Introduction

1.1. Scope of the document

This document is part of the JTCL project set of documentation, it aims at describing how to use the JTCL set of classes to write client/server applications. A knowledge of Tcl/Tk and jTcl, the underlying languages of the JTCL libraries, is required.

A brief introduction to the Tcl language can be found in [2] and a complete tutorial in [3]. The structure of jTcl is described in [2]. A general overview of JTCL is given in [1].

This document has the following structure:

- Chapter 2 – **The `lang` set of packages**: explains the content and use of the packages defining basic function for the jTcl language.
- Chapter 3 – **The `tcp` set of packages**: explains the content and use of the packages providing a functional basis for client/server applications using the HTTP and RPC protocols.
- Chapter 4 – **The `ic` set of packages**: explains the content and use of the packages enabling the use of the rule-driven graphical interface.

Table of Contents

1.2. Glossary

The following terms and abbreviations are used throughout this document.

FTP	File Transfer Protocol
HTML	Hypertext Mark-up Language
HTTP	Hypertext Transfer Protocol
HTTPd	Hypertext Transfer Protocol (HTTP) daemon
jTcl	Java-like Tool Command Language, an object-oriented extension to the Tcl language.
MIME	Multipurpose Internet Mail Extensions
RPC	Remote Procedure Call
Tcl	Tool Command Language, a free platform-independent scripting language designed in the late 1980's by Prof. John Ousterhout of the University of California, Berkeley.
TCP/IP	Transmission Control Protocol / Internet Protocol, a set of network protocols used in the Internet.
Tk	Tool Kit, a graphical tool-set (also platform-independent) associated with Tcl.
URL	Universal Resource Locator

1.3. Bibliographic references

The following references are used throughout this document.

[3] Brent B. Welch
Practical Programming in Tcl and Tk, 2nd edition
Prentice Hall, 1997
ISBN 0-13-616830-2

[4] T. Berners-Lee, R. Fielding, H. Frystyk
Hypertext Transfer Protocol -- HTTP/1.0
RFC 1945, May 1996

[5] R. Fielding, J.Gettys, J.Mogul, H. Frystyk, T. Berners-Lee
Hypertext Transfer Protocol -- HTTP/1.1
RFC 2068, January 1997

[6] M. St. Johns
Authentication Server
RFC 931, January 1985

[7] M. St. Johns
Identification Server
RFC 1413, February 1993

[8] N. Freed, N. Borenstein
Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies
RFC 2045, November 1996

[9] N. Freed, N. Borenstein
Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types
RFC 2046, November 1996

[10] K. Moore
MIME (Multipurpose Internet Mail Extensions) Part Three: Message Header Extensions for Non-ASCII Text
RFC 2047, November 1996

[11] N. Freed, J. Klensin, J. Postel
Multipurpose Internet Mail Extensions (MIME) Part Four: Registration Procedures
RFC 2048, November 1996

[12] N. Freed, N. Borenstein
Multipurpose Internet Mail Extensions (MIME) Part Five: Conformance Criteria and Examples
RFC 2049, November 1996

1.4. Typographic conventions

The following typographic conventions apply throughout this document.

- Names of Tcl keywords are printed with a typewriter-like font, for example: “the `else` part of an `if` command is optional”.
- In extract of Tcl code, the characters typed at the shell prompt are shown preceded by the percent character (which is the default prompt character for the Tcl shell). If a single command is spawn onto several lines, each continuing line is preceded by the “>” character. Output from the shell is shown preceded by the “⇒” character.

1.5. Trademarks and copyrights

Windows, Windows 3.x, Windows 95, Windows NT are trademarks from Microsoft Corporation.

Macintosh, MacOs are trademarks from Apple Computers.

Tcl/Tk is copyrighted by the Regents of the University of California, Sun Microsystems, Inc., and other parties.

jTcl is copyrighted by FRIDU, a free software company, South Brittany University and other parties.

2. The `lang` set of packages

2.1. Overview

This set of packages performs basic tasks related to the jTcl language. It consists of the following packages:

lang.debug	contains several debug-oriented procedures.
lang.serial	contains several serialization procedures used to store permanently object contents on disk.
lang.doit	test routines.
lang.object	contains the definition of a root class named `Object`.
lang.search	contains the search mechanism used to retrieve the jTcl file distribution.

package name	**associated file**	**imported packages**	**content**
lang.object	Core/Lang/Tcl/objectLang.jTcl	lang.debug	Object class
lang.debug	Core/Lang/Tcl/debugLang.jTcl	-	
lang.serial	Core/Lang/Tcl/serialLang.jTcl	lang.object	
lang.doit	Core/Lang/Tcl/doitLang.jTcl	-	
lang.search	Core/Lang/Tcl/searchPkgLang.jTcl	-	

Table 2-1: contents of the `lang.*` packages.

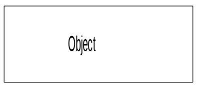
2.2. Reference

This chapter is intended to be used as a reference guide for all classes defined in the `lang` packages.

Object

package: `lang.object`

This is the root class of the inheritance tree of the JTCL package. It provides some basic mechanisms useful for debugging.



instance variables:

clog	name of the log file used by the `log` method. Initial value is `stdout`.
errno	last error status. Initial value is `{}`.
level	trigger for the logging mechanism: the logging mechanism is enabled only if the `JTCL_LOG` environment variable is set to a value greater or equal to the value of `level`. Initial value is `0`.

public methods:

log {args}

according to the values of the `JTCL_LOG` environment variable and `level` instance variable, may write an output message to the log file defined by the `clog` instance variable. If `JTCL_LOG` is greater or equal than `level`, then the message is written. The format of the message is the following:

```
_C_class._O_instance:errno: args
```

where *`class`* is the name of the class, *`instance`* is the concatenation of the name of the class and the instance counter (automatically incremented with each creation of a new object of that class), *`errno`* is one of the instance variables of the `Object` class (see above) and *`args`* is the list of arguments passed by the user to this method (usually it consists of a human-readable message).

dump {{TYPE Object}}

dump the content of the object according to the value of the `TYPE` parameter. `TYPE` can take one of the following values: “`Object`” (dumps only instance variables), “`Class`” (dumps only class variables and public methods), “`All`” (dumps everything). For any other value of `TYPE`, nothing is dumped. Default value for `TYPE` is “`Object`”. The dump is always directed to the standard output (`stdout`). The format of the printed message is the following:

```
dump Object=_O_instance
*Object owned
+name->value
...
*Class owner
+name->value
...
```

	where *`instance`* is the concatenation of the name of the class and the instance counter (automatically incremented with each creation of a new object of that class). The first part named "`*Object owned`" appears only if `TYPE` is "`Object`" or "`All`", in that part all instance variables are listed, including the inherited ones and the default ones (such as `class` and `my`). The second part named "`*Class owner`" appears only if `TYPE` is "`Class`" or "`All`", in that part all class variables and all public methods are listed, including the inherited ones and the default ones (such as `extends`, `super`, `instanceCounter` and `my`). For both parts, *`name`* is the name of the variable or the method and *`value`* is the value of the variable or, for a method, the name of the method preceded by the name of the class in which the method is defined.
signal {MSG}	stops the interpretation and displays an error message to the standard output. The message consists of the name of the instance (under the form "`_O_`", followed by the name of the class, followed by the instance counter) and the message provided by the user via the `MSG` parameter.

2.3. Examples

For an example of use of the Object class, refer either to the "Gum machine" sample given in [2] or to the following chapters.

3. The `tcp` set of packages

3.1. Overview

These packages provide means to create client/server applications using the HTTP and RPC protocols. The following classes are defined: `TcpChannel`, `TcpServer`, `TcpServerCl`, `TcpHttpd`, `TcpHttpdCl`, `TcpHttpdRpc`. Relations between these classes are described in the following diagram.

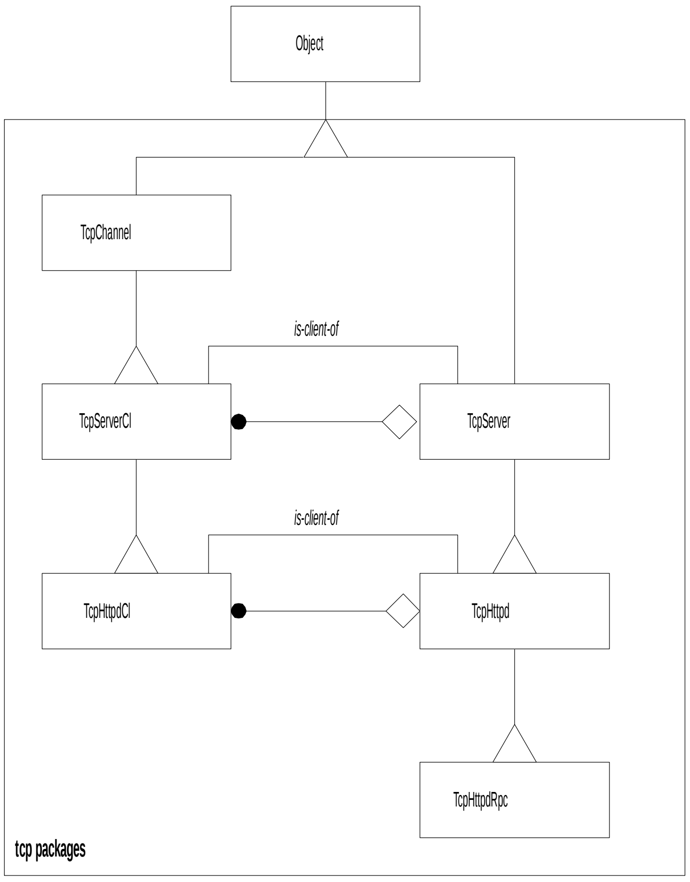


Figure 3-1: OMT diagram of the `tcp` classes

The class are organized into the following packages:

tcp.channel provides the basic functionality (management of a virtual channel) for all the network classes defined in the next packages. Defines the `TcpChannel` class.

tcp.server provides the basic classes for TCP/IP communication (management of actions of the server). Defines the `TcpServerCl` and `TcpServer` classes.

tcp.httpd	Provides a small HTTP server. Defines the `TcpHttpdCl` and `TcpHttpd` classes.
tcp.rpc	Provides RPC communication. Defines the `TcpHttpdRpc` class.

package name	associated file	imported package	content
tcp.channel	Core/Tcp/Tcl/channelTcp.jTcl	lang.object	TcpChannel class
tcp.server	Core/Tcp/Tcl/serverTcp.jTcl	tcp.channel	TcpServerCl and TcpServer classes
tcp.httpd	Core/Tcp/Tcl/httpdTcp.jTcl	tcp.server	TcpHttpdCl and TcpHttpd classes
tcp.rpc	Core/Tcp/Tcl/rpcTcp.jTcl	tcp.httpd	TcpHttpdRpc class

Table 3-2: contents of the `tcp.*` packages.

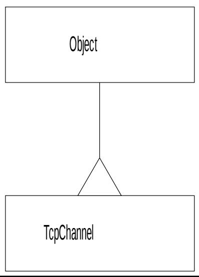
3.2. Reference

This chapter is intended to be used as a reference guide for all classes defined in the `tcp` packages.

TcpChannel

package: `tcp.channel`

This class provides basic management of a communication channel based on the use of sockets. It is used as the base class for all client-side communication classes.



class variables:

blocking	configuration parameter for the output socket. This option determines whether I/O operations on the channel can cause the process to block indefinitely. The value of the option must be a proper boolean value (0 for non-blocking, 1 for blocking). Channels are normally in blocking mode; if a channel is placed into non-blocking mode it will affect the operation of the `gets`, `read`, `puts`, `flush`, and `close` Tcl built-in commands. Initial value for this variable is `1`.
translation	configuration parameter for the output socket. In Tcl scripts the end of a line is always represented using a single newline character (\n). However, in actual files and devices the end of a line may be represented differently on different platforms, or even for different devices on the same platform. The Tcl I/O system performs all the necessary translations. The default translation mode, `auto`, handles all the common cases automatically, but it is possible to provide explicit control with the following values: `binary`, no end-of-line translations are performed, `cr`, the end of a line in the underlying device is represented by a single carriage return character, `crlf`, the end of a line in the underlying device is represented by a carriage return character followed by a linefeed character, `lf`, the end of a line in the underlying device is represented by a single newline character. Initial value for this variable is `auto`.

instance variables:

buffer	input buffer. Initial value is `{}`.
bufOut	output buffer. Initial value is `{}`.
in	input socket.
out	output socket.

constructor:

TcpChannel {SOCK_IN SOCK_OUT}

this method (1) sets the `in` and `out` instance variables to the values given by `SOCK_IN` and `SOCK_OUT`, (2) configures the output socket using information provided by the `blocking` and `translation` class variables and (3) assigns the `read` method as the call-back for the input socket, i.e. as soon as data is present in the input socket, the `read` method will be called.

`SOCK_IN`: input socket, shall be defined using the `socket` Tcl built-in command, no default value.

`SOCK_OUT`: output socket, if used, shall be defined using the `socket` Tcl built-in command, otherwise, if `{}` is given as a value, then the output socket will be the same as the input socket, default value is `{}`.

destructor:

free {} closes the sockets.

return value: none.

public methods:

read {} this procedure is associated with an event handler, it is called whenever the input socket (`in`) becomes readable. In that case the characters read in the socket are appended to the input buffer (`buffer`). In case an "End-Of-File" (EOF) character is read from the socket, this procedure is release from the event handling binding and the `errno` instance variable is set to EOF.

return value: none.

write {BUFFER} this procedure enables to write data to the output socket. The procedure does not write directly to the socket (to write directly to the socket, one should use the `writeAndFlush` method with the buffer passed as argument) but rather install the `writeAndFlush` method as an event handler associated with the output socket in order to write the data contained in the `BUFFER` argument in non-blocking mode.

BUFFER: the buffer to be written to the output socket, can contain any data, no default value.

return value: none.

writeAndFlush {{BUFFER "\000"}}
this procedure can be invoked either directly, either asynchronously. If it is invoked directly, then the content of `BUFFER` is written to the output socket (`out`) in only one shot, which can lead to some blocking problems since if there is no available room for data in the socket, the program will freeze until the socket becomes writable again. If the procedure is invoked asynchronously, then the data from the output buffer (`bufOut`) is written to the output socket byte by byte until the buffer becomes empty.

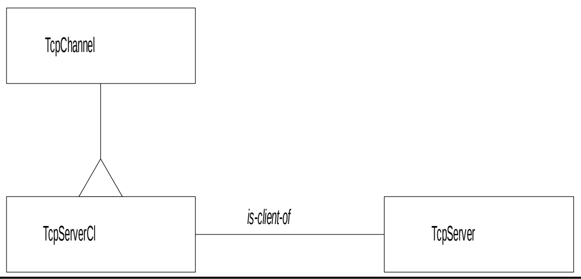
`BUFFER`: the buffer to be written to the output socket in case of direct call of the procedure, in the case of asynchronous use, `BUFFER` shall have the "`\000`" value. Default value is "`\000`".

return value: none.

TcpServerCl

package: `tcp.server`

Generic client channel class. An object of this class is located in the server side and is created by the `TcpServer` class to handle all communications with a given client connected by TCP socket.



instance variables:

server	identifier of the server the object is connected to, set by the constructor.
host	TCP/IP name of the remote host the object is connected to, set by the constructor.
port	TCP/IP port number on the remote host, set by the constructor.
user	user name, set by the `auth` method.

constructor:

TcpServerCl {SERVER_ID HOST SOCK PORT}

this method (1) sets the `server`, `host` and `port` instance variables to the values given by (resp.) `SERVER_ID`, `HOST` and `PORT`, (2) calls the constructor of the superclass (`TcpChannel`) passing `SOCK` as argument. This method is most always called by the `TcpServer` class.

`SERVER_ID`: object identifier (as returned by the `new` jTcl command) of the server the client is connected to, no default value.

`HOST`: name of the remote host, no default value.

`SOCK`: socket for communication between the client and the server, shall be defined using the `socket` Tcl built-in command, no default value.

`PORT`: port number on the remote host, no default value.

destructor:

free {} call the `clientLeaved` method of the associated server.

return value: none.

public methods:

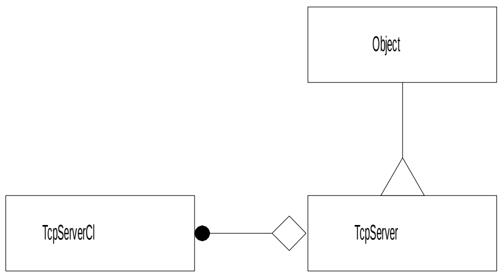
auth {} this method is called by the associated server in order to authenticate the client. It follows the procedure defined in RFC 931 [5] (which is now superseded by RFC 1413 [6]). A new socket is temporarily created to the port #113 of the server, query data is sent on this socket and the result is read on the same socket. The result is parsed to retrieve the user name,

	stored in the `user` instance variable. If this procedures fails, then `user` is set to “`Unknow`” [sic]. return value: always returns “`OK`”.
refused {args}	this procedure is called by the associated server each time a client connection is refused by the server. It simply outputs a message and then terminates `args`: the list of arguments passed by the user to this method (usually it consists of a human-readable message). It is sent to the output socket of the client. return value: none.

TcpServer

package: `tcp.server`

Generic server class using TCP/IP connection.



class variables:

clientsClass	name of the class representing the clients allowed to connect to this server class. Initial value is `TcpServerCl`.
vwait	name of the variable to wait for in the event handling process. Initial value is `_TCP_SV_VWAIT`.

instance variables:

clientsAddr	list of the currently connected clients. Initial value is `{}`.
clientsNum	number of currently connected clients. Initial value is `0`.
clientsMin	minimum number of clients allowed to connect, if actual number of clients goes below that limit, then the server terminates. Initial value is `0`.
clientsMax	maximum number of clients allowed to connect. Initial value is `64`.
port	port number on the server host. Set by the constructor.
socket	server socket (listens for new connections). Set by the constructor.

constructor:

TcpServer {PORT}

this method (1) sets the `port` and `socket` instance variables to the values given by (resp.) `PORT` and a server socket opened to that port, and (2) assigns the `accept` method as the call-back for the server socket, i.e. as soon as a client tries to connect to the socket, the `accept` method will be called. Note: if the socket cannot be created (e.g. an illegal port number is given), then an error message is displayed and the execution is aborted.

`PORT`: port number on which the server will be listening to client connections, no default value. Note: port numbers below 1024 are reserved by most systems (e.g. port #80 is used by HTTP servers).

destructor:

free {}	kills all the connected clients and closes the socket.

return value: none.

public methods:

vwait {}

this is the main event loop. It loops until the variable defined by the `vwait` instance variable is set. This variable is set when a new client is disconnected. If there are less clients than defined by the `clientsMin` instance variable, then the server terminates.

return value: none.

clientLeaved {CLIENT_ID}

this procedure is called by the client just before it terminates. The client is then removed from the list of connected clients and the variable defined by the `vwait` class variable is set to some value in order to interrupt the main event loop.

`CLIENT_ID`: identifier of the client about to leave, no default value.

return value: none.

accept {SOCK CLIENT_HOST PORT}

this procedure is automatically called when a client wants to establish a new connection. It creates a new object of class `TcpServerCl` which will be used to handle all the communication with the client. Arguments are passed “as-is” to the constructor of `TcpServerCl`. Authentication is performed calling the `auth` method of `TcpServerCl`. The connection can fail if (1) the client authentication fails, or (2) the maximum number of authorized clients to connect is reached. In case of failure, the client is informed via the `refused` method of the `TcpServerCl` class.

`SOCK`: identifier of the socket created by the client trying to connect, no default value.

`CLIENT_HOST`: host name of the client trying to connect, no default value.

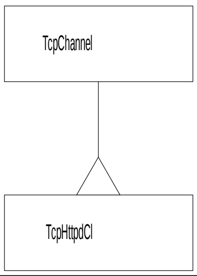
`PORT`: port number on the server, no default value.

return value: none.

TcpHttpdCl

package: `tcp.httpd`

Generic client class. A object of his class is located in the server side and is created by the `TcpServer` class to handle all communications with a given client connected by TCP socket.



instance variables:

req	request method of the client to retrieve information on the server, possible values are `GET` or `POST`, set by the `process` method of the associated server class.
url	URL on the server the client wants to connect to, set by the `process` method of the associated server class.
query	query on the server the client wants to connect to, set by the `process` method of the associated server class.
length	contains the value given by the "`Content-length`" field of the HTTP header, set by the `process` method of the associated server class.
keep-alives	contains the value given by the "`Keep-Alives`" field of the HTTP header, set by the `process` method of the associated server class.
cookie	contains the value given by the "`Cookie`" field of the HTTP header, it is a pair name and value, set by the `process` method of the associated server class.
if-modify	contains the value given by the "`If-Modified-Since`" field of the HTTP header, set by the `process` method of the associated server class.
data	data following the HTTP header sent by the client, contains `length` bytes, set by the `process` method of the associated server class.
page	contains the body of an HTTP page.
header	contains the header of an HTTP page.

constructor:

TcpHttpdCl {HTTP_ID HOST SOCK PORT}

this method does nothing but send the arguments to the constructor of its superclass.

`HTTP_ID`: class identifier (as returned by the `new` jTcl command) of the server the client is connected to, no default value.

`HOST`: name of the remote host, no default value.

SOCK: socket for communication between the client and the server, shall be defined using the socket Tcl built-in command, no default value.

PORT: port number on the remote host, no default value.

public methods:

fCopyDone {FD LEN}

this method closes the file which descriptor is FD and destroys the object (except if the keep-alives instance variable is defined).

FD: file descriptor of the file to be closed, no default value.

LEN: length of the file, only used to print debug message, no default value.

return value: none.

read {}

this method overrides the read method defined in the TcpChannel class. It is automatically called whenever there is new data to read in the input socket. This method reads data from the input socket, if this data corresponds to the End-Of-File character, then the execution is stopped. If the data corresponds to a blank line, then the HTTP page is processed by the server (using the process method of the server class) and effectively sent to the client (using the write method). if the data corresponds to something else (non-blank line), then the data is appended to the buffer instance variable.

return value: EOF if the End-Of-File (EOF) character is read.

write {PAGE}

this method is called by the read method in order to send back to the client the requested page. An action is performed depending on the value of the first part of the PAGE parameter, possible values are the following:

"PAGE": in that case, the body of the data is contained in the page instance variable and the header in the header instance variable.

"HTML": in that case, the data to send back is located in the second part of the PAGE parameter.

"FILE": in that case, the body of the data to send back is contained in a file on the server which descriptor is given by the second part of the PAGE parameter. The header of the data is located in the header instance variable of the client channel.

After that, the class is destroyed (since HTTP/1.0 is a stateless protocol) except in the case where the keep-alives instance variable is defined.

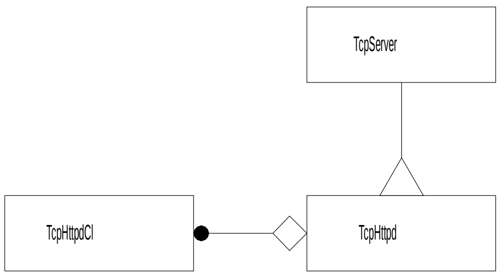
PAGE: this parameter is a pair consisting of a keyword ("HTML", "FILE" or "PAGE") and a pair (file descriptor, full path name of the file) as returned by the openUrl method, no default value.

return value: none.

TcpHttpd

package: `tcp.httpd`

An HTTP server class.



class variables:

clientsClass Name of the class used for communication with the client. A new instance of that class is created each time a new client tries to connect. Initial value is `TcpHttpdCl`.

mimes name of an array variable defining all the supported MIME types, the index of the array shall be the file extensions. Initial value is `_HTTPD_MIME_TYPE`. This variable contains the following:

index	*content*
.gif	“image/gif”
.jpg	“image/jpg”
.html	“text/html”
.htm	“text/html”
.txt	“text/html”
.class	“image/gif”
.tcl	“application/x-tcl”
.tk	“application/x-tk”

MIME media types are defined in RFC 2046 [9].

errors name of an array variable defining all the error messages, the index of the array shall be the error numbers. Initial value is `_HTTPD_ERROR_MSG`. This variable contains the following:

index	*content*
204	“204 No Content”
304	“304 Not Modified”
400	“400 Bad Request”
404	“404 URL/File Not Found”
503	“503 Service Unavailable”
504	“504 Service Temporarily Unavailable”

The complete list of error codes is given by RFC 1945 [4] for HTTP/1.0 and RFC 2068 [5] for HTTP/1.1.

errFmt name of the variable containing the format of an error message. Initial value is `_HTTPD_ERROR_FMT`. This variable is a string containing a fragment of an HTML page to be displayed in case of error. Various information are provided, such as error code, error message, URL, etc.

instance variables:

rootDir root directory of the HTTP file tree, set by the constructor.

index name of the index file in an HTTP file tree, initial value is "`index.html`".

host name of host as given by the "`host:`" field in the HTTP header the client sends, set by the `process` method.

constructor:

TcpHttpd {PORT ROOT_DIR}

sets the `rootDir` instance variable to the value provided by `ROOT_DIR` (a check is perform, if `ROOT_DIR` does not match a valid directory, then the execution is aborted) and call the constructor of the superclass (i.e. `TcpServer`) passing `PORT` as argument.

`PORT`: port number on which the server will be listening to client connections, no default value.

`ROOT_DIR`: root directory of the HTTP file tree (only files and directories found below this level will be available to the clients), shall be a valid directory, no default value.

public methods:

date {SECONDS} generate a date in HTTP format from a system date given by the CLICKS parameter. The format is "%a, %d %b %Y %H:%M:%S %Z", where:

%a:	abbreviated weekday name
%d:	day of month
%b:	abbreviated month name
%Y:	year with century
%H:	hour in 24-hour format
%M:	minutes
%S:	seconds
%Z:	time zone name

for example: "Wed, 31 Dec 1997 09:17:58 WDT". This format is referred as the preferred of the three authorized formats in the HTTP specification document [5].

`SECONDS`: an integer time value, typically returned by one of the following Tcl built-in command: `clock seconds`, `clock scan`, or the `atime`, `mtime`, or `ctime` options of the `file` command, no default value.

return value: the date formatted as described above.

error {CODE URL MSG {ACTION {Html/error.html}}

builds an HTTP page (header and body) which body is an HTML page to be displayed in case of an error.

`CODE`: HTTP error code, shall be present in the array indicated by the `errors` class variable, no default value.

`URL`: the list of arguments passed by the user to this method (usually it consists of a human-readable message). It is sent to the output socket of the client.

`MSG`: the list of arguments passed by the user to this method (usually it consists of a human-readable message). It is sent to the output socket of the client.

`ACTION`: the list of arguments passed by the user to this method (usually it consists of a human-readable message). It is sent to the output socket of the client.

return value: the HTTP message as defined above.

header {CLIENT_ID MIME FILE}

this method generates an HTTP header. This header is appended to the header instance variable of the associated client (given by `CLIENT_ID`). The generated header has the following format:

```
HTTP/1.0 200 Data follows
Date: current-date
Last-Modified: last-modif-date
Content-Type: mime-type
Content-Length: file-size
```

where:

- *`current-date`* is the current date, formatted by the `date` method,
- *`last-modif-date`* is the date of last modification of the file which name is given by `FILE`, this date is also formatted by the `date` method,
- *`mime-type`* is a MIME type given by `MIME`,
- *`file-size`* is the size in bytes of the file which name is given by `FILE`.

`CLIENT_ID`: instance identifier (as returned by the `new` jTcl keyword) of the associated client, shall be of type given by the `clientsClass` class variable, no default value.

`MIME`: shall be a valid MIME type, no default value.

`FILE`: shall be a valid file name, no default value.

return value: none.

openUrl {URL}

tries to open the file given in the URL provided as parameter. This function (1) tests if the provided file name is legal (valid file or directory name located below the HTTP root directory), (2) creates a full path name out of the provided file name, by concatenation with the HTTP root directory given by the `rootDir` instance variable and, in the case of a directory, by appending the default index file name given by the `index` instance variable, and (3) opens the specified file.

`URL`: shall be a valid URL as described in [4] and [5], no default value.

return values:
- "`Error BelowRoot`" if the URL indicates a path name located below the HTTP root directory,
- "`ERROR` *`error-msg`*" where *`error-msg`* is a string describing the error, if another kind of error occurs (e.g. the file does not exist),
- a pair consisting of the file descriptor (as returned by the `open` Tcl built-in command) and the full path name of the file, if the command proceeded correctly.

mime {EXT CLIENT_ID}

gets file type from extension (given by `EXT`) and retrieve URL from associated client (given by `CLIENT_ID`).

`EXT`: shall be a valid MIME extension listed in the array referenced by the `mimes` class variable, default value is `.html`.

`CLIENT_ID`: identifier of the associated client, no default value.

return value: an error page (formatted with the `error` method) if the given extension is unknown or if the URL cannot be opened (`openURL` returns an error message), a pair (`HTML`, HTTP header) if the file has not been modified since last load as indicated by the `modify-since` instance variable of the client channel, otherwise returns the pair (file descriptor, full path name) corresponding to the URL.

process {CLIENT_ID}

process HTTPD request from client. The line contained in the input buffer of the associated client channel (`TcpHttpdCl` class) is parsed, it should have the following format:

```
REQ URL?QUERY HTTP/1.0
```

where *`REQ`* is equal to `GET` or `POST`, *`URL`* is a string containing any characters except "?" and *`QUERY`* is a string containing any character except space. *`URL`* shall have at least one character and *`QUERY`* can have zero character. The `req`, `url` and `query` instance variable of the associated client channel are set accordingly.

After that, the parsing of the input buffer goes on. The following fields and their associated values are retrieved:

```
Content-length:
Keep-Alives:
End-Data:
End-Eol:
Cookie:
If-Modified-Since:
Host:
```

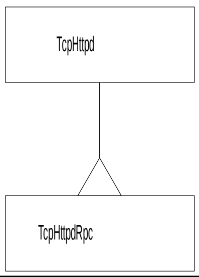
Then the data following the HTTP header in the input buffer is retrieved.

	`CLIENT_ID`: identifier of the associated client, no default value. return values: an error page displaying “Invalid HTTPD-1.0 header” if an error occurs during the parsing of the input buffer of the associated client channel, otherwise, if no error occurs, return the file descriptor of the URL (using the `mime` method).

TcpHttpdRpc

package: `tcp.rpc`

This class provides a server which waits for commands on a TCP socket. Commands are one-line lists and are executed in a private slave interpreter attached to each client.



instance variables:

timeout	time of no-activity to wait before killing a slave (time is given in seconds and `0` means infinite time). initial value is `900`.
jTclPath	contains the file path to the `package.jTcl` and `slave.jTcl` files, set by the constructor.

constructor:

TcpHttpdRpc {PORT ROOT_DIR {AUTO {}} {INDEX {}}}

sets the `jTclPath` and `index` instance variables using the values given by `AUTO` and `INDEX` and then calls its superclass constructor (`TcpHttpd`) passing `PORT` and `ROOT_DIR` as arguments.

`PORT`: port number on which the server will be listening to client connections, no default value.

`ROOT_DIR`: root directory of the HTTP file tree (only files and directories found below this level will be available to the clients), shall be a valid directory, no default value.

`AUTO`: contains a list of file path where the `slave.jTcl` file may be found, default value is `{}`.

`INDEX`: alternative name for the index file, if `{}` is given as a value, then the value from the `index` instance variable will be used, default value is `{}`.

public methods:

logout {INTERP} frees all client session resources. This method is usually called after the timeout period has expired.

`INTERP`: handle of the slave interpreter the client agent is running in, no default value.

return value: none.

exec {CLIENT_ID} executes a remote command. The command is given by the `data` instance variable of the client channel.

`CLIENT_ID`: object identifier of a client channel, no default value.

return value: return value of the executed command.

eval {CLIENT_ID} processes an RPC request. If the slave interpreter associated to the client does not exist, then try to retrieve it from the `interp` field of the cookie associated with the client. If the associated interpreter cannot be retrieved, then an error page is sent. After that, the timeout is reset and the `exec` method is called.

`CLIENT_ID`: object identifier of a client channel, no default value.

return value: error page (with error 400) if something goes wrong.

parse {CLIENT_ID TYPE}
Parse a file in order to add a slave interpreter cookie.

`CLIENT_ID`: object identifier of a client channel, no default value.

`TYPE`: type of the client, can be one of the following values: `TCLET`, `JAVA` or `URL`, no default value.

return value: error page (with error 400) if something goes wrong, otherwise `PAGE`.

alias {CLIENT_ID} this procedure does nothing. It is automatically called when a new client logs in. It is useful to add new authorized commands in the slave interpreter.

`CLIENT_ID`: not used, no default value.

return value: none.

login {CLIENT_ID} if the client is already logged, an error page is sent back, otherwise a Tcl slave interpreter is created and associated to the client (for the occasion, a `interp` instance variable is created in the client object)

`CLIENT_ID`: object identifier of a client channel, no default value.

return value: error page (with error 400) if the client is already logged.

mime {EXT CLIENT_ID}
performs a given action according to the value of the `EXT` parameter.

`EXT` value:	action performed:
`.login`	call the `login` method
`.logout`	call the `logout` method
`.rpc`	call the `eval` method
`.htt`	call the `parse` method with `TYPE` = `TCLET`
`.htj`	call the `parse` method with `TYPE` = `JAVA`
`.htcl`	call the `parse` method with `TYPE` = `URL`

`EXT`: indicates which action to perform, possible values are: `.login`, `.logout`, `.rpc`, `.htt`, `.htj`, `.htcl`, no default value.

	`CLIENT_ID`: object identifier of a client channel, no default value. return value: return value of the action performed.

3.3. Examples

Examples will be provided at a later stage.

4. The `ic` set of packages

4.1. Overview

These packages provide means to create rule-driven graphical user interfaces. The following classes are defined: `IcClient`, `IcRpcClient`, `IcError`, `jTkContainer`, `IcRuleClass`, `IcRuleEngine`, `IcPopup`, `IcHelp`, `IcDispatch`, `IcBinder`, `jTkIcWidget`, `jTkIcButton`, `jTkIcEntry`. Relations between these classes are described in the following diagram.

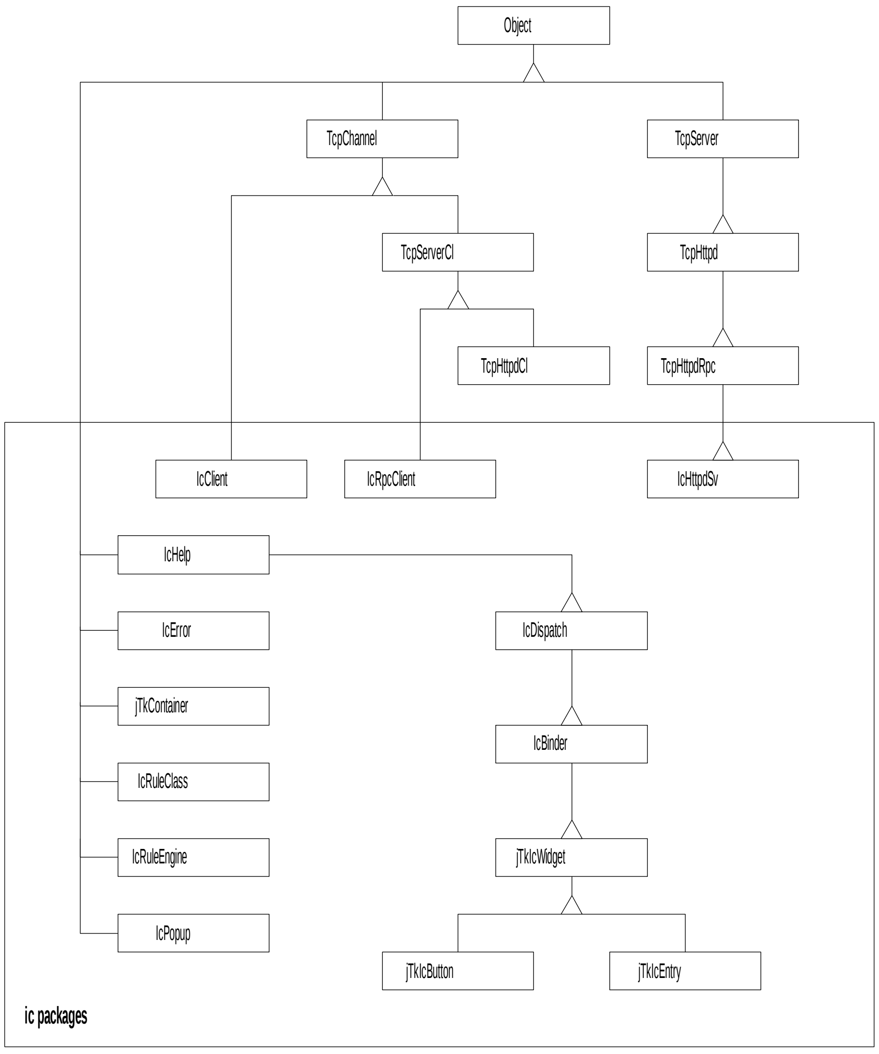


Figure 4-1: OMT diagram of the `ic` classes

package name	associated file	imports	content
ic.client	Core/Ic/Tcl/clientIc.jTcl	tcp.channel	IcClient class
ic.rule	Core/Ic/Tcl/ruleIc.jTcl	lang.object	IcRuleClass and IcRuleEngine classes
ic.server	Core/Ic/Tcl/serverIc.jTcl	tcp.rpc, ic.rule	IcRpcClient and IcRpcServer classes
ic.httpd	Core/Ic/Tcl/httpdIc.jTcl	tcp.rpc, ic.server	icHttpdSv class
ic.help	Core/Ic/Tcl/helpIc.jTcl	lang.object	IcHelp class
ic.popup	Core/Ic/Tcl/popupIc.jTcl	lang.object	IcPopup class
ic.error	Core/Ic/Tcl/errorIc.jTcl	ic.popup	IcError class
ic.dispatch	Core/Ic/Tcl/dispatchIc.jTcl	ic.help, ic.popup, ic.error	IcDispatch class
ic.binder	Core/Ic/Tcl/binderIc.jTcl	ic.dispatch	IcBinder class
ic.jTk	Core/Ic/Tcl/jTkIc.jTcl	ic.binder, ic.client	jTkIcContainer, jTkIcWidget, jTkIcEntry and jTkIcButton classes

Table 4-2: contents of the `ic.*` packages.

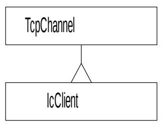
4.2. Reference

This chapter is intended to be used as a reference guide for all classes defined in the `ic` packages.

IcClient

package: `ic.client`

base class for a client channel using the IC rules.



instance variables:

errorId	object of `IcError` class, set by the constructor.
host	name of the remote host, set by the constructor.
status	status of the client channel, set by the `read` method.
_RL_*	contains a list of widget descriptors which are connected with a given rule, the name of the rule is given in the name of the variable (instead of the "`*`" character), set by the `register` method.

constructor:

IcClient {ERROR_ID HOST PORT}

this constructor (1) sets the `errorId` instance variable to the value provided by `ERROR_ID` and the `host` instance variable to the value provided by `HOST`, (2) creates a socket to the port `PORT` of the host `HOST` and passes it to the superclass constructor (`TcpChannel`), (3) if the supported shell (`wish` or `tclsh`) has been started with an option following the "`--`" sequence, then the string "`icClient` *`option`*" is sent to the server (via the socket), otherwise the string "`icClient NoSlaveInterp`" is sent to the server, (4) if the `LANG` environment variable is defined, then the string "`icEngine valid lang LANG`" is sent to the server.

`ERROR_ID`: object of `IcError` class, no default value.

`HOST`: name of the remote host, no default value.

`PORT`: port number on the remote host, no default value.

destructor:

free {} displays an error message using the error object to indicate that the connection with the server is lost.

return value: none.

public methods:

read {} this procedure overrides the `read` method defined in the `TcpChannel` class, it is automatically called whenever there are data available in the input socket. The data read shall be either the End-Of-File (EOF) character, in that case the object is destroyed, either a one item list. If something different is read, then an error is displayed using the associated error object.

The data read from the socket shall have the following format:

`{{`*`RULE`*$_1$ *`FRAME`*$_1$`…} {`*`RULE`*$_2$ *`FRAME`*$_2$`…}…}`

where *`RULE`* shall be a rule previously registered using the `register` method, otherwise an error message is displayed using the associated error object. For each rule, the `update` method of each associated widget is called with *`FRAME`* as parameter.

Each time an error is issued, the `status` instance variable is set to `FX`. Before finishing, the procedure always calls the `current` method of the associated error object.

return value: none.

focus {} dummy method, this function does nothing.

return value: none.

send {ACTION RULE args}
sends the string "`icEngine` *`ACTION RULE args`*" to the output socket.

`ACTION`: name of the action, no default value.

`RULE`: name of the rule, no default value.

`args`: additional arguments, no default value.

return value: none.

query {ACTION QUESTION}
sends the string "`icEngine` *`ACTION QUESTION`*" to the output socket using the `send` method and retrieve the answer from the input socket.

`ACTION`: name of the action, no default value.

`QUESTION`: name of the question, no default value.

return value: answer read on the input socket.

register {JTK_WD RULE}
appends the widget descriptor given by `JTK_WD` to the rule `RULE` represented by the `_RL_RULE` instance variable. If `_RL_RULE` does not exist, then it is created.

`JTK_WD`: widget descriptor, no default value.

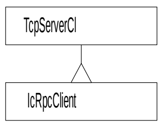
`RULE`: rule name, no default value.

return value: none.

IcRpcClient

package: `ic.server`

base class for a client channel using the IC rules.



class variables:

engineClass	class of the associated rule engine object. Initial value is `IcRuleEngine`.

instance variables:

engine	object of `IcError` class, set by the constructor.
interp	name of the remote host, set by the constructor.

constructor:

IcRpcClient {RPC_ID HOST SOCK PORT}

does nothing specific, simply call the superclass constructor (`TcpServerCl`), passing its arguments.

`RPC_ID`: object identifier (as returned by the `new` jTcl command) of the server the client is connected to, no default value.

`HOST`: name of the remote host, no default value.

`SOCK`: socket for communication between the client and the server, shall be defined using the `socket` Tcl built-in command, no default value.

`PORT`: port number on the remote host, no default value.

destructor:

free {} destroys the associated rule engine if it exists.

return value: none.

public methods:

read {} this procedure overrides the `read` method defined in the `TcpChannel` class, it is automatically called whenever there are data available in the input socket. The data read shall be either the End-Of-File (EOF) character, in that case the object is destroyed, either a one item list. If something different is read, then an error is displayed using the associated error object.

The data read from the socket shall have the following format:

{{$RULE_1$ $FRAME_1$…} {$RULE_2$ $FRAME_2$…}…}

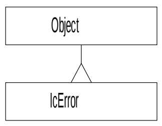
where *RULE* shall be a rule previously registered using the `register`

	method, otherwise an error message is displayed using the associated error object. For each rule, the `update` method of each associated widget is called with *`FRAME`* as parameter. Each time an error is issued, the `status` instance variable is set to `FX`. Before finishing, the procedure always calls the `current` method of the associated error object. return value: none.
auth {}	dummy method, this function does nothing. return value: none.

IcError

package: `ic.error`

Manages an error window which is used to display error messages and allow to navigate through the stack of current errors.



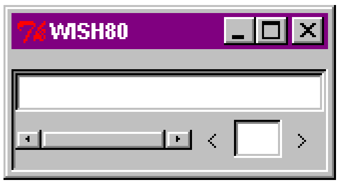
instance variables:

color-Message	color used to display an informational message in the text entry of the error window. This variable should be a list of two values, the first value is the name of the foreground color and the second value is the name of the background color. Initial value is `{black Bisque1}`.
color-Empty	color used when there is no message to display in the text entry of the error window. This variable should be a list of two values, the first value is the name of the foreground color and the second value is the name of the background color. Initial value is `{grey grey}`.
color-Error	color used to display an error message in the text entry of the error window. This variable should be a list of two values, the first value is the name of the foreground color and the second value is the name of the background color. Initial value is `{red Bisque1}`.
count	number of errors currently pending. Initial value is `0`.
num	number of the error currently displayed in the error window. Initial value is `1`.
wnum	widget corresponding to the entry field for the error number in the error window, set by the constructor.
last	last associated `IcDispatch` object referred to, i.e. the one which error message is currently displayed in the text entry of the error window.
tiped	boolean value to indicate whether or not we are in “tipped” mode, set to `0` by the constructor.
widget	widget corresponding to the entry field for the error message in the error window, set by the constructor.
owner	handle of the object which will make use of this class, set by the constructor.
root	root widget as given by the owner's `root` method, set by the constructor.
_err*	represents the associated `IcDispatch` object to a given error, there are as many `_err*` variables as there are errors pending. '`*`' is the same as the value of the variable, i.e. an object handle.

constructor:

IcError {OWNER_ID}

creates a new window containing widgets to display and navigate through error messages. The created window looks like this:



`OWNER_ID`: handle of the owning object, no default value.

public methods:

clean {}

resets all the counters to zero, removes all the errors (represented by the `_err*` instance variables) and displays no message (empty string) in the text entry of the error window.

return value: none.

del {JTK_ID}

removes an existing error, identified by `JTK_ID` (the `_err`*`ID`* instance variable is deleted).

`JTK_ID`: handle of an object of class `IcDispatch` (or sub-class of `IcDispatch`), used here to identify an error, no default value.

return value: none.

add {JTK_ID}

adds a new error, identified by `JTK_ID` (the `_err`*`ID`* instance variable is created with the value `ID`).

`JTK_ID`: handle of an object of class `IcDispatch` (or sub-class of `IcDispatch`), used here to identify an error, no default value.

return value: none.

tip {JTK_ID}

displays an help message in the text entry of the error window. The message displayed is contained in the tip instance variable of the object represented by `JTK_ID`. The message is displayed until the mouse pointer leaves the `JTK_ID` associated widget.

`JTK_ID`: handle of an object of class `IcHelp` (or sub-class of `IcHelp`), no default value.

return value: none.

display {COLOR MSG}

displays the message `MSG` in the text entry field of the error window. The text entry field is in read-only mode.

`COLOR`: indicates the color used to display the message, possible values are “`Message`”, “`Empty`” or “`Error`”, additional values can be added provided that a corresponding `color-`*`xxx`* class variable is created, no default value.

`MSG`: message to display, no default value.

return value: none.

current {{THIS_INCR 999}}
this method is automatically invoked whenever the user clicks on the "<" or the ">" buttons or type a number in the number entry field and press `<Return>` in the error window.

This method does nothing if we are in "tipped" mode (i.e. the `tip` method is currently invoked) or if there is currently no pending error. Otherwise it increments the number of the currently displayed error of the value given by `THIS_INCR` and displays the corresponding error.

`THIS_INCR`: value to increment the currently displayed error counter, default value is `999`.

return value: none.

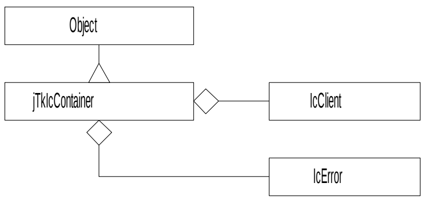
popup {} constructs a new pop-up window (with an `IcPopup` object) displaying the error messages associated with all of the pending errors.

return value: none.

jTkIcContainer

package: `ic.jTk`

Frame containing widgets.



class variables:

packOpt	default options (using standard `Tk` syntax for the `pack` command) for the layout of the container. Initial value is `{-expand y -fill x -side top}`.
frameOpt	default options (using standard `Tk` syntax for the `frame` command) for the border of the frame. Initial value is `{}`.

instance variables:

root	root frame widget, set by the constructor.
frame	frame of the container, child of `root`, set by the constructor.
errorId	associated error object (of class `IcError`), set by the constructor.
icSvId	associated client object (of class `IcClient`), set by the constructor.
wdList	list of component widgets (sub-classes of `jTkIcWidget`), set by the constructor of `jTkIcWidget`.

constructor:

jTkIcContainer {ROOT HOST PORT}

this method (1) sets the `root` instance variable to the value provided by `ROOT`, (2) create a new frame with the options given in the `frameOpt` class variable, this new frame is a sub-widget of `ROOT` and is stored in the `frame` instance variable, (3) creates a new associated error object (stored in the `errorId` instance variable) and a new associated client object (stored in the `icSvId` instance variable).

`ROOT`: root frame widget, no default value.

`HOST`: name of remote host, used for the creation of the associated client object, no default value.

`PORT`: port number on the remote host, used for the creation of the associated client object, no default value.

public methods:

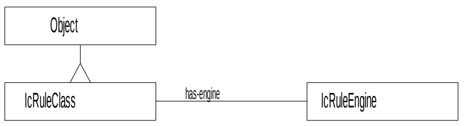
focus {JTK_ID}	gives the default focus to the widget given by `JTK_ID`.

	`JTK_ID`: object of class `jTkIc*`, no default value. return value: none.
pack {args}	uses the packing geometry manager of the Tk toolkit in order to organize the widgets given as arguments inside the frame of the container. Packing options given by the `packOpt` class variable are used. `args`: each argument shall be an object of class `jTkIc*` to be put inside the container frame, no default value. return value: none.

IcRuleClass

package: `ic.rule`

represents a rule.



class variables:

refused-us	American version of the error string to be displayed when the rule cannot be activated. Initial value is `"Corrected dependency error first"`.
refused-fr	French version of the error string to be displayed when the rule cannot be activated. Initial value is `"Corriger les erreurs sur les dependances"`.
depends	List of dependencies. Initial value is `{}`.
phony	Indicates if the rule is phony or not, a “phony” rule does not return any information to the client. Initial value is `0`.

instance variables:

data-extra	list of instance variables to be set by a given action, most of the time, its value is `“value status”`. Initial value is `{}`.
data-temp	contains a reference to a message to display. Initial value is `{}`.
status	status of the rule, possible values are: “`OK`” (no error), “`FX`” (error occurred), “`LCK`” (rule is locked), “`UNK`” (rule is unlocked) or `Unused`. Initial value is `Unused`.
value	value of the rule. Initial value is `{}`.
previous	previous value of the `value` instance variable. Initial value is `{}`.
param	value of the parameter passed to the `valid` method. Initial value is `{}`.
tic	contains CPU time as return by the `clock clicks` Tcl built-in command. Initial value is `{-1}`.
depends	list of dependencies. Initial value is `{}`.
engine	associated engine, shall be an instance of the `IcRuleClass` class, set by the constructor.
oldStatus	saved value of the `status` instance variable, set by the `lock` method.
oldValue	saved value of the `value` instance variable, set by the `lock` method.
locked	indicates if the rule is locked or not. If the rule is locked, then contains a message indicate the reason of the locking of the rule, otherwise, if the rule

is not locked, this variable does not exist. Set by the `lock` method.

constructor:

lcRuleClass {ENG_ID}
sets the `engine` instance variable to the value given by `ENG_ID` and then searches in the dependency list (given by the `depends` class variable). Each rule found in that list is created and added to the `depends` instance variable.

`ENG_ID`: object representing the associated engine, no default value.

public methods:

valid {PARAM args} sets the `data-extra` instance variable to "`value status`", the `value` and `param` instance variables to `PARAM`, the `status` instance variable to "`OK`" and the `tip` instance variable to the current CPU time.

This is the default validation method, it does nothing special (just report that validation is always OK). For an actual use, the class should be derived and this method overloaded to take into account application-specific code.

`PARAM`: value for the `param` instance variable, no default value.

`args`: unused parameter.

return value: none.

message {TYPE} displays a message corresponding to a type indicated by `TYPE`, the language of the message is indicated by the `lang` instance variable of the associated engine object. If the message in the required language cannot be retrieved, then the message in American is displayed as a default. The message is not actually displayed, but stored in the `data-temp` instance variable instead.

`TYPE`: type of message to display, no default value.

return value: none.

list {args} displays a message of type "`list`" using the `message` method.

`args`: unused parameter.

return value: none.

tip {args} displays a message of type "`tip`" using the `message` method.

`args`: unused parameter.

return value: none.

default {args} displays a message of type "`default`" using the `message` method and sets the `data-extra` instance variable to "`value status`".

	`args`: unused parameter.
	return value: none.
help {args}	displays a message of type “`help`” using the `message` method.
	`args`: unused parameter.
	return value: none.
refused {DEP_LST}	displays a message of type “`refused`” using the `message` method and sets the `data-extra` instance variable to “`status`”.
	`DEP_LST`: unused parameter.
	return value: none.
decorate {args}	this method is typically called once when displaying the widget in order to decorate it with all the strings from the rules.
	`args`: arguments passed to the default method, no default value.
	return value: none.
undo {args}	reset the `value` instance variable with the value contained by the `previous` instance variable.
	`args`: unused parameter.
	return value: none.
unlock {args}	Unlocks the rule and restores the previous values of the `value` and `status` instance variables (or set status to “`UNK`” if no previous status was set).
	`args`: unused parameter.
	return value: none.
lock {PARAM args}	locks the rule (set the `status` instance variable to “`LCK`”) in order to avoid any changes of the value during validation process. Previous values of the `value` and `status` instance variables are saved in the `oldValue` and `oldStatus` instance variables.
	`PARAM`: a message/value pair, the message part explains the reason to lock the rule and the value part is the value submitted to validation, no default value.
	`args`: unused parameter.
	return value: none.
query {METHOD PARAM}	
	performs an action (given by `METHOD`) and check the dependency list before.

`METHOD`: action to perform, possible values are: `decorate`, `default`, `valid`, `list`, `undo`, `lock`, `unlock`, `tip` and `help`, no default value.

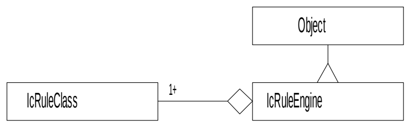
`PARAM`: parameters for the action, no default value.

return value: none.

IcRuleEngine

package: `ic.rule`

Generic rule engine, stores all rule objects.



instance variables:

lang	language used to display the messages, possible values are `fr` for French and `us` for American. To add more languages, new class variables must be added to the `IcRuleClass` class, the name of the class variables shall be under the form *`messagetype-langcode`*. Initial value is `{us}`.
answer	return value for the `doIt` method, set by the called rules.
owner	handler of the class owning the rule engine, set by the constructor.
topRule	name of the top rule, set by the constructor.
_rl_*	set of rules objects (of type `IcRuleClass`), the star (`*`) represents the name of the rule, set by the constructor (for the top rule) and the `getHandle` method (for the other rules).

constructor:

IcRuleEngine {OWNER {RL_TOP all}}

sets the `owner` and `topRule` instance variables using values provided by `OWNER` and `RL_TOP` and creates a new rule: the top rule.

`OWNER`: handle of the owner, no default value.

`RL_TOP`: name of the top rule, default value is `all`.

destructor:

free {} destroys all the rules (referred to by the `_rl_*` instance variables).

return value: none.

public methods:

value {RL_NAME} returns a list consisting of the name of the rule (given by `RL_NAME`) and its value (given by the `value` instance variable of the associated rule object).

`RL_NAME`: shall be a rule object previously defined, no default value.

return value: the pair name/value as described above.

lock {RL_NAMES MSG {VALUE {}}}

calls the `lock` method with `MSG` and `VALUE` as arguments for all rules found in the `RL_NAMES` list.

RL_NAMES: shall be a list of valid rules, no default value.

MSG: message giving the reason of the locking of the rule, no default value.

VALUE: value submitted for validation, default value is {}.

return value: none.

unlock {RL_NAMES} calls the lock method for all rules found in the RL_NAMES list.

RL_NAMES: shall be a list of valid rules, no default value.

return value: none.

rlDump {RL_NAME} displays the content of the rules (calling their dump method) matched by the glob-style pattern given by RL_NAME.

RL_NAME: glob-style pattern matching one or more rule names associated with the engine, no default value.

return value: “ERROR” if the pattern matches no rule, otherwise none.

getHandle {RL_NAME}
returns the object associated with the rule name given by RL_NAME, if there is no rule with that name, then it is created.

RL_NAME: name of a rule, no default value.

return value: rule object corresponding to RL_NAME.

doIt {METHOD QUERY}
calls the query method of all the rules

METHOD: name of the method to call, as accepted by the query method of the rule class, no default value.

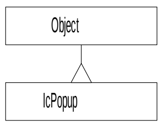
QUERY: list of values, the first value shall be the name of the rule to call, and the remaining values are the parameters to pass to the called rule, no default value.

return value: answer instance variable.

IcPopup

package: `ic.popup`

Pop-up window object which displays a list box in which the user can select a particular string.



instance variables:

popup	pop-up top-level widget, set by the constructor.
w	widget associated with the owner object, set by the constructor.

constructor:

IcPopup {OWNER T}

this method (1) sets the `w` instance variable to the value of the associated widget of `OWNER`, (2) creates a new frame (which is stored in the `popup` instance variable), a list box and a scroll bar are inserted in this frame, (3) the strings given in the `T` list are inserted in the list box.

`OWNER`: owner object, the object shall belong to a class providing the `widget` method which returns a widget handle, no default value.

`T`: list of strings to be inserted in the list box of the pop-up window, no default value.

destructor:

free {} destroys the `popup` widget.

return value: none.

public methods:

select {OWNER VALUE}

this method is automatically called when the user select (with a click of the mouse or by pressing the `<Return>` key) a particular string in the list box of the pop-up widget. The `select` method of the `OWNER` object is called, passing `VALUE` as a parameter and the object is destroyed.

`OWNER`: owner object, the object shall belong to a class providing the `select` method, no default value.

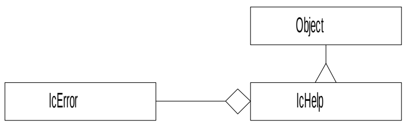
`VALUE`: currently selected string in the list box of the pop-up widget, no default value.

return value: none.

IcHelp

package: `ic.help`

provides the basic mechanism to display a pop-up help window.



instance variables:

errorId	associated error object (of class `IcError`), set by the constructor of `jTkIcWidget`.
widget	associated widget descriptor.
tip	message to be displayed in the help pop-up window.

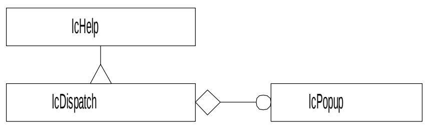
public methods:

message {}	calls the `tip` method of the associated error object. return value: none.
popup {}	displays the `tip` help message for the widget `widget` in a pop-up window. The help pop-up window is automatically destroyed when the mouse pointer will leave the parent widget. return value: none.

IcDispatch

package: `ic.dispatch`

dispatches the bound actions of the user to the rule engines.



instance variables:

status	current status. Initial value is “`Unknow`” [sic].
value	current value. Initial value is “`Unknow`” [sic].
error	error message, set by the `ERR` method.

public methods:

focus {}
gives the default focus to the associated widget (given by the `widget` instance variable).

return value: none.

TIP {MSG}
sets the `tip` instance variable to the value provided by `MSG` and then calls the `popup` method of the superclass in order to display `MSG` in a pop-up window.

`MSG`: help message to be displayed in a pop-up window, no default value.

return value: none.

ERR {MSG}
sets the `error` instance variable to the value provided by `MSG` and then calls the `add` method of the associated error object in order to add the error message to the list of the currently pending errors displayed by the `IcError` object.

`MSG`: error message to be displayed in the `IcError` object, no default value.

return value: none.

LST {LIST}
creates an `IcPopup` object to display the list of strings given by `LIST` in a list box. The user will be able to select a particular string.

`LIST`: list of message strings to be displayed in the `IcPopup` object, no default value.

return value: none.

MSG {MSG}
displays the message given by MSG (with the default colors for a message, as opposed to an error) in the text entry field of the associated error object.

`MSG`: message to display, no default value.

	return value: none.
STS {STATUS}	`STATUS` is stored in the `status` instance variable. If `STATUS` values "`OK`" then the error is removed from the associated error object. The status is then used to paint the widget. `STATUS`: value of the status, no default value. return value: none.
VAL {DATA}	displays the text provided by `DATA` in the associated widget. This method is equivalent to the `LBL` method. `DATA`: message to be displayed inside the associated widget, no default value. return value: none.
LBL {DATA}	displays the text provided by `DATA` in the associated widget. This method is equivalent to the `VAL` method. `DATA`: message to be displayed inside the associated widget, no default value. return value: none.
update {FRAME}	for each pair `{ACTION DATA}` given in the `FRAME` list, performs the action indicated by `ACTION`, providing `DATA` as a parameter to the invoked method. Possible methods called are given below:

value of `ACTION`	method called
"`status`"	`STS`
"`list`"	`LST`
"`message`"	`MSG`
"`libel`"	`LBL`
"`refused`"	no method is called
"`error`"	`ERR`
"`tip`"	`TIP`
"`value`"	`VAL`

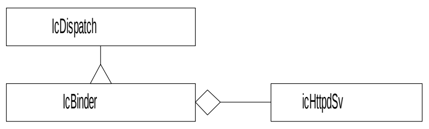
`FRAME`: list of pairs action/data, no default value

return value: none.

IcBinder

package: `ic.binder`

Default binding for actions on widgets.



class variables:

delayList	delay (in milliseconds) to wait before displaying query list window. Initial value is `300`.
delayHelp	delay (in milliseconds) to wait before displaying `queryTip`. Initial value is `3000`.

instance variables:

rule	rule name.
icSvId	associated server object (of class `icHttpSv`), set by the constructor of `jTkIcWidget`.
queryTip	query tip, as returned by the associated server, set by the `queryTip` method.
changed	indicates that data is been changed in the widget, set by the `keyIn` method.

public methods:

queryValid {}

this method is called each time we want to make sure the data in the widget is valid.

return value: none.

queryList {}

displays a query list after a small delay, the delay is specified by the `delayList` instance variable. The waiting process can be interrupted at any time if the user presses a key, otherwise it sends a list command to the associated server.

return value: none.

select {DATA}

this method is automatically called from a list pop-up window, when a selection is made by the user. The selection is stored in `DATA`. This method displays the selected string on the widget (by a call to the `VAL` method) and test if the data is valid (by a call to the `queryValid` method).

`DATA`: string selected by the user in the list box, no default value.

return value: none.

queryTip {}

if the `queryTip` instance variable is set then it is displayed, otherwise its value is retrieved by sending the tip command to the associated server. For each of these operations, we are waiting a small amount of time (given by the `delayHelp` instance variable).

return value: none.

queryUndo {} sends an undo command to the associated server.

return value: none.

keyIn {} this method is called each time a key is pressed in order to enter data when the widget has the focus. The `changed` instance variable is set to `1` to indicate that data is changed and the widget is repainted.

return value: none.

focusIn {} this method is called each time the focus is given to the widget, it binds user events to internal methods. Possible methods called are given below:

event bound	method called
`<Key>`	`keyIn`
`<FocusOut>`	`queryValid`

return value: none.

bind {} binds user events to internal methods. Possible methods called are given below:

event bound	method called
`<Double-1>`	`queryValid`
`<Button-2>`	`queryList`
`<FocusIn>`	`focusIn`
`<Button-1>`	`queryList`
`<Key-Escape>`	`queryUndo`
`<Enter>`	`queryTip`
`<Leave>`	cancels the execution of the watch-dog
`<Control-n>`	moves the focus to the next widget
`<Control-p>`	moves the focus to the previous widget
`<return>`	`queryValid` and focus to the next widget

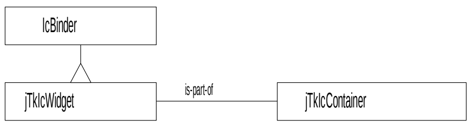
After that, the rule given by the `rule` instance variable is registered to the associated client (given by the `icSvId` instance variable).

return value: none.

jTkIcWidget

package: `ic.jTk`

Generic widget class.



class variables:

frameOpt	default options (using standard `Tk` syntax for the `frame` command) for the border of the frame. Initial value is `{}`.
color-LCK	default color for “`LCK`” mode. Initial value is `grey`.
color-FX	default color for “`FX`” mode. Initial value is `red`.
color-OK	default color for “`OK`” mode. Initial value is `blue`.
color-UNK	default color for “`UNK`” mode. Initial value is `black`.

instance variables:

frame	top-level frame of the widget, set by the constructor.
ownerId	owner object, shall be of type `jTkIcContainer`, set by the constructor.
wdPrevious	previous widget object (as given by the list of widget in the associated `jTkIcContainer` object), set by the constructor.
wdNext	next widget object (as given by the list of widget in the associated `jTkIcContainer` object), set by the constructor.

constructor:

jTkIcWidget {OWNER_ID}

This method:

1. sets the current status (`status` instance variable) to `UNK`,

2. sets the `ownerId` instance variable to `OWNER_ID`,

3. sets the `icSvId` and `errorId` instance variables to the values of the associated server and error objects of `OWNER_ID`,

4. creates a new frame which is a sub frame of the root frame of `OWNER_ID` (the frame is created using the default options given by the `frameOpt` class variable and is set in the `frame` instance variable),

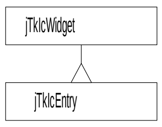
5. sets the `wdPrevious` and `wdNext` instance variables from data given by the list of component widgets of the owner and update this list to add the current widget.

	`OWNER_ID`: object of class `jTkIcContainer`, no default value.
public methods:	
focus {}	gives the focus to the widget. return value: none.
paint {COLOR}	set the foreground color of the widget to the color specified by the color code given by `COLOR`. The color code indicates a color referenced by one of the following class variables: `color-LCK`, `color-FX`, `color-OK` or `color-UNK`. `COLOR`: shall be a known color code, possible values are: `LCK`, `FX`, `OK` or `UNK`, default value is `UNK`. return value: none.
queryValid {}	calls the `queryValid` method of the superclass. return value: none.

jTkIcEntry

package: `ic.jTk`

Text entry widget.



class variables:

entryOpt	default options for the text entry field. Initial value is `{}`.
labelOpt	default options for the associated label. Initial value is `{-anchor w -width 25}`.
packOpt	default options for the packing of the text entry field and its associated label in the widget. Initial value is `{-side left -expand y -fill x -pady 2m}`.

instance variables:

changed	boolean value to indicate if data in the text entry field has been changed, set by the constructor.
label	associated label widget, set by the constructor.

constructor:

jTkIcEntry {OWNER_ID RULE}

this method (1) sets the `changed` and `rule` instance variables to (resp.) `0` and `RULE`, (2) creates a new frame widget containing a text entry field and an associated label.

`OWNER_ID`: object of class `jTkIcContainer`, no default value.

`RULE`: rule name, no default value.

public methods:

queryValid {}

if data has changed (the `changed` instance variable is set to `1`), calls the `queryValid` method of the superclass and reset the `changed` instance variable to `0`.

return value: none.

LBL {DATA}

set `DATA` as the text of the associated label.

`DATA`: string to write on the label, no default value.

return value: none.

VAL {DATA}

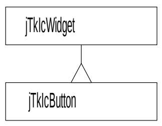
set `DATA` as the text of the entry field.

	`DATA`: string to write in the entry field, no default value. return value: none.

jTkIcButton

package: `ic.jTk`

Button widget.



class variables:

packOpt	default options for the packing of the button in the widget. Initial value is `{-side right -pady 2m}`.

constructor:

jTkIcButton {OWNER_ID RULE}

creates a new frame with a button inside it, `RULE` is used to initialize the `rule` instance variable and is displayed on the button.

`OWNER_ID`: object of class `jTkIcContainer`, no default value.

`RULE`: rule name, no default value.

public methods:

focusIn {}	this method does nothing. return value: none.
TIP {MSG}	sets the TIP instance variable to MSG and call the message method. `MSG`: message to display in ‘tipped’ mode, no default value. return value: none.
VAL {MSG}	this method does nothing. `MSG`: unused parameter. return value: none.