Toolkit for Conceptual Modeling (TCM) Design and Implementation

for version 2.01

Frank Dehne
Division of Mathematics and Computer Science
Faculty of Sciences, Vrije Universiteit
De Boelelaan 1081a, 1081 HV Amsterdam
The Netherlands
frank@cs.vu.nl

Henk R. van de Zandschulp
Department of Computer Science
University of Twente
P.O. Box 217, 7500 AE Enschede
The Netherlands
henkz@cs.utwente.nl

February 12, 2001

Contents

Chapter 1

Introduction

This report describes the design and implementation of the Toolkit for Conceptual Modeling. This is a working document, supplied together with the source code. It is not intended for separate publication.

TCM consists of a number of X/Motif-based diagram and table drawing tools. This document tries to give insight into *how* TCM is accomplished from the designer/programmer's point of view. The user guide and reference manual [2] describes *what* TCM does from the user's point of view.

TCM is implemented in C++ [13] and it uses the standard X Windows libraries Xlib [3] and Xt (X toolkit intrinsics) [7] and it uses the OSF/Motif widget set [12]. TCM can be used under any X11 window manager. TCM has been ported by us to SunOS 4.1.x, Sun Solaris 2.x, Linux 2.x, IRIX 6.x, AIX 4.x, HP-UX 10.x and OSF/1, But TCM is portable to any Unix system that has a C++ compiler and has the development libraries for X Windows and Motif (or LessTif, the free Motif clone). TCM uses the Motif widget library together with the Xlib and Xt libraries for its graphical user interface. These libraries have a C API. Books on Motif programming which have had influence on TCM are [1, 4, 15, 14]. Books on Xlib and Xt programming which are used while writing TCM are [8, 9, 10, 11].

This document is a stepping stone for a designer/programmer who has access to the TCM source code. It expects that you have knowledge about C++, X/Motif and Unix. This document tries to offer:

- An aid for compiling and porting TCM.
- An overview of the system architecture and the source code organization. This should make it easier to understand the software and to make it easier to write additions and modifications.

This document is kept rather short intentionally, because it is not possible to anticipate on every possible question or problem. Furthermore, TCM is rapidly evolving so that many things would be quickly outdated and each new release would be slowed down by a documentation update. When you have any questions or comments about this document you are advised to e-mail them to `tcm@cs.utwente.nl`

Chapter 2

System Architecture

In this chapter a high level overview of the TCM software system is presented. In figure 2.1 TCM is logically decomposed into a number of subsystems. Each subsystem is then worked out further as an UML static structure diagram (SSD) in subsequent figures. All classes in these SSDs also exist in the source code, but not every source code class is mentioned in these SSDs. The classes that are not mentioned are either implementation classes (data structures, user interface widgets etc.) or they are specializations of the classes that are mentioned for specific editors. The same applies to the relationships between the classes. Note also that we do not show the attributes and operations of the classes because that would clutter up the diagrams too much. But a complete and accurate overview of all classes of the source code and their attributes, operations and specialization relationships has been generated automatically and is described in chapter 5.

This system architecture is rather conceptual, it does not say exactly how the source code is organized physically. In chapter 3 the physical source code structure is presented.

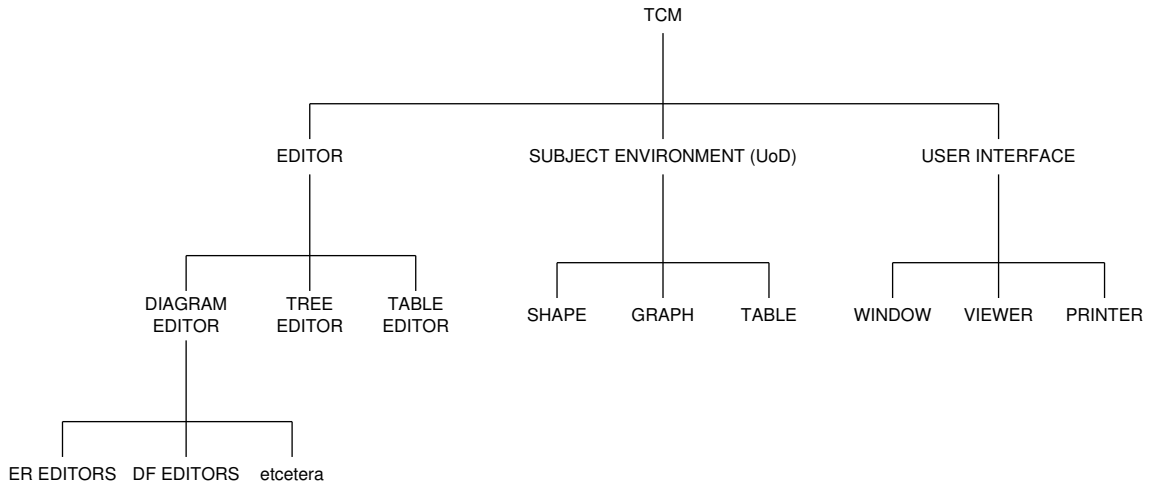

Figure 2.1: Subsystems.

TCM can be subdivided into different orthogonal ways.
The most practical way in C++ is to have classes, grouped into libraries and/or directories. Another way, which will be shown in next SSDs is using subject areas. Each subject area is a subsystem.

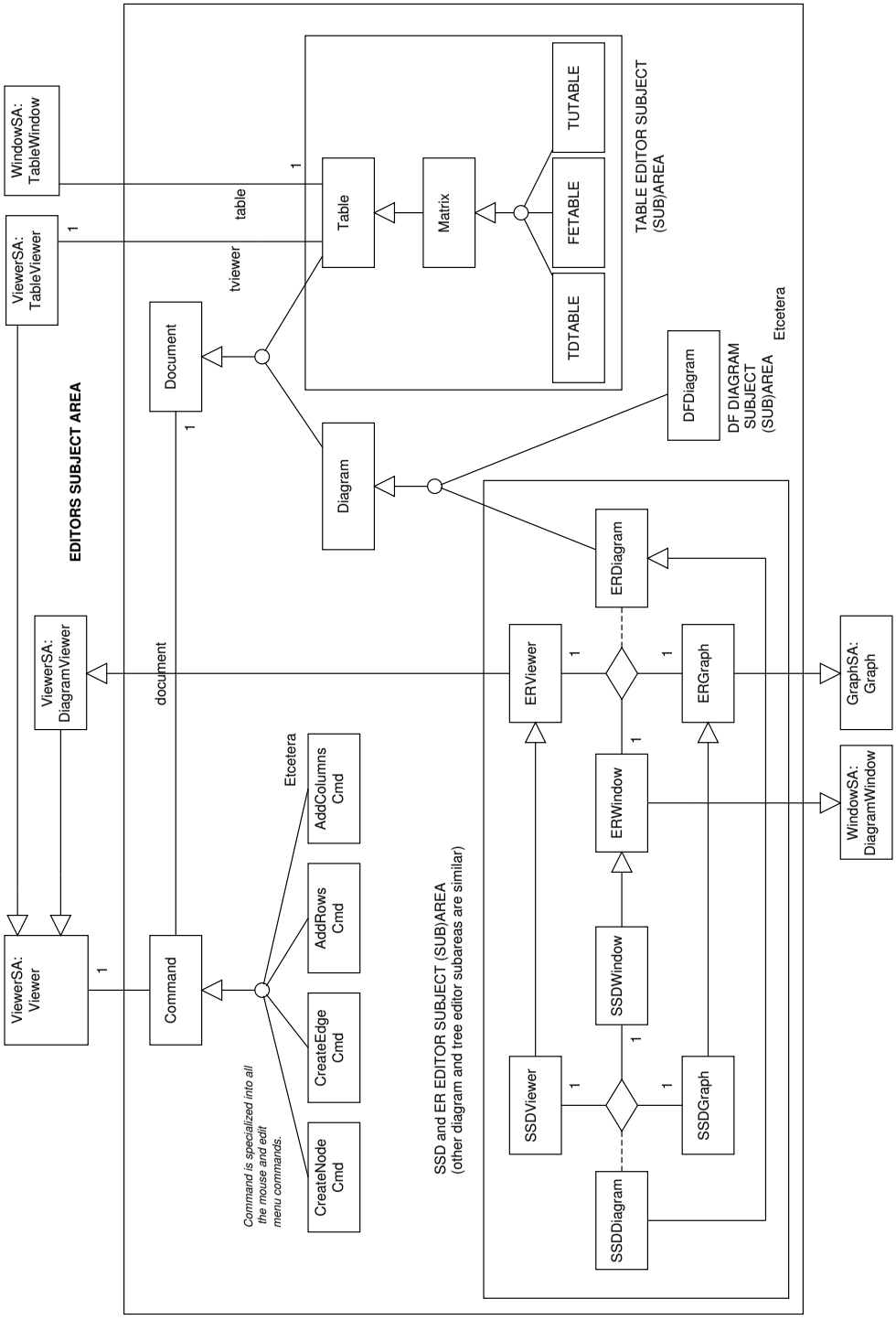

Figure 2.2: Editor subject areas.

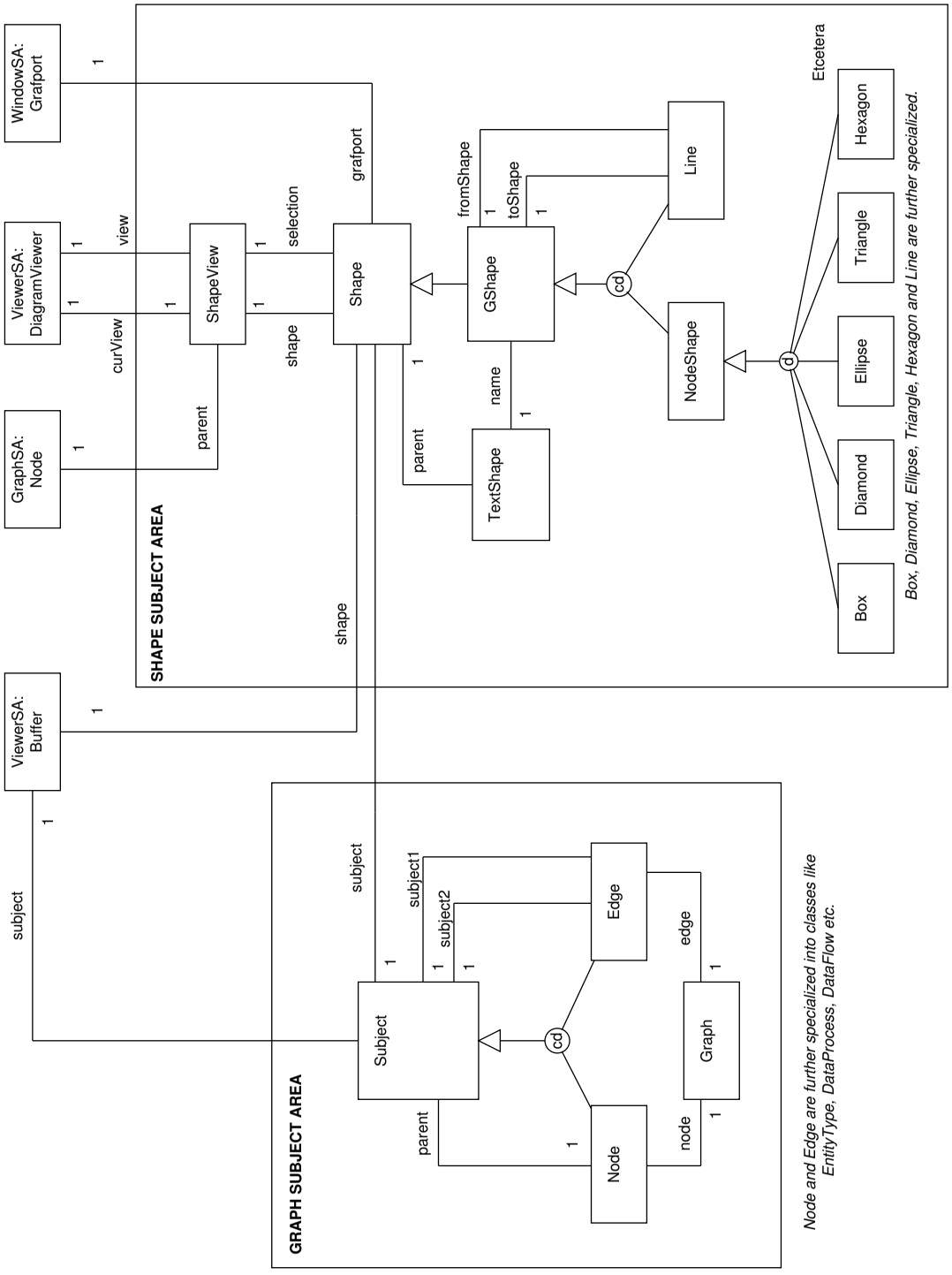

Figure 2.3: Shapes and Graph subject areas.

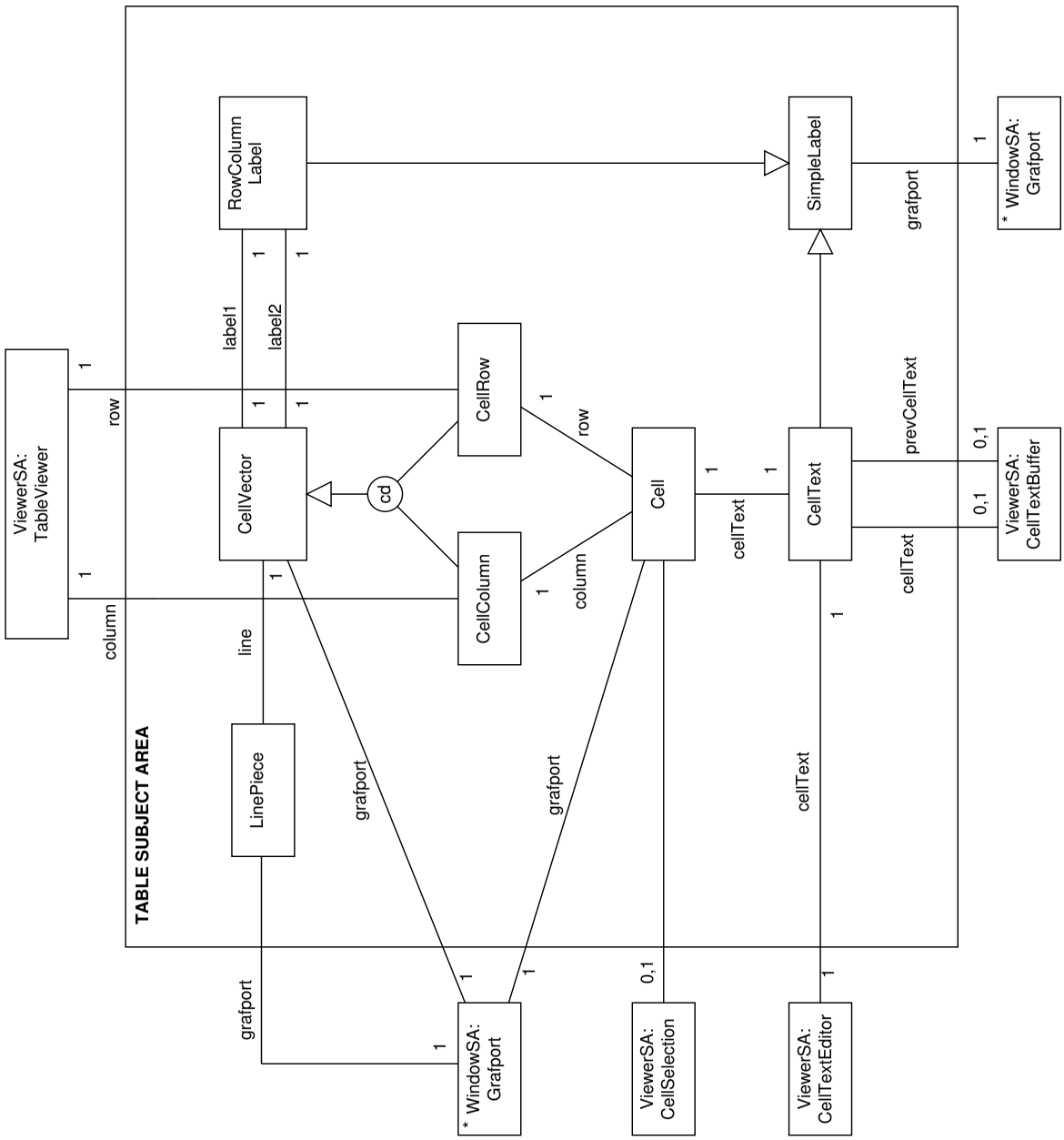

Figure 2.4: Table subject area.

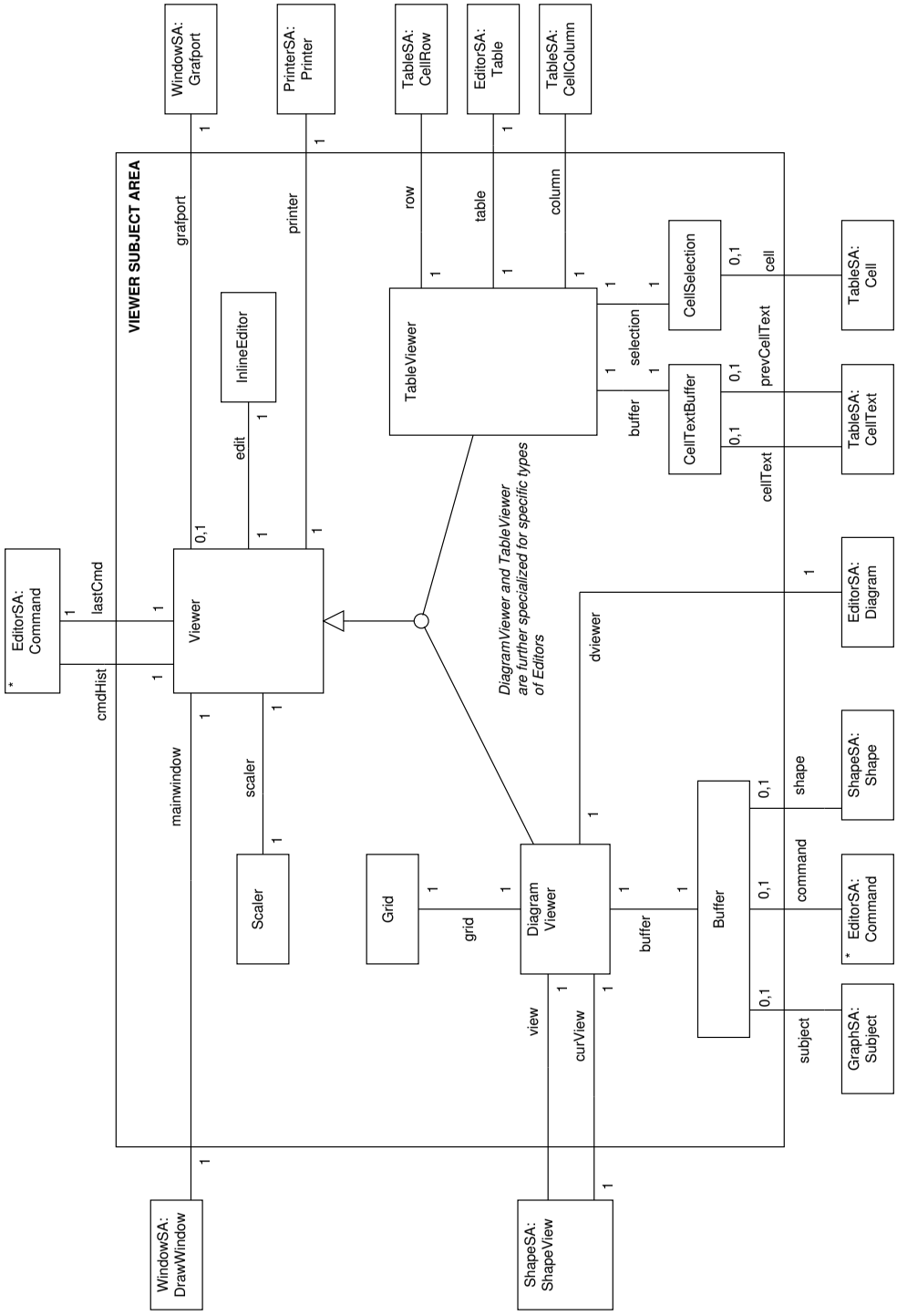

Figure 2.5: Viewer subject area.

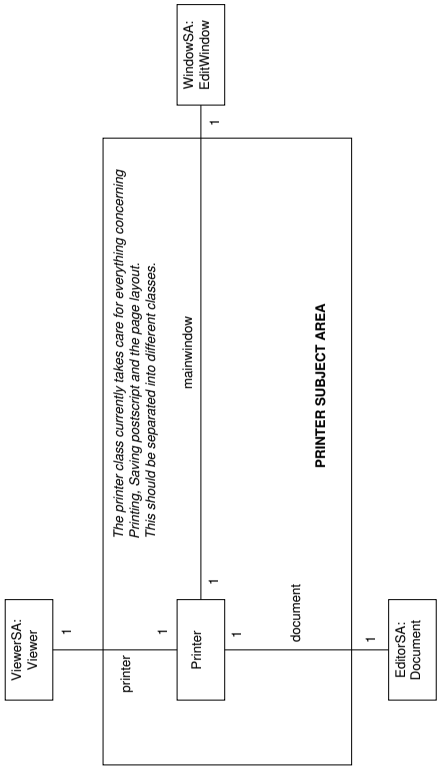

Figure 2.6: Printer subject area.

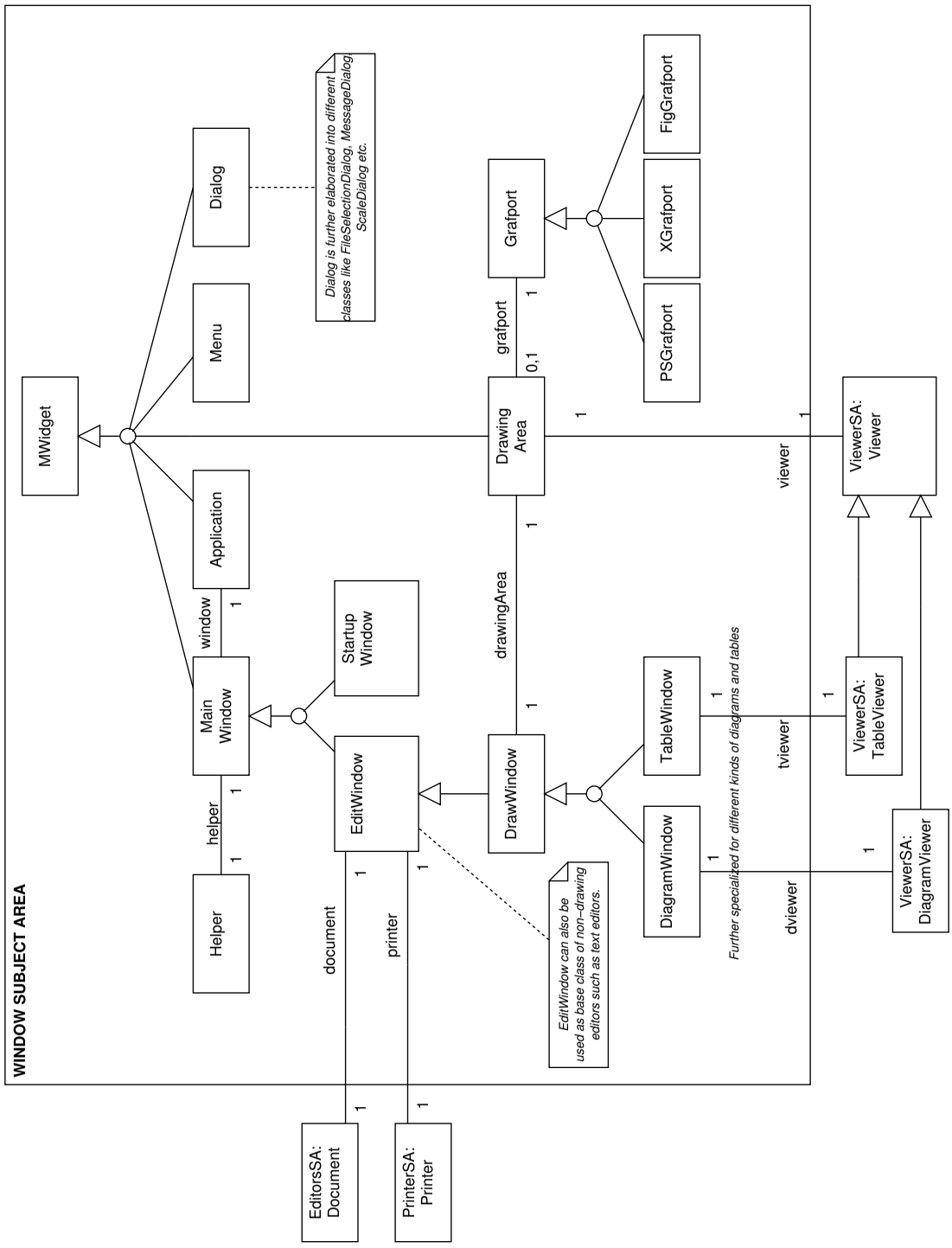

Figure 2.7: Window subject area.

Chapter 3

Source Code Organization

3.1 Source code versus design criteria

In this chapter we explain how the source code of TCM is organized. TCM basically consists of about two dozen graphical editors and a startup program. The source code can be found in the directory `$TCM_HOME/src`. The TCM source code is physically split over several subdirectories. The sources can be compiled into a number of object libraries and executables. The source code is split in order to factor out commonalities between the programs. See figure 3.1 for a Venn-diagram like overview of the commonalities of the source code. The entire TCM source is called *Global*, the sources of the programs that have an X/Motif GUI, are called *GUI*. The sources that are used in all editors, are called *Editor*. The sources that are used in all diagram editors, are called *Diagram*. The sources that are used in all table editors are called *Table*. The sources that are used in the editor TERD are called *TERD* etc. For instance, the editor TSSD uses the sources of the TSSD area, the TERD area, Diagram, Editor and Global, but it does not use Table or TCRD. See figure 3.2 for an overview of the current TCM development directory structure. The sources are physically split over several subdirectories of the src directory. This is done according to the following criteria:

- All code that is global and that is not part of the other areas is collected in the directory `src/gl`. This code is compiled into a library called `libglobal`. This includes common classes for lists, strings, Unix utilities etc. that can be used in any program, not necessarily a TCM tool.
- All code that comprises the graphical user interface but which is not specific for any TCM program is collected in the directory `src/ui`. This code is compiled into a library called `libgui`. This includes a generic application framework for Motif based C++ programs, classes for drawing lines and other shapes under X, classes for building all kinds of pop-up dialog windows, classes for making various pop-up and pull-down menus etc.
- All code that lies in the editor area (and in the TCM area) but not in one of the subareas, is collected in the directory `src/ed`. This code is compiled into a library called `libeditor` and in the executable `tcm`, the startup program. The libeditor library includes things that are applicable to all TCM editors such as saving and loading documents (the generic part of it), printing documents, the on-line help and it contains a number of abstract classes like Document, Viewer and Command on which specific editor classes are based.
- All code that lies in the diagram area and that is not part of one of the diagram subareas, is collected in directory `src/dg`. This code is compiled into a library called `libdiagram`. This includes classes for (abstract) graphs (and classes for nodes and edges), classes for graphical

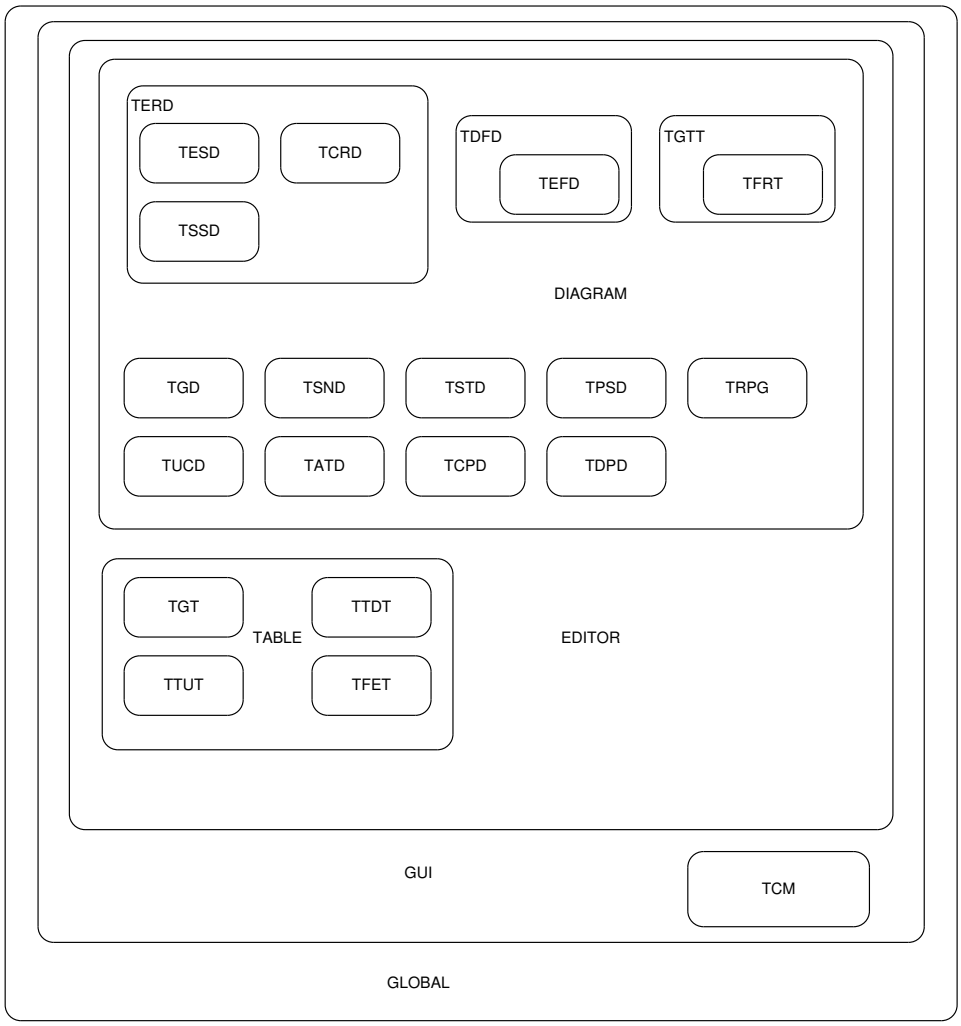

Figure 3.1: Logical source code organization.

shapes (boxes, lines, ellipses) and classes for most diagram edit commands (CreateNodeCmd, CreateEdgeCmd, DeleteShapesCmd,...). This library is used by every specific diagram editor.

- All code that lies in the table area is collected in directory `src/tb`. This code is compiled into a library called `libtable`. This includes classes for (abstract) tables having rows and columns of cells and it includes all the table edit commands. This library is used by each table editor.

- All code that is specific one or more (but not all) of the diagram editors, is collected in the directory `src/sd` (specific diagram). Each specific diagram editor can be compiled into a distinct executable (at least in principle, it is also possible to compile more editors into one executable). As you can see in the picture, the code of some editors includes the code of others (e.g. TSSD code includes TERD specific code). The specific diagram editor code consists of C++ classes derived from (possibly abstract) classes in libdiagram or from other specific diagram editor classes. These classes are (indirectly) derived of the libdiagram classes Graph, DiagramWindow, DiagramChecks, DiagramViewer, Diagram, Shape, ShapeView, Node and Edge. The tool specific constraints are all implemented in the classes derived from Graph, DiagramChecks, Node and Edge. Because the specific diagram editor code is so diverse it is subdivided further into separate subdirectories:

  - `src/sd/bv` is for the behavior view editors (TSTD, TATD, TRPG, TPSD),
  - `src/sd/dv` is for the data view editors (TERD, TESD, TSSD, TCRD, TUCD),
  - `src/sd/fv` is for the function view editors (TDFD, TEFD, TSND),
  - `src/sd/pv` is for the physical view editors (TCPD, TDPD),
  - `src/sd/gd` is for TGD and
  - `src/sd/tr` is for the tree editors (TGTT, TFRT).

  The TGD sources form the most simple diagram editor and it can be used as a basis for developing your own diagram editors. Specific editors that reside in the same directory can share one or more classes (for instance, the class BinaryRelationship is used both by TSSD and TUCD), or, one editor shares/extends all classes of another editor (for instance, the classes of TFRT are all specializations of the classes of TGTT).

- All code that is specific for the different table editors, is collected in the directory `src/st`. The amount of specific table code is rather small. The specific code consists entirely of specializations of the following classes in libtable: TableWindow, TableViewer and Table. Also, most of the tool specific constraints are implemented in these specializations of class Table.

In principle each C++ class is declared in a distinct header file and has a distinct source file for the implementation. The files names are equal to the class name except that file names are in lower case letters by convention. Header files have suffix '.h' and source file have suffix '.c'. The reason that C++ source files have suffix '.c', which is originally used only for C programs, is that some C++ compilers require a suffix '.C', and some require '.cc' or '.cpp'. There is no C++ file name suffix that is accepted by *all* compilers that we have used except the '.c' suffix.

3.2 Individual files and directories

In the previous section we described the contents of the `src` subdirectory. Here we will describe the individual files and directories that are included in the source code distributions of TCM. The TCM distribution top-level directory contains the following files:

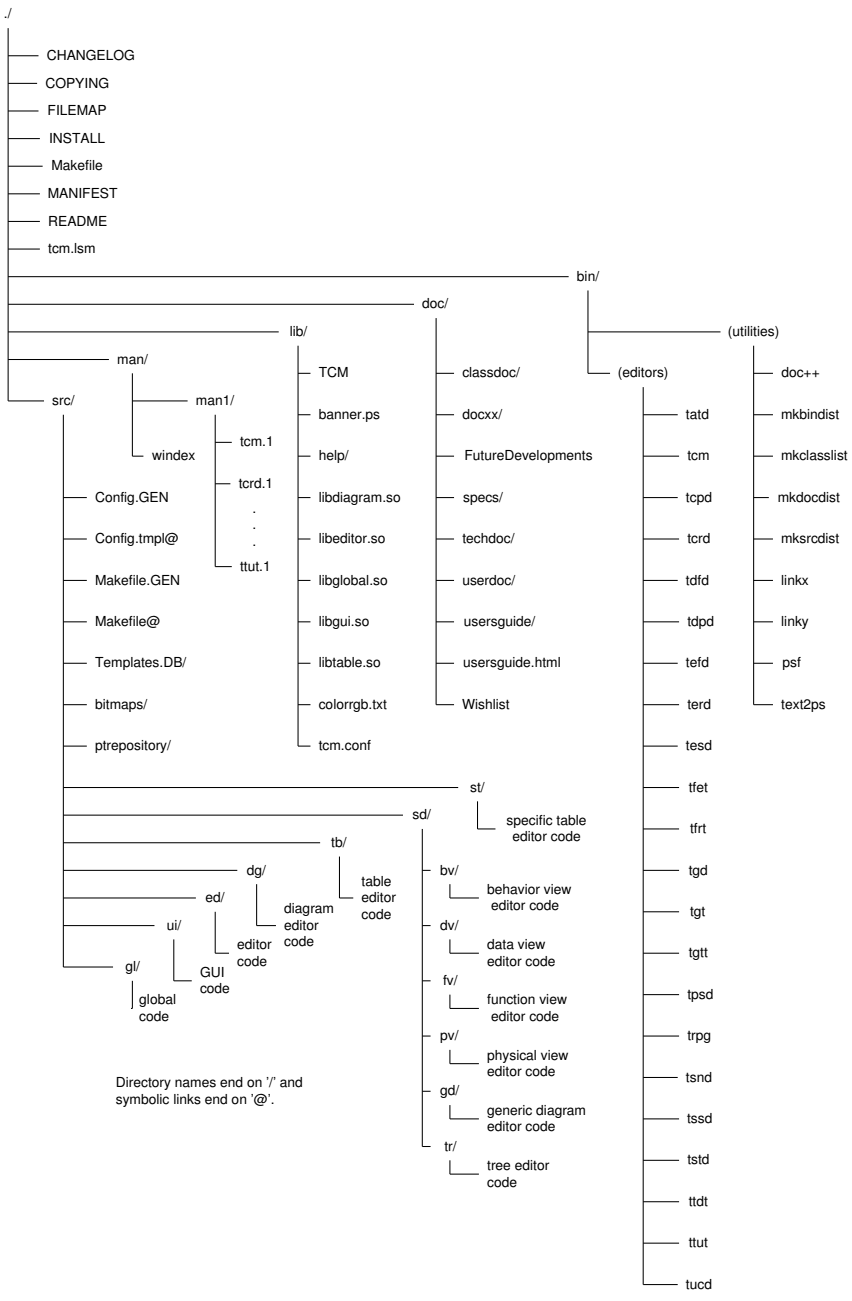

Figure 3.2: TCM directory tree.

- `CHANGELOG`, which contains the differences between the consecutive TCM versions.
- `COPYING`, which is the GNU public license.
- `FILEMAP`, which lists and describes the files and directories in the TCM executable distributions.
- `INSTALL`, which tells how to install the binary and source code distributions.
- `MANIFEST`, which lists all files and directories in the TCM distribution.
- `Makefile`, which is the top-level Makefile. Most sub(sub)directories contain a Makefile too. See chapter 7 for what to do with these Makefiles.
- `README`, which is the README file about the TCM project, the software, manuals, distributions etc.
- `tcm.lsm` is a file that describes TCM in the format that is required by the Linux software map. TCM is registered in the LSM (`http://www.execpc.com/lsm/`).
- `tcm-dynmotif-2.0.spec`. RPM spec file for building a TCM RPM distribution that links dynamically with the Motif/LessTif library. You can use (a modified copy of) this spec-file to build your own TCM RPMs. See `http://www.rpg.org` for more information about RPMs.
- `tcm-statmotif-2.0.spec`. RPM spec file for building a TCM RPM distribution that is statically linked with a Motif library.

The top-level directory contains the following directories:

- `bin/`. Here the TCM editors and other binaries are put after compilation. Also this directory contains a number of utility programs:
  - `bin/mkbindist`. This is a shell script that makes a tar.gz file of a binary distribution of the compiled source code. The script needs two arguments for the version number and the platform, e.g. `mkbindist 2.0 solaris.sparc`. This will create the file `tcm-2.0.bin.solaris.sparc.tar.gz` in `$TCM_HOME`. The files that will be included are listed in the code of the script itself.
  - `bin/mksrcdist`. This is a shell script that makes a tar.gz file of the source code. It needs one argument for the version number, e.g. `mksrcdist 2.0`. This will create the file `tcm-2.0.src.tar.gz` in `$TCM_HOME`. The files that will be included are listed in the code of the script itself.
  - `bin/mkclasslist`. This is a shell script that writes the names of all header files of the source code to standard output in alphabetical order.
  - `bin/psf`. This is a Perl script that is used to filter PostScript output (see `man psf`).
- `doc/`. Here all technical and user documentation can be found. The documentation is in HTML-format and possibly in PostScript and/or PDF format. The file `doc/index.html` links to all the different HTML documents. The documentation includes:
  - **User's guide**. In `usersguide/index.html` you can find an HTML version. A PostScript version can be found in `usersguide-2.0.ps.gz` (large PostScript files in TCM are always gzipped to save disk space). Optionally a PDF copy is put in `doc/usersguide-2.0.pdf`
  - **Developer's guide**, which is the document that you are reading now. In `developersguide/index.html` you can find an HTML version. A gzipped PostScript version can be found in `developersguide-2.0.ps.gz`. Optionally a PDF copy is found in `developersguide-2.0.pdf`

- **Source code documentation**. For each C++ class an HTML page is generated by the program DOC++ (see chapter 5). The HTML index is in `sourcecode/index.html`. Also a PostScript document with all the source code documentation can be found in `sourcecode-2.0.ps.gz`
- **Specifications**. The directory `specifications` contains a number of specifications (made with TCM) of some individual editors.
- **Wish lists**. The directory `wishlist` contains:
  - `FutureDevelopments.html`. An overview of major TCM extensions that we have in mind.
  - `WishList.html`. A list with unfulfilled wishes for TCM.
  - `WishListDone.html`. A list with fulfilled wishes for TCM.
- **Document sources**. In the directory `docsrc` you can find the LaTeX, EPS and TCM files that comprise the sources of the documentation. These files are not included in the source code distribution itself. The sources can be downloaded separately from our FTP-site, from a file `tcm-2.0.docsrc.tar.gz` The sources contain a number of Makefiles to generate a number of documents automatically. The program DOC++ is used to generate HTML and LaTeX from the C++ source code, `LaTeX2HTML` is used to generate the HTML versions of the user's and developer's guide. We have included the sources of DOC++, so that it will be compiled and installed before you build the documentation. `LaTeX2HTML` can be downloaded from `ftp.tex.ac.uk/tex-archive/support/latex2html` `LaTeX2HTML` is written in Perl and you have to configure it yourself before you can use it on your system. This is all explained in the README file of `LaTeX2HTML`.

- `lib/`. Here the object libraries (libglobal, libgui, libeditor, libdiagram and libtable) are stored after compilation. Furthermore this directory contains:
  - `lib/TCM` is the X Resources file. This file is not directly used by TCM (the default X resources of TCM are included in the source code), but it serves as a basis of your own modifications. You can load X resources with the `xrdb` command or by including them in some X-startup file like `$HOME/.Xdefaults`.
  - `lib/banner.ps` is a PostScript banner page. Normally this page is not printed but when you wish to print this banner page in front of your TCM documents, you can indicate this via the Printer Options sub-menu of in your editor or you can make this option the default by updating the `tcm.conf` file.
  - `lib/colorrgb.txt` is an ASCII file that maps TCM color names to red-green-blue (rgb) values.
  - `lib/help/` is a directory with the on-line help text files. The help files are all in ASCII format. These help texts are shown by the commands of the help-menu in the editors.
  - `lib/tcm.conf` is the TCM configuration file that is read upon start-up. It contains also some platform specific configuration options, like the name of the printer and the command to print or preview files. This file is intended to be human-readable and self-documenting. In stead of editing this file (which will affect all users of this TCM installation), you can also decide to override one or more options in a personal configuration file `$HOME/.tcmrc`. `.tcmrc` has the same syntax as `tcm.conf`. Note that some system specific configuration options such as the command to print or preview a document is commented out in `tcm.conf`. That is because TCM itself tries to determine these commands. Only if TCM can't find these commands or it chooses the wrong ones, you should set these as options explicitly by modifying `tcm.conf`.

- `man/` contains Unix man pages for TCM.
  - `man/man1/` contains short manual pages in nroff-format for each of the editors.
  - `man/windex` contains a simple index file that is used by `whatis` and `man -k`.

3.3 Object libraries

Depending on the operating system and the way TCM is compiled, object libraries are either **shared object libraries**, ending on `.so` or **object archive libraries** ending on `.a`. In the first case, the object code in the library is kept separated from the tool executables (they are dynamically linked), which makes executables smaller and run faster. For the Sun CC compiler (the default Solaris compiler) shared object libraries are made by default. For the other compilers such as `gcc` archive libraries are made by default. The contents of archive libraries are physically made part of the tool executables (they are statically linked); the libraries could be removed when compilation is completed.

Executables using object archive libraries tend to be much larger than with shared object libraries. Because of that, distributions that have to use archive libraries are in general compiled into only a few different executables, e.g. one for the tcm startup tool, one for all the diagram editors and one for all the table editors. The individual editors are then *symbolic links* to the collective diagram or table editor. The collective editor will see in its `argv[0]` argument which of the tools has to be launched.

Chapter 4

TCM User Interface

4.1 X/Motif user interface

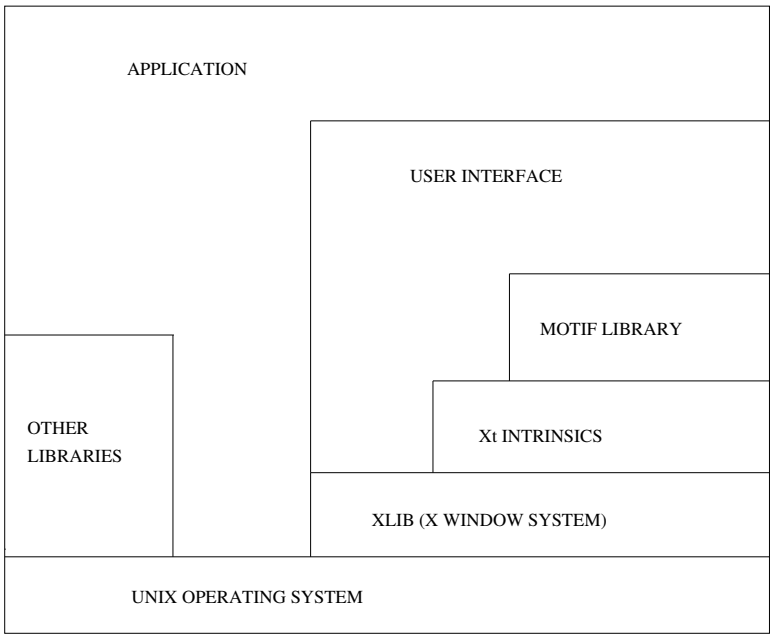

Figure 4.1: typical X/Motif application structure.

Figure 4.1 shows the relationships between the built-in X, Motif and Unix libraries and an arbitrary X/Motif application. All X, Motif and Unix libraries have a C interface (functions, defines, typedefs, structs, enums, unions and variables). When a library is depicted on top of another library, this means that a library is implemented by means of the interface provided by the underlying layer. You are referred to the X and Motif books and manuals in the list of references at the end of this text.

The TCM X/Motif user interface is restricted to that part of the source (classes and functions) that directly uses the Motif, Xt and Xlib libraries. That part consists of most of the classes in the `src/ui` directory and *some* of the classes in the other editor libraries. The names of the editor specific

user interface classes all end on `Window` or `Stubs`, such as `ERWindow`, `DFWindow` and `DiagramStubs`, `TableStubs` etc. The window classes are all specializations of the abstract class `ui/MainWindow`. The stub classes contain functions that are called from the user interface widgets (menus, push buttons, text fields etc.) and these functions in turn call application specific functions of non-GUI objects.

The `ed/DrawingArea` class is not part of libgui but of libeditor because it also has application specific behavior (albeit at a high level). The drawing area needs a viewer class (a descendent of class `ed/Viewer`). The viewer class should provide functions that are called when mouse buttons are pressed or dragged in the drawing area, when the mouse pointer is moved or when a characters are typed in while the mouse pointer is in the drawing area.

When asked what code contains X/Motif calls (and forms the GUI) then the answer is: almost all code of src/ui, all classes that end on `Window` or end on `Stubs` and the DrawingArea class.

The entire TCM distribution of the current version consists of about 450 classes and about 83000 lines of code (source and headers files, including white spaces and comments, but excluding the 20-line header part of the GNU General Public License). The X/Motif dependent part is about 19000 lines of code. This is the part that should be rewritten when TCM is translated to a non-X Windows environment.

The TCM user interface is more or less insulated into a restricted part of the source code, put in a number of C++ classes that are clearly distinguishable from the rest. But the user interface code is spread over all the TCM libraries (except libglobal). The reason for this organization is that factoring out the commonalities between all different tools was for us more important than a complete separation of the entire application code on one hand and the entire user interface code on the other. This means that each diagram editor has its own specific node and edge buttons. The code for creating tiled buttons in general is part of libgui but the specific initialization is done in the code of a specific diagram editor.

The approach of factoring out commonalities was a design for **evolution** and that has proved to work for TCM. It makes it relatively easy to add new tools or modify individual existing tools while maintaining a stable core of libraries and unrelated tools. When features are added to some library or to some tool, all tools that are included in the area of that library or tool (see figure 3.1) are also updated minimizing the risk of code redundancy or inconsistencies.

4.2 User interface implementation

4.2.1 Overview

Like all X11 applications, the TCM tools are event driven. This means they have the following basic structure:

1. Create the widgets.
2. Register the event handlers for these widgets.
3. Go into the main event loop.
4. Activate the appropriate event handler when a specific event occurs on a specific widget.
5. Return to the main event loop when done.

When a TCM tool is started the widgets that form the main window are created. The class `ed/EditWindow` and its specializations contain the functions to create the constituent parts of the main window. These widgets have one or more so-called **call-back functions**: the widget reacts to a certain set of X events (that set is built-in) and when such an event occurs a user-supplied action,

in the form of ordinary C functions, will be called. In our implementation we do not use ordinary C functions but instead we provide C++ static member functions.

An exception to this use of callbacks, is the drawing area widget which, by default, does not react on the events that are needed to draw pictures as we like to do in TCM. Therefore, so-called *translations* are used, an X-Toolkit data structure to define some specified mapping of X events to user supplied actions. All details of translations and drawing area events are hidden in the `ed/DrawingArea` class.

Most pop-up window dialog widgets are created during initialization. When they are popped up they do not have to be created (which is much faster) and when these dialogs are dismissed they are not destroyed, but they are simply kept unmanaged (invisible). An exception to this are so-called message dialogs, for error and warning messages etc. These are created on the fly because we can not determine in advance how many of which kind will be necessary.

A special kind of message dialog is the question dialog which is application modal. This means that the user can only respond to this dialog and other user actions in the applications are prohibited. This is necessary when the application has to have an answer before it can proceed (e.g. the 'save before quit?' dialog).

The main event loop is built-in in Motif, as well as the calling of the event handlers (via callback or translation).

4.2.2 Application startup

In contrast with an ordinary C program the editors don't start by directly entering the `main()` function. Instead, each editor is compiled with a distinct file `??editor.c`, e.g. `gdeditor.c` for TGD. In that file two global objects are created (but the two instance variables are not visible outside this file). For example:

```
Application *app = new Application("Tcm");
MainWindow *mw = new GDWindow("Tcm");
```

This means that the very first thing that is done on start-up is calling the constructor `Application::Application` and then the constructor `GDWindow::GDWindow`. So, for each editor first an instance of class `ui/Application` is created and then an instance of an editor specific main window class is created (but both are not initialized yet). MainWindow and Application are both part of libgui. Specific main window classes such as GDWindow are specializations of MainWindow and are included in the specific diagram editor sources.

There should be just one instance of class Application per editor instance, which can be accessed via the global variable named `theApplication`. The application opens the X display, sets the color map and does some other things that are applicable for the entire application. Furthermore, the application class keeps track of a list of the application (main) windows. The current TCM implementation is limited to a single main window (the other windows are made as pop-up dialog windows) but in principle the framework of libgui can be utilized in programs that have multiple main windows.

After the creation of both class instances, the function `main(argc, argv)` is entered in which `theApplication->Initialize(argc, argv)` is called. The `main` function is generic and is part of libgui. The application then creates an object config of type Config, which reads in the configuration file and keeps tracks of various editor defaults. Then the application initializes the Xt application context and subsequently calls the main window initialization functions.

The main window initialization consists of the creation of the main window widgets. After the main window widgets are created, some other important objects are created (in the DiagramWindow or TableWindow class), notably:

- `printer` of class `ed/Printer`. This object keeps track of all page and print options and takes care of printing and generating PostScript.

- `helper` of class `ed/Helper`. This object shows the on-line help messages. It reads the help messages from file.
- `document` of an editor specific specialization of the abstract class `ed/Document`, depending on the base class of the main window. E.g. GDWindow creates GDDiagram, DFWindow creates DFDiagram etc. The document object keeps track of the document information (like document name, author etc.) and takes amongst others care of loading documents from file and saving them to file.
- Depending on the kind of document that is created some extra objects are created such as a graph object for diagram editors (a GDGraph in the case of TGD), that manages the nodes and edges of the document and a viewer object (GDViewer in the case of TGD) that manages in the case of a diagram editor one or more views containing a set of graphical shapes. In the case of a table editor the viewer object manages the rows and columns of the table.

These objects (of which exactly one instance exist per main window), will be supplied as parameter to the Motif callback functions [1]. The callback function are static member functions of one of the stub classes. When the stub is called then the object is casted to the right type (because the parameters to stubs are simply pointers) and the appropriate class member of this object is called possibly with some other parameters that are supplied to the stub.

And finally, after the creation of the main window, the application object and the other main editor objects, the X main event loop is entered, giving X the control.

4.2.3 Main window structure

See figure 4.2 for the basic main window's widget structure. The boxes are Motif widgets (widget type and name between parentheses). The arrows connect widgets with their parents. The root is the top-level shell (TCM). To be more exact: the figure shows the main window of the generic diagram editor, TGD. The other diagram editors look almost the same, the main difference are their node and edge buttons. The table editors are almost similar too but they do not have node and edge buttons of course.

When libeditor.so is compiled with the option `-DDUMPWIDGETTREE`, a textual representation of the widget structure will be written to standard output when the editor is started (its output is also used for making this figure).

The details about creation of the widgets can be found in the classes `ui/MainWindow`, `ed/EditWindow`, `ed/DrawWindow`, `dg/DiagramWindow`, `tb/TableWindow` etc. See the class hierarchy of chapter 5 for the relationships and all the members of these classes.

4.2.4 Calling functions from the user interface

Menus are created by the constructor function of class `ui/Menu`. For an example for how to create a menu see `dg/DiagramWindow::CreateMenuBar`.

The items of a menu are specified by means of the `ui/MenuItem` class. A menu contains a list of menu items, which can be supplied to the menu constructor function. The menus common to all drawing editors are made in `ed/DrawWindow`, the menus common to all diagram editors are made in `dg/DiagramWindow` and the menus common to all table editors are made in `tb/TableWindow`. Likewise, menus that are specific for a single editor, are made in the corresponding main window class.

[1] 'callback' is Motif-speak. 'stub' is TCM-speak and means in this context the kind of callback function that is used in TCM and it forms an intermediate between the GUI (which uses X/Motif) and the other classes which do not depend on X/Motif.

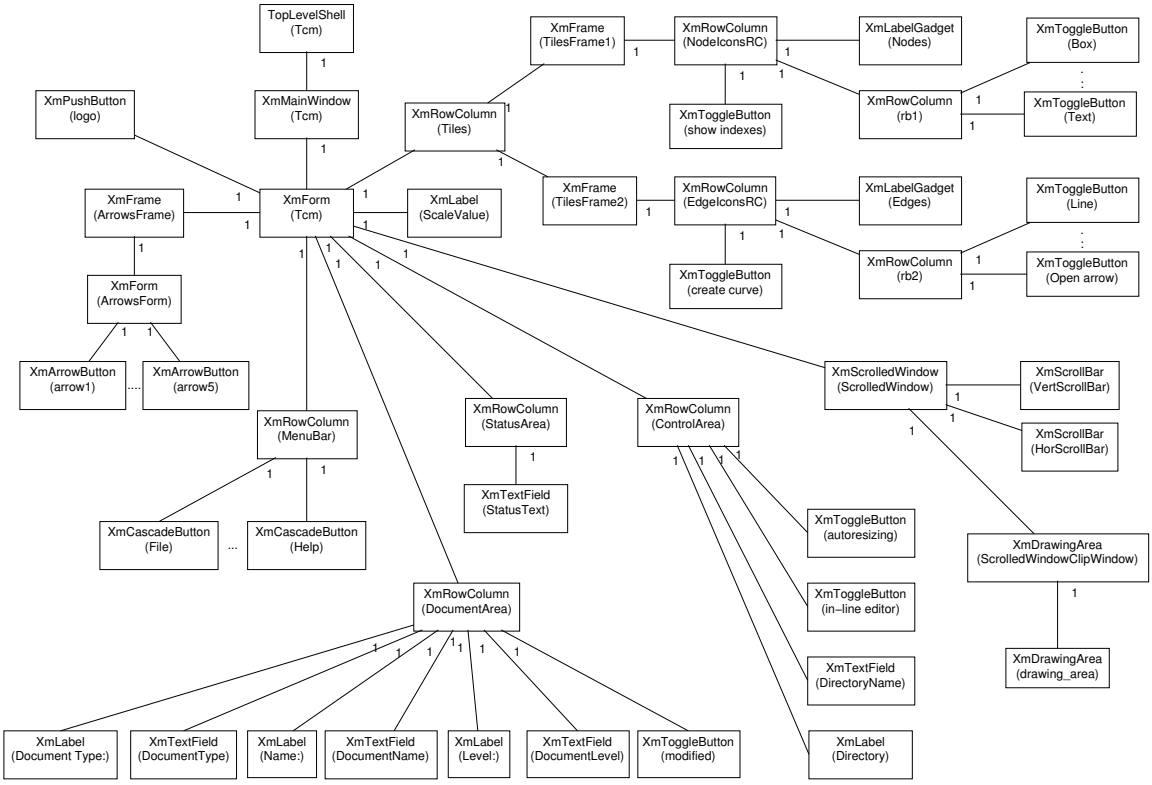

Figure 4.2: Main window of the generic diagram editor.

One of the fields of a MenuItem is the function that will be called when the menu item is selected. These functions are called stubs and have to be static C++ class members. The functions that are supplied are all part of a `...Stubs` class which consist entirely of these static functions. Menu items also have a so-called callback data. The callback data is always used to pass a pointer to the object whose class member should be called. For example, the Print menu item has as callback function the Print function of `ed/Editstubs` and as callback data a pointer to a printer of class `ed/Printer`. In the stubs function `Printer::Print` is called.

You can also call functions via the other main window widgets, for instance the diagram name text field or the node and edge toggle buttons. This works similar as with menus. You have to supply a callback function (`XtCallBack`) after the widget creation. These callbacks are also listed in the stub classes.

4.3 Xlib drawing

This is a very short introduction to drawing lines etc. in Xlib. For the rest you are referred to the documentation mentioned in the references.

X has 16 drawing modes (so called raster operations). A technique for simulating graphical objects moving or stretching (like resizing a box or a "rubber band" line), is to set the Graphics Context to the eXclusive-OR raster operation mode. The Graphics Context is an Xlib data structure which determines how an object will be drawn when a drawing routine is called (XDrawLine, XDrawRectangle etc.). In XOR mode, the new destination pixel is produced by the exclusive or of the old destination pixel with the source pixel. In this mode, you can easily draw and erase a figure. You draw a figure to let it appear, and when you redraw it, you erase it. By sequencing drawing and redrawing you can simulating an object being moved or dragged. Furthermore, in XOR mode, objects can overlap each other without damaging each other. The overlapping part is then white, and when one of the objects is moved or removed it will be redrawn, so that the overlapping part will appear black again. All details of drawing graphics with Xlib is hidden in the `XGrafport` class. An XGrafport has as attributes some GCs.

Note that each Xlib drawing operation is performed twice: one on the drawing area (window) and one on a Pixmap that serves as a backup store. This is needed because X does not automatically save the window contents when the window is overlapped or resized.

All the entire Xlib drawing functionality is hidden in the `ui/XGrafport` class which is a descendant of the abstract `ui/Grafport` class. Because XGrafport only uses a reference to the X Display and an X Window it can be used rather easily in any 2D drawing program under X (it uses Xlib but it does not use Motif nor the other classes of libgui that use Motif).

Chapter 5

TCM Class Hierarchy

A representation of the C++ class hierarchy can be automatically generated with `doc++`. `doc++` is a documentation system for C++ programs capable of generating output for HTML and LaTeX. `doc++` follows the approach of maintaining one source code that contains both the C++ program itself along with the documentation in order to avoid incompatibilities between the program and its documentation. `doc++` documentation is solely hidden in standard C++ comments (the text that appears in the doc++ files is C++ comment that starts with `//`, instead of the standard C++ convention to start comment with `///`). `doc++` is free software, subject to the GNU PUBLIC LICENSE. It is included in the TCM documentation source distribution and it can be found in the `doc/docsrc/docxx` subdirectory or it can be downloaded from `http://www.zib.de/Visual/software/doc++/index.html`.

A `doc++` LaTeX document of the entire TCM source tree can be made automatically and from that HTML and PostScript documents can be generated. These documents can be found in `doc/sourcecode/index.html` and `doc/sourcecode-2.0.ps.gz`.

The classes in the LaTeX document generated by `doc++` are alphabetically ordered. Each class has a section which consists of a picture of its base class and its derived classes, a list of its public and protected members plus relevant comment. As the files reside in different directories and are used in different libraries or executables, its 'scope' is mentioned at the end of the listing of a class. At the end of the `doc++` documentation you can find the entire class specialization graph. In the HTML documentation generated by `doc++`, a distinct HTML page is generated per C++ class. Connections between classes (variables, function arguments, base/derived classes) are automatically translated to HTML-links and there is an index file which contains links to all classes in alphabetic order.

Chapter 6

Output Files

6.1 PostScript output

This is a short description of the structure of the PostScript files that is generated by the current version of TCM. This is not a lesson in programming PostScript nor a detailed explanation of each possible PostScript expression that could be generated by TCM. Instead, you are referred to the official documentation like [6, 5].

6.1.1 Plain PostScript

The PostScript output files for a TCM document have the following structure.

Header

The first part of the plain PostScript output consists of the following header:

```
%!PS-Adobe-1.0
%%Title: <name of document>
%%Creator: <tool+version>
%%CreationDate: <current date>
%%For: <login>
%%DocumentFonts: (atend)
%%Pages: (atend)
%%BoundingBox: (atend)
%%EndComments
/ISOLatin1Encoding[
      ...
] def
EndProlog
```

Lines that start with `%%` or `%!` are PostScript structuring commands. The generated files conform to the PS-Adobe 1.0 file structuring conventions. `(atend)` means that the value is determined in the rest of the program. The ISOLatin1Encoding definition (whose body is not printed here to save space) is necessary to make sure that the entire ISO Latin-1 character set can be displayed.

Pages

Each page starts with a page setup followed by the page contents. Example page setup:

```
%%Page: 1 2
12.8976 9.49604 translate
0.867470 0.867470 scale
0 948.736111 translate
1 -1 scale
1.000000 1.000000 scale
0.750000 setlinewidth
%%BeginPageSetup
gsave
-0.000000 -0.000000 translate
%%EndPageSetup
```

The first line gives the current page label (1 in this example) and the total number of pages (2). The next translate line moves the coordinate system to the drawable paper area. Then the scale line scales X coordinates to PostScript points. The other translate and scale lines change from X to the PostScript coordinate system (in portrait mode and with scale factor 1.0). When the page would be written in LandScape and with scale factor 1.5, those three lines would be:

```
90 rotate
1 -1 scale
1.5 1.5 scale
```

The setlinewidth line sets the width of the line a bit smaller than the default, which is a bit too fat in comparison with the Xlib lines. The lines between BeginPageSetup and EndPageSetup move the page (like it is visible in the X window) that is going be written to the position of the first page (in this case the translation is zero because the example shows the first page).

After that, the lines of the current page are written in rather ordinary PostScript prose (with `newpath`, `moveto`, `rlineto`, `closepath` and `stroke` commands).

Strings are drawn in the following fashion: First the font is loaded when that is not already done, e.g.:

```
dup length dict begin
   {1 index /FID ne {def} {pop pop} ifelse} forall
   /Encoding ISOLatin1Encoding def
   currentdict
end
/Helvetica-ISOLatin1Encoding exch definefont pop
```

This loads the font family (in our case Helvetica ISO Latin1). Then the string is positioned and drawn, e.g:

```
/Helvetica-ISOLatin1Encoding findfont
10 scalefont setfont
(foobar) stringwidth
pop 2 div neg
120 add 144 moveto
gsave
1 -1 scale
(foobar) show
grestore
```

The above lines draw the string foobar in the just loaded font, centered at position (120,144) with point size 10. Each PostScript output page ends with:

```
grestore
showpage
%%PageTrailer
```

End finally the PostScript output ends with:

```
%%Trailer
%%EOF
```

6.1.2 Encapsulated PostScript

The first part of the EPS output consists of the following header.

```
%!PS-Adobe-3.0 EPSF-3.0
%%Title: <document name>
%%Creator: <tool+version>
%%CreationDate: <current date>
%%For: <login>
%%DocumentFonts: (atend)
%%Pages: 0
%%BoundingBox: llx lly urx ury
%%EndComments
/ISOLatin1Encoding[
] def
```

The four numbers (llx lly urx ury) are the top left and bottom right coordinates of the bounding box, i.e. the square area occupied by the drawing. After the header the font is loaded, in the same way as in plain PostScript. EPS does not have separate pages and hence does not need page setup lines. The following lines are generated to change from X coordinates to EPS coordinates:

```
0.867470 0.867470 scale
0 85 2 mul 117 add 0.867470 div translate
1 -1 scale
%%EndProlog
```

0.867470 is the (fixed) factor of X coordinates/PS coordinates. As a remark, I forgot the exact meaning of the second line, but is works in any case. After that the entire drawing is written as PostScript, the same way as it is done with plain PostScript (but without page setups). The output ends with the same two lines as plain PostScript.

6.1.3 PSGrafport

All PostScript generation is hidden in the `PSGrafport` class. It contains the functions that write headers and page setup lines to the PostScript output file and for each `Grafport` function it contains a counterpart that writes a piece of PostScript. For example `Grafport` has a virtual function `DrawRectangle(int x, int y, int width, int height)` that draws a rectangle to the Grafport. `XGrafport` implements this function as:

```
XDrawRectangle(display, window, xorGC, x, y, wd, ht);
```

I.e. it draws a rectangle on an X window. Whereas `PSGrafport` implements this function with:

```
fprintf(fd, "newpath\n");
fprintf(fd, "%d %d moveto\n", x, y);
fprintf(fd, "%d 0  rlineto\n", wd);
fprintf(fd, "0 %d  rlineto\n", ht);
fprintf(fd, "-%d 0 rlineto\n", wd);
fprintf(fd, "closepath\n");
fprintf(fd, "stroke\n");
```

I.e. it writes a PostScript fragment to the file that would result into the printing of a rectangle.

To add another output format to TCM, you probably need only to make another specialization of Grafport and implement each Grafport function in that class to write a piece of output in that format. For instance, for writing the Fig output (Produced by Xfig) we implemented a FigGrafport that writes for every Draw-functions a part of Fig-format to file.

6.2 TCM file format

The TCM file format of the current version is described in full detail in appendix B of the User's guide.

Chapter 7

Compiling and Porting TCM

Before you start compiling or porting TCM you first have to unzip and untar the source code distribution in the same manner as the executable distribution. See then the file `$TCM_HOME/INSTALL` which contains the most up-to-date installation instructions. See chapter 3 for what files are included. Make sure that before compilation the `TCM_HOME` environment variable is set to the directory where the distribution resides (the "root" directory of the entire distribution, not the src directory!).

The TCM directory tree contains a set of Makefiles and configuration files. The `Makefile` in the src directory should be a symbolic link to `Makefile.`*compiler*. `Config.tmpl` should be a link to `Config.tmpl_`*platform*. `src/Makefile` contains the targets specific for some compiler (g++, Sun CC, etc.) and `Config.tmpl` contains settings for the system. For each operating system to which TCM has been ported (Linux, Solaris, HP-UX, etc.) a distinct Config.tmpl file is supplied. The default make targets are in `src/Makefile.GEN` and the default settings of the various Config.tmpl files are in `src/Config.GEN`.

7.1 Compiling TCM

The most simple way to compile and install TCM is to type `make` for the Makefile in `$TCM_HOME`. This default target will set the right links for the `src/Makefile` and `src/Config.tmpl` and then it will write to standard output what are the following targets that you can build. Of course it wise to check the setting of `src/Config.tmpl` yourself first before you continue. The next possible targets are:

- `make depend`, for making file dependencies.
- `make execs`, for making the object libraries and binaries and then move them in the `lib` and `bin` directories respectively.
- `make install`, for copying the binaries and all the other files of a binary distribution to the directory where you would like to install TCM. `/opt/tcm` is the default directory but that can be changed in the Makefile by setting `TCM_INSTALL_DIR`.
- `make all`, which is the same as `make depend execs install`.
- `make docs`, which tries to (re)generate all documentation in HTML, PostScript and PDF format in the `doc` subdirectory. This works only of course if you have installed the document sources in `docsrc`.
- `make clean`, which removes all binaries, object and temporary files.

Instead of `make install` you can also call the scripts `mkbindist` or `mksrcdist` to build a tar.gz file with the binaries and the source code respectively. They were treated in chapter 3.2.

For compiling TCM you can also go directly into the src directory. There you are able to build individual editors and individual libraries. As usual, compilation is controlled by a set of Makefiles. In the src directory there are a number of configuration files and Makefiles.

7.1.1 Compilation configuration files

The configuration files in the `src` directory are called `Config.tmpl_`*suffix*, one for each of the different platforms (platform as file name suffix). There is a file called `Config.GEN` which contains reasonable default values and is included in the Config.tmpl files. The configuration file defines which compiler is used, the compiler flags, the location of the Unix and X include files, the needed Unix and X libraries and their locations. Ideally, this is the only file you need to tailor for compiling TCM on a Unix system. To compile TCM you also need `lex` and `yacc` (or the GNU variants `flex` and `bison`). The files that are generated have to be compiled by an ordinary C compiler. Therefore, define in Config.tmpl, `LEX`, `YACC` and `Cc`. Before compiling TCM, make a symbolic link called `Config.tmpl` to the configuration file of the desired platform. The declarations in the `Config.tmpl` will be included in the Makefiles.

Configuration options consist of *Variable name* `=` *Value*. To append a value to a variable name you can use `+=` instead of `=`. A value can contain the contents of a variable that was defined before, by using the notation `$(`*variable name*`)`. The following configuration options can be set:

- `CC`. This is the C++ compiler. This is `/usr/bin/g++` by default.
- `Cc`. This is the C compiler. This is `/usr/bin/gcc` by default.
- `CFLAGS`. These are the flags given to the C++ compiler. By default these are `-Wall -pedantic`. You can add compiler-specific options for C++ templates, optimizing options or add the option `-g` to add debugging information.
- `INCLUDEDIRS`, the system include directories (-I flags) for the compiler.
- `LD_FLAGS`, the library flags (-L and -R flags) for linking.
- `LD_LIBS`, the libraries against which TCM is linked.
- `LEX`, the (f)lex lexical analyzer.
- `MKDEPEND`, the command that is called for `make depend`.
- `MKDEPENDFLAGS`, the flags for `make depend`.
- `MOTIF_HOME`, the directory where Motif or LessTif is installed.
- `STRIP`, the Unix command to strip binaries. When you don't want to strip (because debugging information would be lost), set it to `/bin/echo`.
- `SHAREDFLAG`, special compiler flag, such as `-shared` or `-G`, to generate shared object libraries.
- `SYSFLAGS`, a number of TCM-specific flags for the compiler:
  - `-DLINUX`, `-DSOLARIS`, `-DIRIX`, etc. Set this to the target operating system.
  - `-DLESSTIF`. Set this if you compile with LessTif, don't set it if you use Motif.
  - `-DDEBUG`, `-DDUMPWINDOWTREE`, output some debugging information.
- `XWIN_HOME`, the directory where X windows is installed.
- `YACC`, the yacc, compiler compiler.

7.1.2 Makefiles

The main Makefile contains the rules for how to build the libraries and the executables in the subdirectories. Per type of compiler there is a distinct Makefile. They are called `Makefile.`*suffix*. The default make targets are in the file `Makefile.GEN` and it's included in the other Makefiles. Before compiling TCM, make a symbolic link called `Makefile` to the Makefile of the desired compiler. Each src-subdirectory has its own Makefile which is platform- and compiler-independent. If everything goes normal they need not be changed when TCM is recompiled or ported.

The Makefile in the src directory has the following top-level targets:

- `make` or `make all`. Compiles the entire distribution (default compilation).
- `make allx`. Force compilation into a few executables (i.e. all diagram editors and all table editors are each compiled into a single executable). This is the default for `g++`. You can create the needed symbolic links for the different editors with the script `$TCM_HOME/bin/linkx`
- `make ally`. Force compilation into some more executables than `allx`. Related tools such as all data view editors are compiled into one executable. The individual tools have to be soft links, that can be made with the script `$TCM_HOME/bin/linky`
- `make allz`. Force compilation in which each editor is compiled in a separate executable. This is the default for the Solaris CC compiler.
- `make libs`. Compiles all object libraries. These libraries can be static or dynamic, depending on the compiler that is used.
- `make staticlibs`: make all static object libraries (the libXXX.a versions).
- `make dynamiclibs`: make all dynamic object libraries (the libXXX.so versions).
- `make clean`. All object files in the source directories are removed.
- `make depend`. Create dependencies in the Makefiles. You are advised to do a `make depend` for a `make all` when you compile TCM for the first time.
- `make` *library*. Compile given library.
- `make` *tool*. Compile given tool (only tool specific part, not the libraries that it needs). So you have to type in `make dynamiclibs tgd` when you want to make TGD and the dynamic libraries that it needs.

When a library is compiled, it will be moved to `$TCM_HOME/lib` and when an executable is compiled it will be moved to `$TCM_HOME/bin`. This means it will overwrite the old version.

Which default compilation is performed, depends on the kind of compiler that is used. The Sun CC compiler, whose Makefile is Makefile.suncc, has `make dynamiclibs allz` as default and the GNU g++, whose Makefile is Makefile.gcc, has `make staticlibs allx` as default.

7.2 Porting TCM

In principle, TCM can be ported to any Unix system that has X Windows, Motif and a C++ compiler that can handle templates. The easiest way is to copy and adapt an existing Makefile and Configuration file and then try to do a `make clean`, a `make depend` and then a `make all`.

Most of the source code is platform independent. Only at a few places there are some Unix specific parts which are compiled conditionally. This is indicated in the source code by for instance '#ifdef

LINUX'. The Makefile compiles its targets with the `-D` flag (see the Config.tmpl file) for instance `-DLINUX`. The files that probably need some modification, because they use conditional compilation are: `gl/system.c`, `gl/link.c`, `gl/util.h` and `ed/document.c`. The best thing to do before you port is to do some `grep`s on the sources to see what sources should maybe be changed when ported. For instance: `grep SOLARIS ??/*[hc] ??/??/*[hc] ; grep LINUX ??/*[hc] ??/??/*[hc]` (do this in `$TCM_HOME/src`), will show what part of the sources have things specific for Solaris and Linux.

Likewise, a few lines of code could be different for Motif and LessTif. When TCM is compiled with LessTif then the compiler should be supplied with the `-DLESSTIF` flag. In the source code you will find some `#ifdef LESSTIF`s.

7.3 G++ specific problems

The GNU C++ compiler g++ is somewhat limited in handling template classes. GNU g++ does not implement a separate pass to instantiate template functions and classes at this point; for this reason, it will not work, for the most part, to declare your template functions in one file and define them in another. The compiler will need to see the entire definition of the function, and will generate a static copy of the function in each file in which it is used. G++ does not automatically instantiate templates defined in other files. Because of this, code written for cfront will often produce undefined symbol errors when compiled with g++. You need to tell g++ the file where they are defined.

The solution for TCM was, when `__GNUC__` is defined, to include in `gl/llist.c` the template declarations that you need from the file `gl/instances.h`. u Because different groups of editors need different declarations, several files with instances are created in different source code directories and during compilation the needed file is copied to `gl/instances.h` and `gl/llist.c` is compiled and `libglobal.a` is generated again. The instance files contain not much more than declarations of template instances. This process is controlled by the Makefiles. This solution only works with static linking of libglobal (because different libglobal libraries are needed by the executables). In principle the other libraries can be linked dynamically by `g++` This can be achieved by issuing the target: `make semistaticlibs`.

By the way, the List class is in the file `gl/llist.[hc]` because the file name `list.[hc]` caused some strange name clashes while using Sun CC compilers.

When you want to compile TCM with g++ on another system (use version 2.7.2 or higher), then take `Makefile.gcc` as a basis for your Makefile and take a look in `Config.tmpl_linux` because on Linux the g++ compiler is used by default.

Chapter 8

Wish List and Future Plans

There are some additional texts in the `doc/wishlist` directory. The file `WishList.html` contains a list of wishes for the current set of editors. At this moment it contains over 200 items in categories and they are given priorities. Chances are big that your wishes are already mentioned there. If not, let us know. Adding an item to this list won't cost us a lot of time :).

Furthermore, there are lots of plans for further developments, which are listed in a file called `doc/wishlist/FutureDevelopments.html`. Our vision is to make from TCM a full featured toolkit, both for Structured Analysis and UML.

Bibliography

[1] Marshall Brain. *Motif programming – The essentials and more.* Digital Press, 1992.

[2] F. Dehne, H. van de Zandschulp, and R.J. Wieringa. Toolkit for conceptual modeling, user's guide for tcm 2.0. Technical report, Faculty of Computer Science, University of Twente, September 1999.

[3] J. Gettys and R. Scheiffler. *Xlib - C Language X Interface*, 1996. for X version 11, release 6.1.

[4] Dan Heller. *Motif Programming Manual*, volume 6 of *The Definitive Guide to the X Window System.* O'Reilly & Associates, 1991. For OSF/Motif Version 1.1.

[5] Adobe Systems Incorporated. *PostScript Language - Reference Manual.* Addison Wesley, 16th edition, 1990.

[6] Adobe Systems Incorporated. *PostScript Language - Tutorial and Cookbook.* Addison Wesley, 16th edition, 1990.

[7] J. McCormack, P. Assente, and R. Swick. *X Toolkit Intrinsics - C Language X Interface*, 1994. for X version 11, release 6.1.

[8] Adrian Nye. *Xlib Programming Manual*, volume 1 of *The Definitive Guide to the X Window System.* O'Reilly & Associates, 1990. for version X11R4.

[9] Adrian Nye. *Xlib Reference Manual*, volume 2 of *The Definitive Guide to the X Window System.* O'Reilly & Associates, 1990. for version X11R4.

[10] Adrian Nye and Tim O'Reilly. *X Toolkit Intrinsics Programming Manual*, volume 4 of *The Definitive Guide to the X Window System.* O'Reilly & Associates, 1990. for version X11R4.

[11] Staff of O'Reilly and Associates. *X Toolkit Intrinsics Reference Manual*, volume 5 of *The Definitive Guide to the X Window System.* O'Reilly & Associates, 1991. for version X11R4.

[12] OSF (Open Software Foundation). *OSF/Motif Programmer's Guide*, 1992. For OSF/Motif Release 1.2.

[13] B. Stroustrup. *The C++ programming language.* Addison-Wesley, 1991.

[14] Douglas A. Young. *Object Oriented Programming with C++ and OSF/Motif.* Prentice Hall, 1991.

[15] Douglas A. Young. *The X window system, programming and applications with Xt.* Prentice Hall, 1994. OSF Motif edition.