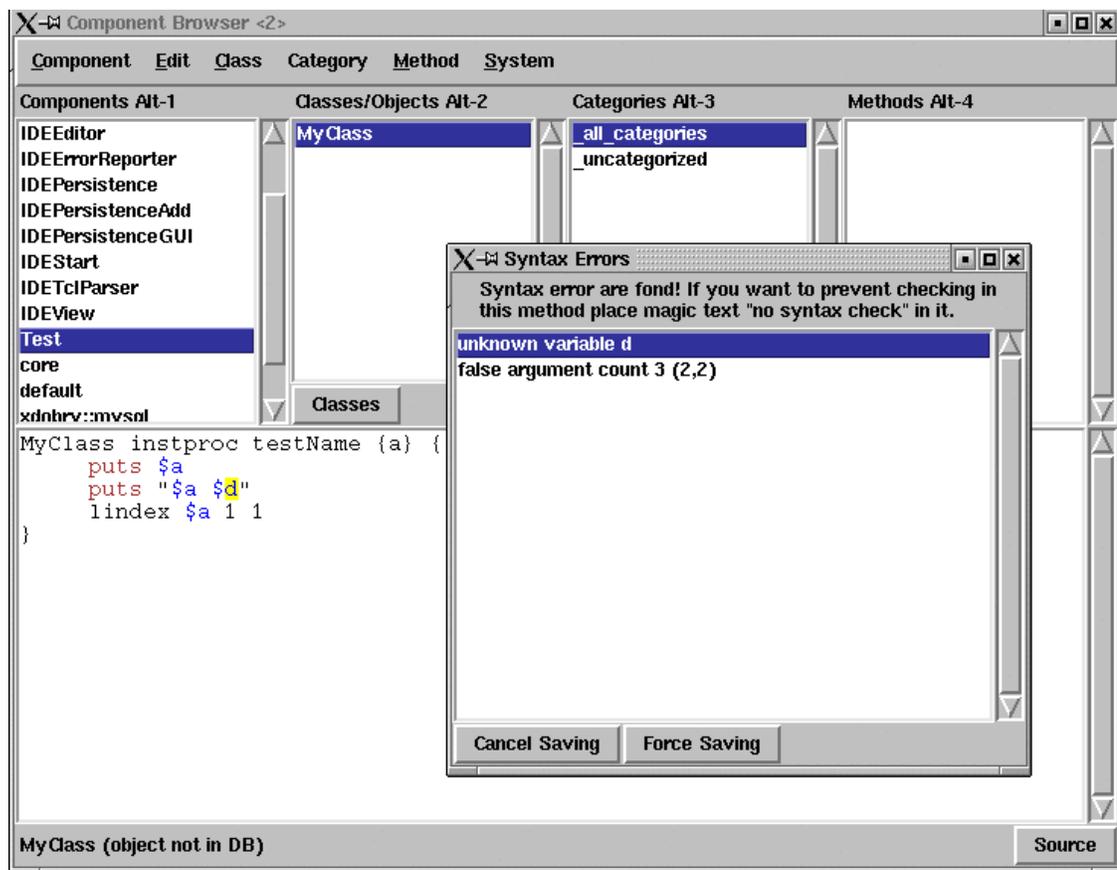
To enable syntax checking while editing, click on the check-box in menu Edit->Syntaxcheck on Save. All accepted (saved) code will be syntax-checked. If errors are found a new window with syntax mes-

sages are displayed. You can see the corresponding code in the editor by clicking on the list items. (see Figure 4.4, “Syntax Checker Dialog”)

If the errors shown by the syntax checker are not really errors or you have already corrected the errors you can press the button Force Saving. The code shown in the editor will be accepted without syntax checking.

If you want to force syntax checking without accepting choose menu Edit->Syntax Highlight->Force Syntax Check.

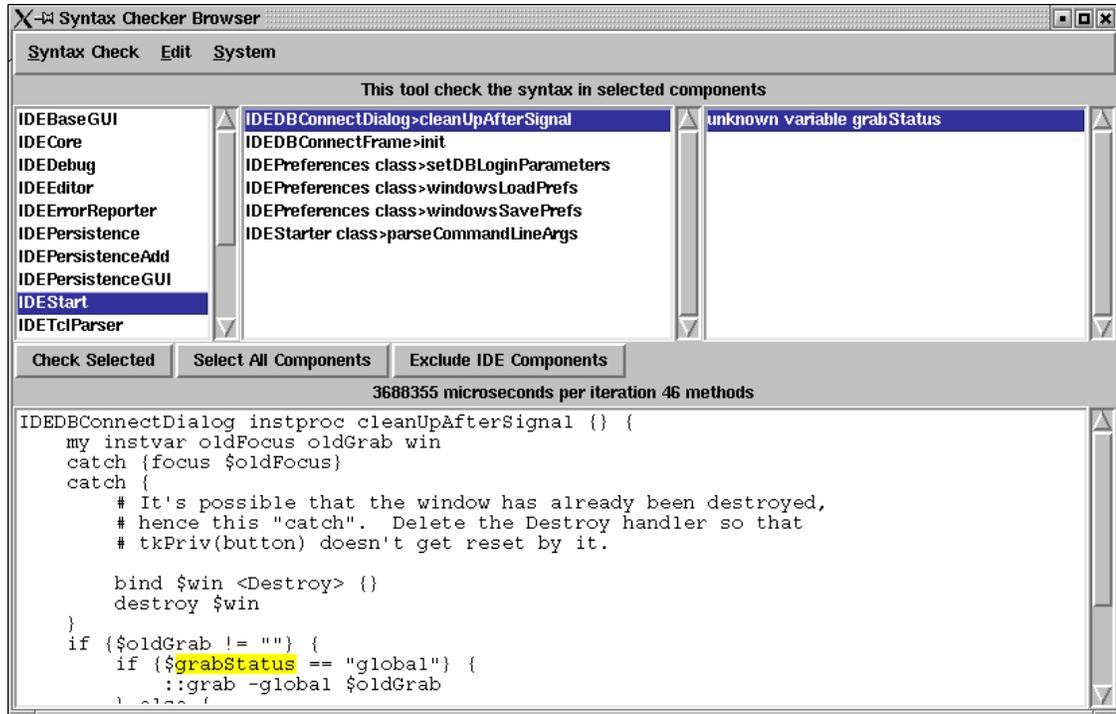
Figure 4.4. Syntax Checker Dialog



Syntax Checker Browser

You can launch this tool from the menu System->Syntax Checker. Choose the components you will check and run the check with button *Check Selected*. You can browse the errors by clicking the other two lists.

Figure 4.5. Syntax Checker Tool



You can produce a protocol of checking as text file with menu Syntax Check->Protocol to file.

Tcl/XOTcl Parser

The XOTclIDE syntax checker works by using its Tcl parser programmed in XOTcl (see component IDETclParser). It produces a parser tree that can also be used for other purposes. At this time the syntax highlighting is also based on this parser.

Other ways of using Tcl parser in Tcl.

- Normalize source code (pretty print)
- convert source codes from or to another object oriented Tcl. (ITcl)
- Refactoring tools in the manner of Smalltalk.

How to extend syntax interpretation

See the *PrsContext>checkTclCommand* method. The syntax of all Tcl control command are coded as simple pieces of code.

```
# while proc
[$command getElem 1] substituteContents
[$command getElem 2] evalContents
# set proc
if {$count==2} {
    my addVariableFrom [$command getElem 1]
} else {
    my checkVariableFrom [$command getElem 1] $notifier
}
```

It should not be difficult to extend the semantics for more commands.

Problems

The syntax checker cannot simulate the full power of a Tcl interpreter. For example, it interprets double substitution as:

```
set a putd
$a hallo
set a c
set $a 2
puts $c
```

"\$a hallo" will be not reported as an error but "puts \$c" will report the error "unknown variable c".

Magic strings for checker

If you want to avoid syntax checking for one method place the string *"no syntax check"* in the method (probably as a comment).

If you want to force the checker to accept a variable use *"add variables (varName varName2)"*

```
# add variables (c)
set a c
set $a 3
puts $c
```

Checking Referenced Object Calls

It is also not possible to check referenced object calls:

```
set a [MyClass new]
$a doJob
# also direct call by object name
MyClass myObject
myObject doJob
```

The first method call will be not checked. The checker has no information about what is \$a. The second method call will be reported with the error "no such proc" (myObject). This second type of call should be very rare in XOTcl programs (besides global singleton objects).

To solve the problem the checker would need more type information. Type information could be coded as meta information in the class. For example:

```
Class A
A addMetaVariable drawContext DrawContext
A instproc draw {} {
my instvar drawContext
$drawContext drawLine 0 2 0 50
}
```

In this case the Syntax Checker would know that "drawContext" references an object of class "DrawContext". The same thing could be done for method arguments or even all variables by using special in-line

directives

```
set a [MyClass new]
# variableType a MyClass
$a doJob
```

This could be a back door to make Tcl type-safe if you want. In fact, the meta type information could be collected by doing analysis of a running system (for example by using XOTcl filters). This type information could also be used to build XOTcl assertions.

There is a chance of making a very powerful Tcl development system even with type safe syntax checking.

Configurations Management

Configuration Management relates to two other areas in the software development process. They are Deployment and Release Management. In XOTclIDE Configuration Management is based on “Configurations Maps” that are used to specify a particular application (that is commonly deployed as one unit) as a collection of components. Versioned configuration maps allow performing release management. With XOTclIDE it is very easy to detect all changes (and their causers) between two application versions. You need no additional bureaucracy in your project.

Main Features

- organize the groups of components which should be loaded and used together
- build application (distribution) as a set of XOTclIDE components and a start script
- build the base file for **CompLoader.tcl** to load your application directly from a database (thin-clients).
- specify the exact version numbers and order of components which should be loaded.

Two tools for configuration management are available in XOTcl. The *Configuration Map Browser* can be used without the version control system and the extended *Configuration Browser* (Menu System->Version Control->Configuration Browser) can only be used with the version control system. The Configuration Browser saves all information in the version control system. You can import and export data between the two systems.

Configuration Map - Without Version Control System

Warning

It is strongly recommended to use the Configuration Browser if you work with the version control system.

A *Configuration Map* is a file (Tcl-script) that exactly specifies the components to be loaded and the start scripts. One script (preStartScript) will be invoked before loading components and one (startScript) will start your application after components are loaded. (see file Sample.cfmap)

The component list has the structure

```
{IDECore 10}
```

```
{IDEBase 12}
{IDEGUI newest}
{IDEView package}
IDEViewDB
```

The loader will search the components *IDECore* versionId=10 and *IDEBase* versionId=12 in the database. The newest version of the component *IDEGUI* will be loaded. The Component *IDEView* will be loaded with *package require IDEView*. For component *IDEViewDB* the loader will search first in the database and if not found it will try to load the package with *package require IDEViewDB*.

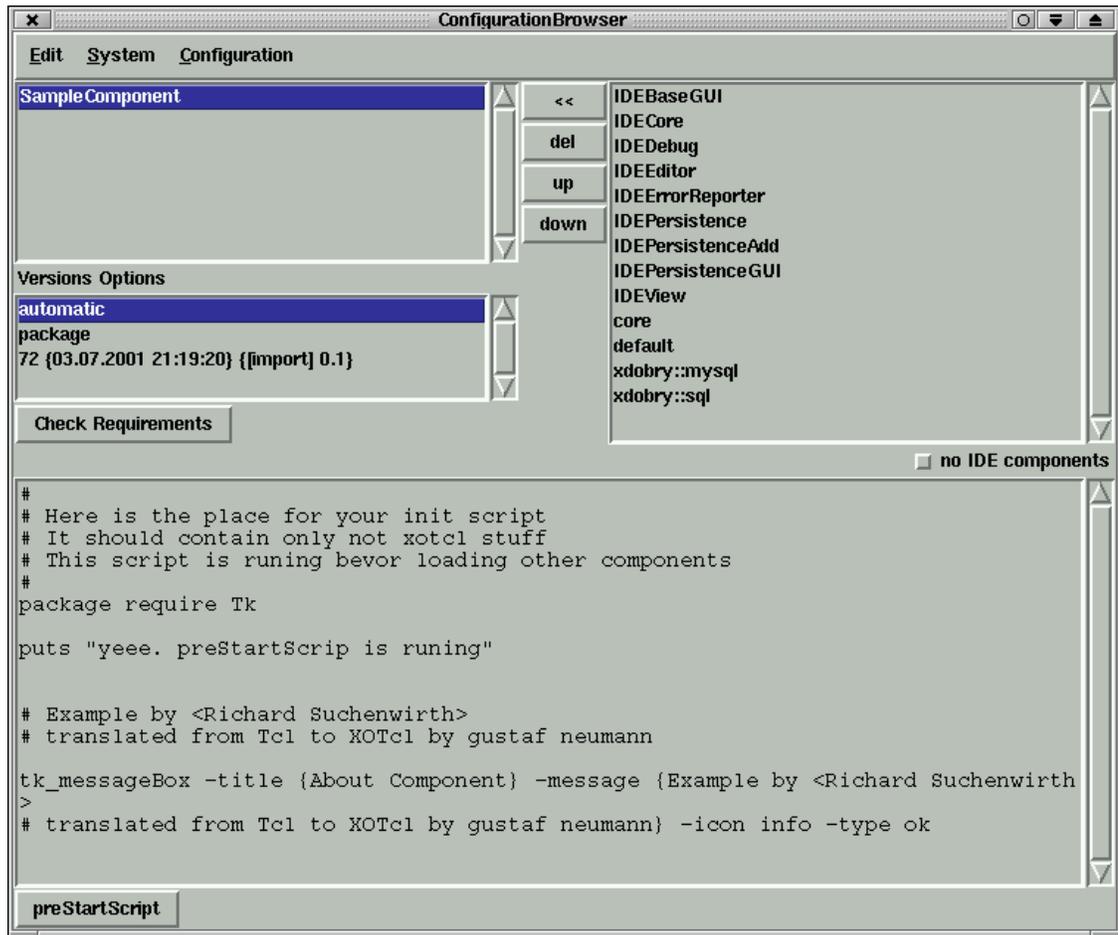
Using Configurations Maps

Launch the **Configuration Browser** by selecting menu System->Configuration Browser. Load the configuration map (Sample.cfmap) by selecting menu Edit->Load Configuration Map. Press the toggle button. You can see the aspects of configuration maps

- preStartScript - tcl code that will be started before loading components
- startScript - tcl code to start your application. Evaluated after loading all component.

You can use list boxes and buttons to specify the configuration components

Figure 4.6. Configuration Map Browser



You can edit them. Press Control-S to apply changes. Select Edit->Load Components to load components into the system (interpreter). Select Edit->Run Start Scripts to evaluate *preStartScript startScript*. You can make a new configuration map by using Edit->Init From System

Deploying Application

Distribution is the set of files that you can bind for example as a tar or zip file and distribute to other parts. A Configuration map is a base to specify the distribution. First load the component with Edit->Load Components than select menu Edit->Make Distribution. Select the directory (or create a new one). The system generates the set of files:

```
Sample Sample.cfmap SampleComponent.xotcl pkgIndex.tcl
```

Sample is the executable file to start the application. Take a look at it.

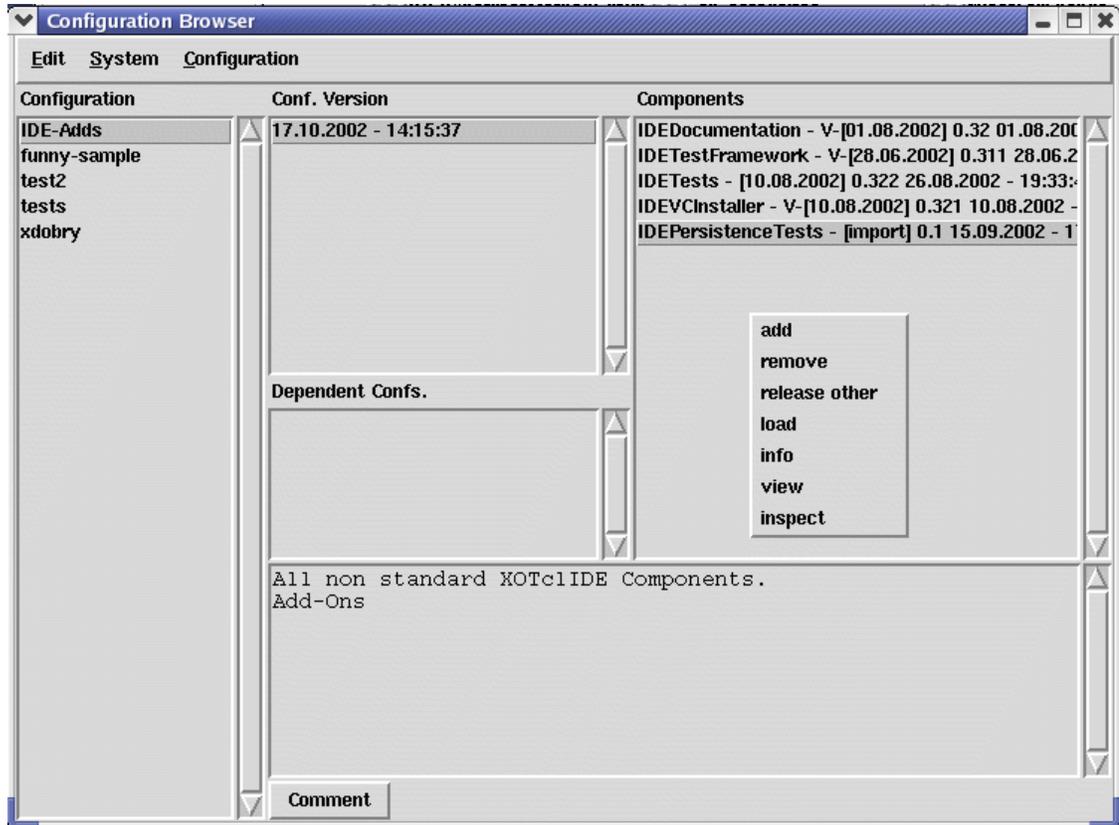
```
#!/usr/local/bin/xowish
# File generated by xotclIDE
# edit if you want

set sname [info script]
if {$sname==""} {
    # Run interactive for develop purposes
    set progdir [pwd]
} else {
    file lstat $sname stats
    # follow sym links
    if {$stats(type)=="link"} {
        set sname [file readlink $sname]
        if {[file pathtype $sname]=="relative"} {
            set sname [file join [file dirname [info script]] $sname]
        }
    }
    set progdir [file dirname $sname]
}

lappend auto_path [file dirname $progdir]
package require PlatformLogDumper
generateTclPlatformProtocol out.log
```

Configuration Browser - with Version Control System

Figure 4.7. Configuration Browser



This tool saves the configuration-maps in the version control systems so configuration-maps can also have editions and versions. Therefore you can have many editions of one configuration-map. You can browse changes among different configuration-map editions.

Warning

Unlike other browsers, all Configuration Browser functions are available only in pop-down menus.

For example you have built a program named **sqleditor**. In Version 0.1 your program is built from components as below

configuration-map - sqleditor (version 0.1)

- guisystem - version 0.1
- sqlparser - version 0.2
- persistence - version 0.3

And the additional configuration-map (sub configuration map) sqlinterfaces - version 1.2

In version 0.2 sqleditor has the following configuration

- guisystem - version 0.2
- sqlparser - version 0.2

- persistence - version 0.4

And the additional configuration-map (sub configuration map) sqlinterfaces - version 1.2

The main idea of Configuration Browser and Configuration-Maps is to have a medium for component based programming. This means you have many components that are parts of many products. All these components are in one version system.

It is not yet possible to use the **Component Browser** to generate a distribution. You must export a versioned configuration map to configuration map file with pop-up menu Conf. Version->export to map. Then you can use this file with **Configuration Map Browser**

Debugging

Debugger Browser

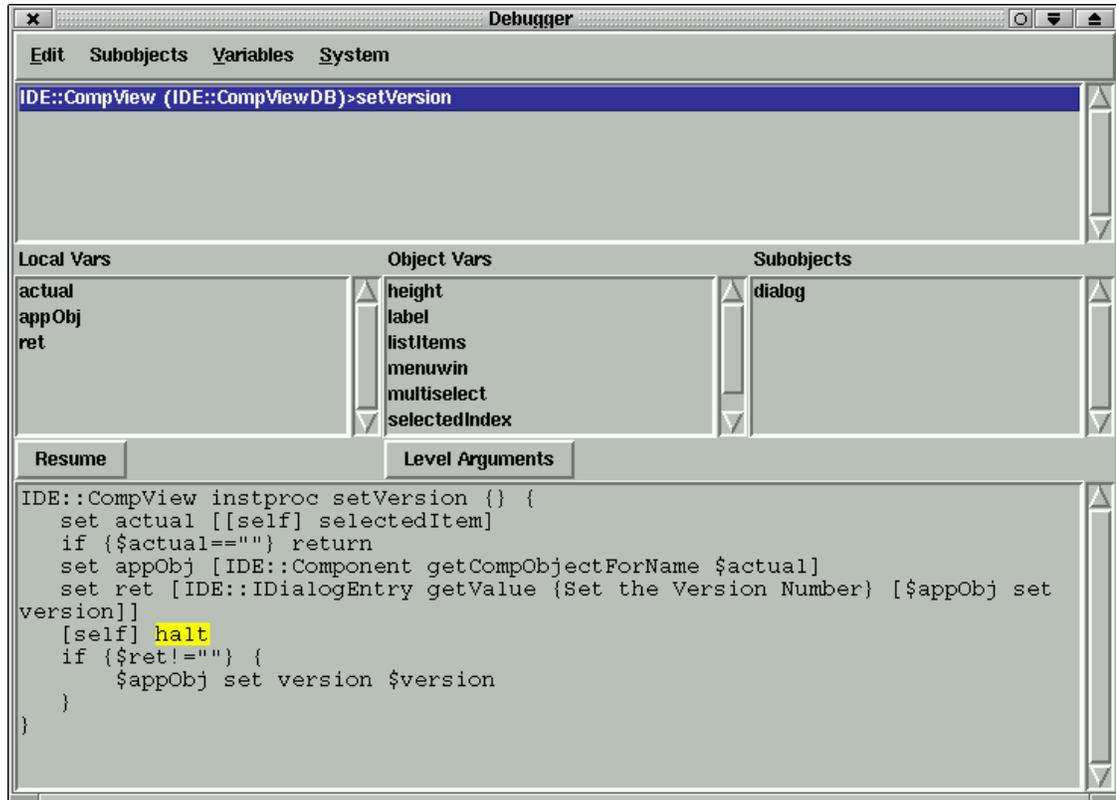
This Browser lets you stop the program flow at a point and view calling stack methods and local variables. To set a break point call the halt method

```
[self] halt
```

in Tcl procedures you need to use

```
Object halt
```

Figure 4.8. Debugger

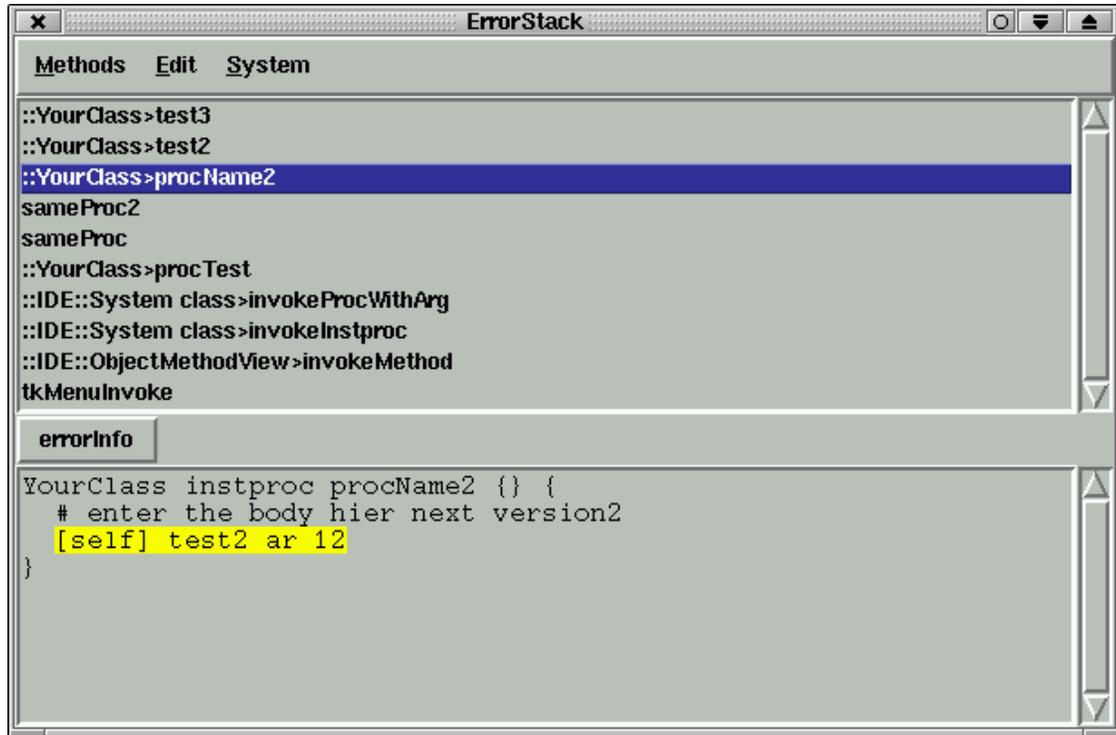


You can resume the program flow with button Resume or terminate the flow by exiting the browser (close window).

Stack Error Browser

This browser can parse the error stack info "errorInfo". XOTclIDE modifies the `bgerror` method and adds a new button to the standard error dialog (XOTclIDE browser). Tcl does not let you inspect the calling stack after errors so this is all the information available after an error. You can view (Button `errorInfo`) the original `errorInfo` text. The browser tries to highlight the called methods in the stack. In the extended debugger an error causes the debugger to be invoked directly at the site of the error.

Figure 4.9. Error Stack Browser

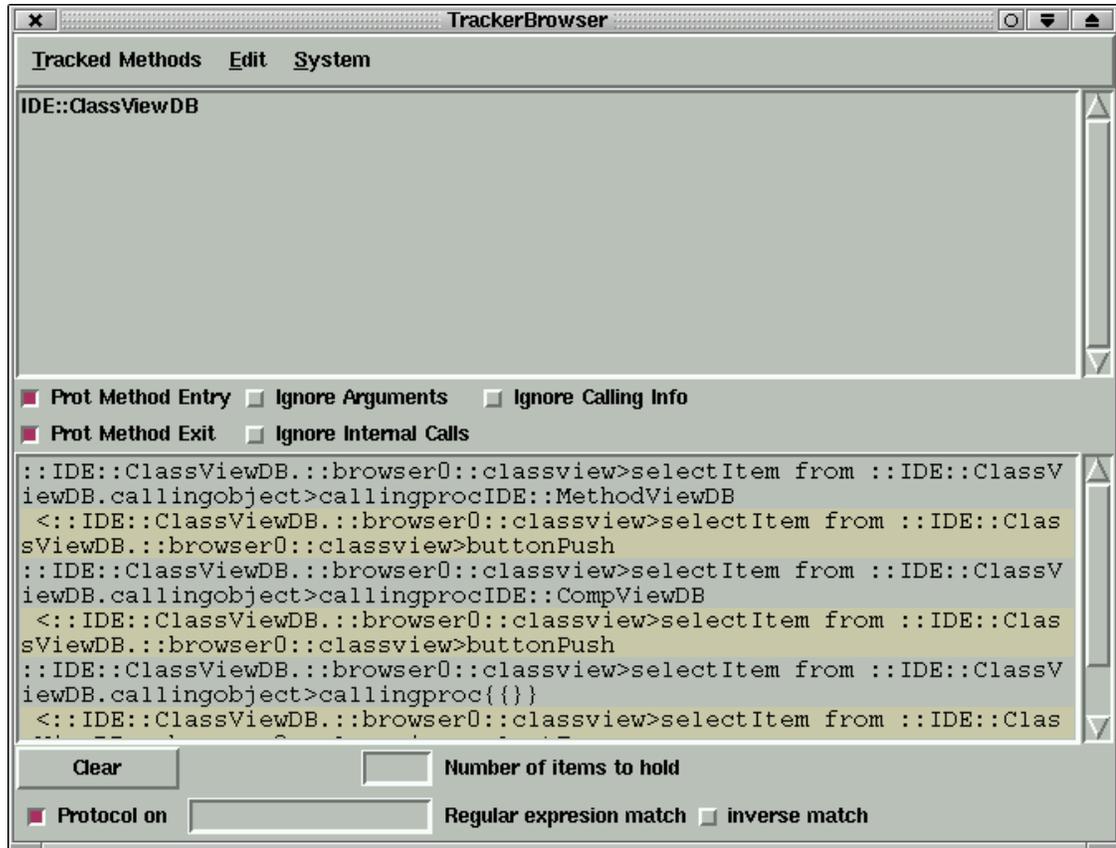


Tracker Browser

This is a GUI for a browser like the tracker from the XOTcl package `xotcl::tracker`. You can track calls to every object of a chosen class. To Track a class select menu "Track Class" from the Class Menu in the Component Browser. You can customize the tracker to show

- only method entry or exit points
- ignore internal calls from the same object
- do not show arguments or return values
- do not show calling information

Figure 4.10. Method Call Tracker



Variable access tracking and watching

XOTclIDE can be used to track read or write access on global variables or object variables. To add some track use variable menu in global variables browser or object inspector. The some selected access occurs the debugger will be invoked in access place. You can inspect in debugger the access context and resume the operation per button resume. The variables can be also watched in entry Tk widget link per - textvariable with chosen variable.

Figure 4.11. Variable Tracker and Variable Watch

In this screen-shoot are 4 tracked variables. By variable *tk_library* the debugger are invoked on every read access. There are also one XOTcl variable (object variable) *objectVar* from object *c* written in XOTcl style as `::c::objectVar`. Local procedures variables can not be tracked or watched.

To add new tracks for chosen variable use Object Inspector or Global Vars Browser. For example Variables->Debug on Write Access to add write access track. You can have multiple tracks (read, write, watch) on one variable.

Warning

Watches on variables can prevent proper unset of variables. Therefore watches can influence program flow.

Importing Tcl Projects into XOTclIDE Compon-

ents

In XOTclIDE there are three ways to import your existing Tcl sources into XOTclIDE components. This section describes how these importing function work and what their limits are. Generally if your Tcl program is well structured and has no commands or a few commands in the global script context other than class or procedure definitions the importing works out of the box.

Importing by definition tracking

There are two main importing functions in XOTclIDE, both accessible from the **Component Browser**

Load Package Component->Load Package With this function you can load any Tcl package accessible in your Tcl system by calling **package require name**.

Import Source Component->Import Source This importing function can evaluate any script in the file system. It is the same as using the command **source filename**.

The importing functions in XOTclIDE do not parse Tcl scripts but evaluate them with the Tcl Interpreter. Therefore the importing works very reliably. Any package or script can be loaded into XOTclIDE. XOTclIDE tracks the definition of Tcl procedures with the **proc name arguments body** command and the definition of XOTcl Classes, Objects and their instance methods and class methods. The importing tracker notices every newly defined procedure, creates components, and adds the procedure to this component. All procedures are also normally evaluated by Tcl interpreter. The importing tracker does not notice any other script evaluation in the global context.

Limits of Source Importing and manual adaptation

Let's examine a Tcl script in the file `myapp.tcl` that should be imported.

```
# This is great script that I want to reuse in XOTclIDE
# Author: old Tcl'er
# Revision:
package require Tk

# set debug 1
set color red
set configfile myapp.conf
if {![file exists $myapp.conf]} {
    # error "can not find config file $configfile"
}

# Starting Application
# parameters: None
proc startApp {} {
    button .re -text "Quit" -command "quite"
    # ... your program
}
... many other defined procedures

# next line starts the application
startApp
```

The importing function will create a new component with the name *myapp* with all your procedures. The new component can be seen as follows

```
# automatically generated from XOTclIDE
package provide importExample 0.1
```

```
@ tclproc startApp idemeta struct importExample default
proc startApp {} {
    # button .re -text "Quit" -command "quite"
    # ... your program
}
```

As you see, all comments, global context commands and the application starting command are lost. Comments (# lines) in the global context are not imported during evaluation because Tcl ignores them. You can use the comment importer (see the section called “Importing Tcl comments”) to add these comments to the component.

Another problem is the evaluation of Tcl commands in the global context. In XOTclIDE you should have just one such line coded in the configuration map (see the section called “Configurations Management”). To enable proper import the definition block should be moved to a special new procedure. The example above could appear as follows:

```
# This is great script that I want to reuse in XOTclIDE
# Author: old Tcl'er
# Revision:
package require Tk

# set debug 1
proc defineGlobalConstans {} {
    global color configfile
    set color red
    set configfile myapp.conf
}
proc checkConfigFile {} {
    global configfile
    if {![file exists $configfile]} {
        # error "can not find config file $configfile"
    }
}
# Starting Application
# parameters: None
proc startApp {} {
    button .re -text "Quit" -command "quite"
    # ... your program
}
proc basicStartApp {} {
    defineGlobalConstans
    checkConfigFile
    startApp
}
... many other defined procedures

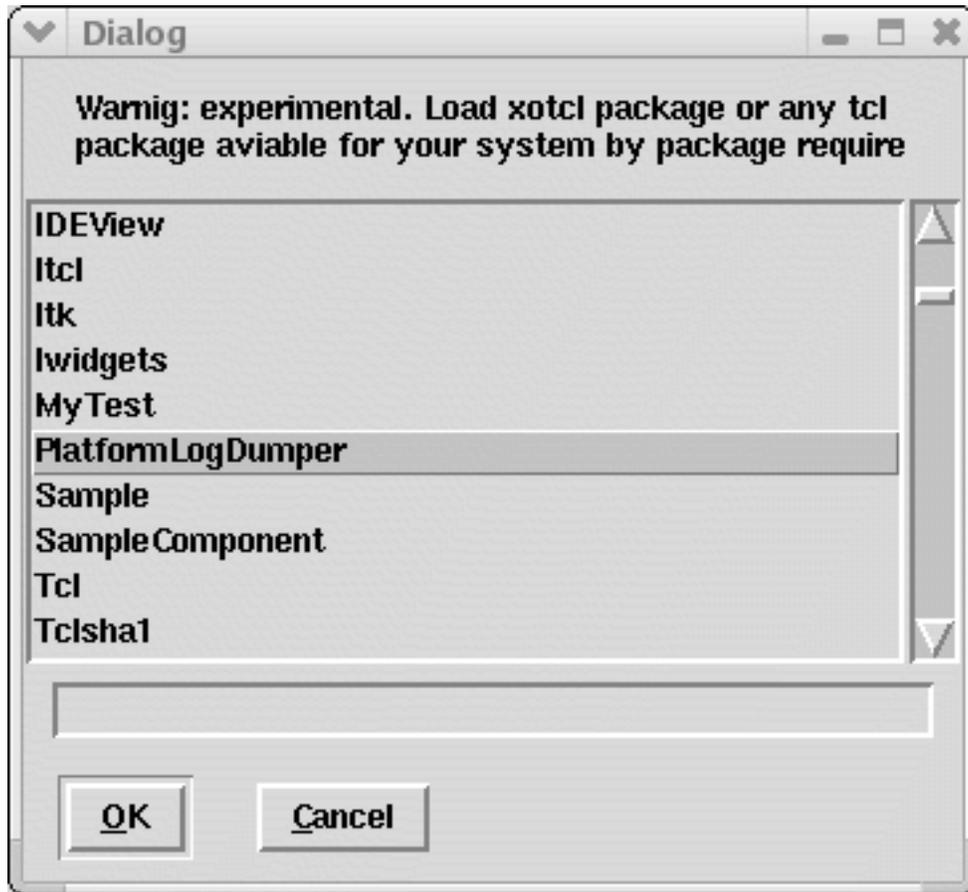
# next line start the application
basicStartApp
```

In this case we have only one line with direct script evaluation and this can be imported into XOTclIDE without losing information. I think good Tcl programs should be written this way - no evaluation in the global context - anyway.

Importing - Load regular Tcl Package

To import a regular tcl package use menu Component->Load Package in **Component Browser**. A new component with the same name as the imported package will be created, along with nested packages that are loaded from the imported package using package require.

Figure 4.12. Load Package Dialog



Importing - Import Source

Newly created components will have names that correspond to the script file name, without extension. Any nested script evaluation or package require commands are respected. Before sourcing the script XOTclIDE changes the working directory to the path of the script file. The application will start normally. A problem may appear when the application uses a toplevel window, as the toplevel window is already used by the XOTclIDE Transcript window. If an error occurs while importing, importing is interrupted with the error message.

Procedures defined in the `::` namespace will be added to a Tcl-Proc-Group named "default". Procedures with names "mynamespace::myname" will be added to a Tcl-Proc-Group named "mynamespace".

Importing by System Introspection

Another way to import your application is to start XOTclIDE from your application and introspect it with XOTclIDE. XOTclIDE can import procedures and XOTcl object classes directly from a running Tcl interpreter. To start XOTclIDE from your application you can use the `START.tcl` script in the XOTclIDE directory. Change the working directory to the XOTclIDE directory with the `START.tcl` script.

To import code from your running application you must first create a new component where all proced-

ures and classes from the Interpreter will be stored. To import a Tcl procedure from the Tcl interpreter, first select the component where you want to import to and use menu Command->Low Level Functions->Register Tcl Proc from Interp in **Component Browser**

To import XOTcl Classes from the Interpreter select the component and choose menu Command->Low level Functions->Register Class from Interp in **Component Browser**

Although importing an application like this is more work, it's a good choice if you want to import only part of an application or the application is not in a readable format.

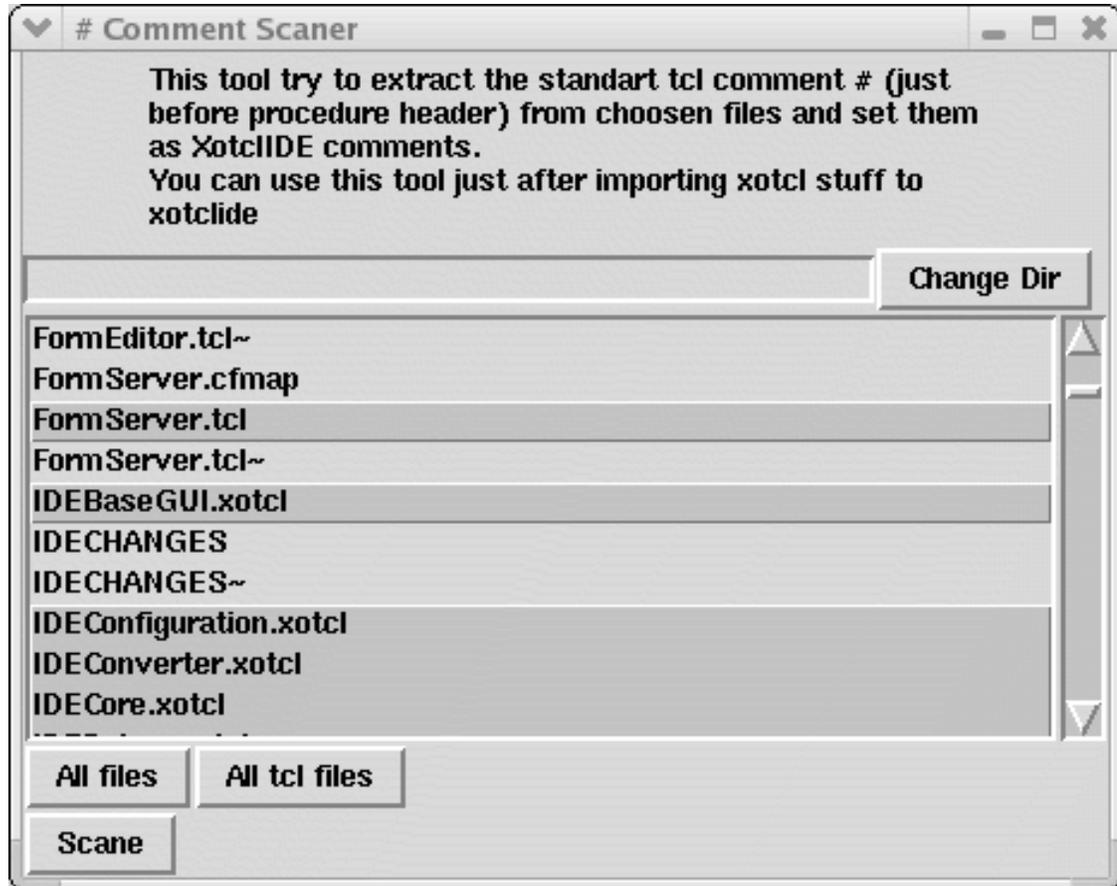
Importing Tcl comments

Consider the example below

```
# This procedure make magic initialization of
# X Module.
# Warning:
proc initModuleX {{path {}}} {
    #
    #
}
```

The three line comment belongs to procedure initModuleX. XOTclIDE has a special parser that can scan Tcl script files and associate the comments to previously imported components. To launch the # Comments Scanner use the menu System->Extras-># Comments Scanner Unlike the source importer, this tool does not evaluate the selected scripts but scans all lines after a leading # character.

Figure 4.13. Comment Scanner Tool



Plug-ins Architecture

One of main advantages of XOTclIDE is easy customizing of XOTclIDE for users needs. Many of XOTclIDE components are loaded dynamic at runing time only on demand. In menu System->Plug Ins are all currently registered plug-ins accesible. The plug ins are normal components the registration and start scripts are specified per file `pluginslist.txt` in XOTclIDE directory.

Following Plug-Ins are currently delivered with XOTclIDE

Unit Test Framework	Unit Test Framework programmed after Smalltalk SUnit (JUnit, NUnit). See also Unit Tests Homepage [http://www.xprogramming.com/testfram.htm]
XOTclIDE Unit Tests	Tests of XOTclIDE itself
Funny Graphics Example	Small XOTcl Application taken from Tcl Wiki
HTML Doc Generator	Generate HTML Source Code Documentation from Source Comments
# Comments Scanner	Importing tool described in the section called "Importing Tcl comments"
Tcl Wiki Reaper	Can import code sniplets form Tcl wiki Tcl Wiki [http://mini.net/tcl/8179]

TclKit Deployer Tool	This tool extend the functionality of Application Deployer Wizard. It can generate TclKit Distributions or standalone Starpacks directly from XOTclIDE. It work properly only form XOTclIDE TclKit version or if TclKit enviroment are installed properly in your Tcl system
Tk Win Inspector	This tool can inspect all Tk windows. It can be used to view and change all configuration of every Tk window. Tk Inspector includes also widget serializator that can be used to serialize every windows and their descend to Tcl script that can be used as code snippet.
Tcl Script Editor	Ideal for edit and test short Tcl-scripts that all contents are evaluated in global context. You can use all advantages of XOTclIDE: syntax highlighting, syntax check, code completion. The script can be evaluated in slave interpreter.
SQL Browser	GUI helper for SQL access to all databases supported by XOTclIDE. Additional 2 Lists-Views show all table names and columns names (schema of DB). The result will be displayed in TkTable-Widget. Every cell can be also viewed in separately view.
Visual Regexp	This plug-in is adapted GPL program written by L. Riesterer original source [http://laurent.riesterer.free.fr/regexp/]

Chapter 5. Additional Information

Author and License of XOTclIDE

The program XOTclIDE was written by Artur Trzewik and is GNU Public License Software (GPL) The documentation of this program (this document) is licensed under GNU Free Documentation License (GFDL).

The original GPL License text can be found in the file `LICENSE` included in XOTclIDE source package or at the GNU Homepage [<http://www.gnu.org>]

XOTclIDE WWW Resources

- XOTclIDE [<http://www.xdobry.de/xotclIDE>] XOTclIDE Home Site
- Tcl [<http://tcl.tk>] Main Tcl Site
- XOTcl [<http://xotcl.org>] XOTcl Site
- Active State Tcl [<http://www.activestate.com/tcl>] Commercial Tcl Maintainer. Offers free Tcl binaries (recommended for Windows)
- Tcl Wiki [<http://mini.net/tcl>] Tcl Wiki - User Forums