Evolution 200

Dokumentation

Informatikprojekt
von Bernhard Nemec

Universität Bern

Januar 2000

Mentor:
Prof. Dr. O. Nierstrasz

Contents

List of Figures

List of Tables

Acknowledgments

I would like to thank Werner Döbelin, owner of LabSource/ProLab GmbH and constructor of the *Evolution* μHPLC Pump, for taking the risk of employing a "beginner" like me and giving me the chance of many experiences. I enjoyed the pleasant and uncomplicated style of working together.

Chapter 1

Introduction

1.1 History

I started to work for LabSource as a freelance software developer in 1995 when Werner Döbelin began to project his "Evolution μHPLC Pump System" and needed someone who could implement a user interface for it.

The first prototype was implemented with Delphi 1 – which was quite new back then – on Windows 3.1. While the development of the pump itself was subject to certain delays due to refactoring, the software prototype was developed further only sporadically until in summer 1997 I decided to rewrite it from scratch and migrate to Delphi 3 and Windows 95/NT 4.0.

As of the end of 1999, several pumps have already been sold, and the system has shown to be work well in productive environments.

1.2 Scope of this Document

This document is meant to

- document the structure of the "Evolution 200" software in a way that an other software developer could manage to maintain it in the future, and
- give insight to my personal experience from the task of writing the software.

It is *not* meant as a user documentation, which is provided separately. Also does it not introduce all use concepts of the software, which can be learnt from the User Manual (see [Manual]).

Figure 1.1: The Evolution 200 Pump

1.3 Terms

Pump The LabSource Evolution 200 μHPLC Pump. This is a hardware device controlled by a built-in microprocessor board that receives commands from, and sends status messages to, a PC over a serial line.

A picture of the pump is shown in figure 1.1.

Evolution Software (Software) The software program that runs on the attached PC. It provides a user interface to the pump functions. This is my part of the work, and it is the actual subject of this document.

Method A "program" that can control the pump automatically over a given time. The term "method" is common in the field of chemical analytics. (See [Manual], section "Concepts | Evolution Methods")

This has nothing to do with "methods" known from OO programming.

Chapter 2

Analysis

2.1 System Overview

The context diagram in Figure 2.1 gives an overview of the whole System and the place held by the Evolution Software.

The whole μHPLC System consists of the Pump on the one hand and of a PC (Notebook) with the Evolution Software on the other hand, the two connected through a common RS323 serial cable. Yet from our point of view, the pump is nothing else but an external object that interacts with "our" system (the PC software), and thus we can regard it as a "black box". We model the pump as an entity for itself in the Overview Context Diagram and depict it with a special symbol derived from the common UML $\ll$active object$\gg$ stereotype icon.

There are two other entities on the diagram: a User and an External Device.

The external device can communicate with the PC "through the pump" by means of triggering the pump's external digital inputs. An common use of this is that a mass spectrometer (MS, the "external device" in this case) can signal its readiness for the next sample to be injected by the pump. It just has to short-circuit a special digital input of the pump, causing the pump to send a message to the software saying that the method should be started now.

2.1.1 The Pump Interface

The PC is connected to the pump through a serial cable (RS-232).

All data that can be exchanged between the PC and the pump is represented in a set of 256 *registers* each of which holds a 16-bit word value. The registers #0 to #139 are defined as "downward" registers (i. e., that can only be sent from the PC to the pump), while the registers #140 to #255 are "upward" registers (i. e., to be sent from the Pump to the PC). The communications protocol has been defined in

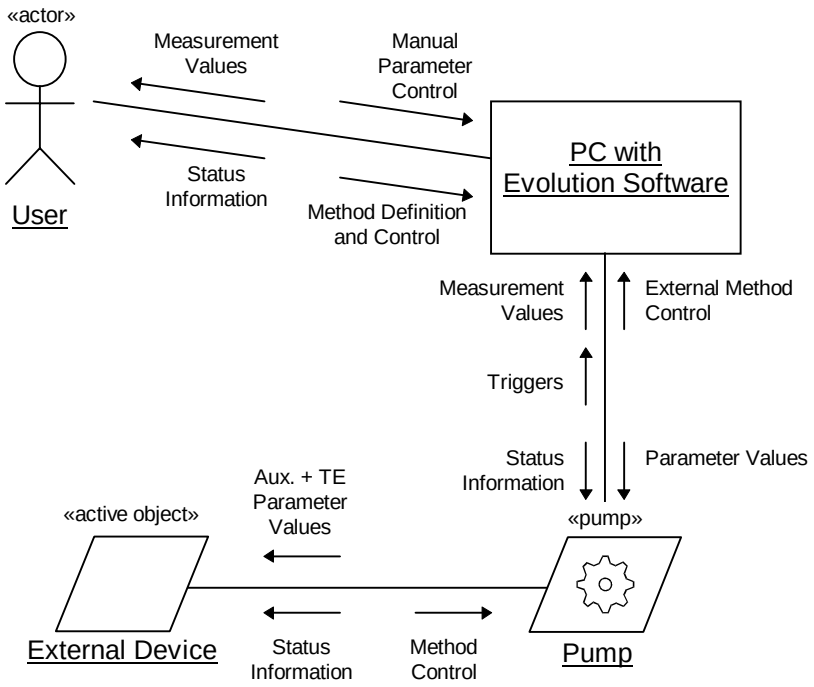


Figure 2.1: Overview Context Diagram

Figure 2.2: PC ↔ Pump Messaging

Flow: Total flow rate of both pump heads A and B in $\mu l/min$
%A: The share of pump head A on Flow in percent
Temperature: Temperature in $^{\circ}C$
Voltage: Voltage of the built-in high voltage power supply (HVPS)
Current: Current of the built-in high voltage power supply (HVPS)
Degaser: Switching the built-in eluent degasser on and off
Auxiliary: Voltage of the external analogous output connector in percent.
Time Events 1-6: Switching the 6 external digital output connectors on and off

Table 2.1: Pump Control Parameters sent from the PC to the pump. These are interpreted as *set points* for the pump.

Pressure: Pump pressure in bar
Actual Temperature: Actual Temperature in $^{\circ}$ C
Actual Voltage: Actual Voltage of the built-in high voltage power supply (HVPS)
Actual Current: Actual Current of the built-in high voltage power supply (HVPS)

Table 2.2: Basic measurement parameters that are sent from the pump to the PC.

Method Init: This sends the signal sgInitialize to the state machine.
Method Start: This sends the signal sgStart to the state machine.
Method Stop: This sends the signal sgStop to the state machine.
Comm. Error: This means that the pump has detected some failure in the communication between pump and PC. Of course it is not sure that the PC receives this message at all in this case. It is more likely that the PC detects a communication error by itself. In either case, the user is informed and advised to check the cable.
Pump failure: This message means that the pump detected some malfunction of its hardware. The user is advised to call a service person in this case.

Table 2.3: Error and method control messages sent from the pump to the PC.

ValueMsg: Contains the new value of some parameter (either control (downwards) or measurement (upwards)).
TriggerMsg: Sent upwards whenever a trigger is released. Triggers cover the arrival at a pressure limit or target value, or internal pump errors that are reported to the PC.
ExtMethodControlMsg: Triggered from an external device, controls the flow of method execution (e. g., *start*, *stop*).
StateMsg: Sent downwards whenever the state of a method is changed (e. g., *initializing*, *running*, *finished*) in order to inform any external devices.

Table 2.4: PC $\leftrightarrow$ Pump message types

a separate document ([Rudin97]) and shall not be discussed here. It just transports pairs of register numbers (1 byte) and their values (2 bytes) between the pump and the PC software.

More interesting here is the "application level" network layer, i. e., the logical view of messaging between the pump and the software. Figure 2.2 and Table 2.4 show various types of such messages.

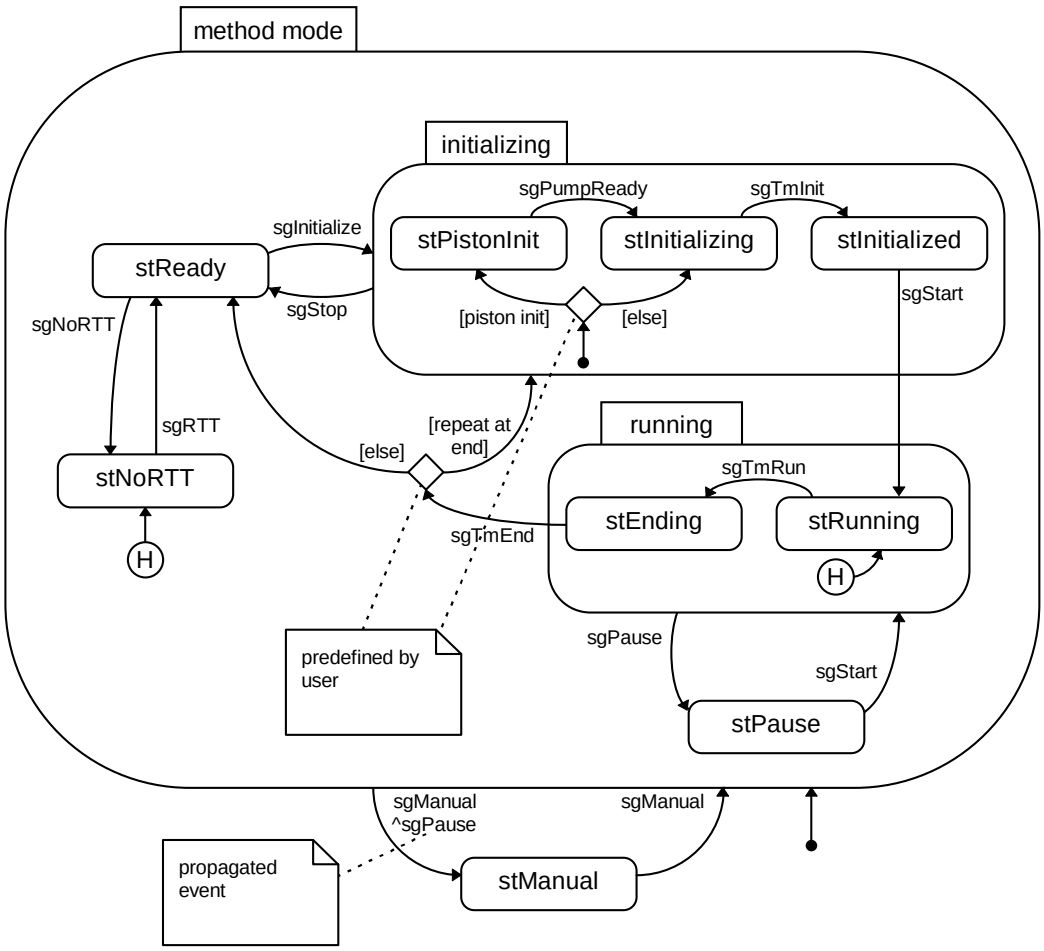


Figure 2.3: State Chart

2.1.2 The Main State Machine

Many things in the software depend on a few clearly defined states. They describe a “method-centric” view of the system.

These states are listed in table 2.5.

Transitions between these states depend on the “signals” (or “events”) listed in table 2.6.

In the state chart (figure 2.3) one can see how the states are related, and which signals they “listen” to. They are grouped into superstates (Method Mode, Initializing, and Running) just to simplify the chart.

stManual: *Manual Mode*, this means that no method is active and the pump is only controlled by hand.
stNoRTT: This means that we're not in Manual Mode, but there is also no method with a valid runtime table that could be started.
stReady: This means that a method with a valid runtime table is loaded. It is ready to be initiated.
stPistonInit: This is the first step of a method initiation, if the chose this option for the currently loaded method. It Consists in resetting the position of all pump pistons. We have to wait for a message to say they're ready to start.
stInitiating: This means that the pump control parameters from the first line in the runtime table are active for the duration specified as "initial configuration time" for this method.
stInitiated: After initiation, the pump is ready to go on with the rest of the runtime table. Yet it waits for the sgStart signal.
stRunning: The runtime table is being worked off. New values are sent to the pump every second until the runtime table ends.
stEnding: The runtime table has been worked off, and the values of its last line remain active for the duration specified as "end configuration time" for this method.
stPause: A method that was running is "frozen", and the pump keeps working using the last active values.

Table 2.5: States of the state machine

sgManual: This occurs when a user wants to enter or to exit Manual Mode (selecting the menu entry, pushing the button, or the hotkey).
sgNoRTT: The runtime table is cleared (for example when the user creates a new method).
sgRTT: The runtime table newly contains more than one line (for example when the user loads a method from disk, or he is writing the runtime table).
sgInitialize: The user wants to initialize the method, or a "method init" message is received from the pump.
sgPumpReady: The pump newly is ready to start pumping. This occurs when the pump pistons have finished initializing.
sgTmInit: This occurs as soon as the method has remained stInitializing for the "initial configuration time" specified for this method.
sgStart: This occurs when the user presses the "run" button (or menu), or when a "method start" message is received from the pump.
sgTmRun: This occurs as soon as the runtime table of the method has been worked off.
sgTmEnd: This occurs as soon as the method has remained stEnding for the "end configuration time" specified for this method.
sgStop: This occurs when the user wishes to stop a method early.
sgPause: This occurs when the user wants to hold the execution of a method. This also happens when he switches to the manual mode while the method is running.

Table 2.6: Signals of the state machine

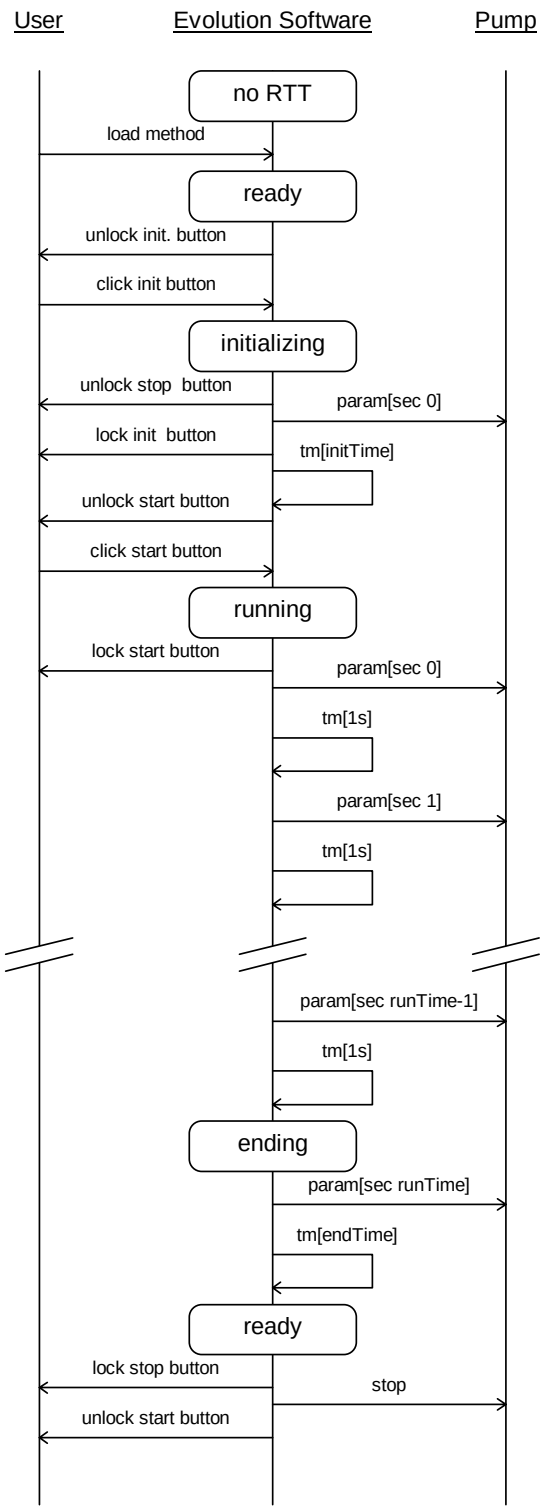


Figure 2.4: Method Execution Sequence Diagram

2.2 Software Requirements

The list of requirements was developed in many conversations with Mr. Döbelin. It was many times adjusted, in some cases as a reaction to user feedback. Although I regard it as mainly finished now, I am sure that there will always be new wishes and improvement ideas from customers, and the software development process will never be at its end.

The requirements for the software in brief:

- Control of all Pump Control Parameters (see Table 2.1) by the user, either manually or programmatically by means of a method.
- Presentation and capture of measured values that are periodically sent by the pump (see Table 2.2).
- Definition of “Methods” to automatically set the Pump Control Parameters over a certain period of time.
- Graphic visualization of the Method’s “Runtime Table”, in form of curves.
- Automatic execution of multiple methods in sequence (“method batches”).
- Printing hardcopies of Method files.
- User documentation (on-line help).

Chapter 3

Design

3.1 Overview

Figure 3.1 shows the principal architecture of the application which is broken up into several subsystems (*Packages*) for this documentation.

The class diagrams in this chapter show the classes of each package in their relationship to each other. They do not display private and unimportant class members. 3rd-party classes appear grayed-out.

Members of Windows often are widgets and are not shown in the class diagrams either. This is because they are directly visible in the windows (screenshots of which are depicted in [Manual]), their functions are clear, and sometimes they are just too numerous to be good for a quick overview.

The application consists of a *Main Window* (`TMainWnd`) which is instantiated by the `Application` thread, and which instantiates several Objects of "subsystem" classes. These are depicted in Figure 3.1.

The main window provides a user interface with menus, toolbar buttons and a "status line" at the bottom of the window which displays important values of the pump and the state of the program.

The main window also acts as a *MDI Parent Form* containing the four windows of the *Method User Interface* package (section 3.5).

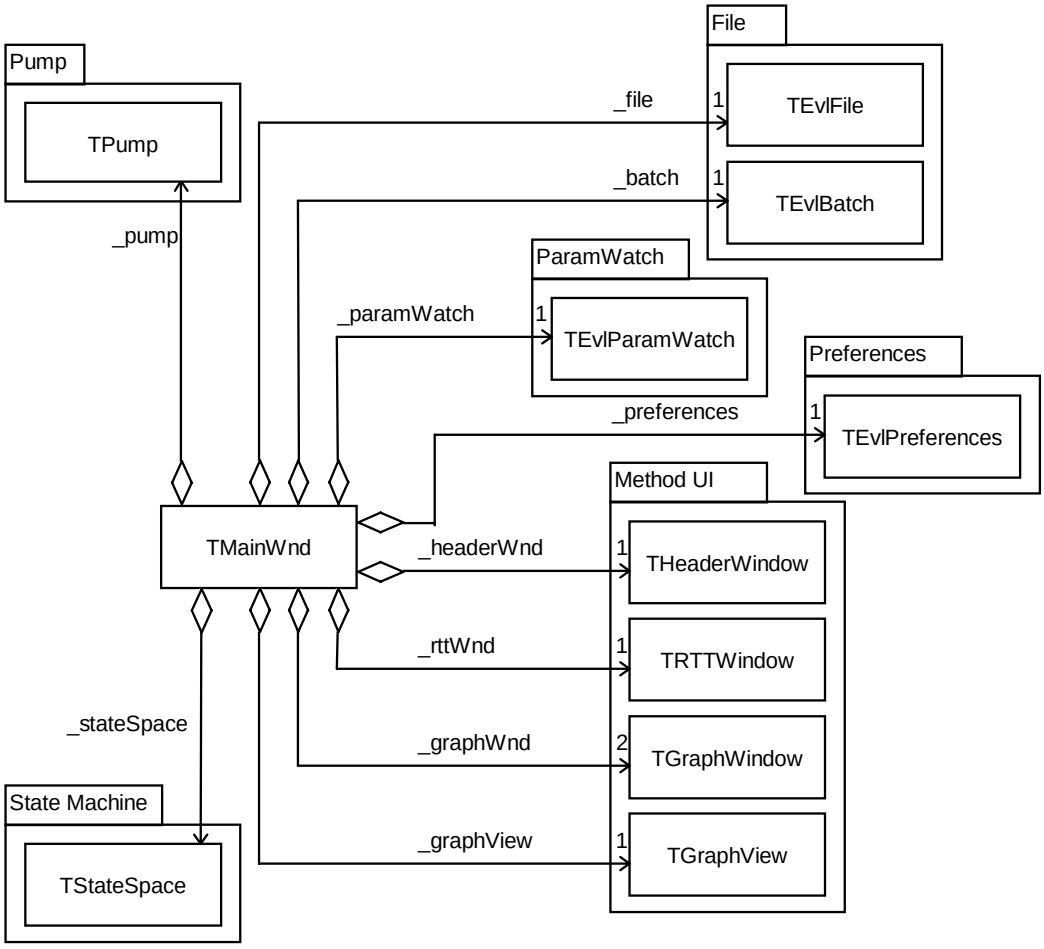


Figure 3.1: Packages Overview. Only classes connected with TMainWnd are depicted here.

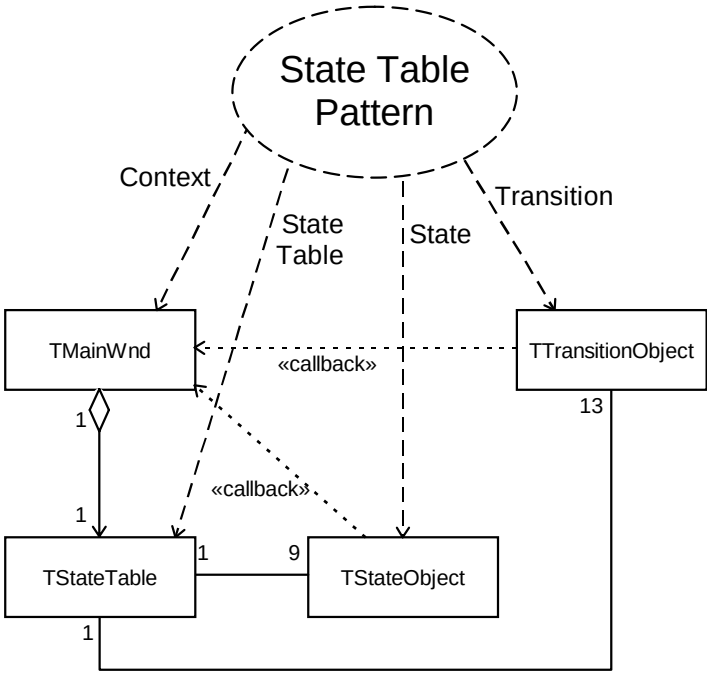


Figure 3.2: State Table Pattern

3.2 *State Machine* Package

I use the *State Table Pattern* as presented by [Douglass98], pp. 286–301, for a 1:1 implementation of the state machine given in Figure 2.3. Figure 3.2 shows the applied pattern.

[Douglass98] gives no example how to implement *History Connectors*. In my state machine, I made use of them twice and thus had to invent some way of implementing them. I used a very straight-forward approach by simply adding two variables (`_manualStateHistory, _pauseStateHistory: TState`) in order to remember the old state before switching to stManual or stPause, respectively. Saving the old state is done within the guard functions of the transitions from "method mode" to stManual and from "running" to stPause. (These are rather "pseudo" guards since they do nothing but this and always evaluate to true.) Restoring the saved state is done within the normal guards of the transitions from stManual to "method mode" and from stPause to "running".

Many of the main window's widgets (toolbar buttons, menu items) are enabled, disabled, checked or unchecked dependent on the *current state*. To simplify the management of setting these properties, there are four constant sets of indexes for these controls for each state.

As an example, the four constants for stManual are:

`MANUAL_ENABLED_CONTROLS` contains the indexes of all visual controls that are always enabled when stManual is entered.

`MANUAL_DISABLED_CONTROLS` contains the indexes of all controls that are always disabled when stManual is entered.

`MANUAL_CHECKED_CONTROLS` contains the indexes of all controls that are always checked when stManual is entered.

`MANUAL_UNCHECKED_CONTROLS` contains the indexes of all controls that are never disabled when stManual is entered.

Analogous constants exist for all other states. Of course the sets of disabled/unchecked controls do not need to be the countersets of the respective enabled/checked control sets, since there are controls that should not change their properties when the state changes.

The actual setting of the controls' properties is done in the *entry action* of each state.

The clear advantage of these constants is that it is easy to define and change them, and all state entry actions are simple and very similar to each other.

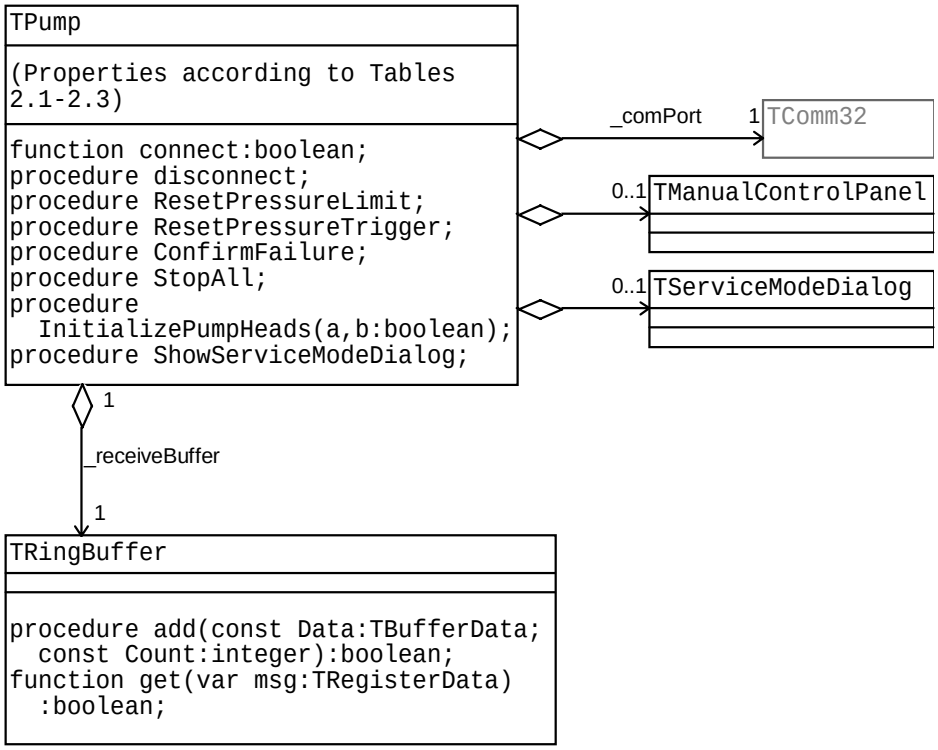


Figure 3.3: Pump Package

3.3 *Pump* Package

The `TPump` class encapsulates the interface to the real pump. It provides access to all relevant control parameters and measured values of the pump (Tables 2.1, 2.2). Furthermore, it has a user interface that provides direct access to the control parameters.

Internally, the register set of the pump (see section 2.1.1 is replicated 1:1 by an array of 16-bit words, `_regs`. All access methods of the pump parameters read and write to/from this array. Whenever a word in the lower part of the array is changed through an access method, it is immediately sent to the pump. In the reverse case, whenever a message is received from the pump, its value is written to the respective word in the upper part of the array. If the value actually did change, the change is propagated through a 'change' callback method. The callback methods are private `TMainWnd` members like the `_pump` object itself.

There are some register numbers that never need to be sent by the software because they either have no function or are only for service purposes. Yet as soon as the pump is connected to the software, all relevant registers have to be initialized (sent to the pump) once. In order to avoid the overwriting of the pump's default

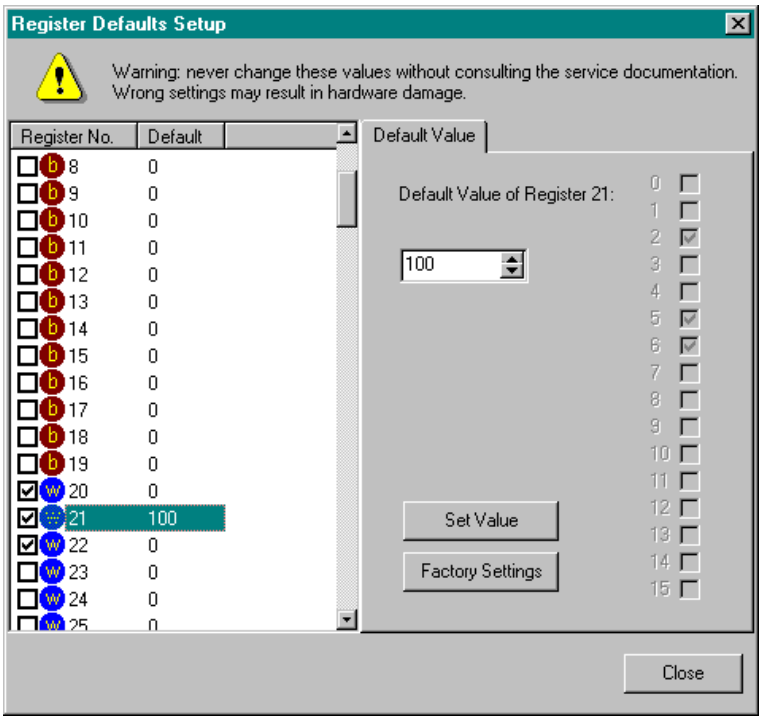


Figure 3.4: Service Mode Dialog

values of some registers with 0, there is a boolean array (`_registersToSend`) that indicates whether a register should ever be sent at all. Further, there is an array of words that contains the *default (initial) values* of all registers that can be sent (`_registerDefaultValues`).

In order to enable a service person to change these default values, there is a *Service Mode Dialog* available upon entering a "service password" (Figure 3.4). By means of this Dialog, the arrays `_registersToSend` and `_registerDefaultValues` can be changed. Their contents are saved in the Windows registry upon destruction and restored from there upon creation of the `TPump` object.

In order to control basic pump operation manually, the *Manual Control Panel* (see [Manual], Concepts | Manual Mode) is displayed upon the invocation of the `TPump.setManualMode` method. It provides access for all pump control parameters that can also be controlled from a method. Its input controls modify the public `TPump` properties only.

3.4 *File* Package

3.4.1 Method Files

The method-file handling subsystem is based on the class `TEvlFile` which can read and write method (`.evl`) files and provides `TEvlData` descendant objects to transport method-specific data and propagate changes in it. Figure 3.5 shows the classes of the *File* Package, while Figure 3.9 shows the `TEvlData` descendants in their functional context.

The user can store his method in a file that behaves just like a document file from a word processor or spreadsheet. The file contains all information given in the Header Window and the Runtime Table as well as a list of past runs of the respective method with references to *Graph Files* (see next section).

The functionality of reading and writing these files is implemented in the `TEvlFile` class.

The saved file itself is an ascii text file with the extension "`.evl`". The first 14 characters of its first line contains some "meta information" about its contents.

Figure 3.6 shows an example of such a first line.

The meaning of the sections 1–7:

1. indicates the file format version, starting with '`A`' for version 1, '`B`' for version 2 and so on.

2. checksum of all bytes of the following lines of the file (all bytes XORed), in hexadecimal notation.

3. a '`.`' at this position means that the check sum should be ignored when the file is read. This is useful for debugging (the file can be manually hacked and the program will not bring up an error message when loading the file). Any other character means that the checksum will be verified. Default is '`x`'.

4. the number of lines in the following that belong to the Method Header (the part that is displayed in the Header Window), in two-digit hexadecimal notation.

5. the number of lines in the following (after the Header-lines) that belong to the Runtime Table, in two-digit hexadecimal notation.

6. the number of lines in the following (after the RTT-lines) that belong to the text notes for the method (which appear in the notepad window), in two-digit hexadecimal notation.

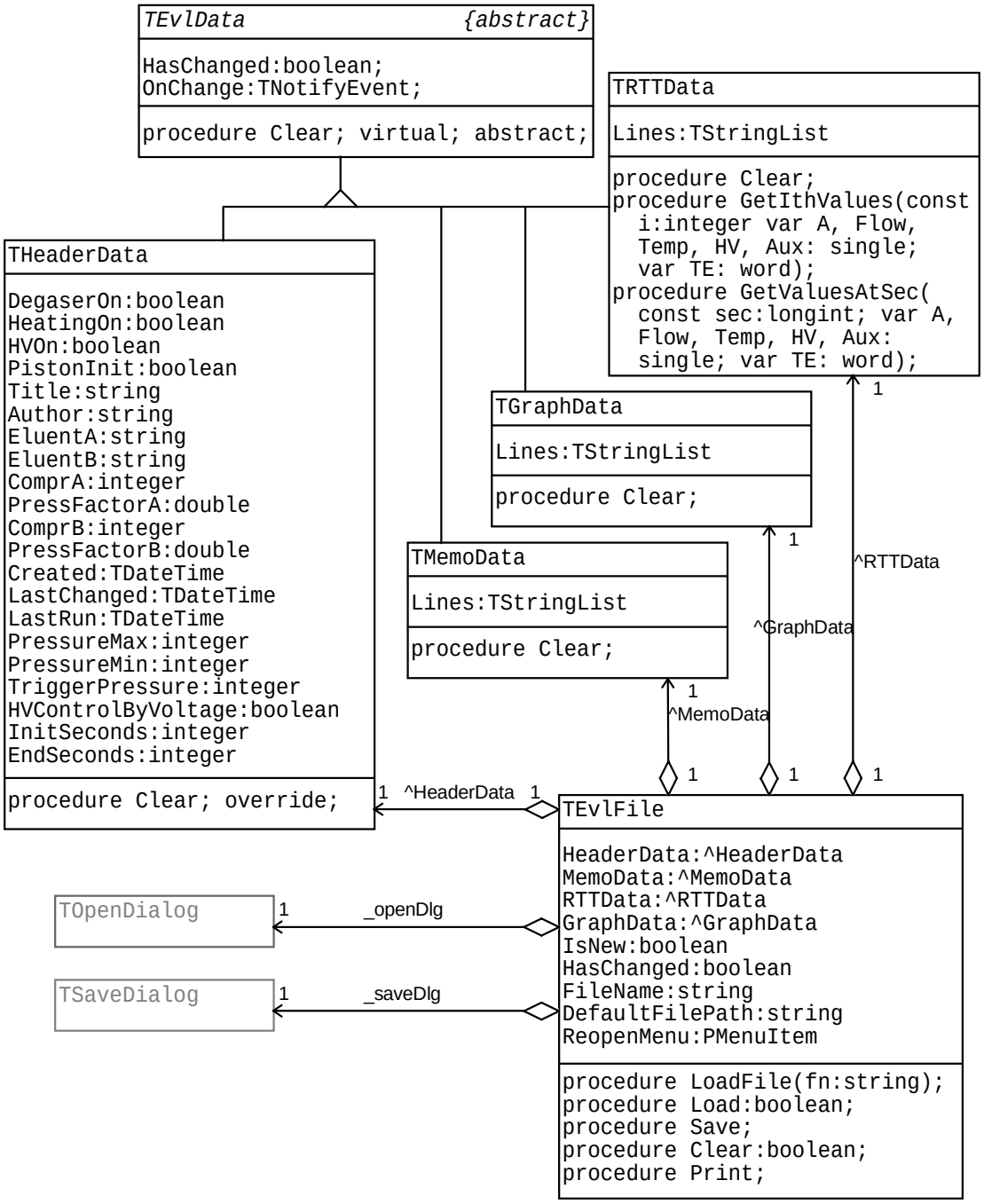


Figure 3.5: File Package: Method-Related Classes

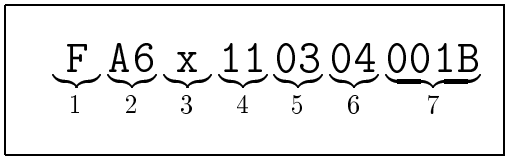


Figure 3.6: First line of a sample method file

7. the number of lines in the following (after the Text-lines) that represent past runs of the method, including date, time, and a reference to a Graph File (see next section), in four-digit hexadecimal notation.

The actual mapping between the ascii lines found in the file and the `TEvlData` descendant data structures is done in the `TEvlFile.LoadFile(const fn:string)` procedure. It contains a `case` statement over the file format version character found in the first line. This way, for every new file format version there is only the need of adding one new case, and the software will always be able to read files from all previous versions.

In case the version character found in a file is higher than the highest case handled in the `case` statement, an exception is raised to properly inform the user ("You need a newer Version of Evolution to read this file").

Of course the files are always *written* in the most current format.

Advantages of the chosen file format are

- plain ascii ⇒ human readable, easy to debug
- disk I/O through built-in methods of `TStringList`
- file format can easily be updated

Method files can also be printed out using `TEvlFile.Print`. This is implemented in a very simple way; it uses Delphi's `TPrinter` class and prints the information contained in `THeaderData`, `TRTTData`, and `TMemoData` as plain text.

3.4.2 Batch Files

Multiple method files that are to be executed in sequence are defined as a *Method Batch* in the *Batch Dialog* (see [Manual], section "Concepts | Method Batches"). The class `TEvlBatch` provides `TMainWnd` with the method file names that are to be executed next. `function TEvlBatch.ShowBatchDialog:boolean` shows the *Batch Dialog* (`TBatchDlg`) in which a method batch can be composed. If this dialog returns with the modal result `mr_OK`, `ShowBatchDialog`'s return value is true, and `TMainWnd` can continue with loading and starting the first method of the batch (`_batch.First; _batch.CurrentMethod`).

```
case fileInfoString[1] of

// ...

'F': with _headerData do
  begin
  DegaserOn       := HeaderStringList.Strings[0][1] = '1';
  HeatingOn       := HeaderStringList.Strings[0][2] = '1';
  HVOn            := HeaderStringList.Strings[0][3] = '1';
  PistonInit      := HeaderStringList.Strings[0][4] = '1';
  HVControlByVoltage:= HeaderStringList.Strings[0][5] = '1';
  Title           := HeaderStringList.Strings[1];
  Author          := HeaderStringList.Strings[2];
  EluentA         := HeaderStringList.Strings[3];
  EluentB         := HeaderStringList.Strings[4];
  ComprA          := StrToInt(HeaderStringList.Strings[5]);
  PressFactorA    := StrToFloat(HeaderStringList.Strings[6]);
  ComprB          := StrToInt(HeaderStringList.Strings[7]);
  PressFactorB    := StrToFloat(HeaderStringList.Strings[8]);
  Created         := StrToFloat(HeaderStringList.Strings[9]);
  LastChanged     := StrToFloat(HeaderStringList.Strings[10]);
  LastRun         := StrToFloat(HeaderStringList.Strings[11]);
  PressureMax     := StrToInt(HeaderStringList.Strings[12]);
  PressureMin     := StrToInt(HeaderStringList.Strings[13]);
  TriggerPressure := StrToInt(HeaderStringList.Strings[14]);
  InitSeconds     := StrToInt(HeaderStringList.Strings[15]);
  EndSeconds      := StrToInt(HeaderStringList.Strings[16]);
  end;
else raise Exception.Create('You need a newer Version of '+
          +'Evolution to read this file');
end;
```

Table 3.1: Mapping between `.evl` file and `THeaderData` object

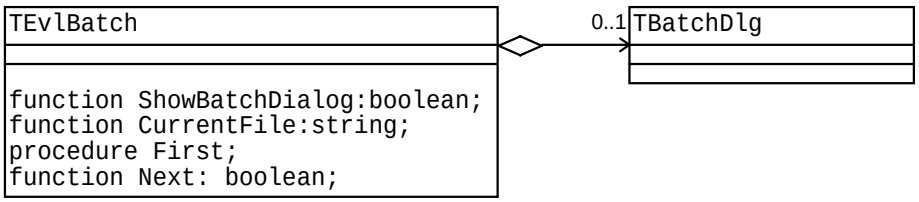


Figure 3.7: File Package: Batch-Related Classes

```
C:\Program Files\LabSource\Evolution\methods\Prepare.evl
1
C:\Program Files\LabSource\Evolution\methods\SampleMethod1.evl
15
C:\Program Files\LabSource\Evolution\methods\Wash.evl
1
```

Figure 3.8: Sample method batch file

The *Batch Dialog* also has a menu allowing to save a once defined method batch to disk (or read it from there). The saved files' name extension is "`.evb`". The file structure is very simple: The first line contains the file name of the first method file in the batch, the second line contains the number of repetitions. This scheme is repeated for all methods in the batch.

Figure 3.4.2 shows a typical example of such a file.

3.4.3 Graph Files

Graph files are saved at the end of each run if the preferences option "Save Graph" is checked. These files, ending with "`.evg`", are written by the `WriteToFile(fn:string)` method of the (third-party) `TXYGraph` component (see sections 3.5.3, 4.3). They contain all data that is displayed in the graph, especially all measured values that were "recorded" during one method run. The actual format of these files is not of interest in this place, since all we need to do with them is reading them into a `TXYGraph` object later using its `ReadFromFile(fn:string)` method. This is so done in the `TGraphView` class (see section 3.5.4).

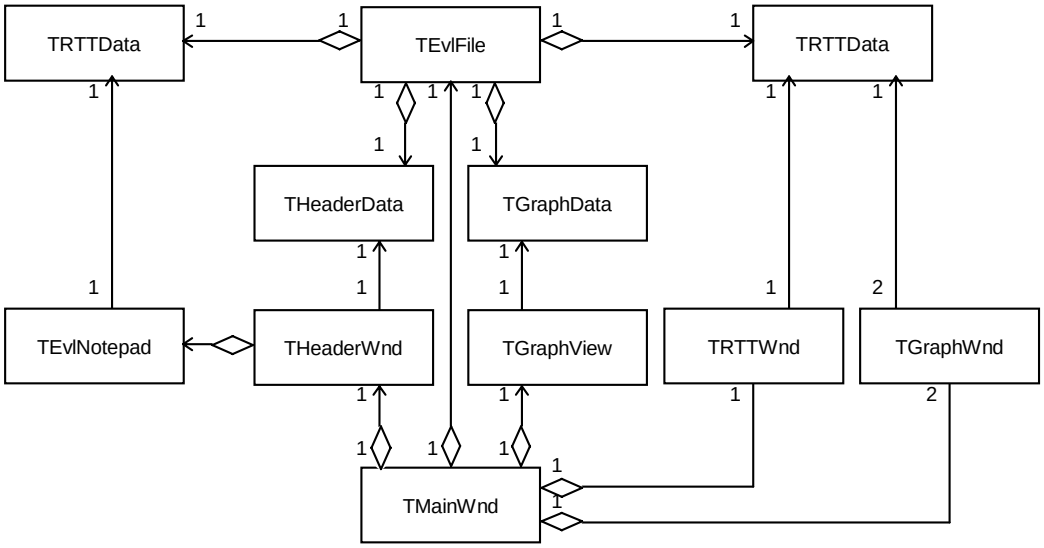


Figure 3.9: `TEvlData` Descendants and Method User Interface Classes in their Relationship

3.5 *Method UI* Package

An overview of the method user interface classes in their relation to method-data containing classes is given in figure 3.9.

3.5.1 Header Window

The *Header Window* (see [Manual], User Interface | Windows | Header) is the user interface of the `THeaderData` class which is instantiated by `EvlFile` (see section 3.4). The header window holds a pointer to this object and displays its values in a grid. Changes are propagated immediately in both directions.

The *Header Window* also contains an instance of `TEvlNotepad` which serves as user interface for `TMemoData`.

3.5.2 RTT Window

Analogous to the *Header Window*, the *RTT Window* (see [Manual], User Interface | Windows | Runtime Table) is the user interface for `TRTTData`. It displays the runtime table as a list of strings which can be edited with a special editor by double-clicking a line (see [Manual], User Interface | Windows | RTT add/edit Dialog).

3.5.3 Graph Windows

The two *Graph Windows* (see [Manual], User Interface | Windows | Graphics Windows 1/2) are read-only user interfaces of the runtime table (`TRTTData`). They contain "`TXYGraph`" chart components to display the information of the runtime table in form of curves.

In addition, measured values can be displayed as overlay curves during the run time of a method. For documentation purposes, these curves can automatically be saved to a file at the end of each method run. The saved files can be viewed with the *GraphView Window* (see next section).

`TGraphWnd` is instantiated *twice* as `TMainWnd._graphWnd1` and `TMainWnd._graphWnd2`. The parameters actually displayed are determined in the *Preferences*. They are set within the `TMainWnd._preferencesChange` method. Users can freely select subsets of the available parameters to be displayed in either of the windows.

3.5.4 GraphView Window

This is the user interface of `TGraphData` which contains a list of strings divided into two columns by a tab (#9) character. The strings contain the date of a run of the method, followed by a file name which specifies a *Graph File* (see section 3.4.3) that was saved at the end of the respective past method run.

The GraphView Window (see [Manual], User Interface | Windows | GraphView) displays the first column of these strings in a `TListBox` in the left. In its `OnClick` event handler, the file indicated in the second part of the selected string is loaded into the `TXYGraph` component on the right.

The data stored in the Graph Files can also be exported to a comma-separated values ascii file. This is done using the `TXYGraph`'s `bla` method.

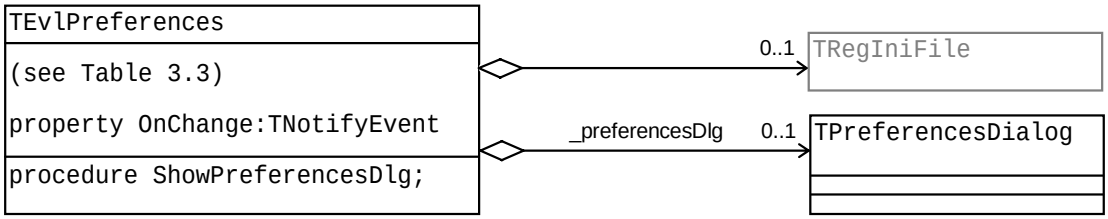


Figure 3.10: Preferences Package

3.6 *Preferences* Package

The user is allowed to set a variety of preferences in the program. These preferences are stored in the Windows Registry Database and will be retrieved from there every time the program is started.

This functionality is provided by the `TEvlPreferences` class, which provides an interface to the Windows registry database as well as a user interface (see [Manual], User Interface | Preferences | Preferences Dialog). It reads the values from the registry upon creation and writes them back in its destructor. In the meantime, it makes them available in its published properties. When the `TPreferencesDialog` is instantiated by `TEvlPreferences.ShowPreferencesDialog`, its input widgets are set to these values. When it gets closed through an "OK" button click, the values are written back to the `TEvlPreferences` member variables before the dialog is destroyed.

Figure 3.11 illustrates this process.

3.7 *ParamWatch* Package

This package allows the observation of measured values at any time by means of small windows displaying a "rolling" curve (see [Manual], User Interface | Windows | Parameter Watch Windows). There is one such window for every parameter, and the user can choose which of them to display at any time. This is especially useful if one wants to observe these values while no method is running, e. g. for method development in Manual Mode.

The `TEvlParamWatch` class has a timer object which invokes the `OnGetValues` event callback method every second. In case there is no callback method assigned to `OnGetValues`, an exception is raised. When the callback method returns, for every instance of `TParameterWatchWnd` in the array `_watchWindows`, the window's `TCurveWatch` object is told the new value by means of its `NewSecondValue` method.

This method shifts the displayed curve one pixel to the left and draws a new

```
// General
PressUnit:string
VoltageLimit:integer
CurrentLimit:integer
Password:string
PasswordEnabled:boolean
RepeatAtEnd:boolean
SaveAtEnd:boolean
NeverStopPump:boolean
Timeout:TTimeoutAfter
LoadTimeoutMethod:boolean
TimeoutMethod:string
TE1AliasName:string
TE2AliasName:string
TE3AliasName:string
TE4AliasName:string
TE5AliasName:string
TE6AliasName:string
GraphViewerName:string
// Communication
ComDevice:integer
// Graph
AColor:TColor
BColor:TColor
PressColor:TColor
FlowColor:TColor
TempColor:TColor
VoltageColor:TColor
CurrentColor:TColor
AuxColor:TColor
BackgroundColor:TColor
ShowAB1:boolean
ShowPress1:boolean
ShowFlow1:boolean
ShowProgTemp1:boolean
ShowActualTemp1:boolean
ShowProgHV1:boolean
ShowActualVoltage1:boolean
ShowActualCurrent1:boolean
ShowAux1:boolean
ShowAB2:boolean
ShowPress2:boolean
ShowFlow2:boolean
ShowProgTemp2:boolean
ShowActualTemp2:boolean
ShowProgHV2:boolean
ShowActualVoltage2:boolean
ShowActualCurrent2:boolean
ShowAux2:boolean
// Method defaults
DefaultAuthor:string
DefaultMaxPressureLimit:integer
DefaultMinPressureLimit:integer
DefaultTriggerPressure:integer
DefaultInitTime:TDateTime
defaultEndTime:TDateTime
// Directories
FilesPath:string
// Toolbars
ButtonDisplayMode:TButtonDisplayMode
FlatToolButtons:boolean
// GraphView
ShowPressGV:boolean
ShowFlowGV:boolean
ShowProgTempGV:boolean
ShowActualTempGV:boolean
ShowProgHVGV:boolean
ShowActualVoltageGV:boolean
ShowActualCurrentGV:boolean
ShowAuxGV:boolean
```

Table 3.2: Properties of `TEvlPreferences`

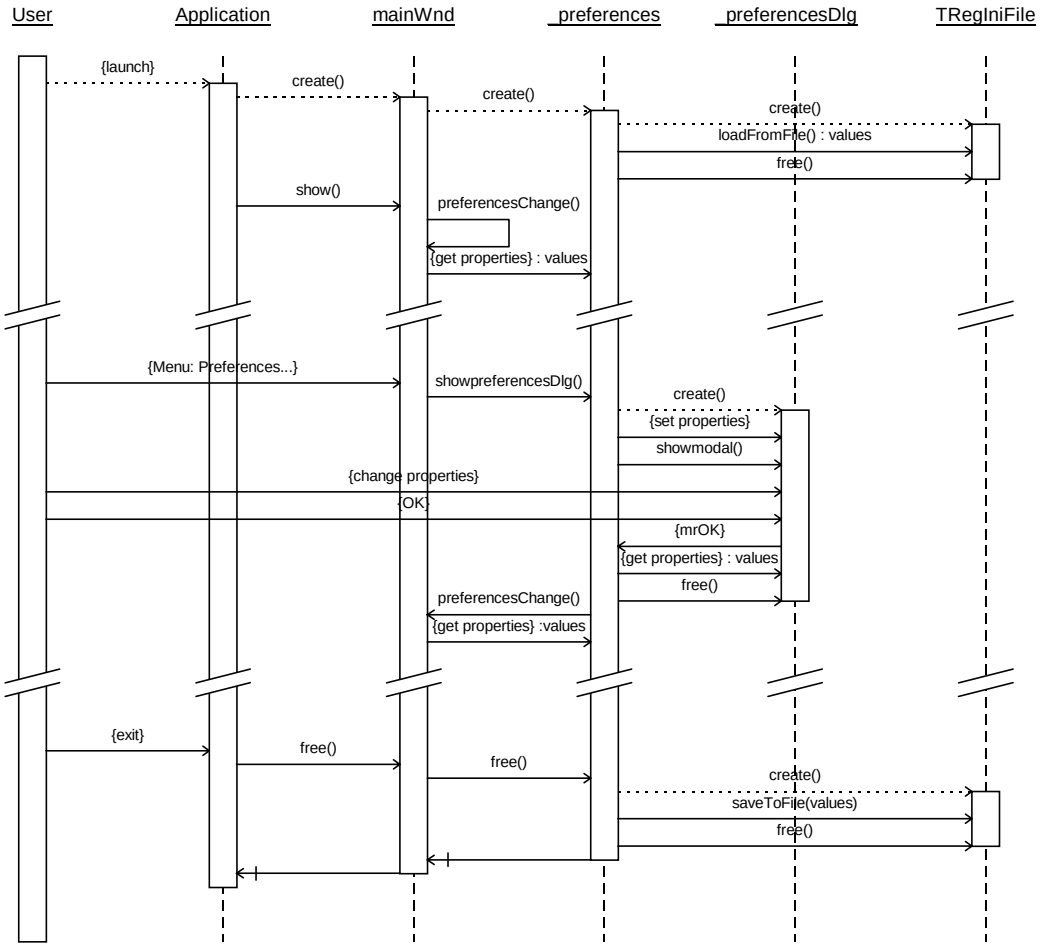


Figure 3.11: Preferences Sequence Diagram

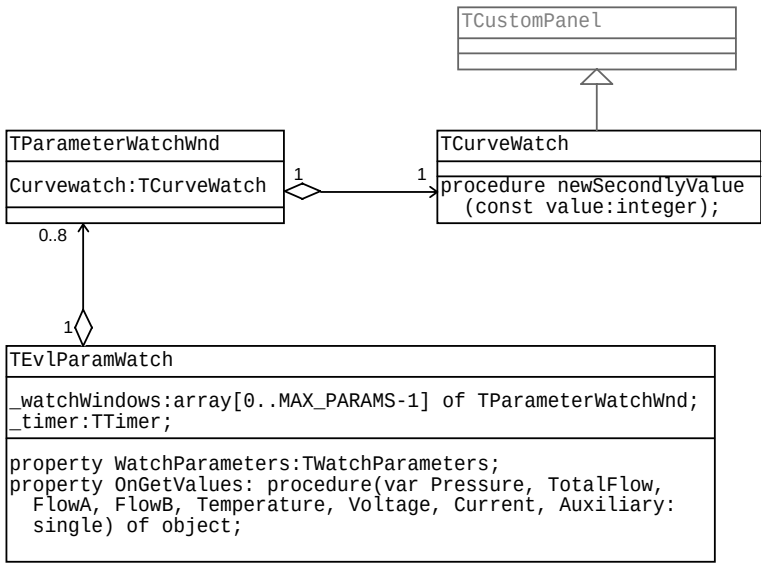


Figure 3.12: ParamWatch Package

curve segment to the new value.

This way all displayed “parameter watch windows” are updated every second.

The windows are dynamically created in the set method of `TEvlParamWatch.WatchParameters` and put into the `_watchWindows` array.

`TEvlParamWatch.WatchParameters` is updated by `TMainWnd` when the user selects form the main menu which parameters he wants to observe.

	User Interface	Method Timer
stManual	•	
stNoRTT	•	
stReady	•	
stPistonInit		•
stInitializing		•
stInitialized		
stRunning		•
stEnding		•
stPause	•	

Table 3.3: Threads Controlling `TPump` Properties

3.8 Concurrency Issues

There are two different threads that can write to the public members of the (only) `TPump` instance `TMainWnd._pump`: the *user interface* and the *method timer*. The latter is instantiated in the "entry action" procedures of initializing or piston init and then fires an event each second.

Its event-handling procedure does the following:

- It retrieves the target parameter values depending on `_runtimeSecond` from the `HeaderWnd` through interpolation between the time points given in the runtime table.
- It sets the members of `_pump` to the appropriate values, depending on the current state.
- It either decreases or increases `_runtimeSecond`, depending on the current state: while initializing or ending, it counts down the respective amounts of seconds, while it counts upwards when the method is running. The value of `_runtimeSecond` is always displayed in the upper right corner of the main window in hh:mm:ss format.

The `_methodTimer` thread gets destroyed again in the entry action of the ready state.

Considering table 3.3 which indicates which of the two threads has write access to `TPump` properties in which states, we can see that there is no danger of inconsistency through concurrent writing of two threads to the same object because no state allows more than one thread to change the pump properties.

The pump object is the only object that is written to from more than one thread in the whole application.

Chapter 4

Implementation Issues

4.1 Coding Conventions

- Names of *Classes* always begin with `T`, for they are declared as ‘types’. This is a Delphi convention.
- Names of *Pointer data types* begin with `P`.
- Names of *private class members* begin with an underscore (_).
- Names of *constants* are writen in uppercase.
- All names are meaningful, even if this results in long names. Exceptions are local counters which are called `i`, `j`, `k`, `m`, or `n`.

4.2 Version Numbering

Version numbers consist of three parts separated by two points, meaning “major version”.“minor version”.“build”. The current version number is 1.5.3.

The “build” number is increased for small updates (bugfixes or small changes that don’t require the user to learn a new behaviour of the software), while “minor version” and “major version” are increased with small/heavy changes which are important for the user to know.

4.3 Third-Party Software

Table 4.1 contains all third party software that is needed to compile the software.

Name	Vendor	Used in
Delphi 3 Professional (IDE, compiler, components)	Borland/Inprise	
Async32 (RS-232 component)	Varian	TPump
XYGraph (Graph plotting component)	Kestral	TGraphWnd
MemProof	A. Stoyanov	

Table 4.1: Third-Party Delphi Components

4.4 Testing

4.4.1 KT

Because I didn't have a real pump at home where I mostly developed the communications interface of `TPump`, I wrote a small program that can play the role of the pump when running on a separate PC connected with a serial cable to the PC running the Evolution software. It is very simple; its only window is depicted in Figure 4.1. It allows you to directly edit the values of its 255 16-bit registers by hand and send them to the connected PC using the pump protocol. Of course it can also receive values and display them in the same way.

On the other hand this program can also be used as a substitute for the software, since the protocol is entirely symmetrical. Thus it can be used to manually test the individual pump functions.

4.4.2 MemProof

The freeware tool *MemProof* (fig. 4.2) is an excellent memory/resource allocation sniffer that helps to easily find memory leaks and other bugs that would be hard to detect without. It helped a lot to make the software free of memory leaks.

4.5 User Documentation

I used the help authoring software "Help & Manual" by EasyCash Software to write the user documentation. It allows the writing of Windows help files in an easy (WYSIWYG) way and offers convenient export possibilities for paper and HTML versions.

The HTML version of the user documentation is available on-line on [Manual].

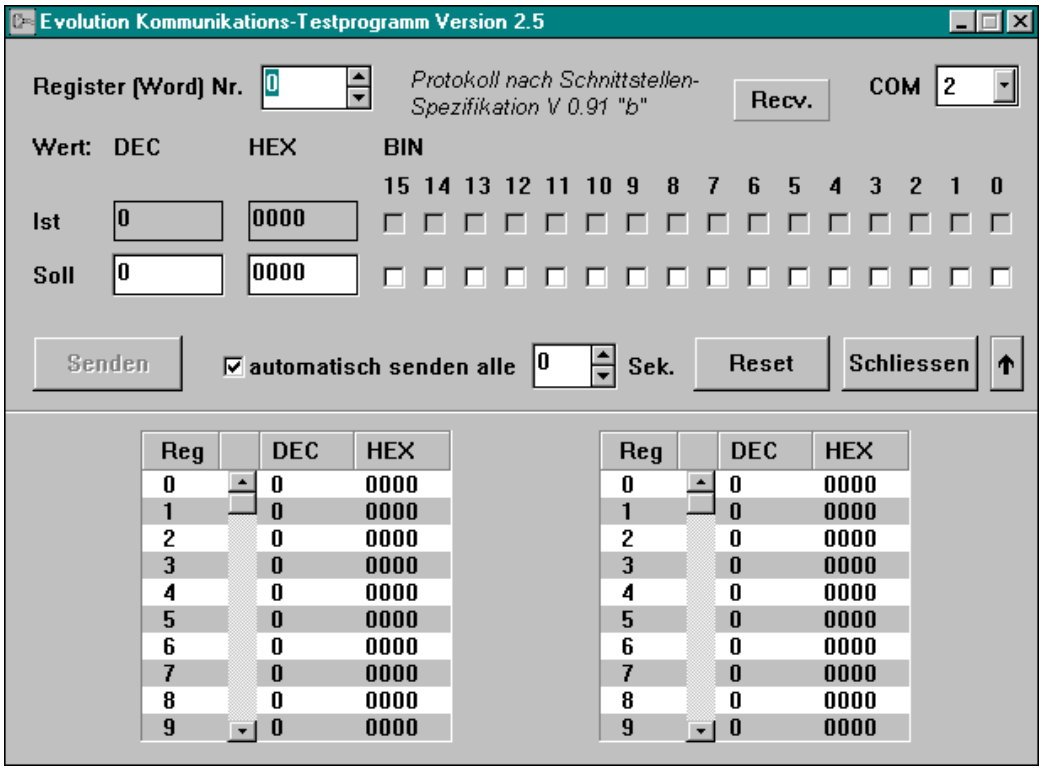
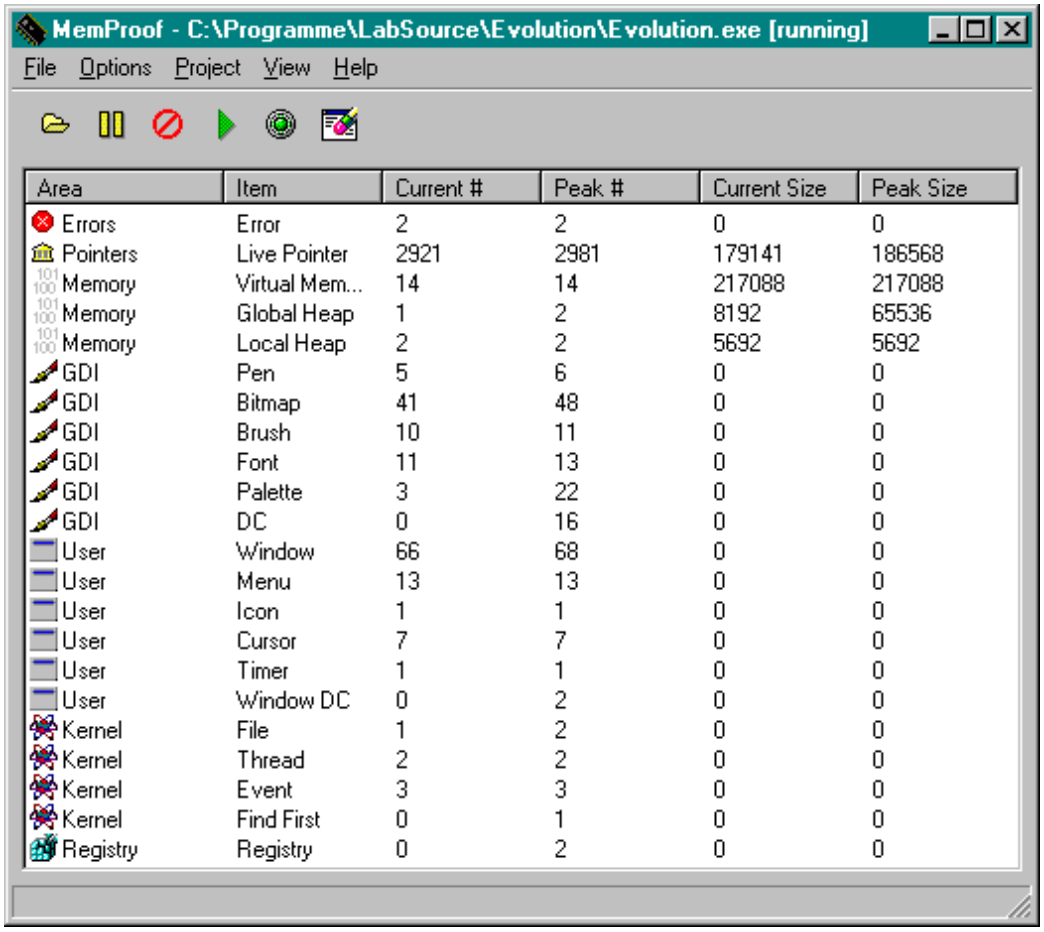


Figure 4.1: Simulating the pump: “Kommunikations-Testprogramm”



Area	Item	Current #	Peak #	Current Size	Peak Size
Errors	Error	2	2	0	0
Pointers	Live Pointer	2921	2981	179141	186568
Memory	Virtual Mem...	14	14	217088	217088
Memory	Global Heap	1	2	8192	65536
Memory	Local Heap	2	2	5692	5692
GDI	Pen	5	6	0	0
GDI	Bitmap	41	48	0	0
GDI	Brush	10	11	0	0
GDI	Font	11	13	0	0
GDI	Palette	3	22	0	0
GDI	DC	0	16	0	0
User	Window	66	68	0	0
User	Menu	13	13	0	0
User	Icon	1	1	0	0
User	Cursor	7	7	0	0
User	Timer	1	1	0	0
User	Window DC	0	2	0	0
Kernel	File	1	2	0	0
Kernel	Thread	2	2	0	0
Kernel	Event	3	3	0	0
Kernel	Find First	0	1	0	0
Registry	Registry	0	2	0	0

Figure 4.2: Finding memory leaks with MemProof

Chapter 5

Personal Experience

Over all, I enjoyed this job very much. I especially liked to work in close cooperation with engineers of other fields than computer science and have insight to the whole development process of the pump itself. Also I was lucky enough to work together with people who were interested in my own ideas and propositions rather than forcing me to implement what they initially imagined how the software would look like.

This was my first 'big' software project, and thuse I learnt a lot of very basic things with it. Most important: it always takes more time than you expect. After quite some experience with software programming, I still make the mistake of planning in too little time.

Another important experience is that things will always change sooner or later in unexpected ways. If you have to decide whether to implement something in a general and flexible way or to do it more straight-forward, the general and flexible implementation will almost certainly pay off later when it comes to requirement changes. Of course this sounds trivial, but I experienced so many unexpected changes that I learnt to always follow this principle even in situations where I can not think of an actual use.

Of course there are situations where you have to deliver a change as quickly as possible. In such situations it is often better to modify the software as straightforward (and fast) as possible in order to satisfy the customer and afterwards do it "right" when you have more time.

5.1 4GLs vs. traditional Languages

Initially I chose Delphi because I already knew TurboPascal for DOS, and so Delphi was the easiest way for me to start developing Windows applications. When I learnt more about the object oriented paradigm later, I began to realize that there

are some differences between the "good" way of OO programming and the way of programming Delphi implicitly propagates.

This has to do with the concept of "components". Components are normal Delphi classes that inherit from `TComponent` instead of `TObject`. The advantage of components over simple objects is that they can be made available in Delphi's "component palette" and from there can be added to a form using just the mouse. As soon as a component is added to a form in the form designer, it already gets instantiated at design time. Thus you can see the effects of different parameters (published class member's values) at design time already.

In the following I want to give two examples of this concept's drawbacks.

The UI designer makes all components public. If one uses Delphi's UI designer to populate forms with components, these components become public class members of the form. This may be convenient to use when one's developing UI prototypes or really small applications, but when it comes to "real" coding, many of these components should be declared private.

If one wants to declare anything as private, one is stuck to instantiating and initializing it by hand, and the convenient UI designer can't be used.

Components exist as long as the application runs. Another drawback of adding components to forms "by mouse" is that these components get created/destroyed as soon as their underlying form is created/destroyed.

For most *visual* components this is exactly what one wants normally. Yet for many nonvisual components dynamic creation and destruction would make more sense.

The same thing applies to forms as such: by default Delphi creates all forms upon the start of the application, even if they are not immediately shown. Here again it makes more sense to avoid automatic creation and instantiate the forms dynamically in the moment before they are shown. Of course one should also destroy them as soon as the windows get closed.

This is not a serious problem since one can easily do without adding *nonvisual* components to a form with the form designer. Also can forms easily be prevented from being instantiated automatically upon program start.

In the development process of "Evolution 200", I started with prototypes of the user interface early. This was greatly simplified by the RAD mechanisms of Delphi.

While implementing new features and refactoring old code I began more and more to dynamically instantiate components and make them private. In the current release, all windows are created and destroyed dynamically, and only visual components on forms are still automatically created.

This is what I consider the best way of well implementing an object-oriented design while still taking some advantage of Delphi's "4GL" qualities.

Bibliography

[Cooper95] Cooper, Alan, *About Face: The Essentials of User Interface Design*. Foster City, California: IDG Books Worldwide, 1995.

[Borland97] *Delphi 3 Object Pascal Language Guide*. Scotts Valley, California: Borland International, 1997.

[Doberenz95] Doberenz, Walter, and Thomas Kowalski, *Borland Delphi*. München; Wien: Hanser, 1995.

[Douglass98] Douglass, Bruce Powel, *Real-Time UML: Developing Efficient Objects for Embedded Systems*. Reading, Massachusetts: Addison Wesley Longman, 1998.

[Manual] Nemec, Bernhard, *Evolution 200 User's Manual*. Reinach: LabSource/ProLab GmbH, 1999.
`http://iamexwiwww.unibe.ch/studenten/nemec/manual/evodoc.html`

[Rudin97] Rudin, Lucas, *Evolution Schnittstellen-Spezifikation*. Reinach; Muttenz: LabSource/Hans Meyer Engineering AG, 1997.

[Rational97] *UML Notation Guide, Version 1.1*. Santa Clara, California: Rational Software Corp., 1997.