Command Based Interpreter and Developer's Library COMPEL

By

Elias Bachaalany

A Senior Project Submitted in Partial Fulfillment of the Requirements for the Degree of Bachelor of Science

AMERICAN UNIVERSITY COLLEGE OF SCIENCE & TECHNOLOGY

Approved by the Computer Science Department and the Supervisory Committee

Program Authorized to Offer the Degree of Bachelor of Science in Computer Science

11/11/2006

We certify that we have read and tested this project and that, in our opinion, it is satisfactory in scope quality as a senior year project for the degree of Bachelor of Science in Computer Science.

Project Committee

Aziz M. Barbar, Ph.D.

Elie Nasr, MS.

Antoine Aouad, MS.

Michel Owayjan, MS.

Acknowledgments

In this short space, we would like to take the opportunity to thank all those who helped and contributed to the making of this project. We would like to acknowledge Dr. Aziz M. Barbar, Mr. Elie Y. Nasr, Dr Majd F. Sakr, Mr. Michel Owayjan, Mr. Antoine Aouad, Mr. Jimmy Badawi, Mr. Ilfak Guilfanov and the rest of the American University College of Science and Technology staff behind the stage, thank you all.

Abstract

Graphical User Interface (GUI) applications are a mark of professionalism in today's market. When using GUI programs the user's attention is required in order to carry a specific set of actions and this can be cumbersome especially if the commands are always repeated, or require little or no intelligent user intervention. This project offers a mechanism to the application developers allowing them to enrich their GUI applications by providing their end-users a facility to script (or type) the required commands. COMPEL provides a scripting engine library for developers, making it easy, feasible and convenient to incorporate scripting features into a GUI application turning it into a scriptable and extensible application. Consequently, COMPEL’s engine implementation proved the effectiveness and flexibility achieved through a modular and hierarchical design.

Table of contents

Table of Figures

List of Tables

Chapter 1
Introduction

Outline

In the past, computer programs were not always user friendly in the sense that one had to be computer literate in order to operate a simple program by commanding the system what to do using special commands that are typed in the shell prompt. With the advance of computer sciences, a new concept of working with the PC was introduced" Graphical User Interface" or "GUI" in short.

Shortly after the introduction of GUI and event driven programming, backed by the operating system, programmers began designing interfaces for their applications in a user friendly manner. Nowadays GUI programming is the norm, completely the opposite of what was 20 years ago. Though GUI programming makes it fun and simple to operate computers, it can impose limitations, decrease the creativity and speed of certain system users (such as administrators, programmers, power users). COMPEL, in short, is a command based interpreter and developer's library. It allows application developers to equip, in addition to their Graphical User Interface, their applications with a Command Line Interface (CLI), so that users can type-in the commands or feed the application a complete set of instructions (through a script file) to be carried of automatically.

1.1 - Objectives

COMPEL's primary objective is to tackle the lack of flexibility of today's applications (few are the applications that provide GUI and CLI at the same time) and to provide applications developers a library to facilitate the integration of CLI and scripting abilities within their systems.

1.2 - Project Description

COMPEL as defined in a dictionary literally means "Make someone do something". The project's name is carefully selected to denote that the syntax used is a command rather than a full syntax like other programming languages. COMPEL is a command based interpreter, that is, when it receives a command it interprets it accordingly, the name also rhymes with the world compiler, and borrows some of the compiler constructions techniques.

COMPEL is a double faced project, the first part is a developer library, facilitating the CLI and scripting facilities integration within one's application. While the second part allows COMPEL, when used with its integrated development environment (IDE), to be a simple standalone command based programming language with a very straightforward syntax.

The developer library is a set of APIs that programmers can use while writing their software, so that they can allow their application to accept commands either typed by the user or commands read from a file (script file). The programmer is saved from the burden of reading the input file and analyzing the commands and interpreting them. The developer's task is then trimmed down to just two steps, the first is to implement the function (or command) from the high-level programming language and the second step is to bind the function with COMPEL's scripting engine.

When COMPEL is used as a standalone interpreter, through COMPEL's IDE, serves as an unsophisticated programming language well suited for teaching programming concepts such as allowing novice users to easily grasp programming concepts, creating a program from scratch, translating a thought or idea into a program, locating bugs and errors using the integrated debugger, using breakpoints and watches to control the program's flow and least but not last how to execute an instruction at a time (single stepping) in a complex program in order to assimilate its overall function.

The following chapter will introduce the problem and show the proposed solution. Chapter Three explains the system's functional specification, overview of the system's engine and the function performed by COMPEL. Chapter Four goes into the details of the project analysis phase from the development model used to the system's components architecture. Chapter Five delves into more details by showing more technical aspects of the system and the choice of the implementation programming language. Concluding by Chapter Six which presents the results and proposes future work plan for COMPEL.

Chapter 2
Problems and Solutions

Outline

Applications usually ship with GUI or CLI but rarely with both versions. There are no technical restraints as to why they should be either GUI or CLI. The most common reason is that to program a GUI or a CLI, the programmer is required to have a different set of skills. Usually GUI programming is easier especially when RAD tools are employed during development [4].

2.1 - Problem Statement

In order to comprehend COMPEL's dual functionalities and solutions, let us illustrate through simple scenarios the problems faced in real life and their solutions when COMPEL is implemented.

2.1.1 - Scenario I - COMPEL Developer's Library

You are a developer, writing architecture software (something like AutoCAD). After you design and develop your application and now it works fine through the menus and the mouse. User can now select the tool and draw shapes on the screen.

After a while your end-users and customers start to complain, and asking for features like how to avoid repeating a given task over and over, or ask if the application can be extended through the use of plug-ins and power users or system administrators start to ask if it is possible to write scripts and get them interpreted by the application in order to automate some tasks. You look back at these suggestions and begin to feel frustrated, since you already developed your application and spent a considerable time fine tuning it and making sure it is the best of its kind, only to look and see that your work is not appreciated and you are required to code more features and spend more time and money during extra development. Now if we look at this scenario, we can realize that indeed the developer has to spend more time coding a CLI and adding scripting facilities to his application. This is when COMPEL developer library comes to the rescue. COMPEL library is designed in such a way, that a finished solution though it was not designed to work with a system such COMPEL, allows a developer to easily integrate this library and address all of the user’s requirements stated above.

2.1.2 - Scenario II - COMPEL as standalone Tool

You are an instructor in a university teaching computer programming, you are required to introduce programming concepts to your students. Naturally, the choice of selection falls on high-level languages such as C++, Pascal or Java. You realize that your students, who were never exposed to programming, find it difficult to grasp these languages. Students even find it incomprehensible that they have to memorize a lot of syntax in order to command the computer to do a simple action. You begin to face problems such as the inability of inexperienced students to easily find suitable C++ compilers, facing problems when configuring the compiler, writing wrong syntax and becoming unable to test programs. Also students might face limitations and feel unable to create something new other than what they learned in class. You as an instructor, having tried your best to introduce programming concepts, you still failed to appropriately convey your message to the students because what is clear for you is totally new for them. This is when you were introduced to COMPEL IDE, and you realized how simple the syntax is and how one can translate his thoughts into programs without all the hassle faced by complex high level programming languages, especially when one is a beginner.

2.3 - Proposed Solutions

COMPEL library is not one of its kind. Many scripting engines and libraries can be integrated into one’s program to provide similar functionalities as COMPEL, however some languages are based on high-level languages (such as Pascal Script, JavaScript, VBScript, …), some are not specific for command based language, and most importantly they are not very simple to be extend or implemented within user’s applications. COMPEL IDE is also not new since all computer languages come with an IDE, however most professional IDEs are not for free; they have lots of functions that are not needed by beginners and are so big in disk space terms and need an expert to operate them and sometimes to properly install them.

2.4 - Our Solution and Target Audience

COMPEL developer library is designed to save programmers the time needed to develop their own command based scripting engines. Also through COMPEL’s library’s extensive API one has to learn just a little in order to use the library. While COMPEL IDE will help instructors teach programming concepts in a simpler manner with fewer constrictions imposed by the syntax of high-level programming languages. Thus when COMPEL API is used, one saves time and benefits from the many features provided by the COMPEL engine and extensible API.

Since COMPEL is a two faced project, the target audience and needed skills differ. When COMPEL Library (API) is used in 3rd party applications, the user is expected to be a developer, with the ability to read documentation about 3rd party APIs and be able to integrate and program this API in his own software. On the other hand, when COMPEL is used as a standalone program, COMPEL turns into a powerful tool for teaching programming concepts, thus we need an instructor or a person that understands COMPEL’s syntax and can then teach programming concepts to others through the help of COMPEL IDE.

In conclusion, having stated the problem and an outlook of the solution that COMPEL provide, we will now head into the next section talking about the functional specification of the system, how it is engineered and how it works.

Chapter 3
System Functional Specifications

Outline

This chapter discusses COMPEL's system functional specifications, what functions are performed and in brief how they are performed. This chapter serves a good introduction for Chapter Four which talks in more details about the project's architecture and functionality.

3.1 - COMPEL Engine Overview

COMPEL engine is the core of the project. The engine is able to interpret single commands and complete script files and because the engine can do this, a high-level library was built on top of the engine to wrap all the functionality into a set of APIs. After the library was developed, an IDE was built on top of it. This design clearly shows the hierarchical nature of COMPEL and its internals. The following section will dissect the parts of COMPEL.

3.2 - Functions Performed

This project performs different functions based on what aspect of the project is used. We will cover all functions covered by the library which is the part that allows developers to extended their applications and give scripting abilities and then by the IDE part which is supposed to be a programming teaching tool.

3.2.1 - COMPEL Library Functions

COMPEL Developer library provides a set of application programming interfaces needed to control and program COMPEL engine. The following table illustrates the functionality provided by the library.

Table 3.1 - COMPEL Library Functions

Function Name	Function Description
Variable manipulation	A set of functions that allows the developer to create, update and delete variables
Object manipulation	A set of functions allowing the developer to create, update and delete objects and their corresponding attributes
User command registration	A set of functions allowing the registration of user commands into the COMPEL engine. By that, new commands become recognizable by the engine
Script Flow manipulation	A set of functions that allow developer to control the flow of the engine execution:

	- Run - Pause - Set executing line - Run one script line (step)
Script Lines manipulation	A set of functions allowing developer to: - Load more script lines into the engine - Load more lines from an external files - Clear script lines
Engine Internals	A set of functions allowing you to query the engine internals such as: - Retrieving the whole symbol table - Retrieving the list of built-in extensions
Utility functions	A set of miscellaneous that are for helping purpose such as: - String to number manipulation - String tokenizing functions - Script initialization and de-initialization routines

These are the basic functionalities covered by the developer library. They are powerful functions as they allow the developer to manipulate almost every aspect of the engine.

3.2.2 - COMPEL IDE Functions

COMPEL Integrated Development Environment (IDE) is a standalone tool that is built on top of the COMPEL developer library. The IDE provides most of the functionalities provided by today's IDEs. In the following section we will describe the major functionalities provided by the IDE.

3.2.2.1 - Per Project/Per Run States Workspace

Per project workspace means that every project has its own settings and configuration, such as windows dispositions, watch window variables and breakpoints list. Whereas the per-run state workspace means that each IDE state has its own settings as well. For example when the project is running, a given workspace is saved and restored then when the project is being edited, another workspace is applied.

3.2.2.2 - Editor

An editor window is provided, allowing the user to develop the script from scratch. The editor has automatic code indentation feature, line numbering, undo facility and copy / paste functionality.

3.2.2.3 - Debugger

A debugger is an essential tool for every developer. COMPEL IDE provides a feature full debugger that allows the user to single step, step over, run-to cursor, manage watches, manage breakpoints and view prepared scripts (interpreted state of a given script). Essentially the debugger is a great tool in assisting one to learn because novice users will benefit from the stepping facility in order to trace a complex program and study it bit by bit.

3.2.2.4 - Run Externally

Run externally functionality allows the user to run his program through the console version of COMPEL interpreter. The console interpreter can be used from outside the IDE in order to run COMPEL scripts from DOS prompt or batch files. The console interpreter has many features not provided by the IDE, such as the ability to dump the complete symbol table of COMPEL, or to pass program arguments to the script and even to dump the prepared script into an output file for later analysis.

After proposing an initial overview of the system's functionality in this chapter, the next chapter presents a more in-depth analysis of the project and its components accompanied with descriptive system screenshots.

Chapter 4

Project Analysis

Outline

Every project is possible only because it was previously engineered and analyzed using well known engineering and analysis methods. In this chapter we shall talk about the development model, Efficiency, Maintainability and portability, System View Points and different architectures of COMPEL's different components.

4.1 - Development Model Used

COMPEL was developed using two commonly used software process models which are the iterative and the component based model. The iterative development approach interleaves the activities of specification, development and specifications. An initial system is rapidly developed from very abstract specifications. The system is then refined based on the needs and requirements. Alternatively, may be re-implemented using a more structured approach [4]. Whereas the component-based software engineering technique (CBSE) assumes that parts of the system or sub-subsystems already exist. The system development process focuses on integrating these parts.

The COMPEL library mainly uses the iterative development and little of CBSE (when coding the built-in extensions), whereas the COMPEL IDE is developed using only the CBSE.

4.2 - Efficiency

In this section, we measure the efficiency of the system from the logical point of view covering how efficient is the system through the architecture and design decisions used, and then from the technical specification point of view specifying how effective is the current implementation.

4.2.1 - Speed and size

COMPEL is written in a compact manner and efficient code in order to provide speed and optimal code size. The compel library is only 350kb in size, which can be easily distributed with your application while the COMPEL IDE is 800kb in size. The results chapter will present the speed of script interpretation.

4.2.2 - Built-In Extensions

COMPEL comes with built-in commands and functionality, saving you a lot of time, thus eliminating the need to re-create them from scratch. Refer to the appendix to learn more about the built-in extensions.

4.2.3 - Operating System

COMPEL library is designed for Windows 32bit operating systems. It is implemented as a DLL file. The engine, with little modification can be compiled and ran under different operating systems. COMPEL IDE is implemented as a standalone Windows GUI executable [5]. Fortunately porting COMPEL IDE into another platform is not as hard as one can initially anticipate since there exists now Kylix, a Linux port of Delphi [5]. For other operating system a complete re-write and/or re-implementation would be necessary.

4.3 - Reliability

In this section we will explain about the reliability metrics used to measure the system then we will cover to what extent is failure tolerated and recoverable.

4.3.1 - Reliability Metrics

COMPEL engine and IDE both belong to the “Mean Time to Failure” (MTTF) metric, when the system is used for a long period of time. The system might have unpredicted memory leaks and hidden bugs that might appear after prolonged usage [4].

4.3.2 - Allowable/Acceptable Failure Rate and Recovery

COMPEL engine supports 3 types of failures, which are Script Pre-Parsing failure, Script runtime error and Logical Errors. The script Pre-Parsing failure happens when the script format is fundamentally incorrect, thus there is no possibility to run it in the first place. The script runtime error occurs when the script has errors in a specific line of code but the error is not fatal and can be recovered then the user can continue/ignore this error. Finally the logical error which manifests when the user types a logically wrong code that does not fulfill its goal; this sort of errors is not detected by COMPEL or any other programming language to-day, however, with the use of the IDE and debugger the user can spot the logical script errors. Thus, the failure classification of COMPEL system are transient when errors might occur only with certain inputs, recoverable when the system can be recovered in most of the cases and non-corrupting when a failure does not cause any corruption or loss to user's data.

4.4 - Maintainability and Portability

The COMPEL engine is delivered as a modular system allowing the system to be easily maintainable. Every module and functional unit is separated, thus every part can be isolated, tested and improved apart. The system currently uses Win32 libraries, however the Win32 library calls are centralized in specific source code files, making it possible to replace those source files with platform specific files (such as Linux library calls).
The programming language used is highly portable, and the style and standards in COMPEL allow the easy of portability.

4.5 - System View Points

The key strength of viewpoint-oriented analysis is that it recognizes multiple perspectives. The system view points can also be used as a way of classifying stakeholders and other sources of requirements [4]. In this section we will illustrate two view points influencing the Library and IDE parts.

4.5.1 - COMPEL Library View Points

The following figure displays the view points of the COMPEL Library system. As [4] defines it, we have Indirect, Interactor and Domain View points each having its own sub view points.

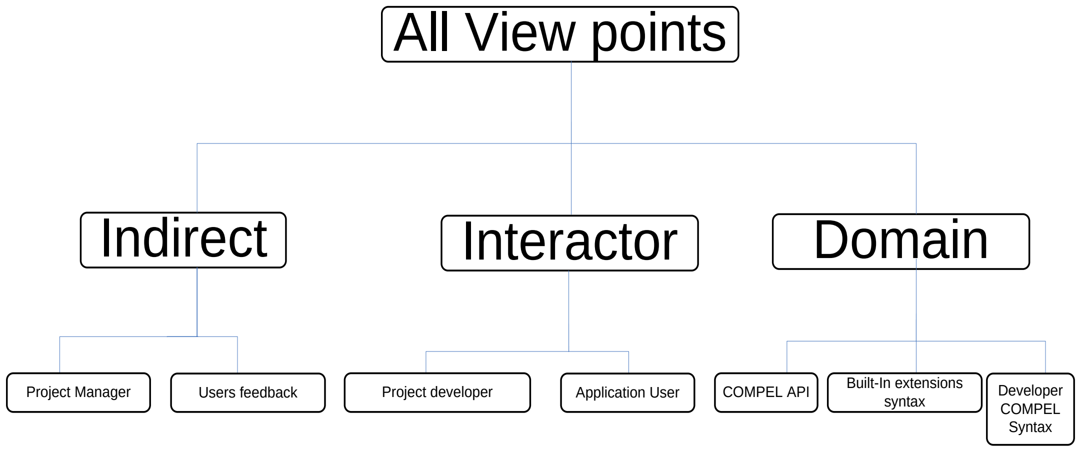


Figure 4.1 - COMPEL Library Viewpoints

4.5.1.1 - Indirect VP

In the indirect viewpoints we have “Project Manager” and “User feedback” viewpoints. The project manager dictates what is to be implemented and how COMPEL Library is to be used, where as the user feedback viewpoint is when/how the users of the system influence the system indirectly.

4.5.1.2 - Interactor

In the Interactor perspective we have Project Developer and Application user viewpoints. The project developer is directly involved with the COMPEL library, since he will be coding and employing the library. The application users will interact with COMPEL directly through the command line interface provided by the project developer.

4.5.1.3 - Domain

From the domain perspective, we have the COMPEL API, Built-In extensions and Developer COMPEL syntax viewpoints. The COMPEL API dictates how the system is to be used, the built-in extensions impose restrictions and rules on how the system is employed and finally, the developer’s new extensions will impose new COMPEL syntax and restrictions.

4.5.2 - COMPEL IDE View Points

In the COMPEL IDE, different usage of the system is introduced, thus we have different perspectives and viewpoints, unlike the COMPEL library roles and viewpoints, roles are completely changed as we will see below.

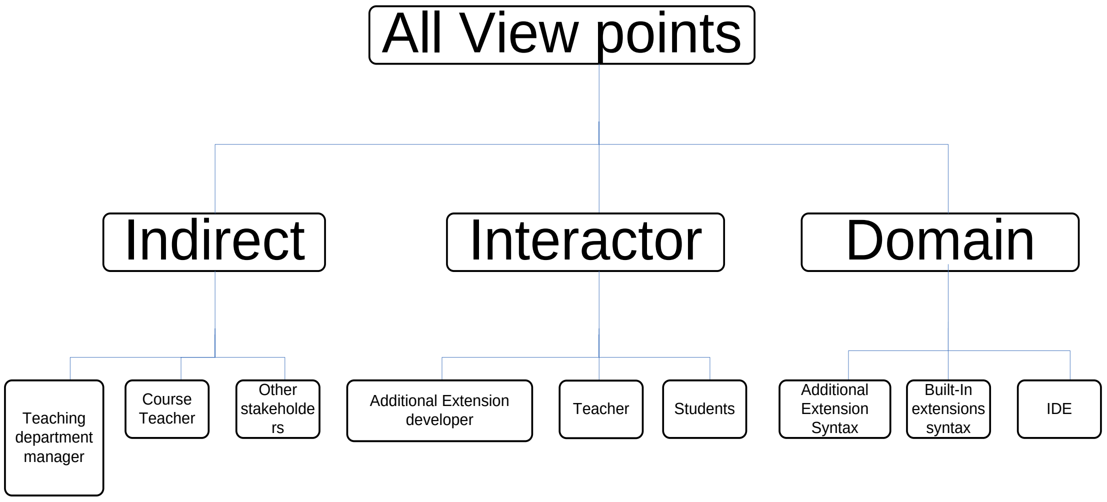

All View points
Indirect
Interactor
Domain
Teaching department manager
Course Teacher
Other stakeholders
Additional Extension developer
Teacher
Students
Additional Extension Syntax
Built-In extensions syntax
IDE

Figure 4.2 - COMPEL IDE Viewpoints

4.5.2.1 - Indirect

In the indirect perspective we have the teaching department manager, course teacher and other stakeholders viewpoints. The Teaching department manager will influence the system indirectly by providing teaching strategies and methods to be employed, the course teacher will be indirectly involved since he will be selecting what and how COMPEL should be used, whereas the other stakeholders in the system, such as other teaching departments will also influence the system indirectly by giving suggestions and hints.

4.5.2.2 - Interactor

In the Interactor perspective, we have the “Additional Extension developer”, “Teacher” and “Students” view points. The additional extension developer interacts with COMPEL directly by introducing new extensions, the teacher will be using the IDE to teach his students whereas the students will interact directly by typing programs and testing them.

4.5.2.3 - Domain

The domain perspective entails two viewpoints, the built-In/Additional extensions syntax that impose restrictions and rules on how the system is employed, and the IDE itself which can also affect the system.

4.6 - System Architecture

Systems of the same type have similar architectures, and the differences between these systems are in detailed functionality. In this section we will classify the COMPEL system components into one of the well known system architectures (Data-processing applications, Transaction-processing applications, Event-processing systems, Language-processing systems).

4.6.1 - COMPEL Library Architecture

The COMPEL engine (or library) belongs to the “Language-Processing” systems where the user’s intentions are expressed in a formal language. The language-processing system processes this language into some internal format and then interprets this internal representation [4]. The engine design is very similar to that of a compiler, however it is much simpler.

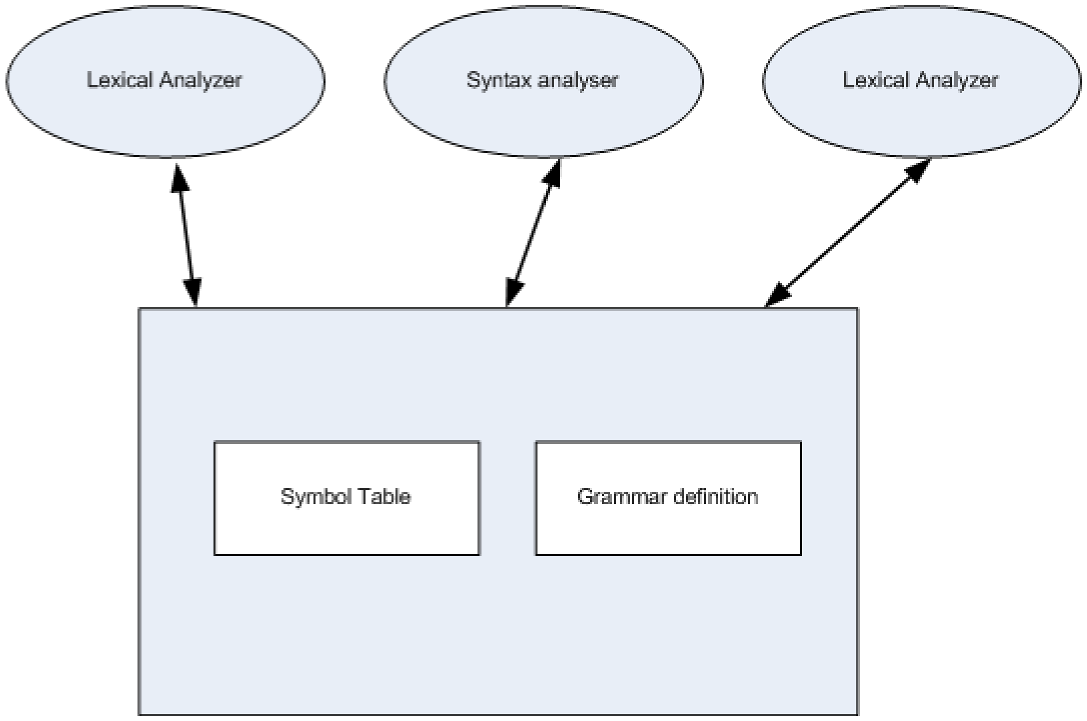


Figure 4.3 - COMPEL's Language-Processing System Architecture

4.6.2 - IDE Architecture

The COMPEL IDE belongs to the “Event-Processing” systems; the system detects and interprets events. User events represent implicit commands to the system, which takes some action to obey that command [4]. The core of the COMPEL IDE is the application is the “Editor” and the “Debugger Window” In the figure below. We illustrate the architectural model of an editing system since COMPEL IDE harbors an editor.

Figure 4.4 illustrates the general architecture of an editor [4]. The File system is responsible for physically expressing the document (loading from disk and saving to disk), the ancillary is responsible for some editor commands such as spell checking or auto-save features, the Display keeps track of organization of the screen display, the Editor data tracks the changes in the document and calls on to the Display when needed, the Command processes the commands from the event object and calls the appropriate method in the Editor data, whereas the Event is triggered by an event arriving from

Screen, the Screen monitors the screen related inputs and events then passes them to the Events object.

4.6.2.1 - Editor Architecture

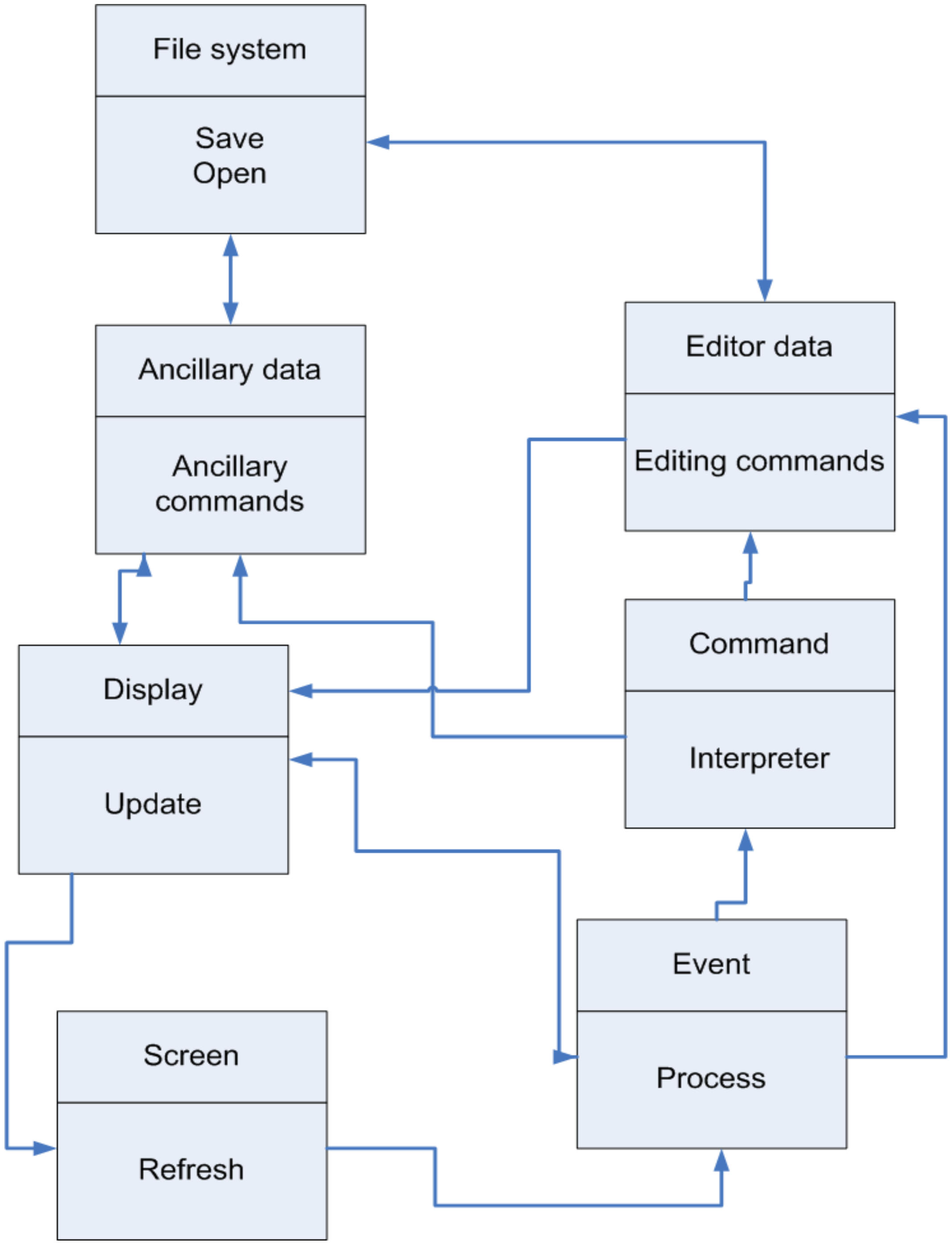


Figure 4.4 - An architectural Model of an Editing System

4.6.2.2 - Debugger State Machine

We illustrate in this figure how the IDE works and interacts with the debugger through a state machine [2]. The transition from state to state happens when an event occurs. For example, when a user opens a file, the state changes from empty IDE state to “Editing IDE” state.

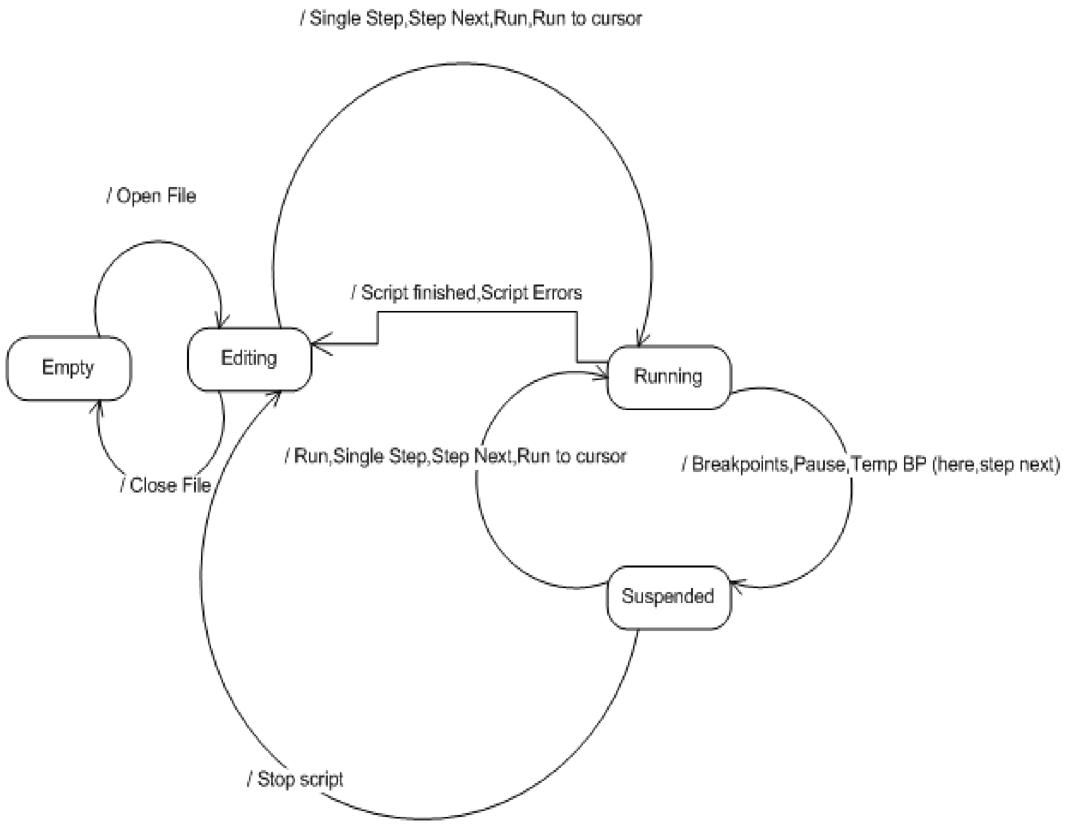

/ Single Step,Step Next,Run,Run to cursor
/ Open File
/ Script finished,Script Errors
Empty
Editing
Running
/ Run,Single Step,Step Next,Run to cursor
/ Breakpoints,Pause,Temp BP (here,step next)
/ Close File
Suspended
/ Stop script

Figure 4.5 - IDE State Machine

4.7 - User Interface Design

In this section we will illustrate all relevant interface designs related to COMPEL library and IDE. The IDE graphical interface design took into consideration the following principles of Consistence, User familiarity, user guidance and recoverability. The interface is consistent, similar functionalities are activated in the same way, it also uses terms and concepts drawn from the experience of the people who will make most use of the system. The system is intuitive providing meaningful feedback when error occurs and allow users to recover from errors.

4.7.1 - COMPEL Library Interface

COMPEL Library does not have a graphical user interface since it is a library. Nonetheless, in the system design chapter, we will portray the internals of the library with relevant design plans and figures.

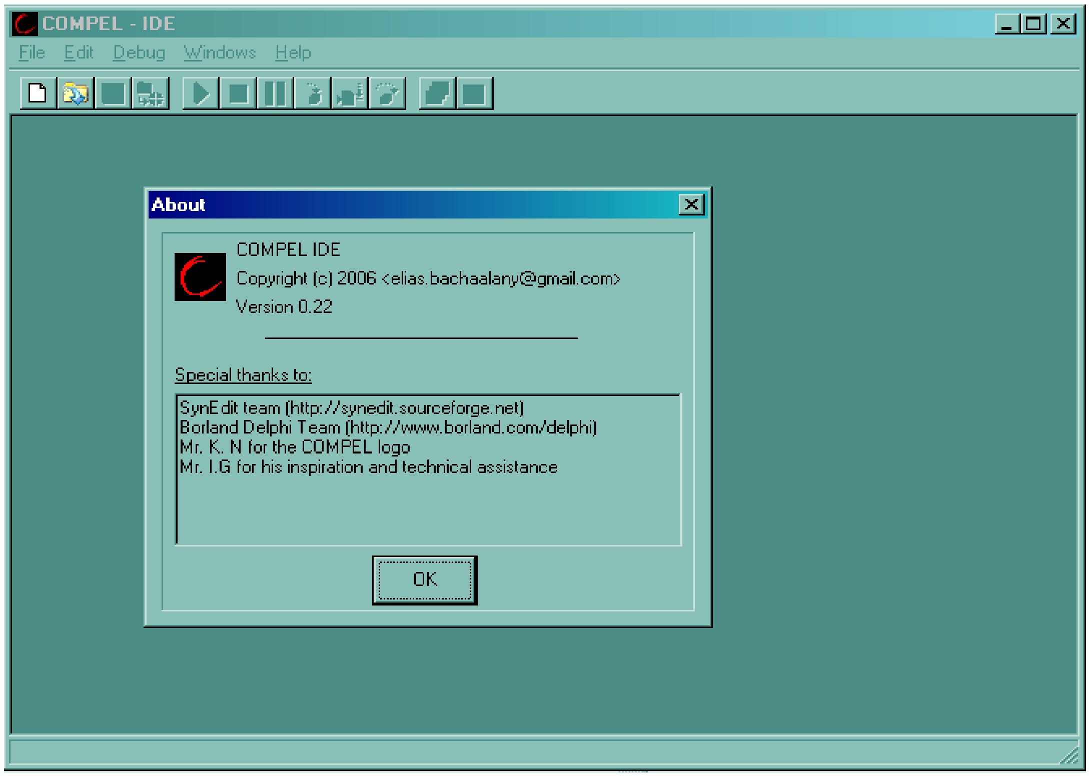
4.7.2 - COMPEL IDE Interface

The IDE is a graphical interface application. Below we will list all the interfaces of the application followed by their functional description. For more information about each screenshot you may refer to Chapter three.
We used the following design principles while designing the GUI:

- Consistency: The look and feel of the IDE is very similar to most windows applications
- Accessibility: Almost every functionality is hooked to a hotkey and an image display to make the interface very friendly
- Simplicity: The IDE menu design is very simple and user friendly

The empty IDE state (Figure 4.6) allows you to simply create a new script or open an existing one. Functionality is limited when in the empty IDE state. However, like advanced IDEs the empty state can be a way to open recent projects, read IDE news (fetched from the internet).

The source code editing window (Figure 4.7) shows the complete path of the file being edited and the line numbers of the script source code. The editor window supports virtual spaces, automatic indentation and the inherent copy / paste functionality.

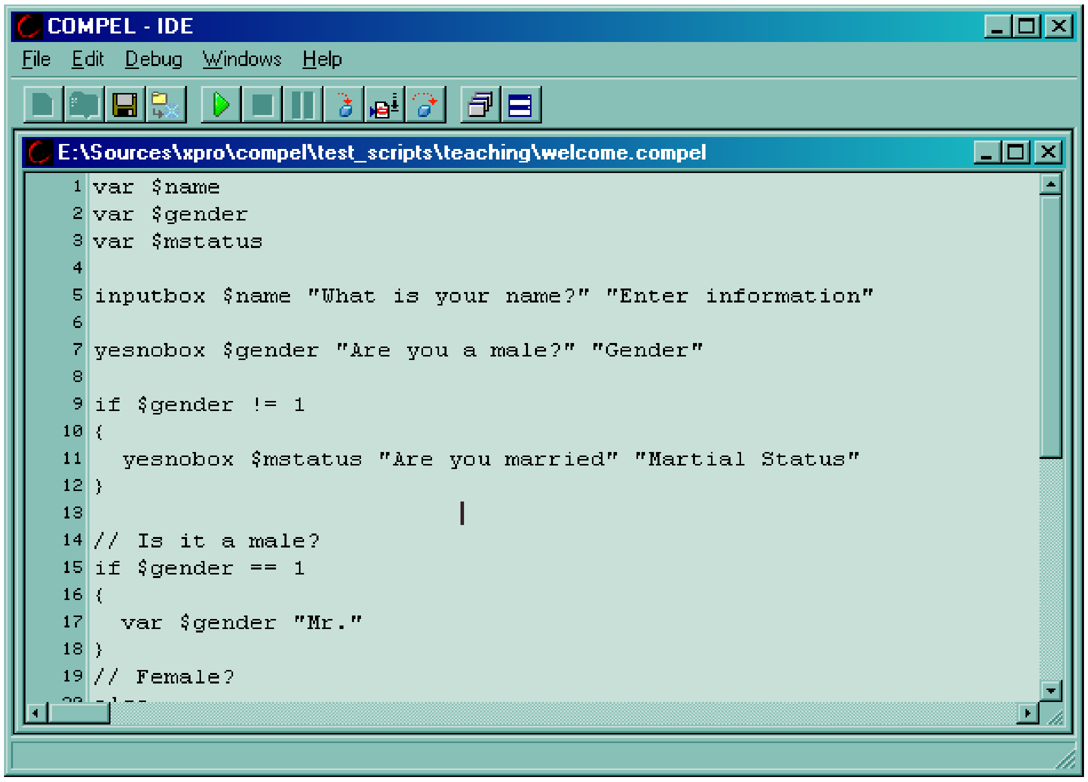


COMPEL IDE
Copyright (c) 2006 <elias.bachaalany@gmail.com>
Version 0.22

Special thanks to:

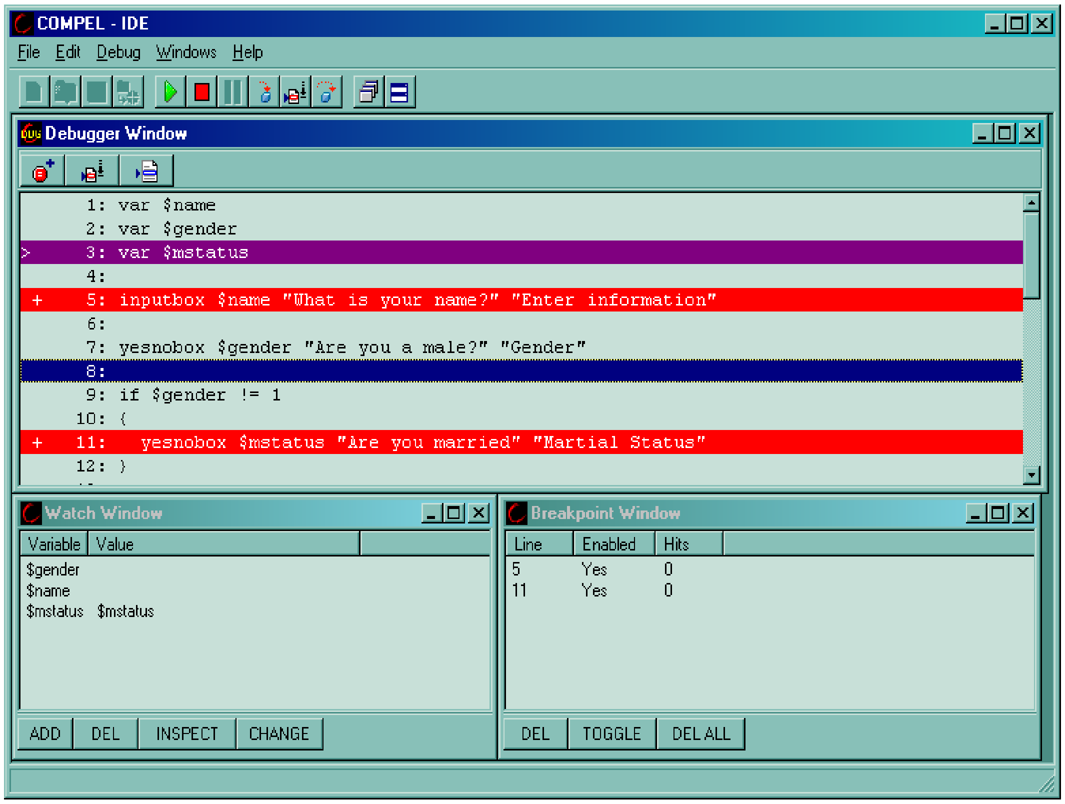
SynEdit team (http://synedit.sourceforge.net)
Borland Delphi Team (http://www.borland.com/delphi)
Mr. K. N for the COMPEL logo
Mr. I.G for his inspiration and technical assistance

Figure 4.6 - Empty IDE (Showing the About Dialog)


COMPEL - IDE
File Edit Debug Windows Help
E:\Sources\xpro\compel\test_scripts\teaching\welcome.compel

```
var $name
var $gender
var $mstatus

inputbox $name "What is your name?" "Enter information"

yesnobox $gender "Are you a male?" "Gender"

if $gender != 1
{
  yesnobox $mstatus "Are you married" "Martial Status"
}

// Is it a male?
if $gender == 1
{
  var $gender "Mr."
}
// Female?
```

Figure 4.7 - Editor Window - Editing a Simple Script “welcome.compel”



```
      1: var $name
      2: var $gender
>     3: var $mstatus
      4:
 +    5: inputbox $name "What is your name?" "Enter information"
      6:
      7: yesnobox $gender "Are you a male?" "Gender"
      8:
      9: if $gender != 1
     10: {
 +   11:   yesnobox $mstatus "Are you married" "Martial Status"
     12: }
```

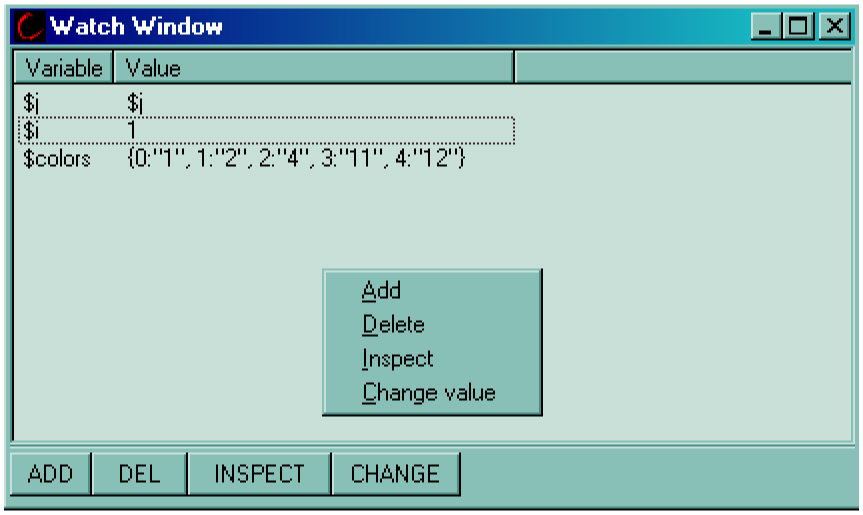
Variable	Value
$gender	
$name	
$mstatus	$mstatus

Line	Enabled	Hits
5	Yes	0
11	Yes	0

Figure 4.8 - Debugger Window

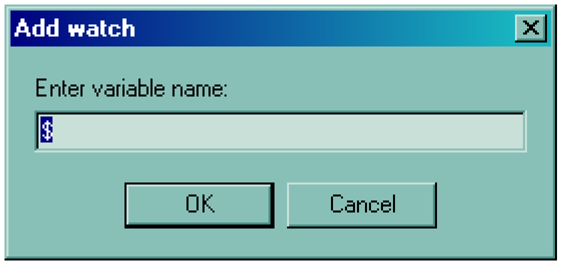
The debugger window is responsible for showing the script’s lines of code and allow you to view or change the order of execution. The line in violet is the currently executing line, also prefixed with the “>” character. The lines in red are the breakpoint lines. The “+” prefix along with the red color denote an active breakpoint. When a breakpoint is inactive (or disabled) it will be in green color and prefixed with “-“. In this screenshot we also see the breakpoint and watch windows.

A watch window (Figure 4.9) is a facility that allows you to select some variables so that their values can be watched through-out the program’s execution. The watch window shows 3 watched variable. $i and $j are variables but $colors is an object with three attributes named “0” to “4”. We also see a popup context menu, toggled when the user right-clicks on a selected menu variable he will be able to adjust the value of the variable, inspect the value by opening the inspection window, deleting a watched variable or even to add a new variable into the watch window.



Variable	Value
$j	$j
$i	1
$colors	{0:"1", 1:"2", 2:"4", 3:"11", 4:"12"}

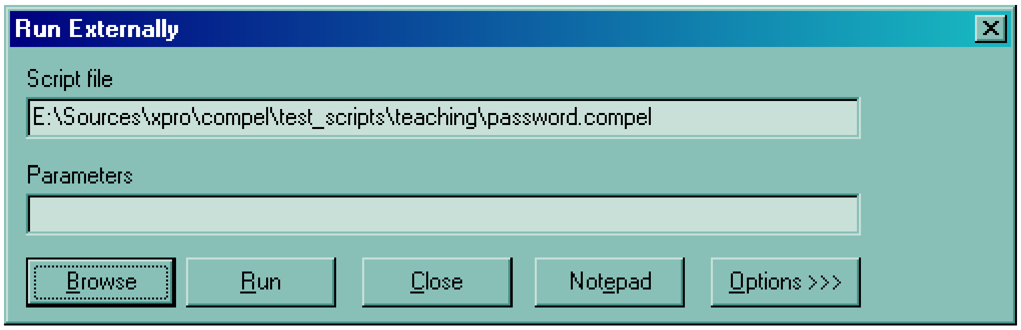
Figure 4.9 - Watch Window



Enter variable name: $

Figure 4.10 - Add Watch variable

The add watch allows you to add more variables to be watched. Every time the script executes a line, all the watched variables will be updated to reflect the changes in the variables.



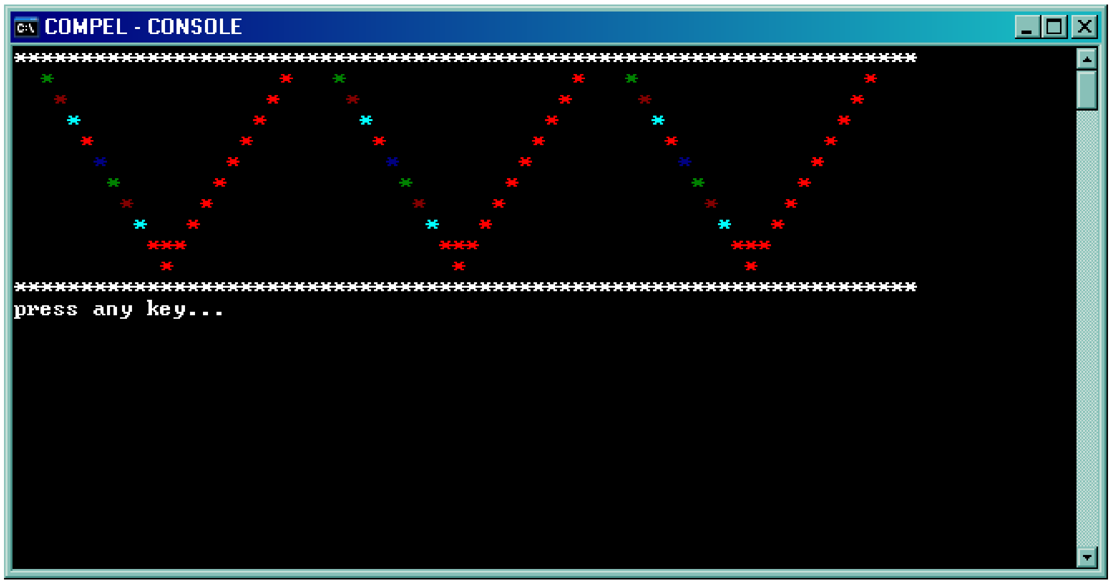
Script file: E:\Sources\xpro\compel\test_scripts\teaching\password.compel

Parameters:

Figure 4.11 - Run Externally: Selecting a Script to Run It Externally

Here, user has selected a script that will be launched from outside the IDE.

No debugger phase will be initiated, instead the program will run and quit on its own. Externally running a script involves running the console version of the COMPEL language interpreter.

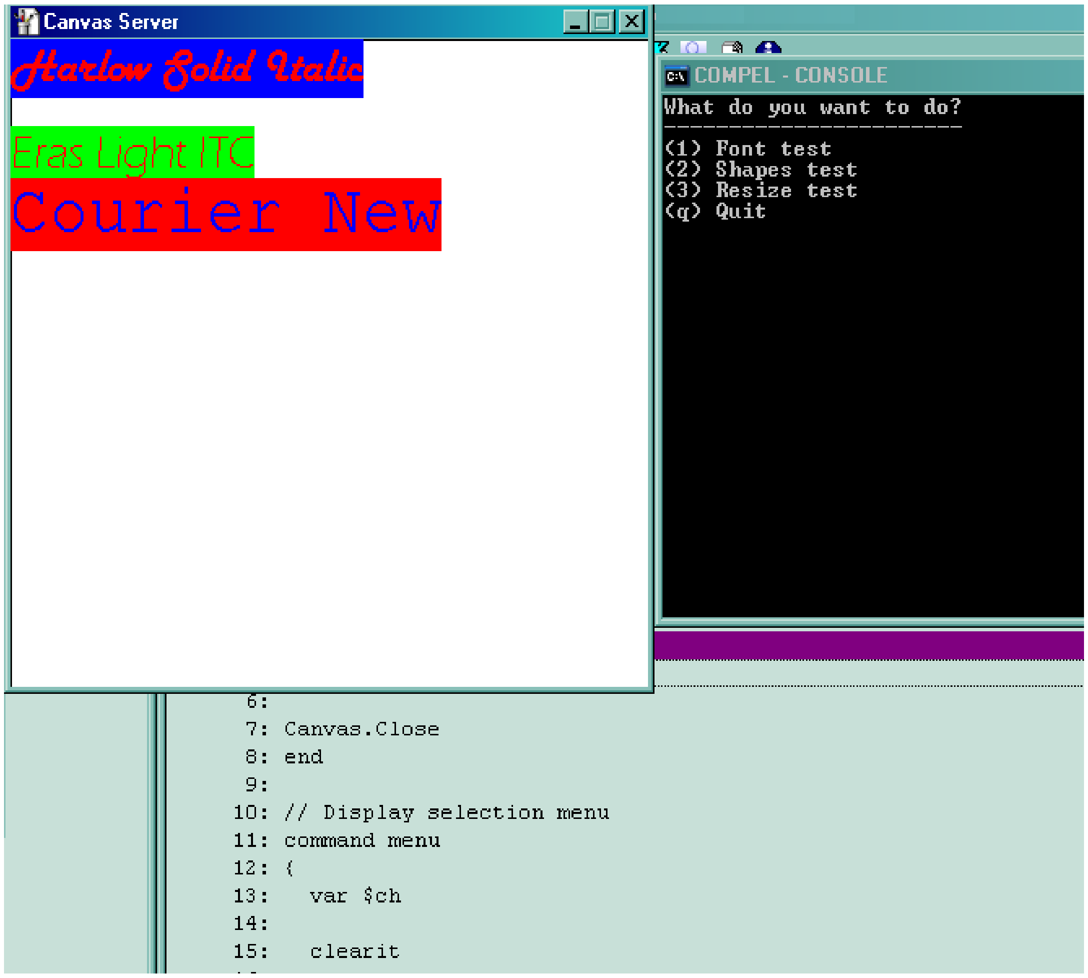


```
********************************************************************
  *                 *   *                 *   *                 *
   *               *     *               *     *               *
    *             *       *             *       *             *
     *           *         *           *         *           *
      *         *           *         *           *         *
       *       *             *       *             *       *
        *     *               *     *               *     *
         *   *                 *   *                 *   *
          ***                   ***                   ***
           *                     *                     *
********************************************************************
press any key...
```

Figure 4.12 - External Run output - After the execution of a sample script

External scripts, as previously noted, are executed via “ccompel.exe” which is a console version of the COMPEL engine. When the script finishes running, a pause prompt will appear waiting the user to press any key to continue. The sample script used in this demonstration is “for_loop_crt.compel”

Figure 4.13 shows the IDE debugging a script that implements / uses an external extension named “canvas”. Again this demonstrates how the IDE is used for general scripting and not bound to anything specific.



```
What do you want to do?
-----------------------
(1) Font test
(2) Shapes test
(3) Resize test
(q) Quit
```

```
Canvas.Close
end

// Display selection menu
command menu
{
  var $ch

  clearit
```

Figure 4.13 - COMPEL External Extension - Canvas

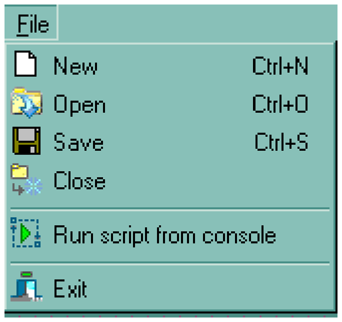


Figure 4.14 - File Menu

The file menu allows you to do file related actions such as creating new files, opening existing files, saving file, closing opened file, running a script externally and quitting the IDE.

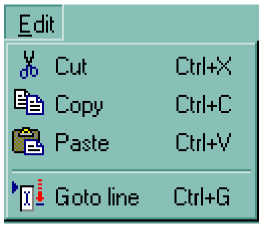


Figure 4.15 - Edit Menu

This is the edit menu, mainly associated with the editor window. The cut, copy and paste are intuitive commands available in all editors. The “Goto line” functionality allows you to jump to a given line number when in editing state. When in debugging state, the debugger window will highlight the target line number.

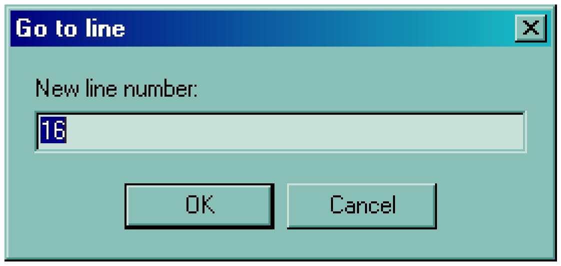
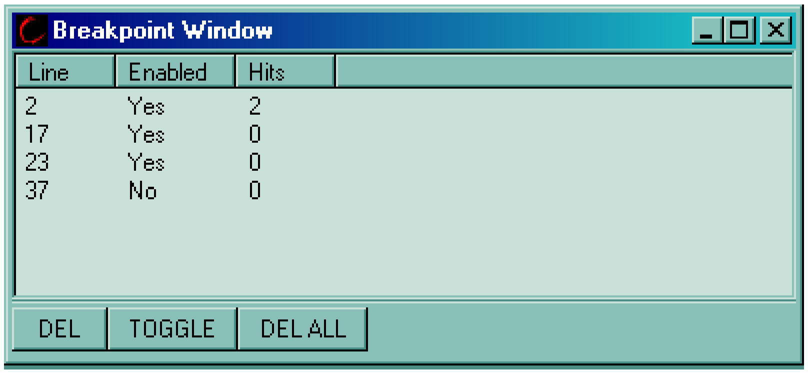


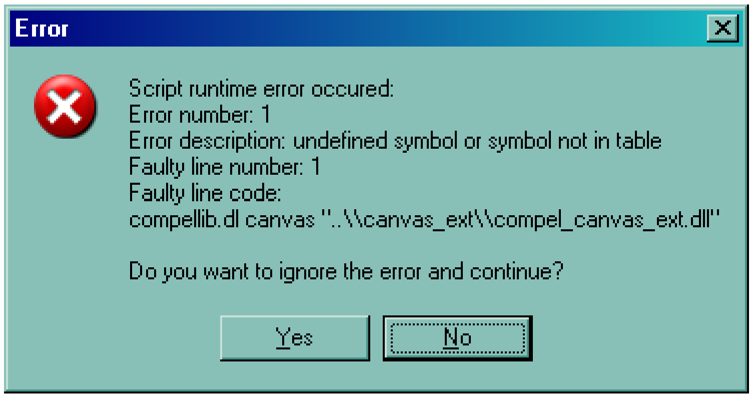
Figure 4.16 - Go to line



Line	Enabled	Hits
2	Yes	2
17	Yes	0
23	Yes	0
37	No	0

Figure 4.17 - Breakpoint window

This window allows you to manage your breakpoints. A breakpoint is a marker you place on a certain line of code which will tell the debugger to stop when that line is reached. This window allows you to delete a breakpoint, toggle the status of the breakpoint (on becomes off and vise-versa), and deleting all breakpoints. The three columns in this window are the script source line number that will cause the debugger to stop when it is reached, the enabled column which denotes whether a breakpoint is active or not (enabled or disabled) and the hits count column which counts the times a breakpoint has been reached.



```
Script runtime error occured:
Error number: 1
Error description: undefined symbol or symbol not in table
Faulty line number: 1
Faulty line code:
compellib.dl canvas "..\\canvas_ext\\compel_canvas_ext.dll"

Do you want to ignore the error and continue?
```

Figure 4.18 - Script error reporting dialog

The error dialog is shown when a pre-parse or runtime script error occurs. A pre-parse error occurs when the script's format is erroneous, such as when the programmer writes a code that is not proper (Example: calling "return" command while not in a user command code), whereas a runtime script error occurs when user mistypes a command name or passed wrong syntax. In both error cases, the user is given the choice to either ignore the error and proceed to next line or to press "No" and abort the debugging session.

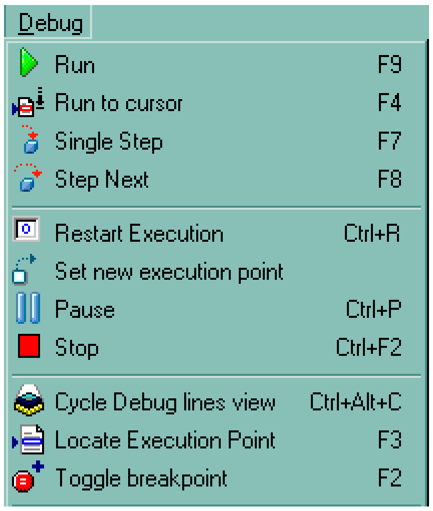


Figure 4.19 - Debug Menu

The debug menu is mainly associated with all the debugging facilities. It also serves a mean to start debugging from an editing state. The following functionality is explained:

- Run: Either starts running a script (if user was previously in editing mode) or resumes running the interrupted script (due to a breakpoint or pause)
- Run to cursor: Allows you to run the script then abandon it when the line number under the editor cursor is reached
- Single Step: Allows you to run one instruction at a time
- Step Next: Creates a temporary breakpoint on the next line, runs the script, then when the next line is reached, execution is suspended. This functionality is useful when you want to skip stepping into function calls
- Restart execution: Allows you to restart execution of the script from first line of code. Useful if you want to start tracing from beginning
- Set new execution point: Allows you to select which line to continue tracing from.
- Pause: Allows you to pauses a running script. Useful when a script is running and you want to interrupt it and continue tracing from which ever execution point it was running from before pause
- Stop: Aborts the script execution and returns to editing mode
- Cycle Debug lines view: Allows you to see the pre-parsed lines view or the script source line views. This will be detailed more in next sections.

- Locate Execution point: While debugging you may browse and scroll to different script lines to overview your code. This functionality allows you to reposition the cursor on the line that is scheduled to be next ran

- Toggle breakpoint: Same functionality provided by the breakpoint window

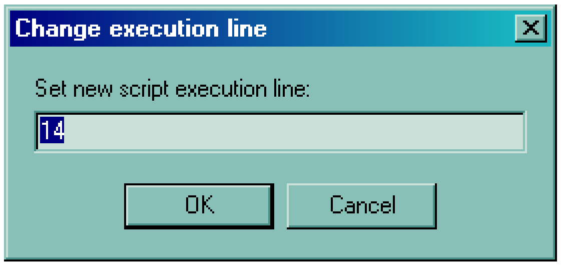


Figure 4.20 - Set New Execution Point

This functionality allows you to set new execution point or line. This means that the next line that will be executed is the line you just specified.

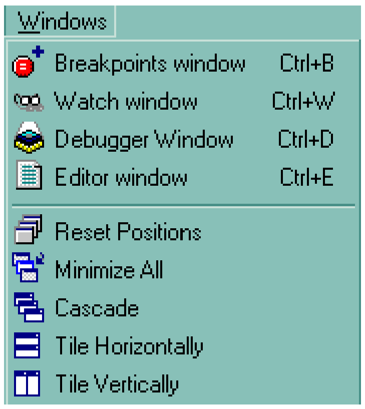


Figure 4.21 - Windows Menu

This menu provides shortcuts to selecting the window you want to work with. In addition, it provides window positioning manipulation routines.

- Breakpoints window: Activates the breakpoint window
- Watch Window: Activates the watch window
- Debugger Window: Activates the Debugger window (only if you are already in debugging state)
- Editor window: Activates the editor window (only if a file was loaded)

- Reset Positions: Re-positions the windows in a smooth way assuring that all necessary windows are shown properly. It acts like default positioning of windows. It positions different when you are in debug or editing mode
- The rest of the menu items are self explained

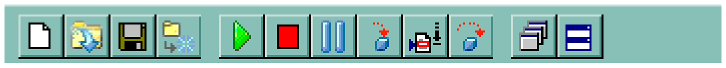


Figure 4.22 - The IDE Toolbar

The toolbar is just a bar with graphical buttons that are shortcuts to menu entries. The entries that we provide in the toolbar are (in the order of appearance): New file, Open file, Save file, Close file, Run script, Stop script, Pause script, Single Step, Run to cursor, Step Next, Window->Reset position, Window->Tile horizontally.

In this section we described all the functional aspects of the project using very little computer jargon. In the coming sections, more details and technical aspects of the system will be displayed.

4.8 – Cost Analysis

In this section we will display the cost incurred during the development of COMPEL project. Cost analysis covers tangible (software and books purchased) and intangible (time spent, algorithms developed) expenses.

Table 4.1 - Cost Analysis

System Analysis (20 hours/ $15/hrs)	$300
Programming and Implementation (350 hours / $10/hrs)	$3,500
Research (10 hours / 10$hrs)	$100
Books purchased	$120
GUI Design (5 hours / $10/hrs)	$50
Total	**$4070**

Based on the previous cost analysis, COMPEL will be sold using the following pricing scheme:

Table 4.2 - COMPEL Selling Price

COMPEL IDE	Non-commercial use (free)	$0.0
COMPEL Developers Library / SDK	Developer License	$350
COMPEL Developers Library / SDK – Support	Per call support / hourly rate	$50

In this chapter we presented the project’s internal from software engineering point of view, the next chapter will present the design specification of the system using diagrams, flowcharts and tables.

Chapter 5
Design Specifications

Outline

In this chapter we emphasis and illustrate the design of COMPEL system using diagrams and flowcharts which are essential for quick understanding of the system. System structures are then explained in UML and other notations.

5.1 - Diagrams

In this section we will illustrate how the system functions through the use of dataflow diagrams for the Engine, COMPEL Library and COMPEL IDE components. In short, the engine is the core of the COMPEL library. It handles all the internal functionality of COMPEL. The engine is composed of 4 sub-systems which are the Symbol Table, Interpreter, Symbol Table Helper and Interpreter Helper. The COMPEL library is a wrapper around the engine. It exports from the engine all the needed functionalities so that developers can manipulate the engine and finally COMPEL IDE is the COMPEL Integrated Developer Environment. It is responsible for allowing users to write scripts and debug scripts as a standalone user.

5.1.1 - Engine Dataflow Diagram

In this diagram we illustrate the data path encountered when the user passes the engine a command.

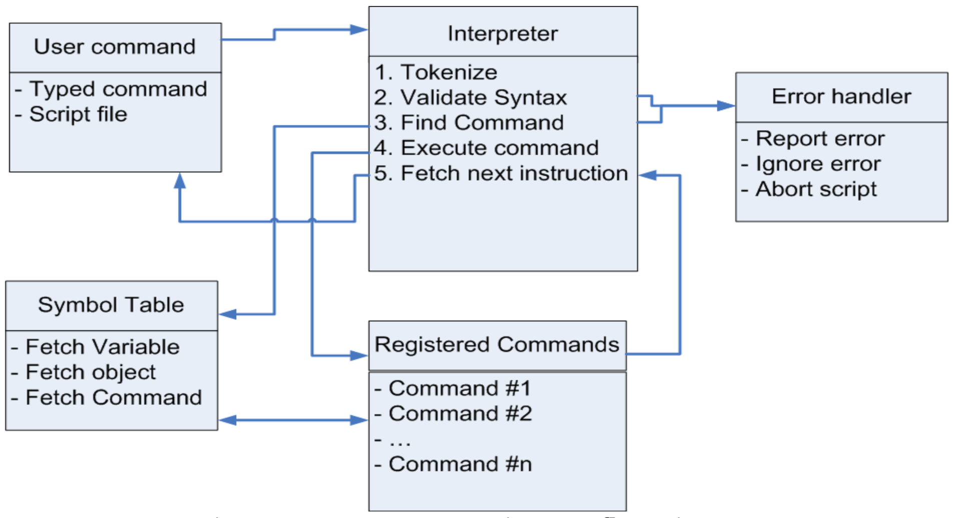


Figure 5.1 - COMPEL Engine Dataflow Diagram

1. User command: A command is entered
2. Interpreter: Tokenizes the command and splits it into separate tokens in order to know what was the typed command and its parameters
3. Interpreter: Validate the syntax typed and check number of parameters (min/max)
4. Interpreter: Find Command will query the symbol table to see if the command exists and is registered or not
5. Interpreter: If command is found in the “registered commands” table then additional syntax checking is done (checks if passed parameters are of correct types)
6. Interpreter: If command is not found or syntax is not correct, then “Error Handler” will be called
7. Error Handler: Provides multiple choices to handling and recovering errors. The COMPEL engine may optionally call the user error handler that was previously registered by the library user. The error handler may decide whether to allow the script to recover from the error (the error can be ignored and script execution line is either on the next line or on the same but correct line) or to stop execution (the error is not handled and interpreter should stop fetching more instructions)
8. Interpreter: Call the specific command in the “Registered commands”
9. Registered Commands: First it executes specific command then returns to interpreter
10. Interpreter: Fetch next instruction if needed
11. User command: Wait for another command (go back to step 1)

5.1.2 - Library Dataflow diagram

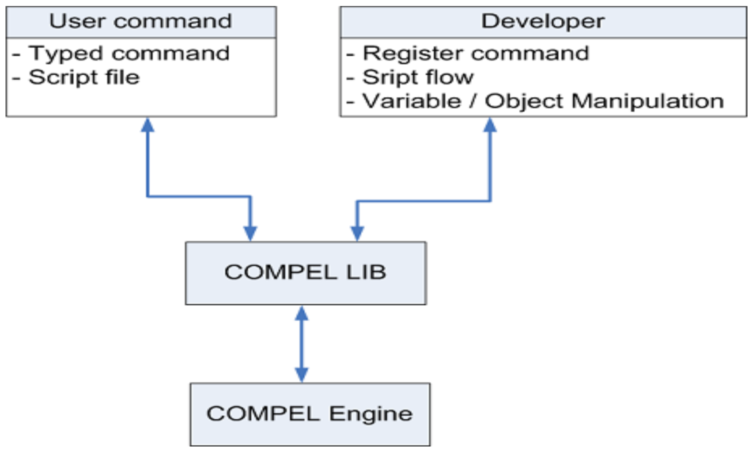


Figure 5.2 - COMPEL Library DFD

This DFD illustrates how multiple clients can talk with the COMPEL Library and how then the COMPEL Library will then talk with the engine, which will in turn, interpret the clients' request and respond accordingly to each one.

5.1.3 - IDE Dataflow Diagram

In the IDE dataflow diagram, we notice how the process starts by waiting for the user input. After the user selects an action, the event handler will decide where to pass the command to. Subsequently the responsible window handles the event by interacting with its internal sub-systems and finally the whole process is repeated by waiting for a newer event. In the case of a debugger event, the subsystem will communicate with another sub-system which is the COMPEL developer library. A sample scenario which can lead to the previous case is when the user starts debugging then selects to single step one script command. It is worthwhile mentioning that the watches window and breakpoint window may also interact with the debugger sub-system.

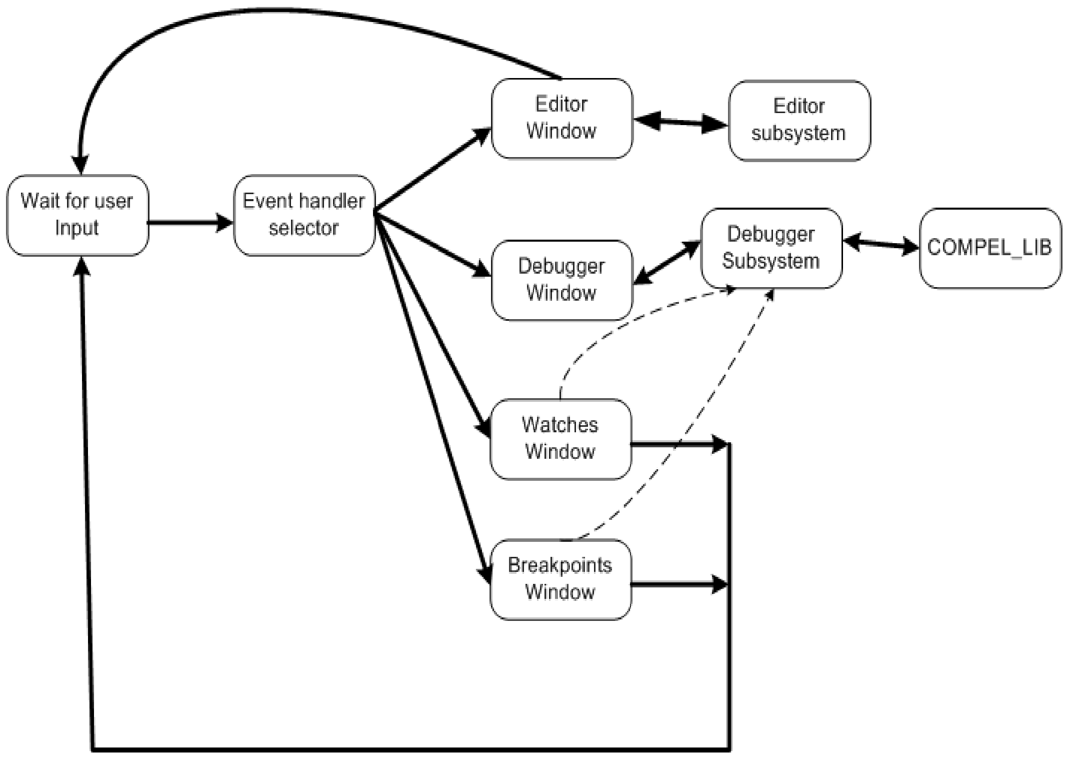


Figure 5.3 - IDE Dataflow Diagram

We notice in Figure 5.3 that the Editor sub-system does not interact with the COMPEL_LIB component, such revelation tells us that the editor is not intelligent enough and cannot provide IntelliSense features or pre-compiling / pre-validation of the syntax such as the functionalities provided by Microsoft Visual Studio IDE. At most the editor may emulate IntelliSense feature through querying the COMPEL engine after script's first run and deduce the created objects then try to identify what are the commands that should be highlighted.

5.1.4 - COMPEL Error Handler Flowchart

COMPEL engine allows developer to register an error handler. An error handler is a mechanism to catch scripting errors and either correcting them or halting the script execution process. The following diagram will illustrate how an error handler is used when a scripting error occurs.

We notice how the process starts by the interpreter fetching first instruction then by checking its syntax it decides whether it has detected the first type of errors known as syntax error. After the instruction is fetched it is executed and its return value is again checked by the interpreter, again based on that error the error handler might be triggered.

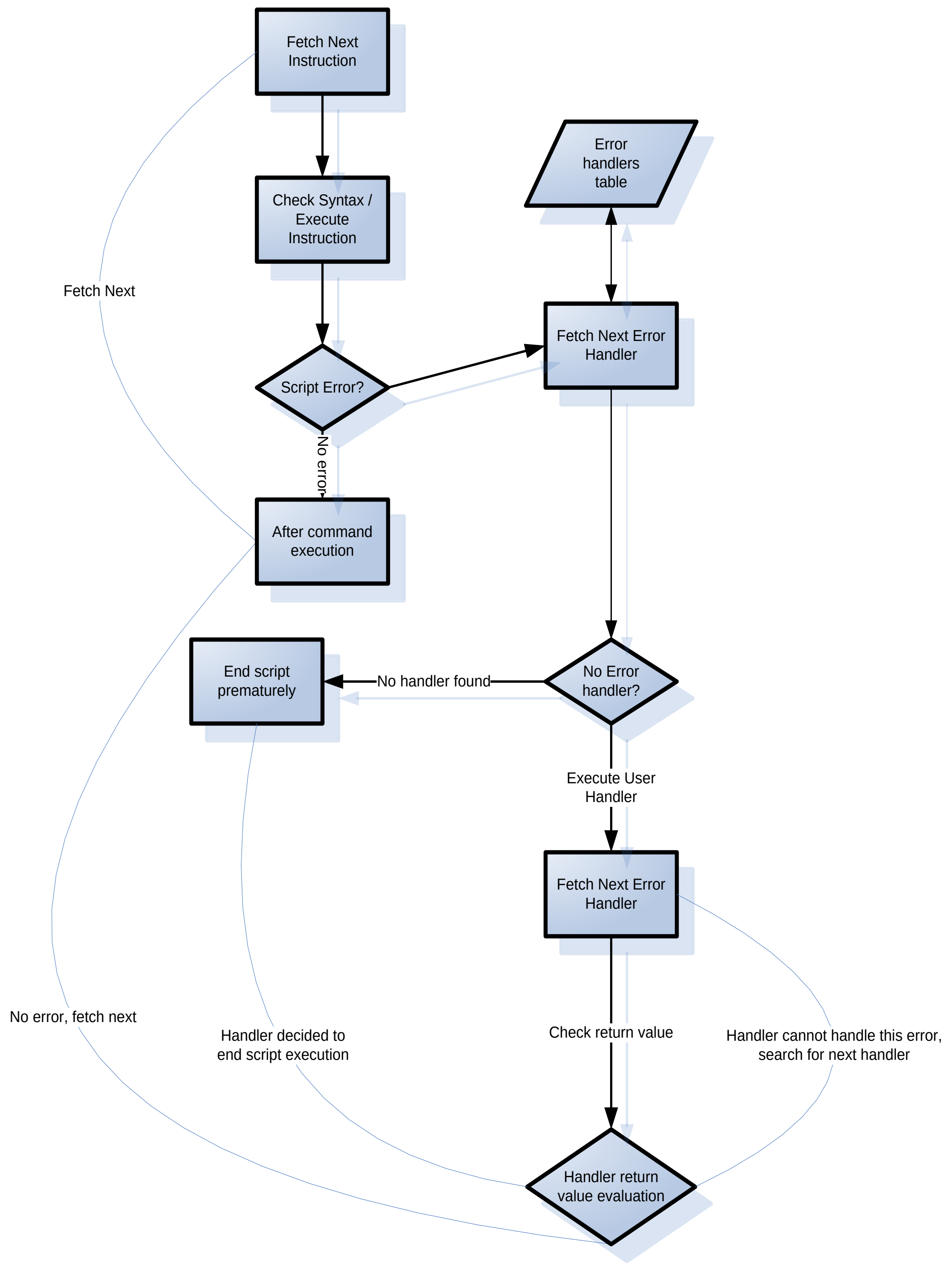


Figure 5.4 - Error Handler Flowchart

The error handler contains a chain of registered error handlers and each handler checks whether it can take care of the error or it passes the error to the next handler. If no handler can handle the error then the script should stop.

5.1.5 - IDE/Debugger GUI states table

The following table shows what GUI components are to be enabled or disabled and in what IDE state. For example, when in "Empty" state you are only allowed to do simple operations such as Exit the IDE, Load a script, Show About dialog, Create New script and show editor options.

Table 5.1 - IDE/Debugger GUI States Tables

Function Name/State	Empty	Editing	Running	Suspended
actDbgStepNext = Step Next (F8)		X		X
actDbgSingleStep = Single Step (F7)		X		X
actDbgRunToCursor = Run to cursor (F4)		X		X
actDbgRun = Run (F9)		X		X
actDbgSetNewExecPoint = Set new execution point				X
actDbgCycleView = Cycle Dbg View (C+V)			X	X
actDbgLocateExecPoint = Locate Exec Point (F3)				X
actDbgStop = Stop debugging (C+F2)			X	X
actDbgRestartExec = Restart script (C+R)				X
actDbgPause = Pause Script (C+P)			X	
actDbgBptToggle = Toggle breakpoint (F2)			X	X
actDbgSetScriptParameters = Set script parameters		X		
actFileNew = New File (C+N)	X			
actFileOpen = Open File (C+O)	X			
actFileClose = Close File (C+F4)		X		
actFileSave = Save file (C+S)		X		
actFileSaveAs = Save as file		X		
actFileExit = File exit program	X	X		
actHelpAbout = Help->About	X	X	X	X
actEditGotoLine = Goto Line (C+G)		X	X	X
actOptionsEditOptions= Options->Editor	X	X	X	X
actWindowBreakpoints = Window->Breakpoints		X	X	X
actWindowDebugger = Window->Debugger			X	X
actWindowResetPositions = Win->Reset positions		X	X	X
actWindowEditor = Window->Editor		X	X	X
actWindowWatchs = Window->Watches		X	X	X

5.1.6 - A Built-in Command Call Scenario

This figure illustrates what happens when a script calls a built-in command and how all the parameters are interpreted by the engine.

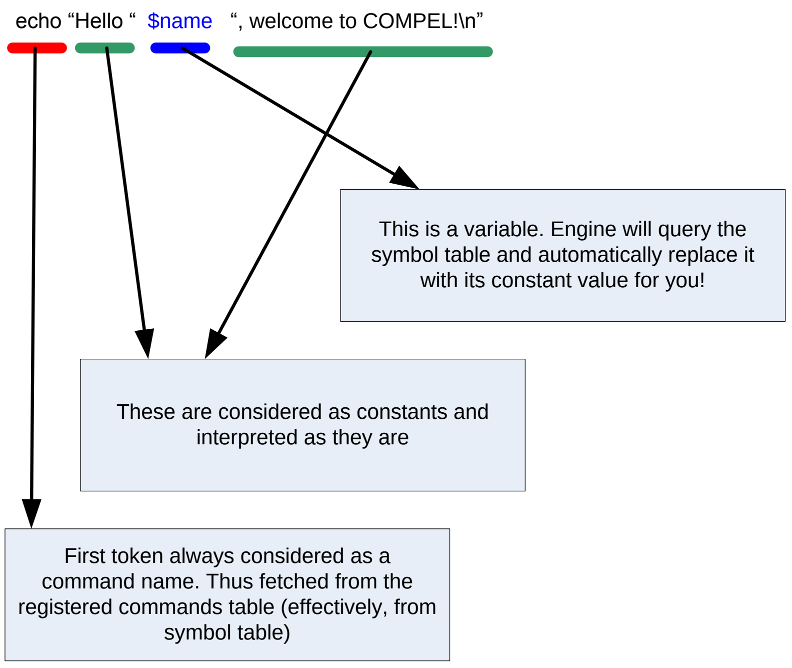

echo "Hello " $name ", welcome to COMPEL!\n"
This is a variable. Engine will query the symbol table and automatically replace it with its constant value for you!
These are considered as constants and interpreted as they are
First token always considered as a command name. Thus fetched from the registered commands table (effectively, from symbol table)

Figure 5.5 - Built-In Command Call Scenario

We notice again how COMPEL breaks the syntax of the script using the space character as a token / separator, interpreting the first token as the command to be executed, the dollar ($) sign as a special symbol to designate variables, and any other parameter as a constant that may or may not contain C language like escape sequence characters.

5.1.7 - IDE/Debugger Interaction

The debugger is implemented as a simple state machine running in its own thread and interacting with the graphical interface of the IDE through messages and events [2]. The debugger's loop, keeps on waiting for new events from the GUI such as pause script, single step, step next or run to cursor, and when any event is received the debugger carries those commands accordingly, and finally notifies the GUI about the action's result.

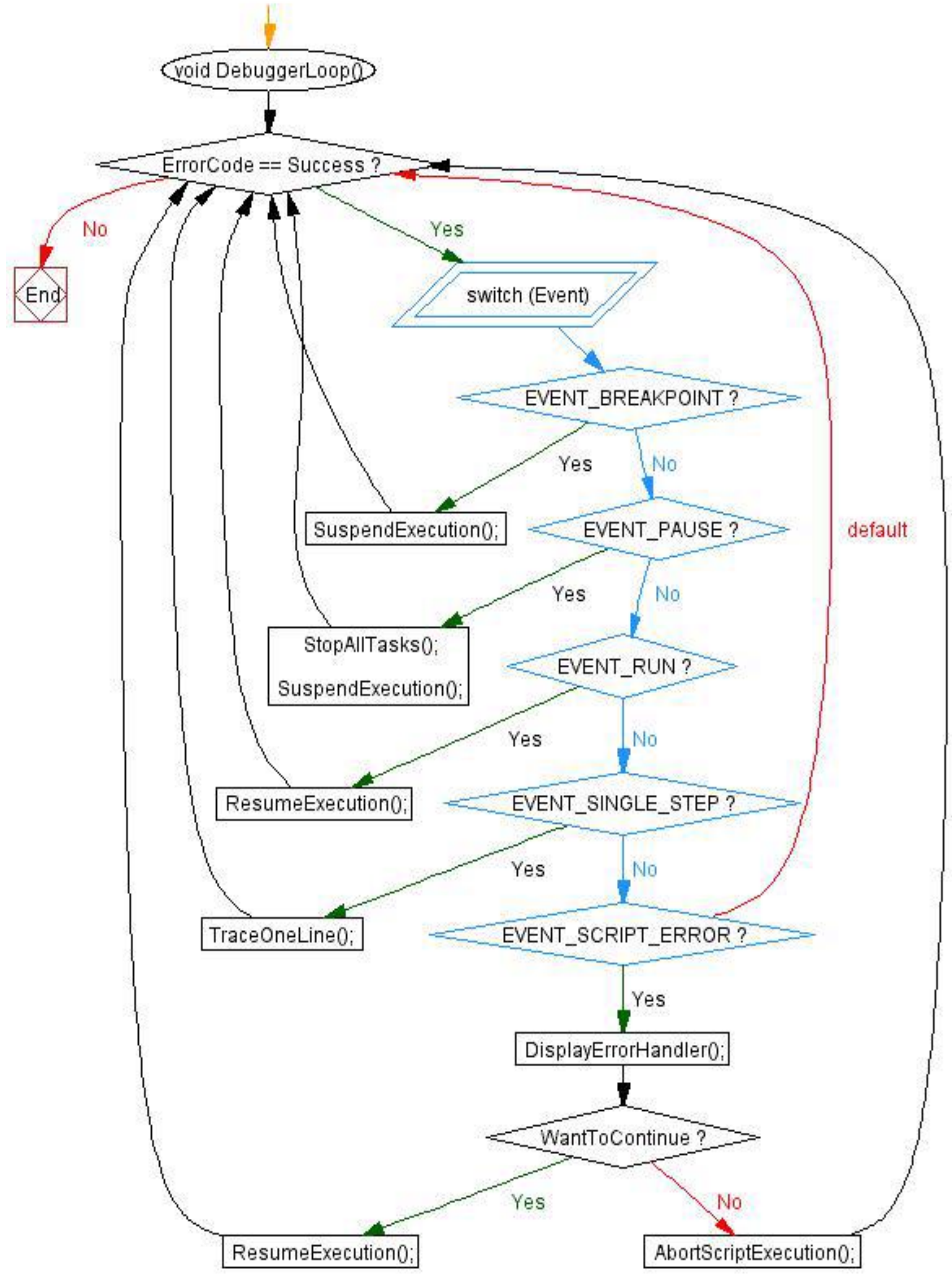


Figure 5.6 - Debugger Subsystem

5.2 - System Structures

In this section we will present UML diagram of the classes and the hierarchical relationship between them. Descriptive text will be provided to make the design strategy used more vivid.

5.2.1 - Engine Structures

The core of the COMPEL engine is the “symbol_t” structure. It is responsible for holding all the four built-in symbols that COMPEL supports and they are the variable, value, object and function.

The “symbol_t” though, not an abstract class, should never be used or instantiated directly, instead it can be used to introduce new kind of symbols other than the “variable_t” and the “function_t” symbols.

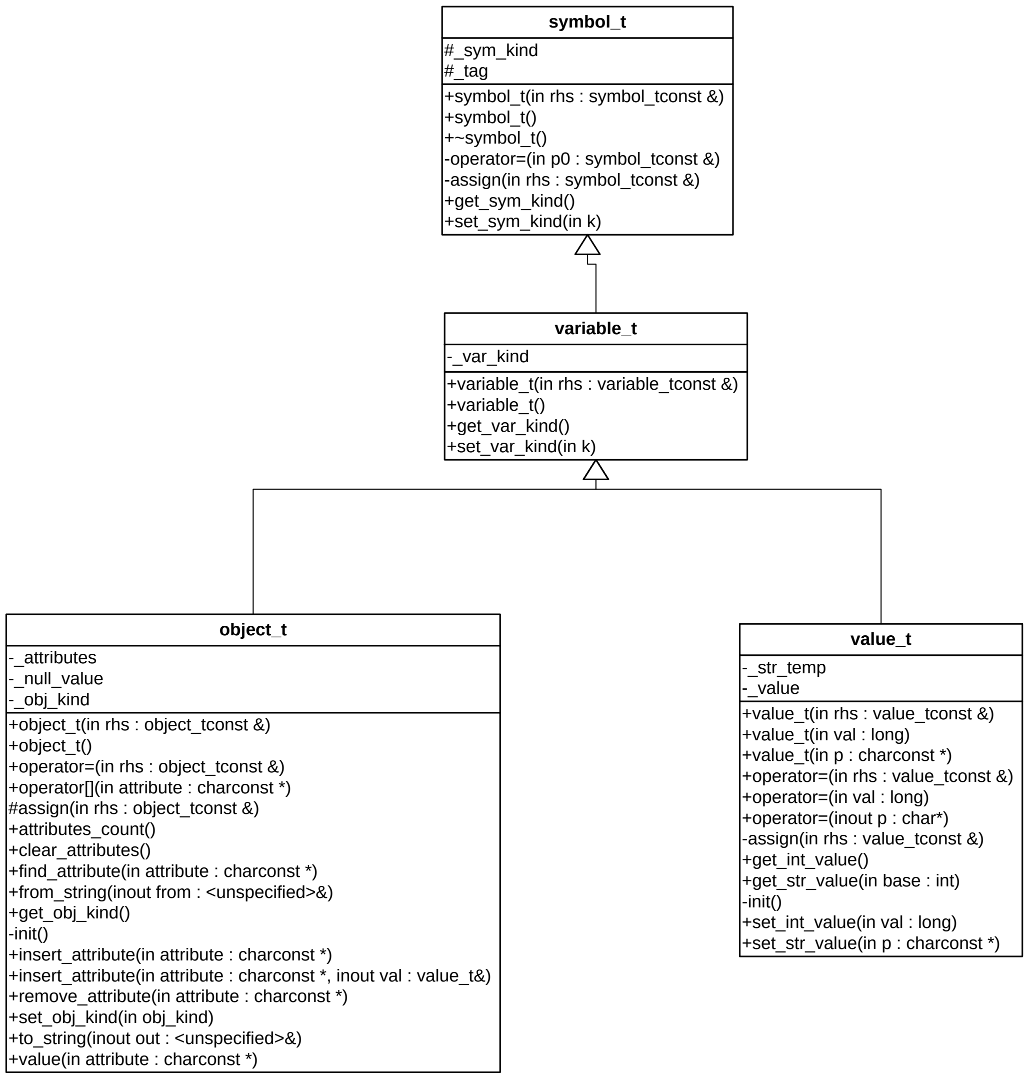

symbol_t
#_sym_kind
#_tag
+symbol_t(in rhs : symbol_tconst &)
+symbol_t()
+~symbol_t()
-operator=(in p0 : symbol_tconst &)
-assign(in rhs : symbol_tconst &)
+get_sym_kind()
+set_sym_kind(in k)
variable_t
-_var_kind
+variable_t(in rhs : variable_tconst &)
+variable_t()
+get_var_kind()
+set_var_kind(in k)
object_t
-_attributes
-_null_value
-_obj_kind
+object_t(in rhs : object_tconst &)
+object_t()
+operator=(in rhs : object_tconst &)
+operator[](in attribute : charconst *)
#assign(in rhs : object_tconst &)
+attributes_count()
+clear_attributes()
+find_attribute(in attribute : charconst *)
+from_string(inout from : <unspecified>&)
+get_obj_kind()
-init()
+insert_attribute(in attribute : charconst *)
+insert_attribute(in attribute : charconst *, inout val : value_t&)
+remove_attribute(in attribute : charconst *)
+set_obj_kind(in obj_kind)
+to_string(inout out : <unspecified>&)
+value(in attribute : charconst *)
value_t
-_str_temp
-_value
+value_t(in rhs : value_tconst &)
+value_t(in val : long)
+value_t(in p : charconst *)
+operator=(in rhs : value_tconst &)
+operator=(in val : long)
+operator=(inout p : char*)
-assign(in rhs : value_tconst &)
+get_int_value()
+get_str_value(in base : int)
-init()
+set_int_value(in val : long)
+set_str_value(in p : charconst *)

Figure 5.7 - Symbol and Variables Hierarchy

It is obvious how the system hierarchy is devised through symbol_t down to object_t and value_t which are the useable types that are implemented in COMPEL engine.

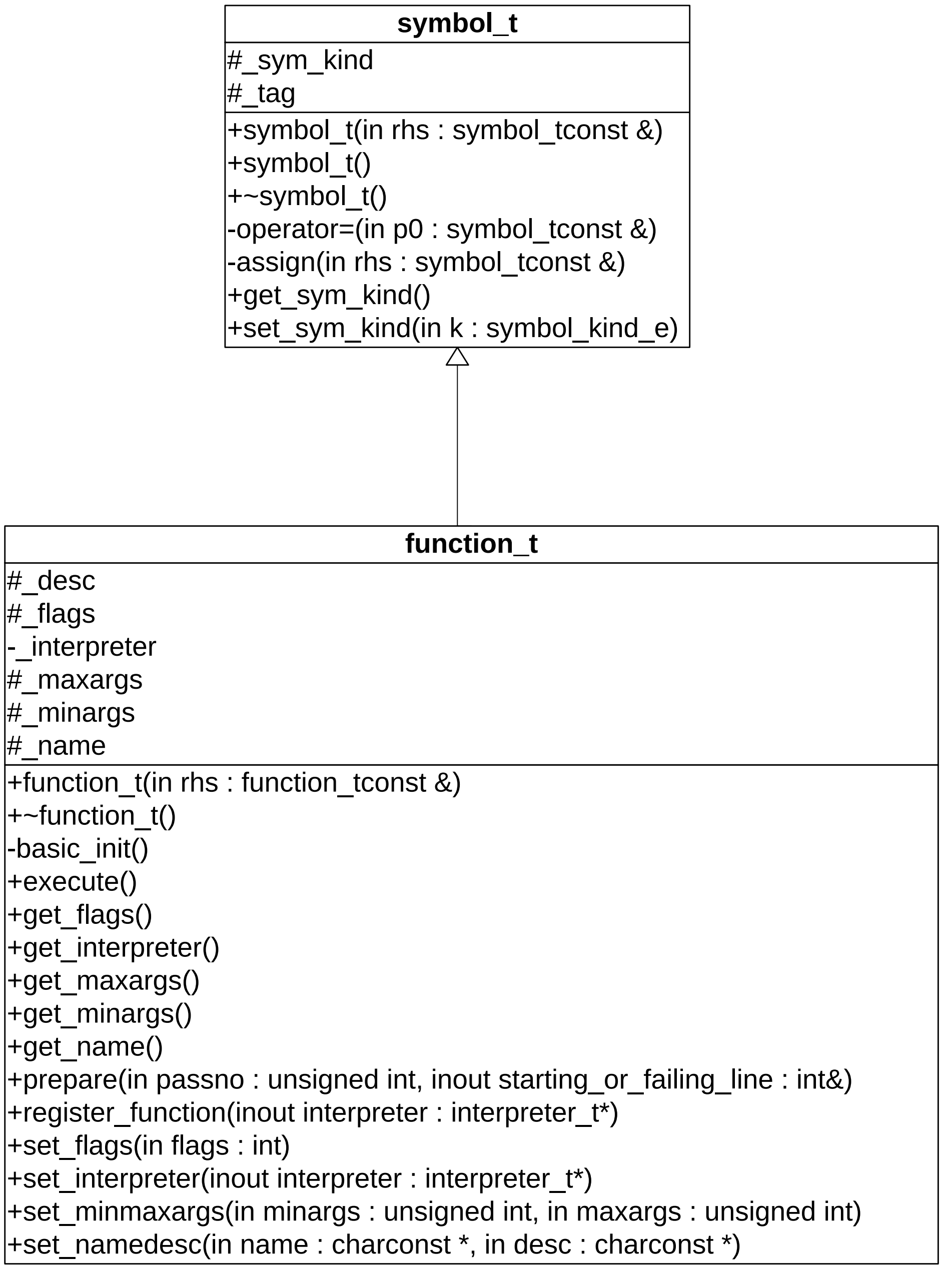
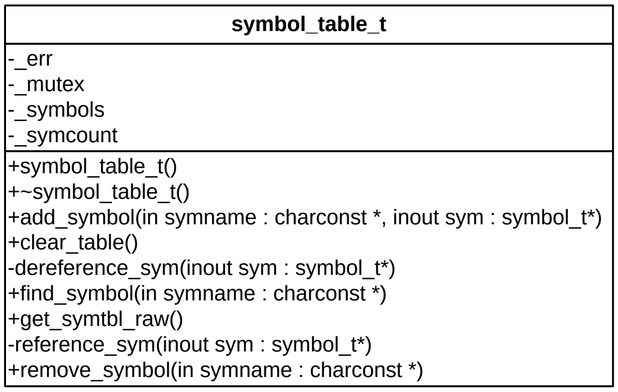

symbol_t
#_sym_kind
#_tag
+symbol_t(in rhs : symbol_tconst &)
+symbol_t()
+~symbol_t()
-operator=(in p0 : symbol_tconst &)
-assign(in rhs : symbol_tconst &)
+get_sym_kind()
+set_sym_kind(in k : symbol_kind_e)
function_t
#_desc
#_flags
-_interpreter
#_maxargs
#_minargs
#_name
+function_t(in rhs : function_tconst &)
+~function_t()
-basic_init()
+execute()
+get_flags()
+get_interpreter()
+get_maxargs()
+get_minargs()
+get_name()
+prepare(in passno : unsigned int, inout starting_or_failing_line : int&)
+register_function(inout interpreter : interpreter_t*)
+set_flags(in flags : int)
+set_interpreter(inout interpreter : interpreter_t*)
+set_minmaxargs(in minargs : unsigned int, in maxargs : unsigned int)
+set_namedesc(in name : charconst *, in desc : charconst *)

Figure 5.8 - Symbol Table and Functions Hierarchy

The two previous figures shows the relationship between symbol_t and the other four types supported by COMPEL. The symbol_t base class has the symbol kind attribute which denotes what symbol kind its subclass is. The variable_t inherits and extends symbol_t. A variable in COMPEL's term is a facility to store either basic values denoted

by value_t or attributed values denoted by object_t. The value_t is a basic data type in COMPEL and can be used interchangibly as a number or a string. The function_t also inherits from symbol_t to allow creation of a built-in extension or user command. A function_t defines a command that COMPEL engine can interpret and execute. Whereas variables are just parameters and values that can be manipulated from within functions.
Now after having illustrated COMPEL's symbols, we will now illustrate the symbol table (symbol_table_t) which is responsible for holding and managing all sort of symbols.

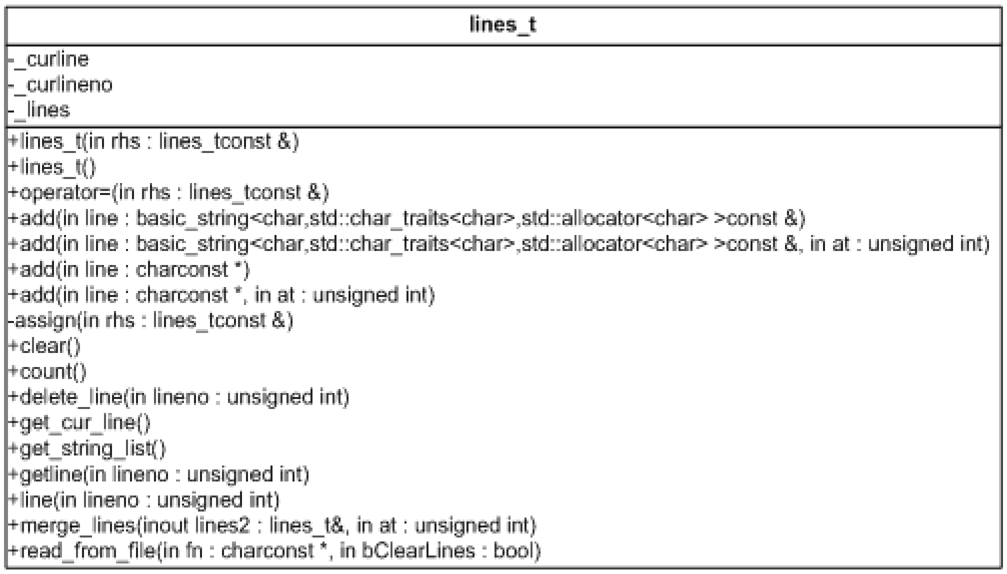


```
symbol_table_t
-_err
-_mutex
-_symbols
-_symcount
+symbol_table_t()
+~symbol_table_t()
+add_symbol(in symname : charconst *, inout sym : symbol_t*)
+clear_table()
-dereference_sym(inout sym : symbol_t*)
+find_symbol(in symname : charconst *)
+get_symtbl_raw()
-reference_sym(inout sym : symbol_t*)
+remove_symbol(in symname : charconst *)
```

Figure 5.9 - Symbol Table Structure

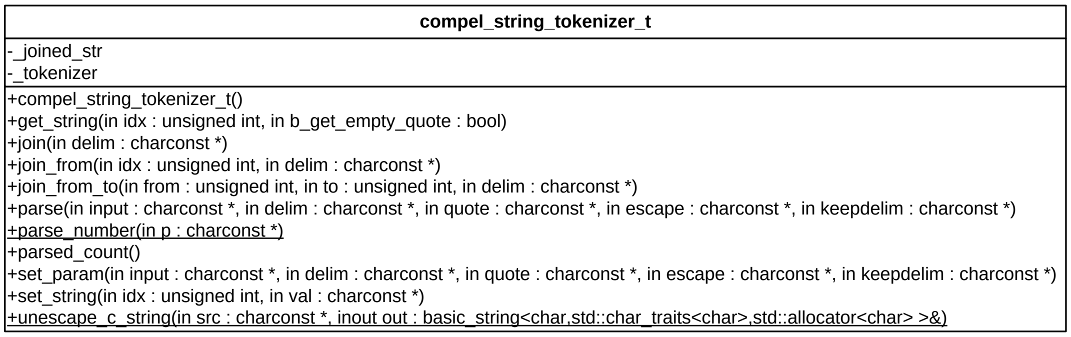
After introducing the symbol types and the symbol table class, we now introduce the structure that will hold all the script lines (Figure 5.10). This structure will allow us to manipulate script lines (by allowing us to add, delete, retrieve and insert lines). The lines class is used as means to separate the script lines management from a specific medium. This way we can read script lines from memory, files or even a remote website.

Now that we have symbol and lines management, we need a way to interpret and analyze each line, for that we will introduce two facilities, the tokenizer (Figure 5.11) and the parser utility module (Figure 5.12).



```
lines_t
-_curline
-_curlineno
-_lines
+lines_t(in rhs : lines_tconst &)
+lines_t()
+operator=(in rhs : lines_tconst &)
+add(in line : basic_string<char,std::char_traits<char>,std::allocator<char> >const &)
+add(in line : basic_string<char,std::char_traits<char>,std::allocator<char> >const &, in at : unsigned int)
+add(in line : charconst *)
+add(in line : charconst *, in at : unsigned int)
-assign(in rhs : lines_tconst &)
+clear()
+count()
+delete_line(in lineno : unsigned int)
+get_cur_line()
+get_string_list()
+getline(in lineno : unsigned int)
+line(in lineno : unsigned int)
+merge_lines(inout lines2 : lines_t&, in at : unsigned int)
+read_from_file(in fn : charconst *, in bClearLines : bool)
```

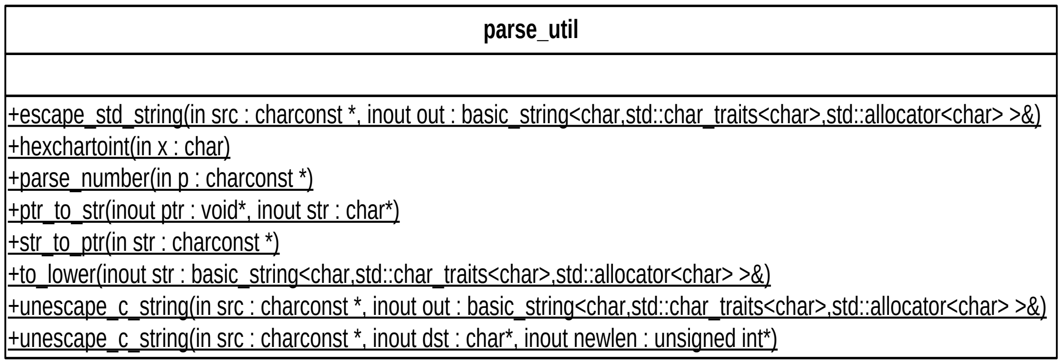
Figure 5.10 - Lines Structure



```
compel_string_tokenizer_t
-_joined_str
-_tokenizer
+compel_string_tokenizer_t()
+get_string(in idx : unsigned int, in b_get_empty_quote : bool)
+join(in delim : charconst *)
+join_from(in idx : unsigned int, in delim : charconst *)
+join_from_to(in from : unsigned int, in to : unsigned int, in delim : charconst *)
+parse(in input : charconst *, in delim : charconst *, in quote : charconst *, in escape : charconst *, in keepdelim : charconst *)
+parse_number(in p : charconst *)
+parsed_count()
+set_param(in input : charconst *, in delim : charconst *, in quote : charconst *, in escape : charconst *, in keepdelim : charconst *)
+set_string(in idx : unsigned int, in val : charconst *)
+unescape_c_string(in src : charconst *, inout out : basic_string<char,std::char_traits<char>,std::allocator<char> >&)
```

Figure 5.11 - COMPEL String Tokenizer

The tokenizer will allow us to split the string into numbered tokens, which we can retrieve and modify later on.

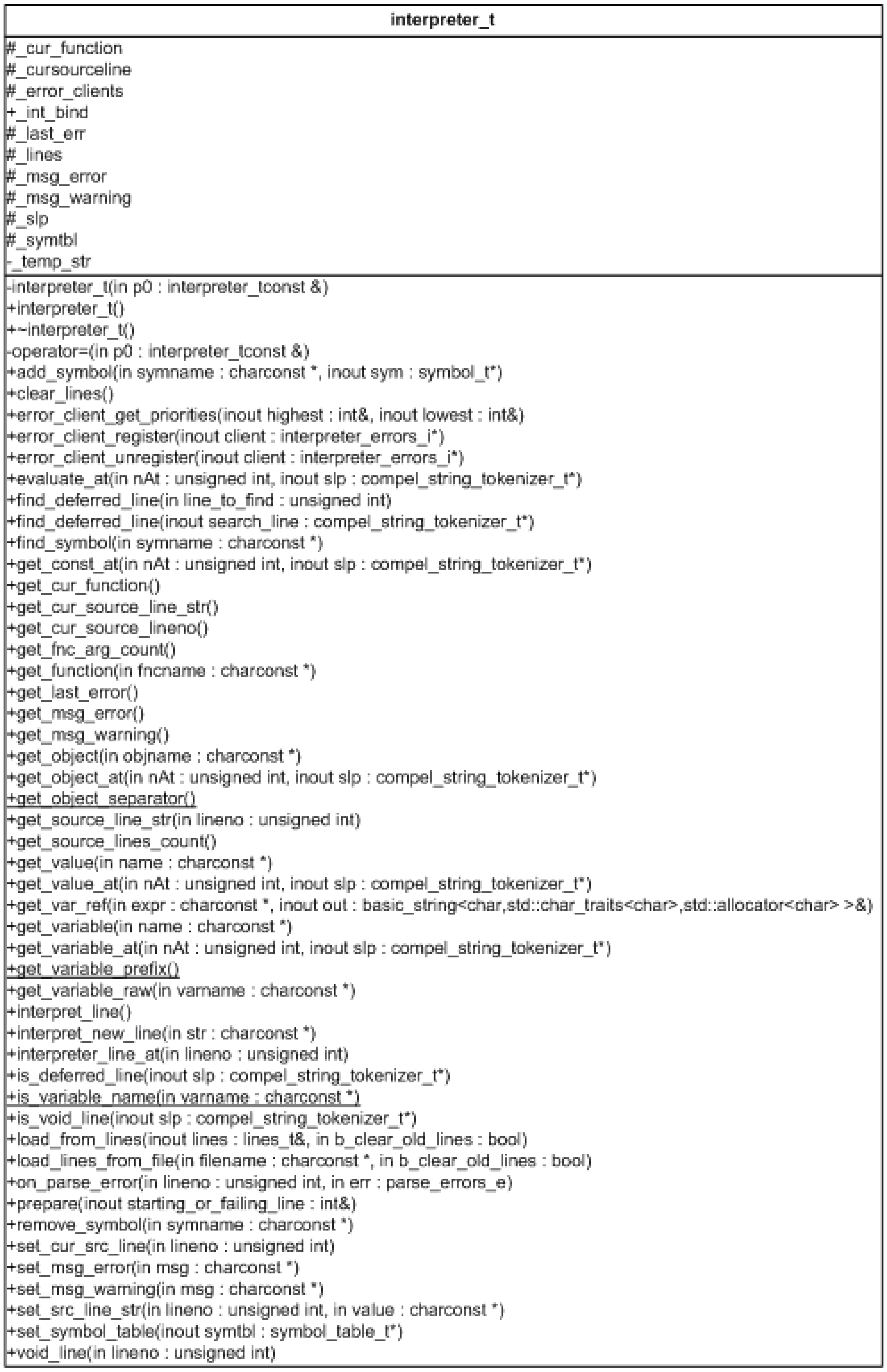


```
parse_util

+escape_std_string(in src : charconst *, inout out : basic_string<char,std::char_traits<char>,std::allocator<char> >&)
+hexchartoint(in x : char)
+parse_number(in p : charconst *)
+ptr_to_str(inout ptr : void*, inout str : char*)
+str_to_ptr(in str : charconst *)
+to_lower(inout str : basic_string<char,std::char_traits<char>,std::allocator<char> >&)
+unescape_c_string(in src : charconst *, inout out : basic_string<char,std::char_traits<char>,std::allocator<char> >&)
+unescape_c_string(in src : charconst *, inout dst : char*, inout newlen : unsigned int*)
```

Figure 5.12 - Parse Util Structure

The parse utility will provide many utility functions such as string to number conversion, string formatting, etc. Now that we have symbols and symbol table, lines and tokenizer to parse the script, we need another mechanism to glue the whole thing as one piece and this is done by the interpreter structure. The interpreter structure holds the lines (script code), error handlers list, the registered symbols (variables, and functions) the current execution point, the last error and other miscellaneous internal variables.

The most important functions of the interpreter class is its ability to talk with all subsystems such as the lines class, the symbol table and the variables types. In addition the interpreter provides a mean to validate a given script line, and to return the value of script variable disregarding its type. Basically, the interpreter is the heart and the driving module of the whole COMPEL engine, since it drives all the other components and modules to run a given script.



```
interpreter_t
-----------------------------------------------------------------
#_cur_function
#_cursourceline
#_error_clients
+_int_bind
#_last_err
#_lines
#_msg_error
#_msg_warning
#_slp
#_symtbl
-_temp_str
-----------------------------------------------------------------
-interpreter_t(in p0 : interpreter_tconst &)
+interpreter_t()
+~interpreter_t()
-operator=(in p0 : interpreter_tconst &)
+add_symbol(in symname : charconst *, inout sym : symbol_t*)
+clear_lines()
+error_client_get_priorities(inout highest : int&, inout lowest : int&)
+error_client_register(inout client : interpreter_errors_i*)
+error_client_unregister(inout client : interpreter_errors_i*)
+evaluate_at(in nAt : unsigned int, inout slp : compel_string_tokenizer_t*)
+find_deferred_line(in line_to_find : unsigned int)
+find_deferred_line(inout search_line : compel_string_tokenizer_t*)
+find_symbol(in symname : charconst *)
+get_const_at(in nAt : unsigned int, inout slp : compel_string_tokenizer_t*)
+get_cur_function()
+get_cur_source_line_str()
+get_cur_source_lineno()
+get_fnc_arg_count()
+get_function(in fncname : charconst *)
+get_last_error()
+get_msg_error()
+get_msg_warning()
+get_object(in objname : charconst *)
+get_object_at(in nAt : unsigned int, inout slp : compel_string_tokenizer_t*)
+get_object_separator()
+get_source_line_str(in lineno : unsigned int)
+get_source_lines_count()
+get_value(in name : charconst *)
+get_value_at(in nAt : unsigned int, inout slp : compel_string_tokenizer_t*)
+get_var_ref(in expr : charconst *, inout out : basic_string<char,std::char_traits<char>,std::allocator<char> >&)
+get_variable(in name : charconst *)
+get_variable_at(in nAt : unsigned int, inout slp : compel_string_tokenizer_t*)
+get_variable_prefix()
+get_variable_raw(in varname : charconst *)
+interpret_line()
+interpret_new_line(in str : charconst *)
+interpreter_line_at(in lineno : unsigned int)
+is_deferred_line(inout slp : compel_string_tokenizer_t*)
+is_variable_name(in varname : charconst *)
+is_void_line(inout slp : compel_string_tokenizer_t*)
+load_from_lines(inout lines : lines_t&, in b_clear_old_lines : bool)
+load_lines_from_file(in filename : charconst *, in b_clear_old_lines : bool)
+on_parse_error(in lineno : unsigned int, in err : parse_errors_e)
+prepare(inout starting_or_failing_line : int&)
+remove_symbol(in symname : charconst *)
+set_cur_src_line(in lineno : unsigned int)
+set_msg_error(in msg : charconst *)
+set_msg_warning(in msg : charconst *)
+set_src_line_str(in lineno : unsigned int, in value : charconst *)
+set_symbol_table(inout symtbl : symbol_table_t*)
+void_line(in lineno : unsigned int)
```

Figure 5.13 - Interpreter Structure

The interpreter makes use of all the previously introduced structure. Generally the interpreter works like this:

1. An instruction is fetched
2. Instruction is tokenized
3. Syntax is checked
4. Command is looked from symbol table
5. If no command found or syntax is wrong, we report error
6. If command and syntax are acceptable we dispatch execution to the corresponding command
7. The command might talk with the interpreter in order to fetch variables
8. Control is returned to interpreter
9. Interpreter fetch another instruction or ends if no more instructions are there

Due to programming disciplines and concepts, the interpreter structure does not have direct access to the symbol table, thus a new structure is introduced "interpreter_helper_t" which acts as a proxy between the interpreter and symbol table.

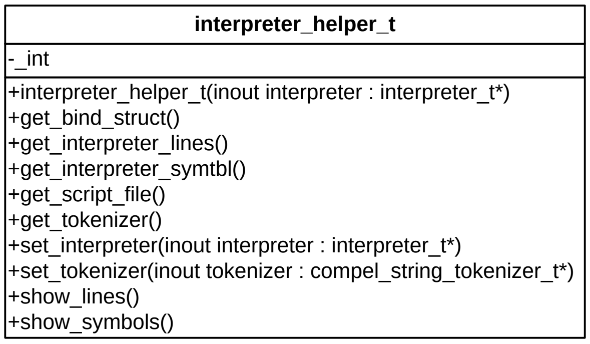

interpreter_helper_t
-_int
+interpreter_helper_t(inout interpreter : interpreter_t*)
+get_bind_struct()
+get_interpreter_lines()
+get_interpreter_symtbl()
+get_script_file()
+get_tokenizer()
+set_interpreter(inout interpreter : interpreter_t*)
+set_tokenizer(inout tokenizer : compel_string_tokenizer_t*)
+show_lines()
+show_symbols()

Figure 5.14 - Interpreter Helper Structure

The following structure shows how one can implement / add a new built-in command to COMPEL engine, through extending the function_t structure.

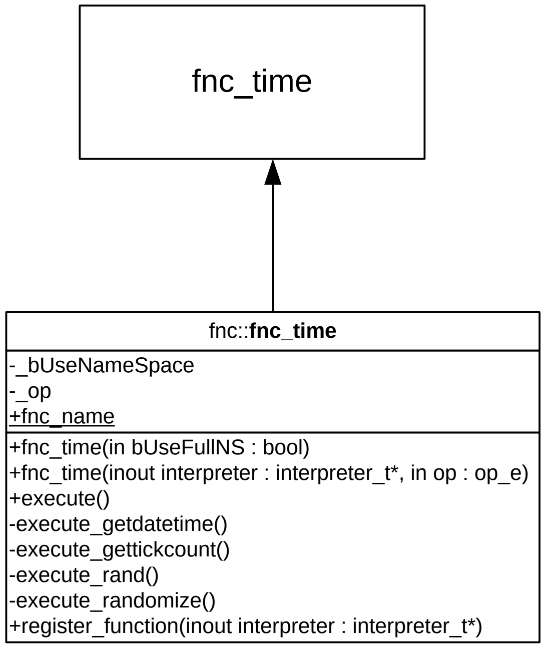
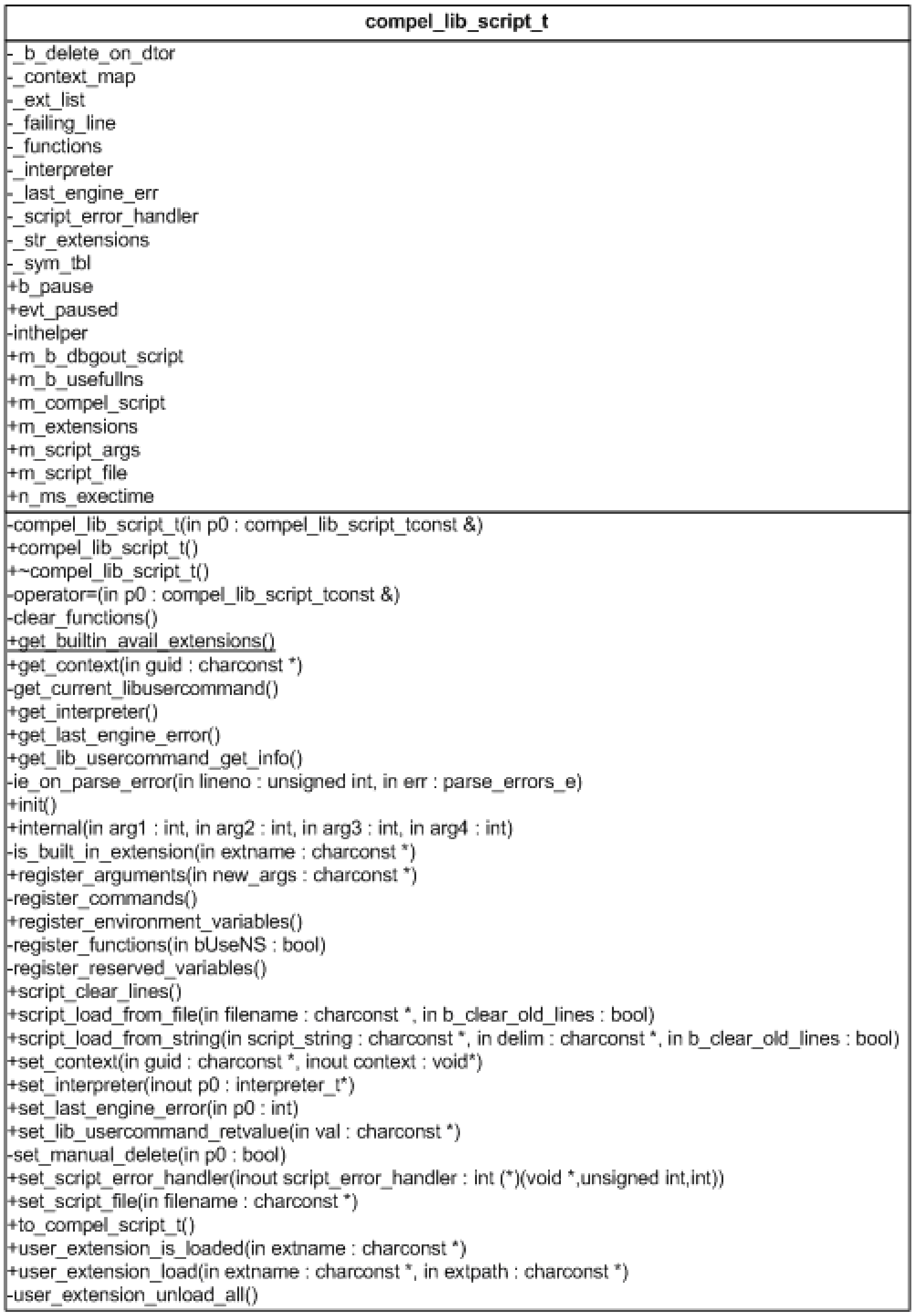

fnc_time
fnc::fnc_time
-_bUseNameSpace
-_op
+fnc_name
+fnc_time(in bUseFullNS : bool)
+fnc_time(inout interpreter : interpreter_t*, in op : op_e)
+execute()
-execute_getdatetime()
-execute_gettickcount()
-execute_rand()
-execute_randomize()
+register_function(inout interpreter : interpreter_t*)

Figure 5.15 - User Extension Sample - Time Extension

Every user extension has to implement the execute() and register_function() methods. After the user's extension class is built, one has to register the class in the symbol table associated with the interpreter. Check the appendix and the package files to learn how to technically achieve that.

5.2.2 - Library Structures

In section 5.2.1 we explained the structure of the COMPEL engine. Now we will explain the structure of the COMPEL developer library. In fact, the library is nothing but a wrapper against the interpreter and the few helper classes.



```
compel_lib_script_t
-_b_delete_on_dtor
-_context_map
-_ext_list
-_failing_line
-_functions
-_interpreter
-_last_engine_err
-_script_error_handler
-_str_extensions
-_sym_tbl
+b_pause
+evt_paused
-inthelper
+m_b_dbgout_script
+m_b_usefullns
+m_compel_script
+m_extensions
+m_script_args
+m_script_file
+n_ms_exectime
-compel_lib_script_t(in p0 : compel_lib_script_tconst &)
+compel_lib_script_t()
+~compel_lib_script_t()
-operator=(in p0 : compel_lib_script_tconst &)
-clear_functions()
+get_builtin_avail_extensions()
+get_context(in guid : charconst *)
-get_current_libusercommand()
+get_interpreter()
+get_last_engine_error()
+get_lib_usercommand_get_info()
-ie_on_parse_error(in lineno : unsigned int, in err : parse_errors_e)
+init()
+internal(in arg1 : int, in arg2 : int, in arg3 : int, in arg4 : int)
-is_built_in_extension(in extname : charconst *)
+register_arguments(in new_args : charconst *)
-register_commands()
+register_environment_variables()
-register_functions(in bUseNS : bool)
-register_reserved_variables()
+script_clear_lines()
+script_load_from_file(in filename : charconst *, in b_clear_old_lines : bool)
+script_load_from_string(in script_string : charconst *, in delim : charconst *, in b_clear_old_lines : bool)
+set_context(in guid : charconst *, inout context : void*)
+set_interpreter(inout p0 : interpreter_t*)
+set_last_engine_error(in p0 : int)
+set_lib_usercommand_retvalue(in val : charconst *)
-set_manual_delete(in p0 : bool)
+set_script_error_handler(inout script_error_handler : int (*)(void *,unsigned int,int))
+set_script_file(in filename : charconst *)
+to_compel_script_t()
+user_extension_is_loaded(in extname : charconst *)
+user_extension_load(in extname : charconst *, in extpath : charconst *)
-user_extension_unload_all()
```

Figure 5.16 - COMPEL Library Internal Class

The compel_lib_script_t structure is just a wrapper around the interpreter, symbol table and utility classes. Every functionality provided by the COMPEL Library is implemented in this class. In order not to export a class to developers (which is not portable among different programming languages), another non-object oriented / procedural style wrapper is created and implemented through compel_lib.dll itself.

One last thing to cover in this session is COMPEL's engine ability to cope with error through registered error handlers.

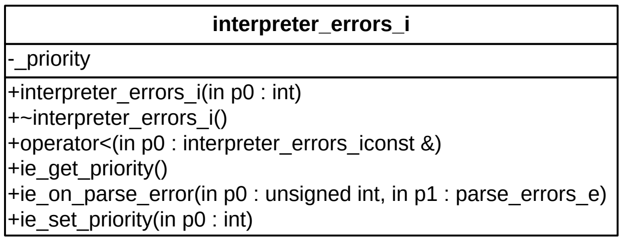

interpreter_errors_i
-_priority
+interpreter_errors_i(in p0 : int)
+~interpreter_errors_i()
+operator<(in p0 : interpreter_errors_iconst &)
+ie_get_priority()
+ie_on_parse_error(in p0 : unsigned int, in p1 : parse_errors_e)
+ie_set_priority(in p0 : int)

Figure 5.17 - Error Handler Structure

To understand more about the interpreter_errors_i interface, please refer to the error handler flow chart for more information.

5.2.3 - IDE Structures

In the previous sections we explained the functionality and internals of the IDE. In this section we will show the data structures employed in the IDE that were not covered previously.

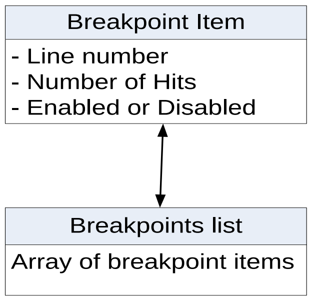


Figure 5.18 - Breakpoint List / Item structure

This structure is used to store all the breakpoints for a given script file. The list is then passed to the COMPEL engine.

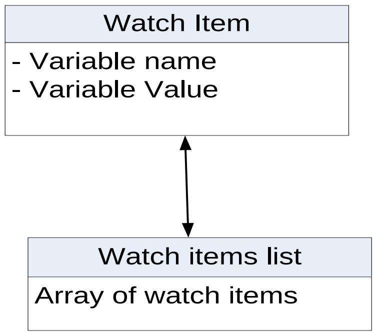


Figure 5.19 - Watch List / Items Structure

This structure is used by the debugger window. On each instruction execution, the IDE queries the COMPEL Library for the values of all the watch items in the list.

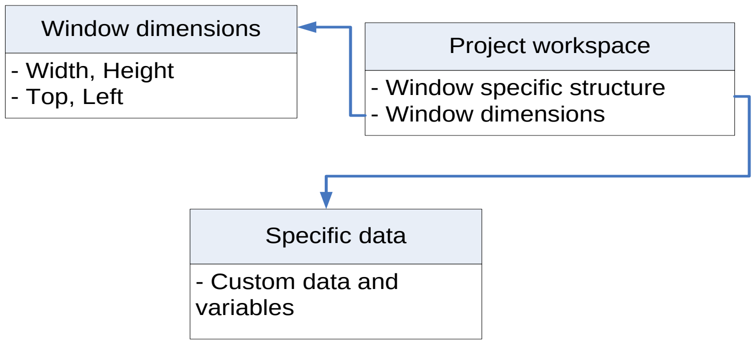


Figure 5.20 - Workspace Data Structure

The IDE saves the workspace information of the Watches, Editor, Breakpoint and main window. For this to happen, each window's dimensions is saved, then for every window, a specific data structure is saved as well in the workspace file.

5.3 - Implementation Language

Careful consideration was taken when choosing the programming language for COMPEL Library (or engine) and the COMPEL IDE. The main concern was the speed of execution and portability of the code.

5.3.1 COMPEL Library Implementation Choice

COMPEL engine was developed in C++ language to address the requirements dictated by Portability, Speed, Inter-language connectivity, OOP design, and closeness to the operating system [3]. When we talk about portability that means the code shall be ported to other platforms in the future, as for speed we chose C++ compilers which are known to generate fast and optimized code (speed and size wise) that runs through the host machine rather than executing inside a Virtual Machine (for instance Java). For inter-language connectivity C++ was chosen because it can easily communicate with most programming languages and because C++ is OOP capable and well suited for the high level of modular design in most of COMPEL's components.

COMPEL Library was developed with Visual C++ 2005 and compiled as a dynamic link library (DLL). DLLs are a very important feature of Windows Operating system. They allow any native windows program to load and use them. COMPEL Library can be used from any programming language that can use windows DLL, such as Visual Basic 6, Borland Delphi, Microsoft .NET, and most Win32 C/C++ compilers.

5.3.1 COMPEL IDE Implementation Choice

The COMPEL IDE was developed using Borland Delphi compiler. Delphi is an object oriented version of the Pascal language [5]. Delphi comes with a VCL (Visual Component Library), which makes Delphi essentially a RAD (Rapid Application Development) tool. For rich and fast Win32 GUI application development, Delphi is a good choice. After the COMPEL IDE GUI was developed, the "Debugger subsystem" was completed through the implementation of COMPEL Library (or engine). Then the editor was implemented through the use of a third-part free editing component, where as the rest of the IDE features are all native within Delphi's VCL. Thus as we see, COMPEL IDE is nothing more than just an implementation of COMPEL library, Delphi's various VCL component and the logic that binds all the subsystems together.

5.4 - Required Support Software

In order to compile and build COMPEL library one has to have only the Microsoft Visual

Studio 2005 C++ compiler with the platform SDK installed. As for compiling and building the COMPEL IDE, one would require Borland Delphi (A rapid application development tool from Borland) and the third party components used through development, and in that case it was only the SynEdit component which is an advanced multi-line edit control for Delphi and Borland C++ Builder.

This chapter presented a more in-depth view of the system using computer science terms, diagrams and flowchart. The next chapter will present the results and conclusions, future development and closing words.

Chapter 6
Conclusions and Future Work

Outline

In this final chapter we display our results, conclusions, future work and problems we encountered, challenges and solutions for those challenges.

6.1 - Results

The design and development of COMPEL Library and IDE took around 720 working hours (what is equivalent to 8 hours per day for 3 month) and almost 11,000 lines of code (between C++ and Delphi code). Our aim throughout the development stage was to make sure that COMPEL meets its end goals and though we have encountered many design problems, coding and technical difficulties, requirements changes and time pressure in order to complete the project in time, we are proud to see that COMPEL indeed met its creation goals:

- A developer library to facilitate command based interpreter creation
- A comprehensive API (COMPEL API) allowing developer to access every aspect of the engine
- Modular design that allows future enhancement and changes in the engine
- A decent IDE with a powerful editor and debugger
- A slick debugger with many features and capabilities

6.2 - Problems and Challenges

Many problems and challenges were faced during the development of the system, most of the problems were solved and some where put-off for future enhancement of the system.

6.2.1 - COMPEL Library Problems and Challenges

During the development of COMPEL system, many challenges occurred from technical challenges such as how to write a specific component using a programming language or how to write and think of an algorithm then express it in computer programming language terms. The following section will list some of the most interesting challenges faced and how they were solved.

6.2.1.1 - Forward Declaration and Pre-Parsing / Multiple passes

COMPEL does not have any built-in commands, however when built-in extensions were introduced, many limitations were imposed. One big limitation was how to implement forward declarations. Initially COMPEL engine did no pre-parsing since a command based interpreter interprets one command at a time without the need to know what is ahead. However, because unconditional branching, scopes and user commands are all useful additions, we had to re-introduce a way to allow pre-parsing of scripts.

6.2.1.2 - Maintain Loose and Independent Connections between Extensions

Since COMPEL is not a language by itself, syntax like “if” or “else” and even the opening and closing curly braces “{“ and “}” were implemented as separate commands each with its own set of logic and strategy. Actually, every extension has a design of its own and documenting those extensions is out of the scope of this document because these extensions are just a convenience and not a part of the COMPEL engine.

6.2.1.3 - Threads Implementation and Script Code and Symbol Table Sharing

Implementing threads extension was another challenge which imposed its own problems which were never thought off. The following issues were faced:

- COMPEL was not designed to handle concurrent accesses to its modules. COMPEL is not thread-safe.
- How to share the same symbol table with multiple threads given that a thread can overwrite or destroy a variable that is being used by another thread.
- How to implement true multithreading from a scripting environment. We know that multithreading is an operating system provided service.
- How to have each thread run at different script lines (i.e: each thread has its own execution point).

Some of these issues were addressed and other issues were left for future work. Now COMPEL have experimental thread support, however one must not use user commands, instead just built-in commands. Threads were implemented through a smart implementation of operating system thread support

6.2.1.4 - User Commands

Creating user commands or functions (as called in high-level programming language) was a big challenge. We faced issues like how to maintain correct command call stack and parameter passing, how to assure inherent and natural support for recursive command calls, the issue of creating and designing a special local variable available only in the scope of a command call to support command calls parameters. Another issue we faced was when we tried to introduce threads and how they should have separate call stacks so that a function called from a thread does not alter the execution of another script in the main thread. We had to write special test scripts to prove that all these problems were tackled successfully.

6.2.1.5 - COMPEL Library API Design

The design and selection of every COMPEL API was a big challenge. We were supposed to select and craft function prototypes that will assure complete access to the scripting engine in a convenient manner to programmers with different sort of background knowledge. Through our experience with working with many APIs (such as Microsoft's Win32 API) and through working with 3rd party SDKs, we were able to create an API as efficient as those libraries.

A proof for our success in creating the API was the development of the COMPEL IDE using a completely different programming language (DELPHI) which implements COMPEL library. The IDE made extensive use of every aspect of the API, and even created a powerful debugger. This demonstrates the wide range abilities that one can do through the API. At a later stage we also implemented a demonstration applications using

Visual C# .NET to illustrate that the COMPEL Library can also be used in managed languages.

6.2.1.6 - COMPEL Library Wrappers

We were challenged to wrap or code classes and libraries so that the COMPEL Library DLL can be used seemingly from other than C++ languages. We successfully managed to wrap the library for Borland Delphi and implemented that wrapper in the IDE. Wrappers for .NET framework were a challenge as well. Here we would like to acknowledge all those who helped us with the creation of COMPEL Lib for .NET

6.2.2 - COMPEL IDE Problems and Challenges

We decided to create an IDE only if we can create a powerful debugger for it. We pondered on how to create the debugger subsystem, how it should interact with the COMPEL Library and the IDE's graphical interface, and how to emulate debugger states (run, suspend and stop) in a scripting engine that has no notion of debugging.

6.2.2.1 - Emulating a Debugger

As you look in the following appendix you will notice that COMPEL_LIB mentions nothing about debugging but we managed to emulate the debugging states through the use of just one COMPEL function, the compel_script_step(), which allowed us to run one instruction at a time. This API was enough for us to implement “Run”, “Run to cursor”, “Pause” and “breakpoints” commands.

6.2.2.2 - Binding Debugger States Correctly with the GUI

Efficiently updating the GUI according to the IDE and debugger states was of a challenge. We successfully solved this problem through the use of a IDE/Debugger GUI state table. The state table is similar to a state machine, however we were able to implement that state machine in a tabular form.

6.2.2.3 - Writing Custom GUI Components

Not every visual aspect in the COMPEL IDE is part of Delphi RAD environment. We had to write our own components and sometimes use free 3rd party components. Such was the case with the Debugger Window and how it highlights the breakpoint lines and the current executing line. Also in the case of the editor control, a 3rd part component was used.

6.2.2.4 - Writing Additional Extensions to Promote the IDE

In order to promote the IDE, we wrote an additional extension named “canvas” so that we enrich the COMPEL language with more extensions and commands. Check figure “COMPEL External Extension - Canvas”

6.3 - Enhancement and Future Work

An idea is like a snowball, it begins a small one then when enough force and intention is applied unto it, it rolls and grows till it becomes huge. This was the case when we developed COMPEL, everything started small, then ideas started to flow and forced us to change the requirements and initial plans, however, we had to put-off lots of ideas for later implementation. In this section, we will talk about future enhancement and work that can be done to both the library and the IDE.

6.3.1 - COMPEL Library Enhancements

The COMPEL Library was developed using the iterative and throw-away prototyping methods. This means that the design and requirements underwent lots of changes during the development process. After we coded this initial version of COMPEL, we can re-write the whole thing all over again taking into consideration all the problems and challenged we passed through. We can improve the following:

- Make COMPEL thread safe
- Make some built-in extensions as reserved words and allow COMPEL to handle them as part of its language processing
- Better engineering of script threads
- Faster pre-parsing techniques
- Better and faster tokenizing techniques
- Advanced type promotion and demotion between values and different kind of COMPEL objects

6.3.2 - IDE enhancements

We all know that when it comes to GUI, we can improve in an unlimited manner. However there are essential and convenient things that can be improved about the current IDE:

- Better editor with syntax highlighting
- Editor and Debugger become in the same window (no more separate debugger and editor window)
- Support for concurrent scripts editing at the same time
- Support for concurrent scripts debugging
- IDE should be able to attach itself to any COMPEL script running in any process, thus allowing one to debug any script hosted in any program

COMPEL was a big project. We have learned a great deal throughout the three month of extensive work. We hope that by reading this document you get all the inspiration we took from this quest.

References

[1] Larry Nyhoff, *C++: An Introduction to Data Structures*. New Jersey: Prentice Hall, 1996.

[2] Jonathan B. Rosenberg, *How Debuggers Work*, Wiley. New York: John Wiley, 1996.

[3] Bjarne Stroustrup, *The C++ Programming Language*, Special Edition. Boston: Addison-Wesly, 2003

[4] Ian Sommerville, *Software Engineering*, Eighths Edition. Boston: Addison-Wesley, 2007

[5] Borland Developers Network, http://www.borland.com/

Appendices

Outline

Appendix A - Glossary

Application Programming Interface	A set of programming interfaces and libraries for programmers. In other terms, APIs are a set of functions that are to be called by developers.
COMPEL	[verb] To give an order
Compiler	A compiler is a software that translate a program written by a programmer into another program that is comprehensible by the machine.
Console Window	In Windows operating system, a console window is similar to DOS 6. It is a character based window as opposed to GUI windows which is graphical.
Debugger	A program that helps in locating and correcting programming errors.
Dynamic Link Library	It is an external module that can be bound to your module dynamically (while your program is running)
Execution Point	An execution point is a location in a program indicating what line is being executed
Extension / Plug-in	A plug-in or an extension is an additional module added to a program through the use of that program's API.
Function / Command	These are a way to describe a group of instructions understood by the computer program.
Integrated Development Environment	This a program that features a text editor (optionally with keyword highlighting), a set of tools and a debugger all bound in the same interface.
Interpreter	A software or function that translates a given command string into a computer comprehensible action
Language / Syntax / Grammar	Language / Syntax or Grammar specifies the format and keywords used when writing a computer program. Different programming languages have different syntax / grammar.
Object	An object is a symbol that can hold multiple symbols named "attributes". Every attribute in an object behave as if they were standalone symbol. Object can be compared to a box full of items.
Rapid Application Development	A development tool featuring lots of facilities to easily design and programming GUI applications. Usually, RAD employs frameworks and components
Script	A script is a set of instructions that have a defined syntax understood by an interpreter
Software Development Kit	A software development kit is a complete kit for software developers comprised of: - A set of APIs implemented in libraries - API documentation - A set of examples (optionally) illustrating how to use the SDK
Static Library	As opposed to DLLs, a static library is a module that is linked to your module at the time of compilation.
Symbol	A symbol is a name for an entity that can be of any type. For example a variable is a symbol of type variable.
Variable	Like in mathematics, a variable is name for a value that can be changed after a assignment or a certain formula was applied.
Win32	Win32 is a short hand for Windows for 32bit operating systems.

Abbreviations

API	Application Programming Interface

CLI	Command Line Interface
DLL	Dynamic Link Library
GUI	Graphical User Interface
IDE	Integrated Development Environment
RAD	Rapid Application Development
SDK	Software Development kit

Appendix B - COMPEL Built-In extensions reference

Outline

Variables Terminology

COMPEL engine recognizes variables and objects embedded in a command and replaces them with their appropriate values. COMPEL recognizes variables only if they are preceeded by “$” sign. Variables preceeded with “&” are not parsed directly and considered a call by reference similarly to other high level languages.

Value substitution

Let us consider a simple script:

```
var $age 18
echo “The age is “ $age “\n”
```

This script will cause the output:

```
The age is 18
```

Object substitution

Objects are replaced by their “to_string()” representation. Base object of type “object_t” has a default “to_string()” which simply displays all the attributes of the object and their values.

```
var $info.

var $info.name “elias”
var $info.lastname “b.”
echo “The info is:” $info “\n”
```

This will cause an output similar to:

```
The info is: {name: “elias”, lastname: “b.”}
```

List of built-in extensions

COMPEL comes with a set of built-in extension created to enrich the scripting engine and to diminish the need for developers to create basic extensions. Such built-in extensions deal with variable management, Control Structure and Arithmetic operations.

Arithmetic	Allows you to do basic arithmetic operations
Comments	Adds support for single and multi-line C++/PHP style comments
Comparison	Adds assembly style binary comparison support
COMPEL_LIB	Allows the script to manage the compel library
CRT	Screen and keyboard related operations
Directory	Directory and file system enumeration operations
File I/O	File manipulation
Include	Inclusion of other scripts
Memory objects	Memory buffer and objects manipulation
Scopes	Adds the support of repetition structures and C style conditional branching operations
Shell	Adds support to run commands through system shell
Strings	String manipulation operations
Threads	Thread creation operations
Time	Time management operations
Unconditional Branching	Unconditional branching and GOTO support
Variables	Variables and objects creation and manipulation

Variables - Extensions

The variable extension's purpose is to provide variables creation and manipulation

Command	Description
alias	Creates a variable or command alias
assign	Similar to "var" command but does not create instead it just update the value.
unvar	Deletes a variable or an object
var	- Creates a variable - Creates an object - Creates an attributes within an object - Updates a value

Namespace: none

Check the fcd_vars*.compel sample series

CRT - Extensions

Command	Description
`delay`	Causes the execution of the script to delay a specified number of milliseconds
echo	Outputs a value to the console window. If no console was there, the first call to this command will create a console
echoln	This command is similar to calling "echo" then writing a "\n" (new line)
`echoxy`	This command is the combination of "echo" and "gotoxy"
`getch`	Waits for a key press inside the console window
`getxy`	Returns the console's X and Y coordinates
`gotoxy`	Sets the cursor location in the console window. Where X (horizontal axis) and Y (vertical axis) has (1,1) as origin
`inputbox`	Prompts for a value using a graphical input box
`inputline`	Waits for a line input inside the console window
`msgbox`	Displays a message using a message box.
`textattr`	Changes the text foreground and background color
`yesnobox`	This command displays a graphical message box with "Yes" and "No" prompts

Namespace: "crt"

Check the fcd_crt*.compel sample series

Arithmetic - Extensions

Command	Description
`add/sub/div/mul`	Achieves addition, substraction, integer division, and multiplication
expr	This is a full expression evaluator allowing the following operators: • \ : integer division

	• * : multiplication • % : Modulo • + : Addition • - : Substraction • ^ : Bitwise XOR • & : Bitwise AND • | : Bitwise OR • && : Logical AND • || : Logical OR • !| : Logical NOR • !& : Logical NAND • == : Integer equality test • != : Not equal • << : Bitwise shift left • >> : Bitwise shift right • < : Less than • > : Greater than • >= : Less than or equal • <= : Greater than or equal

The “expr” command may allow complex expressions combined with variables, such as:
expr $result ($delta + 1900) * $height / $width

Namespace: “math”

Check the fcd_arith*.compel sample series

Comments - Extensions

The comments extension is responsible for introducing commands with no side-effects such as comments in High-Level functions.
For example the command:
# This is a command
The “#” does nothing and ignores its parameters.

The following comment syntax is supported: “//”, “#” for single line comment, and “/*” and “*/” for multi-line comment like in C language

Check the fcd_comment*.compel sample series

COMPEL_LIB - Extensions

This extension allows the script to talk with the COMPEL_LIB directly.

Only one command is exposed at the moment.

”dl”: Load external extensions.

Check the “canvas” examples in the “binaries” folder

Mandatory Namespace: “compellib”

Directory - Extensions

Command	Description
`enumfiles`	Enumerate files in the specified directory
`enumdrives`	Enumerates system drives
`Getdir`	Returns the current directory
`chdir`	Changes the current working directory
`fileexists`	Checks if a file exists
`direxists`	Checks if a directory exists

Namespace: “dirs”

Check the fcd_dirs*.compel sample series

File I/O - Extensions

Command	Description
`fopen`	Opens a file
`fclose`	Closes a file
`fread`	Reads from a file
`fwrite`	Writes to a file
`fseek`	Seeks to a new file position

When you open a file, a new variable is introduced to the symbol table.
The variable is an object of type file object.

The following attributes are present in the file object:

ok	A Boolean variable designating whether the last operation succeed or failed. Applies for all file operations
read	Contains the total read count
name	The name of the opened file
eof	A Boolean variable designating whether the end of file has been reached or not
write	The total number of bytes written so far
pos	The current file position. The position is automatically updated after a read/write or seek operation

Namespace: “fileio”

Check the fcd_fobj*.compel sample series

Include - Extensions

The include extension allows one script to include into its line another script.
It works similarly to other high-level languages.
The command is: “include”

Namespace: none

Check the fcd_include*.compel sample series

Memory objects - Extensions

`Memory objects/extensions are closely related to the file objects / extensions.`

Command	Description
`alloc`	Allocates memory
`mfree`	Frees allocated memory

When memory is allocated, a new memory object is created.
The memory object has the following attributes:

ptr	The system address where the new allocated memory resides
size	The size of the allocated memory
length	The length of the zero terminated string located at “ptr”

The memory object’s to_string() method will return a C string of its contents. However, a memory object will preserve its binary contents when used with File I/O routines.

Namespace: none

Check the fcd_mem*.compel and fcd_fobj*.compel sample series

Scopes - Extensions

The scopes extension provides a large set of commands:

Command	Description
`{`	Initiates a scope
`}`	Closes a scope
`for`	Creates a repetition scope by providing a starting count value and ending count value. It supports “to” and “downto” counting, meaning forward counting and backward counting
`command`	Creates a user command scope
`if`	Initiates a conditionally executed scope
`else`	Optionally used with “if” command to create an alternative scope for an “if” scope
`return`	Jumps to the end of a command scope
`break`	Jumps to the end of a scope
`continue`	Jumps to the beginning of a scope

A scope is simply a bunch of script lines enclosed within opening and ending curly braces.

```
1:{
2:   Echo “Hello world!\n”
3: }
```

Only one line exists in this scope. The scope begins at line 1 and ends at line 3.

Scopes can be nested, that means, scopes can exist within each other:

```
1: {
2:  {
3:     echo “Hi”
4:  }
```

5: }

Now we have two scopes; scope 1 [line 1-5] and scope 2 [line 2-4].

A user command scope, is a scope that names a given scope, so that when that name is used, the scope contents are executed. This is similar to procedures/functions in high-level commands.

Namespace: none

Check the fcd_ifelse*.compel, and fcd_for*.compel, fcd_recursive*.compel and other scripts that may contain the { and } commands.

Shell - Extensions

The shell extension allows one to interact with the system.
Such as shelling (or executing) commands.

Only one command is exposed: “shell”

Namespace: “shell”

Check the fcd_shell*.compel sample series

Strings - Extensions

The string extension provides string manipulation routines. One command is exposed: “tokenize” which is supposed to tokenize (break down string) into tokens.

Namespace: “string”

Check the fcd_string*.compel sample series

Threads - Extensions

This is an experimental extension.

It will allow the execution of a user command in parallel with the main script execution.

For example, you may want to create a program that constantly displays the time.

Mandatory Namespace: “thread”

Check the fcd_shell*.compel sample series

Time - Extensions

This extension allows you to retrieve date and time. It creates a basic object and fills it with time values.

Command	Description
`getdatetime`	Returns date and time into a basic object
`gettickcount`	Returns the system’s ticks count (in milliseconds) since the system is started. This is useful to calculate how long the system was up.
`rand`	Returns a random number
`randomize`	Randomizes the random number generator

Namespace: “time”

Check the fcd_time*.compel sample series

Conditional and Unconditional Branching - Extensions

Conditional branching means that we go to a new execution point only if a condition is met. Unconditional branching means we always go to a new execution point with no prior conditions.

Command	Description
`goto`	Jumps to a labeled script position
`gotoline`	Allows you to jump to a script line directly if you know it is number. This is a dangerous call. It is advised that you used named labels with “goto” command
`end`	Ends the script’s execution. It is similar to C’s “exit()” and Pascal’s “halt()” functions.
`label`	Creates a named label at the given line. A label will be a symbolic name holding the current line number
`if_gt/if_gte/if_lt/`	Conditionally jumps to a given label if the

`if_lte/if_eq/if_neq`	condition is met. ”gt” stands for “greater than” and so on.

Namespace: none

Check the fcd_count*.compel, fcd_branch*.compel sample series

Appendix C - COMPEL_LIB C++ Examples

Outline

List of examples

In this section we will be giving a set of examples on how to use COMPEL LIB in the C++ language. The aim of this section is to demonstrate how to get started with COMPEL from the developer point of view.

Script Initialization / De-Initialization

The following script will illustrate how to initialize the scripting engine, how to set up the required extensions and how to get started with COMPEL.

```
void example_initialize(compel_script_t &script)
{
  compel_init_t init = {0};
  init.b_usefullns = false;

  init.b_dbgout_script = true;

  // get all the avail extensions
  init.extensions = compel_script_avail_extensions();

  // something like this:
  //
"binary_arith;binary_comparison;comments;crt;fileio;include;memory;scop
es;shell;unconditional_branching;vars"
  // is returned.

  // initialze the script engine
  script = compel_script_init(&init);

  // do something...
  // ...
  // ...
  //

  // De-initialize script
  compel_script_deinit(script);
}
```

Script Running

The following script will illustrate how to run a single script line using COMPEL or how to load and run a complete set of script lines.

```
int example_run_script(compel_script_t &script)
{
  // script lines
  char *script_lines =
    "echoln \"Hello world\";"
    "echoln \"Welcome to COMPEL\";"
    "echoln internal:> $_COMPEL <\n;";

  compel_script_load_lines(script, script_lines, ";");
  int err = compel_script_run(script);

  return err;
}
```

Script Pausing example

This example shows you how to pause the script and change its state from running to paused. This is an important aspect and functionality of the scripting engine, this function is used inside the COMPEL IDE debugger.

```
static DWORD __stdcall PauseThread(compel_script_t script)
{
  _getch();
  printf("will pause...");
  compel_internal(script, compel_internal_pause, 0, 0, 0);
  printf("paused...\n");
  return 0;
}

void example_pause_script(compel_script_t script)
{
  ::CreateThread(0, 0, PauseThread, (LPVOID)script, 0, 0);
  compel_script_load_lines(script, "var $i 0|for $i 0 to 1000|{|echoln
\"i=\" $i|}|", "|");
  compel_script_run(script);
}
```

Value Manipulation

This sample will illustrate how to create variables and how to manipulate existing variables.

```
void example_value(compel_script_t &script)
{
  compel_script_interpret_line(script, "var $s value!");

  // find the value "$"
  compel_value_t val = compel_value_find(script, "$s");

  // get the variable's value
  std::string s = compel_value_get(script, val);

  // patch string - enclose in angle brackets
  s = "<" + s + ">";

  // update the variable's value
  compel_value_set(script, val, s.c_str());

  // display the "$s" value
  compel_script_interpret_line(script, "echoln s= $s");

  // create a new variable
  compel_value_create(script, "$el", "elias");

  // show the value of the newly created variable
  compel_script_interpret_line(script, "echoln el = $el");
}
```

Object Manipulation example

This sample will illustrate how to create objects and how to manipulate existing objects. Notice that an object can be considered as a variable however it can hold multiple values in its different attributes.

```
void example_object(compel_script_t &script)
{
  // create OBJ1 using SCRIPT
  compel_script_load_lines(script, "var $obj1.|var $obj1.name john|var
$obj1.lastname doe", "|");
  compel_script_run(script);

  // create another object "obj2"
  compel_object_t obj = compel_object_create(script, "$obj2");

  // create attribute in obj2
  for (int i=0;i<10;i++)
  {
    char attr[20];
    sprintf(attr, "attr%d", i);
    compel_object_add_attr(script, obj, attr, attr);
  }

  compel_script_interpret_line(script, "echoln obj2: $obj2");

  // delete the object "obj2"
  compel_object_destroy(script, "$obj2");

  // adjust an attribute
  obj = compel_object_find(script, "$obj1");
  compel_value_t val = compel_object_find_attr(
    script,
    obj,
    "name");

  // adjust the attribute's value
  compel_value_set(script, val, "elias");

  // show object info
  compel_script_interpret_line(script, "echoln before_lastname_remove:
$obj1");

  // remove an attribute
  compel_object_remove_attr(script, obj, "lastname");

  // show object info - after removing an attribute
  compel_script_interpret_line(script, "echoln after_lastname_remove:
$obj1");

  // find another object
  obj = compel_object_find(script, "$_COMPEL");
}

void example_object_2(compel_script_t &script)
{
  compel_script_load_lines(script, "var $o.;var $o.a a;var $o.b b",
";");
  compel_script_run(script);

  char *s;
  s = compel_object_to_string(script, "$o");
  compel_string_destroy(s);

  s = compel_script_evaluate_expression(script, "$o.b . $o.a", false,
0);
  compel_string_destroy(s);

  compel_script_interpret_line(script, "echoln $o");
}
```

User Command example

This sample illustrates how to extend the COMPEL commands by creating and registering additional “user command”. The sample shows how a “lowercase” command was registered and shows how the body of the function is coded within the C++ language.

```
/*!
\brief lowercase user command
This command takes the first parameter by value and returns the
lowercase version
*/
int COMPEL_API my_lowercase(compel_script_t compel_script, int argc,
char *argv[])
{
  size_t len = strlen(argv[0]);
  char *s = new char[len+1];
  strcpy(s, argv[0]);
  strlwr(s);

  compel_lu_cmd_set_retval(compel_script, s);

  delete [] s;

  return 0;
}

/*!
\brief uppercase user command by reference
Takes two arguments, the first one is the variable's name that will
hold the uppercase
The 2nd argument is the string that needs to be uppercased
*/
int COMPEL_API my_uppercase(compel_script_t script, int argc, char
*argv[])
{
  compel_value_t v = compel_value_find(script, argv[0]);
  if (v == 0)
    return compel_error_symbol_expected;

  size_t len = strlen(argv[1]);
  char *s = new char[len+1];
  strcpy(s, argv[1]);
  strupr(s);

  compel_value_set(script, v, s);

  delete [] s;
  return compel_error_success;
}

void example_user_command(compel_script_t &script)
{
  int err;

  // register a command, the long way
  lib_usercommand_info_t cmd = {0};
  cmd.cb = my_lowercase;
  cmd.minargs = 1;
  cmd.maxargs = 1;
  cmd.name = "my_lowercase";
  err = compel_lu_cmd_register(script, &cmd);

  assert(err == compel_error_success);

  // register another command the quick way
  err = compel_lu_cmd_register2(script, my_uppercase, "my_uppercase",
2, 2);

  assert(err == compel_error_success);

  char *lines =
    "var $s1 \"tHiS iS a StRiNg\";"
    "var $s2 $s1;"
    "echoln \"before user command calls: \" $s1;"
    "my_lowercase $s1;"
    "echoln \"lowercase: \" $my_lowercase;"
    "my_uppercase &$s2 $s1;"
    "echoln \"uppercase: \" $s2;"
    ;

  // load the test script
  compel_script_load_lines(script, lines, ";");

  // run the script
  compel_script_run(script);
}
```

External extension loading

This sample illustrates how to extend the scripting engine by loading external extensions. External extensions are special user commands that reside in external DLL files. External extensions are very useful if you want to distribute binaries to end-users without giving the them the source code of your extension.

```
int example_user_ext(compel_script_t &script)
{
  int err;

  err = compel_script_load_lines(script,
    "var $s \"HElLo WoRld\";"
    "ext_lowercase $s;"
    "echoln \"x=\" $ext_lowercase;"
    ,
    ";"
    );
  err = compel_extension_load(script, "ext", "..\\Debug\\ext_dll.dll");

  err = compel_script_run(script);

  return err;
}
```

Script Error handling

Error handling sample will illustrate how the programmer can write an error handler to catch all scripting errors and decided what to do when the script error occures. This sort of error handling is also implemented within the COMPEL IDE debugger.

```
// the error handler
int COMPEL_API script_error_handler(compel_script_t script, size_t
lineno, int liberr)
{
  const char *faulty_code = compel_script_get_line(script, lineno);

  printf("error handler caught error %d @ %d:\n>%s<\n", liberr, lineno,
faulty_code);

  return compel_error_success;
}

// shows how to install an error handler to handle script errors
void example_error_handler(compel_script_t script)
{
  compel_script_set_error_handler(script, script_error_handler);
  compel_script_load_lines(script,
    "echoln okay;"
    "some_bad_command;"
    "echoln after_bad_command",
    ";");
  compel_script_run(script);
}
```

Script Internal commands

COMPEL library allows the developer to issue high-level commands via the normally documented commands and also allows low-level commands via the compel_internal() function call. The compel_internal() function allows the user to accomplish multiple functionalities by selecting which functionality via the internal command code.

```
void example_internal_script_load_and_show(compel_script_t &script,
char *fn)
{
  compel_script_load_file(script, fn);
  compel_internal(script, compel_internal_showlines, 0, 0, 0);
}

void example_internal_multiple_scripts(compel_script_t &script)
{
  const int max_scripts = 4;
  char *scripts[max_scripts] =
  {
    "lib_test1.compel",
    "lib_test2.compel",
    "lib_test3.compel",
    "lib_test4.compel"
  };

  compel_script_clear_lines(script);

  for (int i=0;i<max_scripts;i++)
    example_internal_script_load_and_show(script, scripts[i]);
}

void example_internal_1(compel_script_t &script)
{
  compel_script_load_file(script, "lib_test1.compel");

  compel_internal(script, compel_internal_setdbgout, 1, 0, 0);

  compel_script_set_lineno(script, 0);
  compel_script_run(script);

  compel_internal(script, compel_internal_writeraw, 0, 0, 0);

  printf("execution ended @ line: %d in (%d msecs)\n",
    compel_script_get_lineno(script),
    compel_internal(script, compel_internal_exectime, 0, 0, 0)
    );
}
```

Tokenizer example

COMPEL library also provides some utility functions such as the string tokenizer, and the following example demonstrates how to use the tokenizer.

```
void example_tokenizer()
{
  compel_tokenizer_t tok;

  tok = compel_tokenize_init("a;b;c;d", ";", "\"", 0);

  size_t pcount = compel_tokenize_parsed_count(tok);

  for (size_t i=0;i<pcount;i++)
  {
    printf("@%d=%s\n", i, compel_tokenize_get(tok, i));
  }

  compel_tokenize_free(tok);
}
```

Welcome.compel script

This is a simple COMPEL script to get you started. It demonstrates how to show message boxes, do an IF/ELSE and how to trigger a “YES/NO” dialog.

```
var $name
var $gender
var $mstatus

inputbox $name "What is your name?" "Enter information"

yesnobox $gender "Are you a male?" "Gender"

if $gender != 1
{
  yesnobox $mstatus "Are you married" "Martial Status"
}

// Is it a male?
if $gender == 1
```

```
{
  var $gender "Mr."
}
// Female?
else
{
  // Is she married?
  if $mstatus == 1
  {
    var $gender "Mrs."
  }
  else
  {
    var $gender "Ms."
  }
}

msgbox "Welcome " $gender " " $name

end
```

Appendix D - Package files

Outline

Package files

COMPEL package files are listed in this section. Folders and files are briefly explained and a more elaborate description is found in the release readme file.

Folder name	Purpose
binaries	Contains the compiled binaries and some additional examples and user extensions
compel_ide	Contains the COMPEL IDE project which is written in Delphi
compel_lib	Contains the COMPEL_LIB implementation
console_compel	Contains the console version of the COMPEL engine. This is how we created “ccompel.exe”
Engine	Contains base engine files: - Interpreter - Symbol table - Basic types: variable, value, object, function
engine\fnc	Contains all the built-in extensions. The file naming convention is: fnc_xxxx.cpp/.h where the “xxxx” are replaced by the meaningful name of the extension
engine\lib	Contains helper libraries such as the Inputbox library, tokenizer library
engine\obj	Contains additional objects, such as the file and memory objects
experiments	It contains the minicompel.cpp file. This small program show how to write a very small COMPEL interpreter using the engine files
lib_wrappers	Base folder for wrappers of COMPEL_LIB.
lib_wrappers\COMPEL.NET	Contains the .NET wrappers for COMPEL_LIB
lib_wrappers\Delphi	Contains the Delphi wrappers for COMPEL_LIB
test_scripts	Contains COMPEL scripts illustrating all the built-in extensions