View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

A New Approach to Event-Driven Programming

Von der Fakultät für Mathematik, Naturwissenschaften und Informatik

der Brandenburgischen Technischen Universität Cottbus

zur Erlangung des Akademischen Grades

Doktor der Ingenieurwissenschaften

(Dr. -Ing.)

genehmigte Dissertation

vorgelegt von

Master of Science

Mosbah Mohamed ELssaedi

geboren am 24. Dezember 1968 in Nofalia (Libya)

Gutachter: Prof. Dr. rer. nat. Habil. Peter Bachmann

Gutachter: Prof. Dr.-Ing Monika Heiner

Gutachter: Prof. Dr. rer. nat. habil. Karl Hantzschmann

Tag der mündlichen Prüufung: Donerstag den 17.07.2008

I dedicate this work to my parents, my wife, my brothers, and my sisters

Abstract

In many applications, like embedded systems or systems with a strong user interaction, the program is mainly controlled by events. An event handler waits for any event to occur, and then processes that event by ignoring it or calling a corresponding action, a procedure declared in the program. In this way, events are coupled with actions and can influence the global state of the whole program.

In order to design a well structured program, event handling and corresponding actions should be clearly separated. However, this aim is not always ensured.

In this thesis, an approach is introduced, which enforces to design event-driven systems into two main parts:

- An event-handling part, also called *specification part*, and
- An action part, also called *hand-built program part* (hbp).

The specification part is defined as a declarative specification of the event-handling in a special language. It is separated from the *hand-built program part*. Every event can only influence the state of this part, which contains the implementation of control functions, which calls the several actions defined in the *hand-built program part* and so connects both parts together.

The prototype of a framework was implemented which allows to specify the event-handling part by means of a special editor and generates from it some classes and templates. In order to complete the implementation, the programmer has to fill into the templates program code by hand (therefore called *hand-built program part*), which describe the actions.

Zusammenfassung

In vielen Anwendungen, wie z.B. eingebetteten Systemen oder Systemen mit starker Nutzerinteraktion, wird das Programm hauptsächlich von Ereignissen gesteuert. Eine Ereignisbehandlungsroutine wartet auf das Auftreten eines beliebigen Ereignisses und verarbeitet dann dieses Ereignis, indem sie es ignoriert oder eine entsprechende Aktion aufruft, eine Prozedur, die im Programm deklariert ist. In dieser Weise sind Ereignisse mit Aktionen gekoppelt und können den globalen Zustand des gesammten Programms beeinflussen.

Um ein wohlstrukturiertes Programm zu entwerfen, sollten die Ereignisbehandlung und die entsprechenden Aktionen klar getrennt werden. Allerdings wird dieses Ziel nicht Immer erreicht.

In dieser Arbeit wird ein Ansatz eingeführt, welcher die Beschreibung ereignisgesteuerter Systeme in Form von zwei Teilen erzwingt:

- einem Ereignisbehandlungsteil, welcher auch Spezifikationsteil genannt wird.
- einem Aktionsteil, welcher auch als handgeschriebener Programmteil bezeichnet wird.

Der Spezifikationsteil ist als eine deklarative Spezifikation der Ereignisbehandlung in einer Spezialsprache definiert. Er ist vom handgeschriebenen Programmteil getrennt. Jedes Ereignis kann nur den Zustand dieses Teils beeinflussen, welcher die Implementierung von Steuerfunktionen enthält und die verschiedenen Aktionen aufruft, die im handgeschriebenen Programmteil definiert sind, und dadurch beide Teile miteinander verbindet.

Es wurde der Prototyp eines Frameworks implementiert, welches es erlaubt, den

Ereignisbehandlungsteil mittels eines speziellen Editors zu spezifizieren und daraus verschiedene Klassen und Templates zu generieren. Um die Implementierung zu vervollständigen, muss der Programmierer Programmcode von Hand in die Templates einfügen (daher die Bezeichnung "handgeschriebener Programmteil"), welcher die Aktionen beschreibt.

Acknowledgement

First of all and the greatest important, all praises and thanks are due to my ALMIGHTY GOD for all his blessings without which nothing of my work could have been done.

I would like to thank my supervisor Prof. Dr. Peter Bachmann, for his inspiration and encouraging way to guide me to a deeper understanding of my work, and his invaluable comments during the whole work of this dissertation. Without his encouragement and constant guidance, I could not have finished this dissertation. He was always there to meet and discuss about my ideas related to my work and to proofread and make important comments throughout my papers and chapters, and also to ask me important questions that helped me think through my problems. His efforts are very much appreciated.

Also, I'd like to thank Prof. Dr. Monika Heiner, Prof. Dr. Karl Hantzschmann and Prof. Dr. Ingo Schmitt as members of my examination committee.

I would like to thank all my colleagues in the chair of Programming Languages and Compilers at BTU Cottbus, Wolfgang Jeltsch, Mario Schölzel, Angelika Claus, Katrin Ebert, and Gudrun Pehle with whom I have had and still have a wonderful time. They provided me with a very friendly atmosphere and ensured that working at the department was always fun.

Furthermore, I would like to thank many other friends for their time and support, for sharing their ideas, and for giving advice: Alhasan Tijani, .Mohamed Almansary, and Kirill Osenkov.

Last, but not the least, I would like to thank my beloved wife for her firm support in the most important period of my life, and thank my family: my parents, my brothers, and my sisters in Libya for their continual encouragement and concern.

Contents

Chapter 1 Introduction 1

1.1 Problem Statement 1

1.2 Solution Overview 3

1.3 Outline of this Dissertation 4

Chapter 2 Programming Paradigms 6

2.1 Imperative Programming Paradigm 8

2.1.1 Structured Programming 10

2.2 Functional Programming 10

2.2.1 Functional programming Languages 11

2.2.2 The Haskell programming language 12

2.2.2.1 Definitions and evaluation 13

2.2.3 Applications of Functional languages 14

2.2.4 A Comparison of Functional and Imperative Languages 15

2.3 Logic Programming 16

2.3.1 The Origins of Prolog 17

2.3.2 Language Overview 18

2.3.3 Application of Logic Programming 20

2.4 Object Oriented Programming 23

2.4.1 The Basic Principles of Object Orientation 23

2.4.1.1 Encapsulation 23

2.4.1.2 Inheritance 25

2.4.1.3 Abstraction 28

2.4.1.4 Polymorphism 29

2.5 Aspect Oriented Programming....32
2.5.1 Separation of Concerns....33
2.5.2 Aspect-Oriented Software Development....36
2.5.3 AspectJ....38
2.5.3.1 Join Point Model....38
2.5.3.2 Pointcut....39
2.5.3.3 Advice....39
2.5.3.4 Aspects....40
2.6 Event Driven Programming....41
2.6.1 Event Handling in General....43
2.6.2 Advantages and disadvantages of Event-driven programming....45
2.7 Integration of Programming Techniques....46

Chapter 3 Event-Handling in Different Programming System....49
3.1 Events....49
3.2 Delegates....50
3.2.1 Declaration....51
3.2.2 Instantiation....51
3.2.3 Invocation....51
3.3 Events and Delegates....52
3.4 Types of Delegates....53
3.5 Delegates and Their Roles....54
3.6 Events and Delegates in Visual Basic vs. C#....55
3.6.1 Declaring Events in Visual Basic and C#....55
3.6.2 Raising Events in Visual Basic and C#....56
3.6.3 Implementing Event Handlers (VB vs. C#)....56
3.7 Events in Visual C++....59
3.7.1 Declaring Events....59
3.7.2 Defining Event Handlers....59
3.7.3 Firing Events....60

3.8 Delegates and Events in J# ... 60

3.9 Events in Java ... 61

3.10 Event Handler in JBuilder ... 65

3.10.1 Connecting controls and events ... 66

3.10.2 Standard event adapters ... 66

3.10.3 Anonymous inner class adapters ... 67

Chapter 4 The Abstract Model ... 68

4.1 Structure and Semantics ... 68

4.2 Refinements ... 69

4.3 Optimization ... 71

4.3.1 Simplifications ... 71

4.3.2 Check of Contradictions ... 72

4.4 Reordering ... 73

Chapter 5 The Event-Programming-Framework *epro* ... 75

5.1 An Overview ... 75

5.2 Specification ... 80

5.2.1 State Space Specification ... 81

5.2.2 Event Handling Specification ... 83

5.2.3 Control Functions Specification ... 86

5.3 Transformation into C# ... 87

5.3.1 Transforming State Space ... 87

5.3.2 Transforming Event Handling ... 92

5.3.3 Transforming Control Functions ... 97

5.4 Example (Alarm-Clock Application) ... 100

5.4.1 Application Description ... 100

5.4.2 Alarm-Clock Specification ... 106

5.4.3 Generated Parts Of The Alarm-Clock Example ... 112

Chapter 6 Conclusion and Future Work 121

6.1 Overview 121

6.2 Advantages and Disadvantages 123

6.3 Proposed Future research work 123

Appendix A Generated Classes 125

A.1 State Class 125

A.2 Events_Handling Class 128

A.3 Control Class: 133

A.4 AppFunctions Class: 136

Appendix B XML File 139

Bibliography 150

List of Figures

Figure 2.1: Object structure and interaction 25

Figure 2.2: An example of inheritance 27

Figure 2.3: An example of polymorphism 30

Figure 2.4: General form of advice 40

Figure 2.5: Event handling concepts 44

Figure 5.1: Event handling system 76

Figure 5.2: Specification part 80

Figure 5.3: An example of the state space form 81

Figure 5.4: The event handling form 83

Figure 5.5: The example of condition form 84

Figure 5.6: An example of control functions specification 86

Figure 5.7: An example of empty templates of bodies of methods 100

Figure 5.8: Start of the Alarm-clock Application 101

Figure 5.9 a: First alarm time (hours) 101

Figure 5.9 b: Second alarm time (minutes) 101

Figure 5.10: Timer alarm mode 102

Figure 5.11: Alarm-clock ringing 102

Figure 5.12: The main form of the Alarm-Clock application.. 103

Figure 5.13: List of state-variables in Alarm-Clock example.. 107

Figure 5.14: List of events in Alarm-Clock example 109

Figure 5.15: List of control functions in Alarm-Clock example 111

List of Acronyms

ADT	Abstract data type
AOP	Aspect Oriented Programming
AOSD	Aspect Oriented Software Development
COM	Component Object Model
CPU	Central Processing Unit
C#	C Sharp Language
EDS	Event Driven System
EDP	Event Driven Programming
EPRO	Event Programming Framework
GUI	Graphical User Interface
HBP	Hand-built Programming
IN	Natural Numbers
OO	Object Orientation
OOP	Object Oriented Programming
OS	Operating System
RDBMS	Relational Database management System
UI	User Interface
VB	Visual Basic language
VS	Visual Studio
XML	Extensible Markup Language

1

Introduction

1.1 Problem Statement

Most graphical user interface (GUI) systems and embedded control systems (such as Microwave, camera etc) are event driven. That is, the operating system sends events to the program and the program responds to these events as they arrive. Events can include actions performed by user such as clicking the mouse, pressing a key, or actions that the system itself does like updating the clock or refreshing the screen [10].

In conventional programming, the sequence of operations for an application is determined by a central controlling program (e.g., a main procedure). In *event-driven programming*, the sequence of operations for an application is determined by the user's interaction with the application's interface (forms, menus, buttons, etc.) [46]. For example, instead of having a main procedure that executes an order entry module followed by a data verification module followed by an inventory update module, an event-driven application remains in the background until certain events happen: when a value in a field is modified, a small data verification program is executed; when the user indicates that the order entry is complete, the inventory update module is executed, and so on. Event-driven programming, graphical user interfaces (GUIs), and object-orientation are all related since forms and the graphical interface objects on the forms serve as the skeleton for the entire application. To create an event-driven application, the programmer creates small programs and attaches them to events associated with objects. In this way, the behavior of the application is determined by the interaction of a number of small manageable programs rather than one large program.

The programming model of today's GUI requires *event-driven programming*. A GUI program waits for the user to take an action, such as choosing among menu selection, pushing buttons, update text fields, and clicking icons. Each action causes an event to be raised. Other events can be raised without direct user action, such as events that correspond to timer ticks of the internal clock, email being received, and file-copy operations completing [28].

An event is the encapsulation of the idea that "something happened" to which the program must respond. Events and delegates are tightly coupled concepts because flexible event handling requires that the response to the event be dispatched to the appropriate event handler [28].

Event handling is associated with dealing with a situation in which something has happened and the software developer needs to be notified of that situation. Sometimes the code written to deal with these situations are referred to as a callback and sometimes as an event handler, in either case, the same basic principle is applied. That is, the developer must implement a method that matches some specification that allows it to be called when the "event" occurs. This event could be some threshold being reached in some sensor, it could be a message being received from some broadcast mechanism or it could be some user interaction with a GUI. In general, this event handling method will be called when the event occurs and will be passed some data to allow it to determine the sender of the event and any event specific data. The event handler can then determine what action the application should take [1].

Depending on the system used and the type of technology applied, the relationship between events and executed actions are explained in different ways. Most often, events and their respective actions are directly coupled. In C# (.NET), an event is delegated to a sequence of actions by means of delegates. This has been described in the parts of the program (classes) where the event is expected to occur. However, not all the events have really influence on the flow of the program. For instance, mostly, changing the pointer of the mouse is the only effect of moving the mouse over a window. Either the occurrence of

such an event is disabled or it is not delegated to an action. In the former case, this has been expressed by a property of an object, for example a windows button.

That is, the event handling is distributed over the whole text of the program. Thus, there is no distinct overview as to where and under which condition events may occur and what kind of effects they cause. Mostly, it depends on the existing general situation of a program, which is mainly characterized by the existence and values of program objects, i.e. the current memory assignment. In this way the maintenance of the software becomes very complicated [50].

1.2 Solution Overview

The fundamental idea to solve this problem is to distinctly separate event handling from the actions of the program. So, the whole program is divided into two main parts:

- A *specification part*, which is responsible for event-handling and
- A *hand-built program part*, which contains the program actions.

The specification part is defined as a declarative specification in a special language. It has an own state, called *event-state*. Every event may cause the event-state to change [50]. Therefore, the program flow has a cycle, which is determined by raising events and changing the event-state. The event-state is also used to control the relationship between events and actions of the program. A *control functions* is used for selecting program actions, which are executed with respect to the current event-state. The advantages of this approach can be summarized as follows:

- It allows two different programmers to develop the system separately. Depending on the different states, one of the programmers is responsible for developing the part for handling the events, the specification part, while the other is responsible for coding the effects, the hand built part.

- The description of the separation between the event-handling cycle and the control of program actions can be specified in a relative abstract level, which does not require any previous knowledge of programming languages. Rather, this specification is transformed automatically into source code. In this way, one can consider the event handling cycle as a special *transaction* system. Moreover, some optimizations and verification on this transformation can be achieved.

The prototype of a framework was implemented which allows to specify the specification part by means of a special editor and generates from the specification some classes and templates for the *hand-built program part*. By the editor, the correctness of the specification with respect to some aspects is proved. In order to complete the implementation, the programmer has to fill into the templates program code, which describes the actions.

1.3 Outline of this Dissertation

The remainder of the thesis is divided into 5 chapters:

In Chapter 2, an overview of some important programming paradigms such as imperative programming paradigm, functional programming, logic programming, object oriented programming (OOP), aspect oriented programming (AOP) and event driven programming are presented. Most programming paradigms are discussed and illustrated with some examples. Also the integration of the programming paradigms and in which applications they are applied is described.

Chapter 3 focuses on the event handling methods of different programming systems. In this chapter, a description of how the event handling work within the Visual Studio.NET languages such as Visual Basic, C#, Visual C++, and Visual J# is presented. Some examples are given in order to make a comparison of how these languages are used to implement the event handling. It also presents how the event handling works within other systems such as Java and JBuilder.

An abstract model approach is presented in **Chapter 4**. Definitions of the model, demanded and derived properties are also introduced.

Chapter 5 describes the implementation of the abstract model as well as developing the framework on the basis of the .NET. C# language was used in the implementation of the framework system. It presents the special program and its features that are used to implement the event handling system. It gives a detail description of the two main parts of the system, namely specification part and *hand-built program* (hbp) part. It also declares the parts of the specification system such as state space, event handling and control functions specification and also how these specification parts are transformed into C# language. It gives an illustrative application example of how the system works.

Chapter 6 summaries the contribution of the work, and it describes the advantages and disadvantages of the system as well as outline of future areas of research.

2

Programming Paradigms

A Programming Paradigm is defined as a style of programming for a class of programming languages that share a set of common characteristics [37].

A programming language is a systematic way of using signs to communicate a task/algorithm to a computer, which influence the task to be performed. The task to be performed is known as a computation, which follows absolutely precise and unambiguous rules [37].

The fundamental question framed as follows: What does it mean to understand a programming language? What do we need to know to be able to program in a language? There are three fundamental aspects to any language.

The method of specifying what is legal in the phrase structure of the language is known as the syntax of the language; any knowledge of the syntax is analogous to knowing how to spell and construct sentences in a natural language such as English. However, this does not give us any information about the meaning of the sentence. Meaning is given by semantics. Ultimately, without semantics, a programming language is just a collection of meaningless phrases; hence, the semantics forms an integral part of a language.

Finally, pragmatics is the aspect of using the language in a special way, how to meet a special programming paradigm. There are numerous numbers of programming languages which were used for special purposes such as, academic and industrial needs.

In order to demonstrate how certain programming languages reflect on programming paradigms, some of them will be discussed in this section. Some of the major programming language paradigms are as follows:

- Imperative (Procedural) Paradigm
- Functional Paradigm
- Logic Paradigm
- Object-Oriented Paradigm
- Aspect-Oriented programming Paradigm
- Event driven programming

The relationships between programming paradigms and programming languages can be complex because programming language has the ability to support multiple paradigms. For example, C++ is designed to support elements of procedural programming, object-based programming, object-oriented programming, and generic programming. However, designers and programmers decide how to build a program using those paradigm elements. A purely procedural program can be written in C++ where as a purely object-oriented program can be written in C++, or one can also write a program that contains elements of both paradigms [38].

The rest of this chapter is organized as follows: an introduction about imperative programming is presented in section 2.1. A general overview about functional programming is introduced in section 2.2 and the Haskell language is taken as an example of the functional programming language. In Section 2.3, the concept of logic programming system is discussed, in this section, the Prolog language is used as an illustrative example of the logic programming language. The object-oriented system is described in Section 2.4. In Section 2.5, a brief introduction about aspect-oriented system is presented and the AspectJ is taking as one of the different application of aspect oriented system. The event-driven programming is described in section 2.6. In the last section 2.7 the integration of the programming paradigms and the importance of learning these approaches are discussed.

2.1 Imperative Programming Paradigm

The oldest programming style is the imperative programming paradigm which is closely related to the computer and its machine language. The computer is controlled by instructions that are executed in one of the units of the CPU [39].

Imperative programming is a programming style that describes computation in terms of a program state and statements that change the program state. In as much as the imperative mood in natural languages expresses commands to take action, imperative programs are a sequence of commands issued for the computer to perform [23]. The hardware implementation of almost all computers is imperative; nearly all computer hardware is designed to execute machine code, which is native to the computer, written in the imperative style. From this low-level perspective, the program state is defined by the contents of memory, and the statements are instructions in the native machine language of the computer [41]. Even though Higher-level imperative languages use variables and more complex statements, they still follow the same paradigm. Recipes and process checklists, while not computer programs, are also familiar concepts that are similar in style to imperative programming; each step is an instruction, and the physical world holds the state. Most computer languages are in the imperative style because the basic ideas of imperative programming are both conceptually familiar and directly embodied in the hardware.

The four basic types of statements supported by most high-level languages are: assignment, looping, conditional branching, and unconditional branching. Generally, the function of the assignment statements is to perform an operation on information located in the memory and store the results in memory for later use. Also High-level imperative languages permit the evaluation of complex expressions, which may consist of a combination of arithmetic operations and function evaluations, and the assignment of the resulting value to memory. Looping statements permits a sequence of statements to be executed multiple times. Loops can either execute the statements they contain a predefined number of times, or they can execute them repeatedly until some condition

changes. Conditional branching statements allow a block of statements to be executed only if some condition is met. Otherwise, the statements are skipped and the execution sequence continues from the statement following the block. Unconditional branching statements allow the execution sequence to be transferred to some other part of the program. These include the jump, called "GOTO" in many languages, and the subprogram, or procedure call [23].

The earliest imperative languages were the machine languages of the original computers. In these languages, instructions were very simple, which made hardware implementation easier, but hindered the creation of complex programs. BASIC, FORTRAN, COBOL and PASCAL are some examples of the imperative programming languages.

FORTRAN, developed by John Backus at IBM starting in 1954, was the first major programming language to remove the obstacles presented by machine code in the creation of complex programs. FORTRAN was a compiled language that allowed named variables, complex expressions, subprograms, and many other features now common in imperative languages. The next two decades saw the development of a number of other major high-level imperative programming languages. In the late 1950s and 1960s, ALGOL was developed in order to allow mathematical algorithms to be more easily expressed. COBOL (1960) and BASIC (1964) were both attempts to make programming syntax look more like English. In the 1970s, Niklaus Wirth developed Pascal, and Dennis Ritchie created C while he was working at Bell Laboratories. Wirth went on to design Modula-2, Modula-3, and Oberon. The United States Department of Defense began designing Ada in 1974, but did not complete the specification until 1983 [23].

The 1980s saw a rapid growth in interest in object-oriented programming. These languages were imperative in style, but added features to support objects. The last two decades of the twentieth century saw the development of a considerable number of such programming languages. SmallTalk-80, originally conceived by Alan Kay in 1969, was released in 1980 by the Xerox Palo Alto Research Center. Drawing from SmallTalk's concepts, Bjarne Stroustrup designed an object-oriented extension of the C language

called C++, which was first implemented in 1985. In the late 1980s and 1990s, the notable imperative languages drawing on object-oriented concepts were Perl, released by Larry Wall in 1987; Python, released by Guido van Rossum in 1990; and Java, first released by Sun Microsystems in 1996.

2.1.1 Structured Programming

Structured Programming was designed to avoid "GOTO" statements [39]. It is still followed the imperative style. At a low level, structured programs are often composed of simple, hierarchical program flow structures [41]. These are regarded as single statements, which may be one of these structures, or primitive statements such as assignment or procedure calls. The three types of structure identified by Dijkstra were concatenation, selection, and repetition. Some of the better known structured programming languages are Pascal, Modula, and Ada.

The Advantages of imperative programming are its relative simplicity, and the ease of implementation of compilers and interpreters.

The disadvantages of imperative programming are the difficulties of reasoning about programs and to some extent the difficulty of parallelisation. It tends to be relatively low level compared to some other paradigms, and this makes them less productive [40].

2.2 Functional Programming

Functional programming is a style of programming that gives special attention to the evaluation of expressions, instead of execution of commands. The expressions in these languages are generated by using functions to combine basic values. That means, evaluation of an expression is nothing else but the application of functions.

Functional languages support and improve programming in a functional style [42]. A program therefore consists entirely of functions which are generally defined in terms of

other functions, which in turn are defined in terms of still more other functions, until at the bottom level the functions are language primitives [7].

The special characteristics and advantages of functional programming are stated as follows. Purely functional programs do not contain assignment statements, therefore any variable that takes on a value, never change. More generally, they separate side effects from expression evaluation. A function call produces no other effect except to compute its result. This removes many major sources of bugs, and also makes the order of execution irrelevant since no side-effect can alter the value of an expression; it can be evaluated at any time. This removes the trouble of prescribing the flow of control by the programmer.

Because expressions can be calculated at any given time, one can easily substitute variables by their values and vice versa - this means, programs are “referentially transparent”. This degree of freedom assist in making functional programs more easily controlled mathematically than their conventional counterparts [7].

2.2.1 Functional programming Languages

LISP started as a purely functional language but afterwards it gains some important imperative features that increased its performance efficiency. Scheme is a small, static-scope dialect of LISP. COMMON LISP is a combination of many dialects of LISP in the early 1980s. ML is a strongly typed functional language with more conventional syntax than LISP and Scheme. Haskell is the modern functional language and is a purely functional language.

Although functional languages are often executed with interpreters, they can also be compiled [4].

In a functional language, each expression of the language specifies an object. Objects are pure values, always functions in principle. So, constants can be considered as nullary functions (without any arguments).

The functional programming languages can be classified into two classes:

- Languages with eager evaluation
- Languages with lazy evaluation.

Roughly spoken means eager evaluation that a function is evaluated only if all the arguments are completely evaluated. That means, for instance, that a function application is not defined if one of the arguments is not defined. By lazy evaluation, the arguments are only as far evaluated as it is absolutely necessary in order to get the result of the whole function. As a consequence, partially defined as well as infinite values can be dealt with.

The type concept of functional languages changed from a very simple one in LISP to a powerful tool in EPIGRAM [11]. In LISP all the data structures are built by pairs. So, for instance, a list is considered as a pair where the left component is the head of the list and the second one the tail.

Modern functional languages such as ML, MIRANDA, and HASKELL use algebraic data type concepts which allow the user to build his own structures by user defined constructors. In EPIGRAM which is based on Martin-Löf´s type theory [11], the type concept allows to verify the correctness of the defined functions. Moreover, using principles of primitive recursion, by the powerful type system the programming system assists the user to construct correct functions. In addition EPIGRAM allows only structurally recursive definitions which ensure that programs always terminate.

2.2.2 The Haskell programming language

Haskell (Peyton Jones and Hughes 1998) is one of the famous functional programming languages being applied today. Haskell is named after Haskell B. Curry who was one of the pioneers of the λ calculus (lambda calculus), which is a mathematical theory of computable functions. There are many ways of implementations of Haskell available; one of them is the Hugs (1998) system. We have the impression that Hugs

delivered the best condition for the programmer, since it is freely available for PC, UNIX and Macintosh systems, it is efficient, compact and has flexible user interface.

Hugs is used for evaluating expressions step-by-step as might manually be done by writing on a paper, for this reason, it is less efficient compared with a compiler which translates Haskell programs directly into the machine language of a computer [6].

Compiling a language such as Haskell permits its programs to run with a speed comparable to those written in more conventional languages such as C and C++. Details of all the implementations of Haskell can be found at the Haskell home page, http:// www.haskell.org.

We will speak only about the main concepts that used in the Haskell and reflected the meaning of the functional programming paradigm.

2.2.2.1 Definitions and evaluation

A functional program in Haskell consists of:

- Type definitions
- Constant definitions
- Function definitions

Types are primitive such as boolean or algebraic types which are built by means of constructors and corresponding arguments. For some algebraic types, like tuples, lists and functions, special writing conventions are introduced which make programs better readable.

A constant definition consists of a *pattern* and an *application*. The pattern is built like expressions with constructors as operators and variables, constant identifiers and pattern as operands. In order to get the value of the constant, at first the corresponding application (an expression) is evaluated and then the pattern is matched with the value of the application.

Functions in Haskell have only one argument. If functions with more than one argument are needed, then either the different arguments must be combined to a tuple or currying is used in order to define a higher order function which has functions as result. Similar to constant definitions, pattern matching is a main method to define functions. For different pattern of the argument different definitions are given.

If a function must be applied to an actual argument, a definition is searched for which the pattern matched with the actual argument. Because Haskell uses lazy evaluation, the actual argument is only evaluated as deep as a match is found or it becomes true that no match is possible. This means that nothing is evaluated until it has to be evaluated [43]. This can be illustrated with the case of defining an infinite list of primes without ending up in infinite recursion. Only the elements of this list that are actually used will be computed. This enables for some very elegant solutions to many problems. A typical pattern of finding a solution to a problem is to define a list of all possible solutions and then sorting out the incorrect ones. The remaining list will then only contain the correct solutions. Lazy evaluation makes this operation very clean. If we only need one solution we can simply extract the first element of the resulting list - lazy evaluation will make sure that nothing is needlessly computed.

2.2.3 Applications of Functional languages

In the history of high-level languages only a few functional languages have gained widespread recognition of their application. Most common among these is LISP. Because of its higher use of the assignment statement, APL also is often considered as a functional language, purely because of its functional forms. APL has been used for different variety of applications, ranging from description of hardware to management information systems. Because of the complexity in reading a typical APL program, its most natural place in contemporary computing is in the category of throwaway programming. With its powerful collection of array operations, it is an excellent tool for quick but dirty solutions to problems involving many array manipulations [4].

LISP is flexible and powerful language. It was developed for computing symbols and list-processing applications, which is found in the AI area of computing. In AI applications, LISP and its derivative languages are still considered as standard languages.

A number of areas have been developed in the field of AI, primarily through the use of LISP. Although other kinds of languages can be used primarily logic programming languages, most existing expert systems, for example, were developed in LISP. LISP also dominates in the areas of knowledge representation, machine learning, natural language processing, intelligent training systems, and the modeling of speech and vision.

Outside AI, LISP has also been successfully applied. For example, the EMACS text editor is partially written in LISP and can be executed in LISP, as is the symbolic mathematics system, MACSYMA, which does symbolic calculus, among other things. The LISP machine is a personal computer whose entire systems software is written in LISP. LISP has also been successfully used to construct experimental setups in different fields of application [4].

Scheme is widely used to teach functional programming. It is also used in some universities to teach introductory programming courses. Use of ML and Haskell has been, for the most part, restricted to research laboratories and universities.

2.2.4 A Comparison of Functional and Imperative Languages

It is a natural to compare functional programming with programming in imperative languages. Because imperative languages directly related to the von Neumann architecture, programmers using them must deal with the management of variables and assignment of values to them. The results of this are increased in efficiency of execution but difficulty in construction of programs. In a functional language, the programmer does not need to be concerned with variables, because memory cells do not need to be abstracted into the language. One result of this is decreased in efficiency of execution. Another result, however, is a higher level of programming, which should require less

labor than programming in an imperative language. Many believe that this is the case and that it is a definite advantage of functional programming.

Functional languages can have a very simple syntactic structure. The list structure of LISP is an example. The syntax of the imperative languages is much more complex. The semantics of functional languages can also be simple compared to that of the imperative languages.

Concurrent execution in the imperative languages is difficult to use and design. For example, consider the tasking model of Ada, in which cooperation among concurrent tasks is the responsibility of the programmer. Functional programs can be executed by first transforming them into graphs. These graphs can then be executed through a graph reduction process, which can be done with a great deal of concurrency that was not specified by the programmer. The representation of the graph naturally exposes many opportunities for concurrent execution. Cooperation synchronization in this process is not the concern of the programmer [4].

In an imperative language, the programmer must make a static division of the program into its different concurrent parts, which are then written as tasks. This can be a complicated process. Programs in functional languages can be divided into concurrent parts dynamically by the execution system, making the process highly adaptable to the hardware on which it is running. Understanding concurrent programs in imperative languages is much more difficult.

2.3 Logic Programming

Logic programming paradigm is a way of expressing programs in a form similarly to symbolic logic and use a logical inference process to generate results.

Logic programs are declarative the sense that only the specifications of the desired results are stated rather than detailed procedures for producing them.

The essential characteristic feature of logic programming languages is their semantics. The basic concept of this semantics is to describe some knowledge of the objects [4].

In a logic language, constants and variables are defined. However, computation is done by defining data objects that satisfy a set of constraints [5]. The programmer defines facts or relationships regarding data objects as well as inference rules by which conclusions may be drawn about those objects. A query may then be written a question whose truth/answer or false the programmer does not know. The language interpreter attempts to prove that question from the facts and rules previously provided.

During the proof process, variables are instantiated with *terms* which describe constraints. By *unification* it is checked whether or not different constraints can be satisfied in a most general way. If all constraints can be satisfied then a *solution* is computed.

The mathematical basis for logic programming is the technique of refutation proofs in a subset of predicate calculus of first order. The formulas are restricted to so called *Horn clauses*, which have the form of an implication. The proof method developed first by Robinson provides the inference technique [4]. The *resolution rule*, a generalization of the cut rule, uses unification.

Logic programs face serious of machine efficiency. Furthermore, the best form of a logic language has not yet been determined, and good methods of creating programs in logic programming languages for large problems have not yet been developed.

Prolog is used to describe the logic programming paradigm because it is the only widely used logic language,

2.3.1 The Origins of Prolog

During the very early 1970s, Alain Colmerauer and Philippe Roussel of the Artificial Intelligence Group at the University of Aix-Marseille, together with Robert Kowalski of the Department of Artificial Intelligence at the University of Edinburgh,

developed the fundamental design of Prolog, which amounts to a syntax for predicate calculus propositions and an implementation of a restricted form of resolution. The first Prolog interpreter was developed at Marseille in 1972. The version of the language that was implemented is described in Roussel (1975). The name Prolog is from **pro**gramming by **log**ic.

To this day Prolog has grown in use throughout North America and Europe. Prolog was used heavily in the European Esprit programme and in Japan where it was used in building the ICOT Fifth Generation Computer Systems Initiative. The Japanese Government developed this project in an attempt to create intelligent computers. Prolog was a main player in these historical computing endeavors. Prolog became even more pervasive when Borland's Turbo Prolog was released in the 1980's. The language has continued to develop and be used by many scientists and industry experts. Now there is even and ISO Prolog standardization (1995) where all of its individual parts have been defined to ensure that the core of the language remains fixed.

2.3.2 Language Overview

A Prolog program is made of collections of clauses. Even though Prolog has only a few kinds of clauses, they can become complicated. Prolog is commonly used as an intelligent database [3]. This application provides the necessary simple framework for discussing the Prolog language. The database of a Prolog program is made of two kinds of clauses, namely, facts and rules. An example of a fact is

```
mother(mary,john).
```

This intends to the description of the fact that mary is the mother of john. An example of a rule is

```
grandparent (X, Z) :- parent (X,Y), parent (Y,Z).
```

The above statement can be explain as follows: X is the grandparent of Z if it is true that X is the parent of Y and Y is the parent of Z, for some specific values for the variables X, Y, and Z.

The Prolog database can be interactively queried with goal statements, an example of which is

```
father (bill, john).
```

which can be asked, if bill is the father of john. When such a query, or a goal, is presented to the Prolog system, it uses the "resolution process", which uses unification, to attempt to determine the truth of the statement. If it can be conclude that the goal is true, it displays "true". If it cannot be proved, then it displays "false".

Resolution is designed to work with formula in clausal form. Given two clauses that are related in an appropriate way, a new clause can be generated as a consequence of them.

The basic idea is that if the same atomic formula appears both on the left hand side of one clause and the right hand side of another the two clauses, missing out the duplicated formula, follows from them. For example:

```
from:
        angry(chris) :-  workingday(today) , raining(today).
and:
        unpleasant(chris) :- angry(chris),tired(chris).
follows:
        unpleasant(chris) :-
                        workingday(today) , raining(today),tired(chris).
```

Resolution is actually more complicated compared to the above illustrative example. Particularly, the presence of variables in propositions requires resolution to find values for those variables that enable the matching process to be succeeded.

A critical important property of a resolution is its ability to detect any inconsistency in a given set of propositions. This property allows resolution to be applied to prove theorems, which can be illustrated as follows: we can imagine the proof of a theorem in terms of predicate calculus as a given set of pertinent propositions; a new proposition can be generated with the negation of the theorem itself. The theorem is negated so that resolution can be used to prove the theorem by finding an inconsistency. This is proof by contradiction. Typically, the original propositions are called the hypotheses and the negation of the theorem is called the goal [3].

2.3.3 Application of Logic Programming

Logic programming has been applied in a number of different areas. Here are some of them:

- **Relational Database Management Systems:** data in the form of tables is stored in Relational Database Management Systems (RDBMSs). Queries on such databases are often stated in relational calculus, which is a form of symbolic logic. The query languages of these systems are nonprocedural in the same sense that logic programming is nonprocedural. How to retrieval the answer is not describe by the user; instead, he or she only describes the characteristics of the answer. The connection between logic programming and RDBMSs should be obvious. Prolog structures can describe simple tables of information, and also relationships between tables can be conveniently and easily described by Prolog rules. The retrieval process is inherent in the resolution operation. The goal clauses of Prolog provide the queries for the RDBMSs [4].

  One of the advantages of using logic programming to implement an RDBMS is that only a single language is required. In a typical RDBMS, a database language includes statements for data definitions, data manipulation, and queries, all of which are embedded in a general-purpose programming language, such as COBOL. The general-purpose language is used for processing the data and input

and output functions. All of these functions can be done in a logic programming language.

Another advantage of using logic programming to implement an RDBMS is it's built in deductive capability. Conventional RDBMSs can not deduce anything from the database other than what is explicitly stored in them. They contain only facts and inference rules. The primary disadvantage of logic programming compared with conventional RDBMSs is its lower level of efficiency. Logical inferences are simply much slower than ordinary table look-up methods using imperative programming techniques.

- **Expert Systems:** Expert systems are computer systems that are designed to emulate human expertise in some particular domain. They are made of some database of facts, an inferencing process, some heuristics about the domain, and some friendly human interface that makes the system appear much like an expert human consultant. In addition to their initial knowledge base, which is provided by a human expert, expert systems learn from the process of being used, so their databases must be capable of growing dynamically. Also, the property of an expert system is that, it should have the ability of interrogating with the user in order to get additional information when it detects that such information is needed.

  One of the central problems for the designer of an expert system can be associated to the dealing with the inevitable inconsistencies and incompleteness of the database [4]. Logic programming seems to be suitable approach to deal with these problems. For example, default inference rules can be of help when dealing with the problem of incompleteness. Prolog can and has been used to construct expert systems. It can easily fulfill the basic need of expert systems, using resolution as the basis for query processing, using its ability to add facts and rules to provide the learning capability, and using its trace facility to inform the user of the "reasoning" behind a given result. Missing from Prolog is the automatic ability of

the system to query the user for additional information when it is needed. One of the most widely known uses of logic programming in expert systems is the expert system construction system known as APES, which is described in Sergot (1983) and Hammond (1983).

- **Natural Language Processing:** some kinds of natural language processing can be done with logic programming. Particularly, natural language interfaces to computer software systems, such as intelligent databases, and other intelligent knowledgebase systems can be conveniently produced with logic programming. For describing language syntax, different forms of logic programming have been found to be equal to context-free grammars. Proof procedures in logic programming systems have been found to be equivalent to certain parsing strategies. In fact, backward chaining resolution can be used directly to parse sentences whose structures are described by context-free grammars. It has also been discovered that some kinds of semantics of natural languages can be made clear by modeling the languages with logic programming. In particular, research in logic-based semantics networks has shown that sets of sentences in natural languages can be expressed in clausal form (Deliyanni and Kowalski, 1979). Logic-based semantic networks are also discussed by Kowalski (1979).

- **Education:** In the area of education, extensive experiments have been carried out in teaching children as young as seven years old on how to use the logic programming micro-Prolog (Ennals, 1980). Researchers claim a number of advantages in teaching Prolog to young people. Firstly, it is possible to introduce computing with the application of this approach. It also has the side effect of teaching logic, which can result in clearer thinking and expression. This helps students in learning different kinds of subjects, such as solving equations in mathematics, dealing with grammars for natural languages, and understanding the rules and order of the physical world. The experiment in introducing logic programming to the very young have yielded a very interesting result that is easier

to teach logic programming to a beginner than to a programmer with a significant amount of experience in an imperative language.

2.4 Object-Oriented Programming

The concept of object-oriented programming originates from SIMULA 67, but this was not completely developed in the evolution of the SmallTalk language [4]. Computer programming based on objects, in which each object has its own program code and data, and can interact with other objects. Data items are closely related to the procedures that operate on them. For example, a circle on the screen might be an object: it´s data, can be defined as it´s centre point and the radius, as well as procedures for moving it, erasing it, changing its size, and so on. The technique originated with the Simula and SmallTalk languages in the 1960s and early 1970s, but it has now been incorporated into many general-purpose programming languages, including Java, C++, C# and Eiffel.

2.4.1 The Basic Principles of Object-Orientation

2.4.1.1 Encapsulation

Encapsulation or data hiding has been a major characteristic feature of a number of programming languages, Both Modula-2 and Ada provides extensive encapsulation features. But what exactly is encapsulation? Essentially, it is the method of hiding the data behind software "wall". Those outside the wall cannot get direct access to the data. Instead, they must ask usually the owner of the data to provide them with the data.

The advantage of the encapsulation is that the user of the data does not need to know how, where, or in what form the owner of the data stores that data. This means that if the owner modifies the way in which the data was stored, this will not affect the user of the data. The user will still need to ask the owner for the data; it is the owner of the data who changes how the request is fulfilled [1].

Different programming languages apply the concept of encapsulation in diverse ways. For example, Ada enables encapsulation using packages which posses both data and procedures. A set of interfaces are also specified by publish those operations the package wishes to make available to users of the package. These interfaces may apply some operations or provide access to data held within the package.

Object-oriented languages give encapsulation facilities which provides the user of an object with a set of external interfaces. These interfaces specify the requests to which the object will respond (or, in the terminology of object orientation, the requests which the object will understand). These interfaces not only avoid the need for the caller to understand the internal details of the implementation, they actually prevent the user from having access to that information. Users of an object cannot directly access the data held by an object because ít is not visible to them. In other words, a program that calls this facility can treat it as a black box, the program knows what the facility’s external interfaces guarantee to do, and that is all it needs to know.

It is necessary to point out the difference between the object-oriented approach and the package approach used in Ada. In general, a package is a large unit of code providing a wide range of facilities with a large number of data structures (for example, the textIO package). In an object-oriented language, the encapsulation is presented at the object level. While objects may well be as large and as complex as the typical Ada package, they are usually much smaller. In languages such as C# and Java, where (virtually) everything is in the form of an object, the smallest data and code units also naturally benefit from encapsulation. The same level of encapsulation can be introduced in Ada, eventhough it is not natural to the language.

Figure 2.1 is an illustrative example in which encapsulation works within an object-oriented language [1]. It shows that anything outside the object can only have access to the data that the object holds through specific interfaces (the black squares). In turn these interfaces invoke procedures which are internal to the object. These procedures may then

access the data directly, use a second procedure as an intermediary or call an interface to another object.

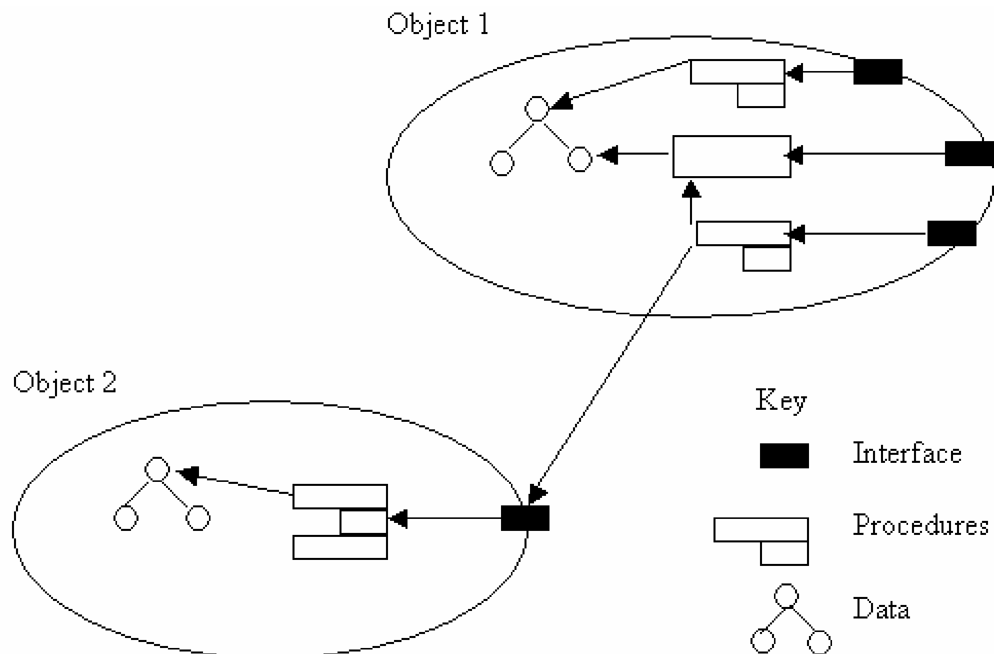

Figure 2.1: Object structure and interaction

2.4.1.2 Inheritance

A class is an example of a particular type of thing (for example, manimal is a class of animal). In the case of object oriented world, a class is defined as the charactristics of that thing. Therefore, in the case of manimals, we might define that the animals have fur, are warm-blooded and produce live young. Animals such as dogs and cats are then instances of the classes manimals. These are all quite obvious and should not give a conceptual problem for anyone. However, in most object oriented languages, the concept of the class is closely related to the concept of inheritance [1].

Inheritance enables us to state that one class is similar to another class, but with a specified set of differences. Another way of stating this is that we can define all the things

which are common to a class of things, and then define what is special about each sub-grouping within a subclass. For example if we have a class defining all the common traits of manimals, we can define how a particular groups of manimals differ. The duck-billed platypus is a quite extraordinary manimals that differs from other manimals in a number of important ways. However, we do not want to define all the things that it has in common with other manimals. Not only is this extra work, but we then have two places in which we have to maintain this information. We can therefore state that duck-billed platypus is a class of manimals that does not produce live young. Classes allow us to do this.

An example which is rather common to most computer scientists is illustrated in Figure 2.2, for this example, assuming that a job of designing and implementing an adminstration system for a small software house that produces payroll, pensions and other financial systems has been given. This system needs to record both permanent and temporary employees of the company. For temporary employees, their department, the length of their contract when they started and an additional information which differs depending on whether they are contractors or students on an industrial placement need to be recorded. For permanent employees, their department, their salary, the languages and operating systems with which they are familiar and whether they are a manger need to be record. In the case of managers, we might also want to record the projects that they run.

A class hierarchy diagram for this application is illustrated in Figure 2.2. It shows the classes we have defined and from where they inherit their information.

- ***inheritance versus instantiation*** Stating that one class is a specialized version of a more generic class, this is different from saying that something is an example of a class of things. For the first case, it can be said that a developer is one category of employee and manager is another. None of these categories can be used to identify an individual. They are, in effect, templates for examples of those catogries. In the second case, we say that "John" is an example of a developer (just as "Chris","Myra" and "Denise" may also be examples of developers).

"John" is therefore an instance of a particular class (or catogery) of things known as developers. It is important to get the concept of specializing a class having a subclass in mind. It is very easy to confuse an instance of a class with a subclass.

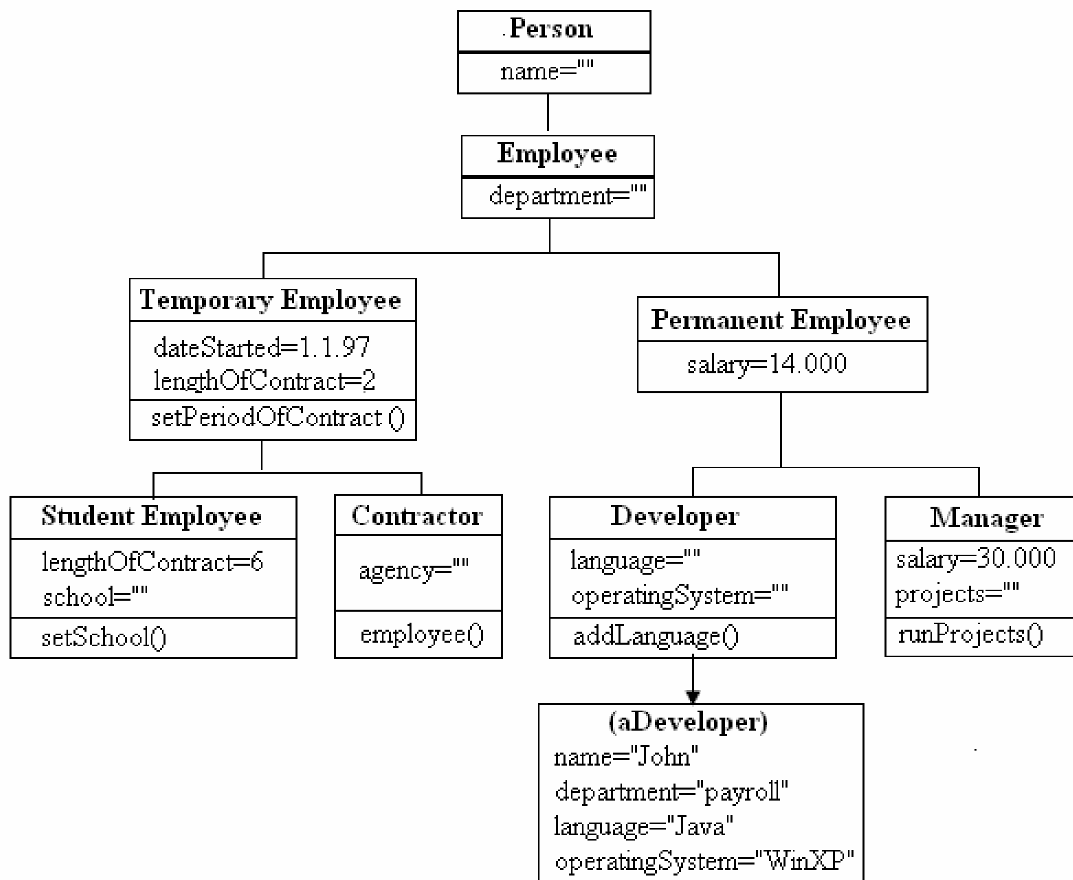

Figure 2.2: An example of inheritance

- ***Inheritance of common information***: common concepts can be placed together in a single class. For example, every individual has a name and all employees have their respective department alocated to them (whether permanent or temporary employee). All temporary employees have a commencement date, whether being contractors or students. In turn, all classes below Employee inherit the concept of

a department. This means that not only do all Managers and Developers have a department, but "John" has a department, which in this case is "Payroll".

- ***Abstract classes***: Figure 2.2 [1] defines a number of classes which has not been intended for giving an example: Employee, Permanent Employee and Temporary Employee. These are termed abstract classes and are intended as a placeholders for common features rather than as templates for a particular catogry of things. This is quite acceptable and is common practice in most object oriented programs.

- ***Inheritance of default***: Having stated that Permanent Employees earn a default salary of 14.000 a year does not imply that all types of employee have that default. In the diagram, Managers have a default of 30.000, illustrating that a class can overwrite the defaults defined in one of its parents.

- ***Single and Multiple Inheritance***: Only a single inheritance has been illustrated in Figure 2.2. That is, a class inharits from only one other class. This is the case in many object oriented programming languages, such as Java and SmallTalk. However, other languages, such as C++ and Eiffel, accept multiple inheritance. In multiple Inheritance, the characteristics of two classes can be brought together to define a third class. For example, we may have two classes, Toy and Car, which can be used to create a third class Toy-Car.

2.4.1.3 Abstraction

Abstraction is much more than just the ability to define categories of things which have in common the features of other things (for example, Temporary Employee is an abstract class of Contractor and Student Employee). It is a way of making a specification regarding what is particular about a group of classes of things. Most this means defining the interface for an object, the data that such an object have and part of the functionality of that object. For example, a class DataBuffer may be defined as the abstract class for things that hold data and return them on request. It may define how the data is held and

that operators such as put() and get() are provided to add data to and remove it from the DataBuffer. The application of these operators may be left to those implementing a subclass of DataBuffer. The class DataBuffer can be used to implement a stack or a queue. Stack could implement get() as return the most recent data item added, while queue could implement it as return the oldest data item held. In either case, a user of the class knows that put() and get() are available and work in the appropriate manner.

Abstraction is related to protection in some languages. For example, in C++ and C#, it can be stated whether a subclass can overwrite data or procedures (and indeed whether it has to overwrite them). The developer can not state in SmallTalk that a procedure can not be overwritten, but can state that a procedure (or method) is a subclass responsibility (that is, a subclass which implements the procedure in order to provide a functioning class).

Abstraction can also be refered to as the ability to define abstract data types (ADTs). In terms of object oriented these are classes (or groups of classes) which provide behaviour that acts as the infrastructure for a particular class of data type (for example, DataBuffer provides a stack or a queue). However, it is worth pointing out that ADTs are most commonly associated with procedural languages such as Ada. This is because the concepts in object orientation essentially supersede ADTs. That is, not only do they encompass all the elements of ADTs, they extend them by introducing inheritance.

2.4.1.4 Polymorphism

Essentially, Polymorphism is the ability to request that the same operation be performed by a wide range of different types of things [1]. The processing of the request depends on the thing that receives the request. How to handle the request should not be a problem for the programmer. This is illustrated in Figure 2.3.

In this example, the variable MotorVehicle can represent an instance of a MotorVehicle and any subclass of the class MotorVehicle (such as car, MotorBike or SportCar etc.).

Since a method motordrive() is defined in the class MotorVehicle each and every one of them will do their own thing. For example, driving a family car can be different from driving a motorbike or a sports car. However, these details should not be of much concern to developers, they just need to know that they will all support the motordrive() method (which they will, as they are Subclasses of MotorVehicle).

```
MotorVehicle motor:
motor= Car:
motordrive();
motor= MotorBike:
motordrive();
motor= SportCar;
motordrive();
```

Figure 2.3: An example of polymorphism

Effectively, this means that many different things to perform similar action can be asked. For example, a range of objects to provide a printable string describing them might be asked. If you ask an instance of the Manager class, a compiler object or a database object to return such a string, the same interface call (ToString in C#) can be used.

Unfortunately “Polymorphism” can be sometimes confusing. It makes the whole process sound rather grander than it actually is. The two types of polymorphism used in programming languages are as follows: overloading and overriding. Each type depends on the mechanism that resolves what code to execute.

- **Overloading Operators:** Overloading is said to occur when procedures with similar names are applied to different data types. The compiler has the ability to

determine which operator to use at compile-time and the correct version can be used.

This type of overloading is used by Ada. For example, if a new version of the "+" operator for a new data type is defined. When a programmer uses "+", the compiler uses the types associated with the operator to determine which version of "+" to use.

In C, although the same function, printf, is used to print any type of value, it can not be classified as a polymorphic function. The correct format option must be specified by the user in order to ensure that a value is correctly printed.

- **Overriding Operators:** Overriding is said to occur when a procedure is defined in a class (for example, Temporary Employee) as well as in one of its subclasses (for example, Student Employee). This means that considering instance of Temporary Employee and Student Employee, each one of them can respond to requests for this procedure (assuming it has not been made private to the class). For example, assuming that the procedure ToString is defined in these classes. The pseudo code definition of this in Temporary Employee might be:

```
public String ToString()
{
     return "I am a temporary employee"
}
```

In Student Employee, it might be defined as:

```
public String ToString()
{
     return "I am a student employee"
}
```

The procedure in Student Employee replaces the version in Temporary Employee for all instances of Student Employee. If an instance of Student Employee is asked for all the result of ToString, the string “I am a student employee” can be gotten.

In Java, the choice of which version of the procedure to execute is not determined at compile-time, because the compiler needs to be able to identify the type of object and then find the appropriate version of the procedure. Instead, the procedure is chosen at run-time. The technical term for this process of identifying the procedure at run-time rather than compile-time is called “late binding”.

2.5 Aspect Oriented Programming

Aspect oriented programming (AOP) or to be more precise, Aspect-Oriented Software development (AOSD) is a new software development paradigm that provides advanced separation of concerns [55]. The concept of AOP is originated by Gregor Kiczales and his team at Xerox PARC. The first and most popular AOP language, AspectJ is also developed by the same group. AOP was invented in order to be able to handle the issues arising by crosscutting concerns. The modularization of the code related to these concerns is not sufficient and it suffers a phenomenon called code tangling and scattering. AOSD made a promise to take care of this problem by means of providing the capability to modularize crosscutting concerns even though this affects the system implementation in a crosscutting way.

The distribution of crosscutting functionalities is achieved by the weaving mechanism at compile or run-time. It enables the implementation of a crosscutting concern at a single place. A system can therefore be constructed by describing each relative concern separately. This provides the usual benefits of increased modularity, e.g. comprehensibility, reusability, facilitated maintenance.

AOP is a new technology for separating crosscutting concerns into single units called aspects. An aspect is a modular unit of crosscutting implementation. It encapsulates behaviors that affect multiple classes into reusable modules [65].

2.5.1 Separation of Concerns

Separation of Concerns is found at the core of software engineering. Since the need of software systems are constantly increased, the software design has attained a level of complexity that the software engineering is trying to tackle with the Separation of Concerns principle. Generally, this principle refers to the capability to identify, encapsulate and manipulate the parts of a system, which are related to a particular concept, purpose or objective.

In the beginning of the seventies, Parnas [66] and Dijkstra [67] dealt with this principle in their publications. Dijkstra describes the need to separate concerns for the design of good software. The term “Separation of Concerns” was not mentioned by Parnas, but he proposes information hiding and encapsulation of software modules as a means of less coupling. Therefore modules become more isolated from change in other software modules because they are hiding behind a narrow interface.

The aim of the Separation of Concerns principle [66, 67] is to divide a complex problem that is difficult to understand into a set of smaller problems, which are less complex and more understandable. This concept tries to accommodate towards the direction that the human mind works, since the human mind has a limitation with respect to problems or concerns that it can concentrate on at a time. At the end, in order to solve the overall problem, each sub problem needs to be solved separately. Regarding the field of software development it means that different areas of interest need to be decompose into separate, independent system modules. The development of software through this way provides the capability to concentrate on all efforts of a programmer on a certain system concern at a time, whilst having in mind that he is dealing with just one part of an entire system. The

advantages of modularizing all different concerns into separated implementation units are greater comprehensibility, maintainability, adaptability and reusability.

Hürsch and Lopes discern between Separation of Concerns at the conceptual level as well as the implementation level [68]. At the conceptual level, Separation of Concerns pursues the goal to give a clear definition and conceptual identification of each concern. The resulting concerns should be correctly differentiated from each other. At the implementation level, the objective of Separation of Concerns is to encapsulate the various concerns into implementation units of the particular programming language and to provide a less coupling between them. A mutual supportive relation is mostly kept by the decomposition of software systems and the applied programming languages. The design process decomposes systems into smaller sub systems. Programming languages gives the mechanisms to generate abstractions out of these sub systems and to compose these abstractions into an overall system.

However, problems occurs due to the current established programming techniques, e.g. OO, procedural or functional programming, do not provide the mechanisms for clear modularization, and above all for composition of all concerns defined at the conceptual level [63]. The separation of concerns at the implementation level proves to be useful if those concerns can be successfully composed to the overall system later.

In addition to the computation of the basic applications, recent applications have become more and more complex due to the integration of nonfunctional requirements such as concurrency, distribution, real-time constraints, persistence and synchronization which contribute to solve the overall application problem domain.

Improving a basic application regarding nonfunctional requirements like real-time capability and distribution leads to intertwined concerns within the modular implementation units (e.g. classes, procedures, methods) of the system. In cases where requirements are interdependent, they cannot be correctly separated with recent

established programming techniques. This results into several problems as described in [68]:

- The application of different concerns leads to increased code complexity. The developer has to focus on different types of views of one functionality at a time.

- OO or procedural languages have the ability to give an appropriate abstraction to decompose the concerns at the implementation level. Therefore, the lack of abstraction decreases the comprehensibility of the program considerably.

- Adoption of the implementation becomes more complex due to the strong-coupled concerns and maintenance among themselves.

- Because of the strong coupling, it is impossible to redefine the intertwined concerns in subtypes separately [69].

The current dominant programming paradigm is the Object-orientation. This concept decomposes a given problem into a series of classes. Classes are the natural unit to encapsulate data and characteristic behavior of a certain concern for OO languages. The mechanism for composition given by OO languages is inheritance and aggregation for structural and message passing for behavioral composition.

The Separation of Concerns problem is improvement on OO paradigm in comparison with earlier technologies. However, the consequent realization of the principle at all levels has not been completely reached. There are concerns that are associated to more than one class in some way, and cannot be clearly encapsulated with the help of OO mechanism. This problem occurs due to the fact that in the object-oriented world a system becomes decomposed into modules along a single dominant concern [70]. The significant functional or basic concerns become encapsulated into first-class abstractions. They can therefore be composed and extended by means of inheritance, aggregation and message passing.

Other non-dominant concerns need to be decomposed along with this dominant one and they therefore get scattered over several modular units. Ossher and Tarr defined this problem as “Tyranny of the dominant decomposition“ [70]. Concerns that could not become encapsulated into a dominant module lead to code that is scattered over several modules, and tangle with other concerns. Those concerns are designated as *crosscutting concerns*.

2.5.2 Aspect-Oriented Software Development

AOSD provides additional mechanisms that enable the fully modularization of crosscutting concerns (or aspects) of a software system.

Crosscutting concerns originate from two main phenomena known as code scattering and code tangling [71]. Scattering occurs when it cannot correctly modularized a code of a certain concern into a system module and remains distributed throughout many modules. This includes the adaptability of the implemented concerns, because it needs to search and edit all scattered code related to a concern under consideration. Due to the scattering phenomenon, the modules (classes and methods) could harbor code pertaining many other concerns, which thereby result in an intertwined mixture of code fragments usually denoted as code tangling. This causes modules that are difficult to comprehend, maintain and reuse.

Many programming languages including object-oriented languages suffer from these phenomena, because decomposition and composition mechanisms are not sufficiently provided to clearly separate all identified system concerns. Their inherent mechanisms enable the hierarchical decomposition of systems with respect to a single dominant concern. That results in code of the non-dominant concerns scattering and tangling across the code of the dominant concern.

Crosscutting refers to the inherent structure of the concerns and it is more than scattering and tangling. The modularization of crosscutting concerns is addressed by AOSD through

the concerns and the composition of all software abstractions in a way that they do not need to be hierarchical.

Crosscutting concerns are those, which cannot be cleanly, encapsulated in the natural units of modularity (e.g. class or method in OO languages). In [63] crosscutting is defined as:
"Whenever two properties being programmed must compose differently and yet be coordinated, we say that they cross-cut each".

Crosscutting concerns cannot be classified as implementations of functional system requirements but instead properties have effect on the system in a non-functional way.

The most famous examples for such concerns are tracing, synchronization of concurrent objects and transaction management. However, that is not to say that mean that crosscutting concerns always have a non-functional nature.

Aspect-oriented programming techniques provide the means of encapsulating crosscutting concerns and thereby it tries to tackle the problems facing traditional programming techniques. Even though aspect-oriented programming is also applicable to object-oriented programming, it is an independent concept whose application is found in other programming styles.

Since AOP is an approach, it is not related to a specific programming language. In fact, it can be useful with the shortcomings of all languages (not only OO languages) that use single, hierarchical decomposition. AOP has been applied in different languages (for example, C++, Smalltalk, C#, C, and Java).

The java language gained the interest of most research community. The following is a list of tools that support AOP with Java:

- AspectJ
- AspectWerkz

- Hyper/J
- JAC
- JMangler
- MixJuice
- PROSE
- ArchJava

In the next section, an overview about the AspectJ will be introduced as an example of the aspect-oriented programming.

2.5.3 AspectJ

The general-purpose aspect language *AspectJ* has been developed by The AOP research project. It is an aspect-oriented extension to the object-oriented base language Java [62]. Some years ago, AspectJ became one of the most popular general-purpose aspect languages. This is backed up by a large and growing number of user communities.

AspectJ is considered to be a practical AOP language that gives a dynamic join point model and a set of new language constructs. This set of constructs is *pointcut*, *advice* and *introduction*, which are mostly encapsulated in modular units of crosscutting implementation so-called *aspects.* Also, aspects can consist of additional methods, fields and initializers like an ordinary Java class.

2.5.3.1 Join Point Model

The central concept in an aspect-oriented language is the join point model. The capability of an AOP language to support crosscutting lies in its join point model. Join points are defined as principled points in the execution of the program [62]. Crosscutting behavior can be attached at those well-defined points. The meaning of principle is that, behavior improvement cannot take place at arbitrary points in the execution of a program.

AspectJ can be used to create a dynamic join point model [61]. The dynamic property can be thought of join points as nodes in a simple runtime object call graph [62]. Alternatively, join points can be imagine as events in the control flow of a program.

The following well-defined join points are introduced by AspectJ [61]: *method call* and *execution*, *constructor call* and *execution*, *read/write access* to a field, *exception handler execution*, *object* and *class initialization* execution. Constructor or method call join points determine those places in the control flow of a call where the method's arguments are already evaluated but the code body is not yet executed. In comparison to the call join point the execution join point is to occur when the control flow has reached the code body. On the other hand, the fundamental difference between them is that, a call join point occurs outside the object or class (for static elements) whiles an execution join point takes place inside the object or class. Also, AspectJ provides join points that takes all access to class attributes and an exception handler execution join point. Finally, the model is made of join points that capture the static class as well as object initialization.

2.5.3.2 Pointcut

A *pointcut* is a set of join points [61,62]. In AspectJ pointcut sometimes can be named or anonymous. An autonomous declaration uses the keyword *pointcut* to define named pointcuts. In comparison to anonymous pointcuts that are defined as part of advice or named pointcut declarations. The term *pointcut designator* is applied by AspectJ team in two-different ways, firstly, as an identifier of a named pointcut and secondly as a language expression. The language expression is used for specifying where to pick out join points from the run-time context.

2.5.3.3 Advice

An *advice* is a method like construct is used to declare that some code should be executed when a join point addressed by a pointcut is reached [61,62]. Syntactically it is made of three important parts: the type of advice denoted by a keyword, a pointcut

designator and the advice body. AspectJ gives different variety kinds of advice, which are denoted by the keywords: *before*, *after,* and *around*.

AdviceType ([Formals]) : Pointcut designator { Body }

Figure 2.4: General form of advice

There are three different kinds of advice defined by AspectJ: one of them is executed *before* its join points; the other one *after* and the last one runs instead (*around*) its join points. The structure of each advice declaration is executed at a suitable time relative to each join point.

Even though not all advice makes sense for every pointcut, the AspectJ syntax currently allows all combinations; in some cases, around advice simply acts like before and/or after advice, depending on when and if the original join point is invoked.

before(): get(int Foo.y) {...}: runs before reading the integer field Foo.y
after() returning(int x): pointcut {...} runs after the join point returns. The join point must return an integer value. The value is bound to x in the body.

int around(): call(int Point.getX()) {...}: runs instead of calls to Point's int getX() method. The getX() may be invoked in the body using proceed(), which has the same signature as the around advice. Around advice may also declare thrown exceptions; these must not break Java's static type safety rules.

2.5.3.4 Aspects

AspectJ is used for providing a modular unit of crosscutting implementation, the so-called *aspect* [61]. Aspects are the native encapsulation unit of the already discussed new language constructs pointcut, advice and introduction. They are made of the information how it's crosscutting implementation cuts dynamically and statically over all

the base program. Classes and aspects have some similarity because both are types and have an implementation that can be made of ordinary Java member with no crosscutting effects like methods, fields and initializers. An aspect can change the way the base code behaves (the non-aspect part of a program) by application of advice (additional behavior) at various join points (points in a program) specified in quantification or query called a pointcut (that identified whether a given join point matches). An aspect can also make binary-compatible structural changes to other classes, like adding members or parents.

2.6 Event Driven Programming

During the early days of computing, a program starts execution and then it continues through its steps until it is completed. If the user of the computer is involved, then the interaction is strictly controlled and limited to data entering into fields [24].

The program may be simple and prints a prompt, wait for the user to enter a name, displays a message with this name in it, and then waits for him to strike a key to exit the program. This style is easy to comprehend and program. The problem arises when the programs need to become more sophisticated and the user must deal with more than just inputting data with the keyboard [27].

Today's embedded system, Graphical User Interface (GUI) programming model, and many other programming needs a different paradigm, known as event-driven programming [24].

Event-driven programming is an easy way to enable the programs to respond to many different inputs or events.

Today's programming presents a user interface and waits for an action to be taken by the user. Many different actions may be taken by the user, such as making selections in menu, pushing buttons, updating text fields, clicking icons, and many others. Each action makes an event to be raised. Other events can be raised without the direct action of the user, such as events corresponding to timer ticks of the internal clock, email being received, file-

copy operations completing, etc. In programming, a situation where a particular action needs to be executed is often presented, but the method or even the object which is called upon to be executed is not known in advanced. For example, a button might know that it must notify some object when it is pressed but it might not know which object or objects need to be notified [28].

In addition to the GUI, the computer operating systems are another classic example of event-driven programs on at least two levels [44]. At the lowest level, interrupt handlers behave like direct event handlers for hardware events, with the CPU hardware performing the role of the dispatcher. Operating systems mostly function as dispatchers for software processes, passing data and software interrupts to user processes that in many cases are programmed as event handlers themselves.

A command line interface can be considered as a special case of the event-driven model in which the system, which is inactive, waits for one very complex event – the entry of a command by the user [44].

Event-driven programs upgrade on sequential programs by acquiring a central event handler and dispatcher that waits for an event (any event) to occur, and then execute that event by calling that event handler [27].

Separation of the event detection and the event handling is an important technique for maintaining the simplicity and flexibility of the program [27].

Applications of Event-driven programs are not bound by the constraints of procedural programs. Rather than the top-down approach of procedural languages, event-driven programs contain logical sections of code placed within events. There is no predefined order in which events occur, and usually the user has complete control over what code to be executed in an event-driven program by interactively triggering specific events, such as by clicking a button. The code which is contained in the event is called an event procedure [29].

2.6.1 Event Handling in General

Event handling is consisting of dealing with a situation whereby something has occurred and the software developer has to be notified of that situation. Sometimes the code written to take care of these situations is known as a callback and sometimes as an event handler. In both cases, the same fundamental principle is applied. That is, the developer has to implement (in C# for example) a method that matches some specification that allows it to be called when the "event" occurs. This event can for example be some threshold being reached in some sensor, it could be a message being received from some broadcast mechanism or it could be some user interaction with a GUI. Generally, this event handling method will be invoked when the event occurs and will be passed some data to enable it identify the sender of the event and any event-specific data. The action that the application must take is then determined by the event handler [1]. Figure 2.5 gives an illustration of the interactions that take place when handling GUI events in a little more detail. The three main steps with respect to a button are illustrated as follows:

- The user clicks on the button.
- An EventArgs object is created by the button. This is an object that contains any additional data that must be made available to the event handler. This step is known as raising an event in C# terminology.
- The button then invokes a suitable handler method (on an object somewhere) passing in a reference to itself (as the sender of the event) and the event args.
- The handler method can then apply any appropriate operation that it needs to perform.

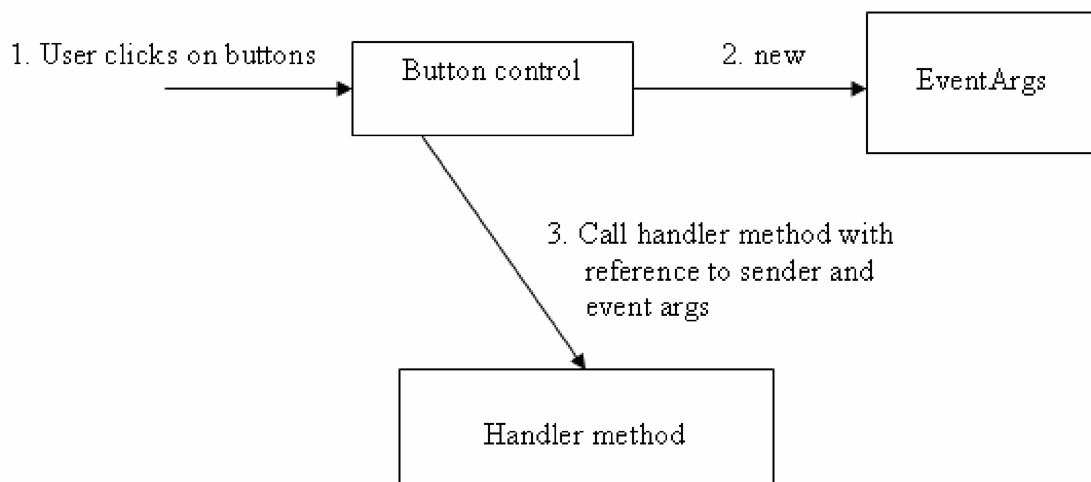

Figure 2.5: Event handling concepts

Delegates are used to implement events in C#. A *delegate* is a reference type that encapsulates methods with a specified parameter list and return type [2]. As discussed in chapter 3, a delegate is formed with the delegate keyword, followed by a return type and the parameter list of the methods that can be delegated to it. As soon as the delegate is defined, a member method can be encapsulated with that delegate by instantiating the delegate and passing in as a parameter the name of a method that matches the return type and parameter list.

In C#, any object can publish a set of events from which other classes can subscribe. When an event is raised by the publishing class, notification is send to the subscribed classes. The publishing class defines a delegate that must be applied by the subscribing classes. When the event is raised, the subscribing class's methods are invoked through the delegate [28].

As previously discussed, any method that handles an event is called event handler. Event handlers can be declared to any other delegate.

By convention, event handlers in the .NET Framework return void and take two parameters. The first parameter is the "source" of the event, that is, the publishing object. The second parameter is an object derived from EventArgs. The EventArgs class contains information regarding the event that can be beneficial to the event handler method.

There are some programming languages that try to apply the style of event driven programming paradigm such as Visual Basic, C#, Visual C++, J#, Java, JBuilder and most modern language but in fact it can not be confirmed that these programming languages are event driven programming. In these languages, the event handling is distributed over the whole program text [50]. Because of that, there is no clear overview where and under which condition events may occur and which effects they cause. It always depends on the existing general situation of a program, which is mostly characterized by the existence, and values of program objects, i.e. the current memory assignment. This procedure complicates the maintenance of software.

In this thesis, as will be seen in chapter 5, an attempt is made to address this problem by clearly separate event handling from the actions of the program. A special program is introduced, which is responsible for event handling alone. Every event may alter the event state. Therefore there exist one cycle in the program flow, which is found by raising events and changing the event state.

In chapter 3, the implementation of the events handling and how it works in different programming languages especially in the .NET Framework is discussed

2.6.2 Advantages and disadvantages of Event-driven programming

- **The primary advantages** of event-driven programming are as stated below:
    - **Flexibility:** Because the flow of the application is monitored by events instead of a sequential program, it is not necessary for the user to conform to the programmer's understanding of how tasks should be performed [46].

- **Robustness:** Event-driven applications happen to be more robust because of their less sensitivity to the order in which users perform activities. In conventional programming, the programmer has to expect every sequence of activities virtually that the user might execute and define feedbacks to these sequences [46].

- **The prime disadvantage** of event-driven programs is that it is often difficult to find the source of errors when they can occur. This problem unfolds from the object-oriented nature of event-driven applications— since events are associated with a particular object, which the user may have to examine many objects before discovering the inappropriate procedure. This is especially true when events cascade (i.e., an event for one object triggers an event for a different object, and so on) [46].

2.7 Integration of Programming Techniques

In recent times, imperative object-oriented languages C++, Java, and Object Pascal supported by many developing tools, are the most popularly selected to carry out large programming projects., On the other hand these languages are not adequately suitable for implementation of large software projects, where by one or many problems often belong to the symbolic processing domain where non-traditional languages, such as Lisp or Prolog, are more adequate [74].

AOP compliments OOP. It relies on cross-cutting concerns or aspects parts of code that are associated with a large variety of objects, whereby logging is the canonical example. Applying an AOP language (for ex. AspectJ) or libraries (for ex. Spring), programmers have the capability of coding this functionality once and then define where to weave it into existing objects. Many security-relevant cross-cutting concerns are found scattered throughout the application logic such as Logging, access control, error handling, transaction management, session management and input/output validation. With the

application of AOP, a large chunk of these concerns can be separated from the code base and centralize them [75].

Therefore an integration of programming techniques from different languages and styles is very much needed so as to facilitate implementation of programming projects. Particularly, it is most attractive to improve the power of popular imperative object-oriented languages with special data structures and control mechanisms from non-traditional languages.

In addition to these paradigms, it should be noted that the event driven programming is an alternative way of integration, i.e. direct integration of programming codes written in different languages, now becomes perspective in connection with the development of Microsoft.NET platform, which allows compiling and linking such different codes [74].

Over the past few years Event-driven programming has gained considerable recognition in the field of software engineering. Many based on this approach have been developed, especially in the application of Graphical User Interface (GUI) and embedded system. Event driven methods facilitate the separation of concerns: the application layer (known as business logic) provides the operations to be executed, whereas the GUI layer initiates their execution in response to human users' actions [76].

Learning Programming Paradigms

The importance of integrating various programming techniques and languages within the same software project is the objective of modern programming. Therefore, an intensive education in the field of computer science should be focused on learning programming techniques of different paradigms. These suggest the importance of learning many different algorithmic languages, because there are no languages that can consist of all possible techniques from various programming styles.

Therefore the idea is to learn modern programming languages and to compare base programming techniques and to explicate distinguishing features of the paradigms and

also acquiring a very good knowledge of programming techniques and programming languages.

The advantages of learning programming paradigms are also illustrated by the fact that the future of popular modern languages is not known. According to the history of programming, many languages became extinct, while the popularity of some other languages have been lost, similar fate awaits some other modern languages. However, the main programming paradigms will remain the same, as well as their base programming techniques, and this makes their learning a permanent constituent of education in the field of computer science [74].

3

Event-Handling in Different Programming System

In this chapter, some principles about the events and delegates in the .NET Framework, especially C# language, will be presented, since the implementation of my framework was implemented in C# language. A few description about how the events and event handler is working in different programming system in Microsoft Visual Studio such as Visual basic, C#, Visual C++ and Visual J# are described. Some examples on how the events was using in other programming languages such as Java and JBuilder are also introduced.

3.1 Events

Technically, the event in the event-driven programming can be defined as a software message in order to specify that something had occurred, such as a keystroke or mouse click [26]. In process control, the event can be defined as an occurrence that has happened and has been registered. So, an event can be defined as an important alteration in a state or any action or occurrence detected by a program. Events can be user actions, such as clicking a mouse button or pressing a key, or system occurrences, such as running out of memory. Events can also be created by state changes of objects. In most recent applications, especially those that run in Macintosh and Windows environments are said to be event-driven, since they are designed to respond to events.

Events are applied in graphical user interfaces (GUIs); specifically, the classes that are represented by controls in the interface contain events that are notified when the user does something to the control (for example, click a button). An event is the results of an action.

The two important terms regarding events are `event source` and `event receiver`. The object that raises the event is called `event source` and the object that responds to the event is called `event receiver`. The communication channel between an `event source` and an `event receiver` is the delegate.

In windows applications, especially in the visual studio programming languages such as Visual Basic, C#, Visual C++, and J#, the implementation of the events is done by means of delegates. Therefore, a discussion is first presented regarding delegates and the relation between the events and delegates. The next step discusses how the events are implemented in the visual studio languages and some comparison between them. Furthermore illustrative examples are also presented. Finally the implementation of the events handling in some other languages such as Java and JBuilder are described.

3.2 Delegates

Delegates work as an intermediary between an event source and an event destination. Technically, a delegate is a reference type used to encapsulate a method with a specific signature and return type. It is possible to encapsulate any method in that delegate. To be more specific, delegates have similarity with respect to function pointers. They can be called as type safe function pointers. Unlike function pointers, delegates are object-oriented and type safe. An event handler is a delegate class that is used as an intermediary between an event source and event receiver [24].

There are three steps in defining and using delegates:

- Declaration
- Instantiation
- Invocation

3.2.1 Declaration

A delegate is generated using the delegate keyword, followed by a return type and the signature of the methods that can be delegated to it, for example in C# the delegate can be declared as follows:

```
public delegate int MyDelegate(object obj1, object obj2);
```

This statement declares a delegate called `MyDelegate`, which will encapsulate any method that takes two objects as parameters and that returns an `int`. After the definition of the delegate, a member method with that delegate is encapsulated by instantiating the delegate, passing in a method that matches the return type and signature. The delegate can then be used to call that encapsulated method.

3.2.2 Instantiation

The delegate needs to be instantiated before it can be used to specify the method that needs to be called.

```
public void MyMethod()
{
      MyDelegate a = new MyDelegate(MyDelegateMethod);
}
```

Here MyDelegateMethod is a method that has a signature similar to that of MyDelegate.

3.2.3 Invocation

A delegate is used to invoke a method similar to how a method call is made. For example:

MyDelegateMethod("This is a test invocation");

3.3 Events and Delegates

Declaration of an event is directly coupled to a delegate. A method can be encapsulated by a delegate object so that it can be called anonymously. An event is a mechanism by which a client class can pass in delegates to methods that need to be called whenever “something happens”. When this happens, the delegates given to it by its clients are called [33].

The syntax below is used to declare an event in C#:

```
public delegate void testDelegate(int a);

public event testDelegate MyEvent;
```

After the declaration of an event, it must have an association with one or more event handlers before it can be raised. An event handler is just a method that is invoked using a delegate. The += operator is used to associate an event with an instance of a delegate that is already existing. For example:

```
MyForm.MyEvent += new testEvent(MyMethod);
```

An event handler can be deleted or removed as follows:

```
MyForm.MyEvent -= new testEvent(MyMethod);
```

In C#, events can be raised by just calling them by its name which has similarity to method invocation, for example

```
MyEvent(10).
```

How Event works?

Once an event is defined for a class, the compiler generates three methods that are used to manage the underlying delegate:

- **add_<EventName>:**
  This is a public method that calls the static Combine method of `System.Delegate` in order to add another method to its internal invocation list. The application of this method is however not explicit. The same result is produced by using the += operator as specified before.

- **remove_<EventName>:**
  This is also a public method that calls the static Remove method of `System.Delegate` in order to remove a receiver from the event's invocation list. This method is also not directly called. Its job is accomplished by means of the "-=" operator.

- **raise_<EventName>:**
  A protected method that calls the compiler produced an Invoke method of the delegate, in order to call each method in the invocation list.

3.4 Types of Delegates

Fundamentally there are two types of delegates. Single Cast delegate and Multi-Cast delegate. A single cast delegate calls only a single function. Whiles a multi-cast delegate is type that can be part of a linked list. The multi-cast delegate points to the head of such a linked list. That means all the functions that form a part of the linked list are called when the multi-cast delegate is invoked. Assuming that one has many clients who would like to receive notification when a particular event happens. Joining all of them in a multi-cast delegate can be helpful for calling all the clients when a particular event occurs [31].

In order to support a single cast delegate the base class library contains a special class type called System.Delegate. To support multi-cast delegates the base class library includes a special class type called SystemMultiCastDelegate.

3.5 Delegates and Their Roles

Delegates are classes mostly applied within the .NET Framework to construct event-handling mechanisms. Delegates have close similarity to function pointers, commonly used in C++ as well as other object-oriented languages. Unlike function pointers however, delegates are object-oriented, type-safe, and secure. Additionally, where a function pointer have only a reference to a particular function, a delegate consists of a reference to an object, and references to one or more methods within the object.

Delegates are used by the event model to bind events to the methods used to handle them. The delegate enables other classes to register for event notification by specifying a handler method. When the event occurs, the bound method is called by the delegate.

Delegates can be bound to a single method or to multiple methods, referred to as multicasting. During the creation of a delegate for an event, a multicast event is typically created by the user (or the Windows Forms Designer). An exceptional case can be an event that is resulted in a specific procedure (such as displaying a dialog box) that would not logically be repeated in multiple times per event.

A multicast delegate maintains an invocation list of the methods it is bound to. The multicast delegate supports a Combine method to add a method to the invocation list and a Remove method to remove it.

When an event is registered by the application, the control raises the event by calling the delegate for that event. The delegate in turn calls the bound method. In most popular case (a multicast delegate) each bound method in the invocation list is called by the delegate, which provides a one-to-many notification. This strategy means that there is no need for the control to maintain a list of target objects for event notification; the delegate handles all registration and notification.

Delegates also permit multiple events to be bound to the same method, thereby allowing a many-to-one notification. For instance, the delegate can be invoked by a button-click

event and a menu-command–click event, which then calls a single method to handle these separate events in the similar way.

The binding mechanism applied to delegates is dynamic — a delegate can be bound at run time to any method whose signature matches that of the event handler. This feature allows the user to set up or make changes to the bound method depending on a condition and to dynamically attach an event handler to a control.

3.6 Events and Delegates in Visual Basic vs. C#

Generally there are small differences between how events are declared, raised, and handled in Visual Basic (VB.NET) versus C#. Events are a simple concept that is not difficult to implement in VB 6.0. All of the message wiring is handled behind the scenes [25].

The implementation of events in the .NET Framework is simple. A group of delegates are kept internally and, whenever an event is occurs each delegate is invoked (i.e. delegated to).

3.6.1 Declaring Events in Visual Basic and C#

In Visual Basic .NET the use of delegates are virtually hidden from the user. The syntax for their declaration is the same as done in VB 6.0:

```
Public Event TotalUpdatedEvent(ByVal Amount As Decimal)
```

It's not so transparent in C#. We first must define a delegate and then declare an event of that delegate type:

```
public  delegate void totalUpdatedEventhandler(decimal amount);
public  event  totalUpdatedEventHandler  totalUpdatedEvent ;
```

There is an additional very important difference. Take note of the `void` in the C# code above. In this specific case, a value can not be returned by event handler; however, it is possible for handlers in C# to return values. This is not permitted in VB.NET where all handlers must be in the form of Subs. Instead, in VB.NET events can pass `ByRef` parameters. A Cancel argument on a Closing event is a common example.

3.6.2 Raising Events in Visual Basic and C#

The basic differences between VB.NET and C# for raising events are rather uneventful as illustrated below:

```
RaiseEvent  TotalUpdatedEvent (123.45)
```

Versus the following in C#:

```
TotalUpdatedEvent (123.45);
```

3.6.3 Implementing Event Handlers (VB vs. C#)

There are two different procedures for implementing event handlers in VB.NET. The most popularly used method is to define a handler at design time with the `Handles` keyword. The `WithEvents` keyword is used to declare the object instance that raise the event, a carryover from VB 6.0 as shown below:

```
Module modmain

      Private WithEvents calculator As calculator = New calculator()

      Public Sub Main()
            Call calculator.DoSomethingEventful()
      End Sub

      Private Sub TotalUpdated(ByVal Amount As Decimal) Handles
            calculator.TotalUpdatedEvent
            Trace.Write("Total was updated to " + Amount.ToString())
      End Sub
End Module
```

```
Public Class Calculator
      Public Event TotalUpdatedEvent(ByVal Amount As Decimal)
      Public Sub DoSomethingEventful()
            RaiseEvent TotalUpdatedEvent(123.45)
      End Sub
End Class
```

In the second method an event is associated with a handler function at run time by using the `AddHandler` together with `AddressOf` as follows:

```
Module modmain

      Private WithEvents calculator As calculator = New calculator()

      Public Sub Main()
            AddHandler calculator.TotalUpdatedEvent, AddressOf
                        TotalUpdated

            Call calculator.DoSomethingEventful()
      End Sub

      Private Sub TotalUpdated(ByVal Amount As Decimal) Handles
                  calculator.TotalUpdatedEvent

            Trace.Write("Total was updated to " + Amount.ToString())
      End Sub

End Module
```

All handlers **in C#** must be associated at run time. This example illustrates the handling of events in C#:

```
public class EventReceiver
{
   static void Main(string[] args)
   {
      EventReceiver demo = new EventReceiver();
      demo.doDemo();
   }
   public void doDemo()
   {
      Calculator calculator = new Calculator();

      Calculator.totalUpdatedEventHandler handler = new
            Calculator.totalUpdatedEventHandler (OnTotalUpdated);
      calculator.totalUpdatedEvent += handler;
      calculator.doSomethingEventful();
```

```
    }

    public void OnTotalUpdated(decimal amount)
    {
        Trace.Write("Total was updated to " + amount.ToString());
    }
}

internal class Calculator
{
    public delegate void totalUpdatedEventHandler(decimal amount);
    public event totalUpdatedEventHandler
    public void doSomethingEventful()
    {
        totalUpdatedEvent(123.45M);
    }
}
```

The following are important points about events:

- At run time, Handlers can be removed with `RemoveHandler` and the -= operator.
- Handlers are executed in their order of association.
- In C#, when a single event is associated with multiple handlers and the handler signature has a return type, then the value returned by the last handler executed will be the one returned to the event raiser. Default delegate types are provided by Visual Basic .NET to the use. But the endless generosity of Microsoft does not stop at this point. Also Default delegates are provided for the events of the .NET Framework's controls and classes. Therefore, the only problem lies with delegates when events in C# are being defined [25].

Sometimes the following question can be asked: What is the difference between an exception and an event? The fundamental difference is that exceptions represent unexpected conditions that are not supposed to happen [34]. For example, the program runs out of memory or encounters divide by zero. These situations are not expected to happen however if they do, then the program has to cope with it. On the other hand, Events are part of normal day to day operation and are fully expected to occur. The user moves the mouse or presses a key. The browser just move to a new page. From the point

of view of a control-flow, an event is a function call, whereas an exception is a long jump across the stack, with unwinding semantics to destroy lost objects.

3.7 Events in Visual C++

The unified event Model is used by event handling in Visual C++. This allows the use of the same programming model for event handling in all types of classes in Visual C++: Native C++ classes, COM classes and Managed classes [35].

An event source and event receiver need to be set up using the attributes `event_source` and `event_receiver` in all types of classes in Visual C++ respectively, specifying *type*=**native** in native C++ classes, specifying *type*=**com** in COM C++ classes, and specifying *type*=**managed** in managed C++ classes. These attributes enable the classes to which they are applied to fire events and handle events in all types of classes in Visual C++.

3.7.1 Declaring Events

In an event source class, the `__event` keyword is used on a method declaration to specify the method as an event. In the case of the native C++ and managed C++, the method needs to be declared, but not to define it. A compiler error is generated if the method is defined; this is because the compiler defines the method implicitly when it is made into an event. In COM classes, the same `__event` keyword is applied on an interface declaration in order to declare that interface's methods as events. The events of that interface are called when they are call as interface methods. The events can be methods with zero or more parameters. The return type can be void or any integral type.

3.7.2 Defining Event Handlers

Event handlers are defined in an event receiver class, which are methods with signatures (return types, calling conventions, and arguments) that match to the event that

they will handle. For COM events, there is no matching for calling conventions. Also in an event receiver class, the intrinsic function `__hook` is used to associate events with event handlers and `__unhook` to dissociate events from event handlers. Many events can be hook to an event handler and vice versa [30].

3.7.3 Firing Events

To fire an event, simply call the method declared as an event in the event source class. If handlers have been hooked to the event, the handlers will be called.

3.8 Delegates and Events in J#

Due to the absence of built-in support for delegates and events in the Java programming language, a set of extensions are provided by the J# implementation for creating this capability. Delegates are tagged with a special comment /** @delgate */ before its definition and the same applies to events (using /** @event */). Also a reference needs to be kept by the class implementing the events to all the listeners assigned for the particular events by creating an ArrayList or a similar collection. After the listeners have been assigned, they are invoked by the Invoke method.

Example:

```
import System.*;
import System.Collections.*;
/** @delegate */
public delegate void EventHandler();

public class Button
{
	ArrayList listeners = new ArrayList();
	public static void main()
	{
		Button button = new Button();
		button.add_OnClick(new EventHandler(Button_OnClick));
		button.Click();
	}
	/** @event */
	public void add_OnClick(EventHandler listener)
```

```
        {
                listeners.Add(listener);
        }
        /** @event */
        public void remove_OnClick(EventHandler listener)
        {
                listeners.Remove(listener);
        }
        public void Click()
        {
                Object[] olisteners = listeners.ToArray();
                for (int i = 0; i < olisteners.length; i++)
                {
                        ((EventHandler)(olisteners[i])).Invoke();
                }
        }
        public static void Button_OnClick()
        {
                Console.WriteLine("Button Clicked");
        }
}
```

3.9 Events in Java

In Java, objects are used for the representation of events. When an event occurs, the system takes all the important information related to the event and an object constructed to contain that information. Different types of events are represented by objects belonging to different classes. For example, when the user clicks a button on the mouse, an object belonging to a class called `MouseEvent` is constructed. There information in the object such as the GUI component on which the user clicked, the `(x,y)` coordinates of the point in the component where the click occurred, and which button on the mouse was clicked. A `KeyEvent` is created when the user presses a key on the keyboard. After construction of the event object, it is passed as a parameter to a designated subroutine. By writing that subroutine, the programmer says what should happen when the event occurs [17].

There are so many processes that happen between the times that the user presses a key or moves the mouse and the time that a subroutine in the program is called to respond to the event. Fortunately there is no need to know much about that processing. But this needs to

be understood. Even though our GUI program does not have a `main()` routine, there is a sort of main routine running somewhere that executes a loop of the form

while the program is still running:
  Wait for the next event to occur
  Call a subroutine to handle the event

This loop is called an event loop. Every GUI program contains an event loop. This loop does not need to be written in Java, because it is part of the system. If a GUI program is written in some other language, a main routine that runs an event loop might have to be provided.

For the effectiveness of an event, the event must be detected by the program and react to it. For an event to be detected, the program must listen for it. Listening for events is a procedure that is performed by an object called an `event listener`. An `event listener` object should have instance methods for handling the events for which it listens. For example, if an object is to serve as a listener for events of type `MouseEvent`, then it must has the following method (among several others):

```
public void mousePressed(MouseEvent evt) { . . . }
```

The structure of the method defines how the object responds when it is notified that a mouse button has been clicked. Information about the event is found in the parameter `evt`. The `listener` object uses this information to determine its response.

The methods that are needed in a mouse event listener are defined in an interface named `MouseListener`. An object must implement this `MouseListener` interface in order to use it as a listener for mouse events.

Every event in Java is contains a GUI component. For example, when the user clicks a button on the mouse, the associated component is the one that is clicked by the user. The listener object must be registered with the component before it can "hear" events

associated with a given component. If a MouseListener object mListener needs to hear mouse events associated with a component object comp, the listener must be registered with the component by calling "comp.addMouseListener(mListener);".

The addActionListener() method is an instance method in the class Component. Particularly, because an applet is a component, all applet has an addMouseListener(), and so it is possible to set up a listener to react to clicks on the applet.

The event classes, such as MouseEvent, and the listener interfaces, such as MouseListener, are specified in the package java.awt.event. This means that if the program needs to work with the events, then the line "import java.awt.event;" should be included at the beginning of the source code file.

In principle, there are a so many of details to tend to when events needs to be used. In summary:

- Place the import specification "import java.awt.event;" at the start of the source code.
- Specify that some class implements the appropriate listener interface, for example MouseListener.
- Definitions in the class for the subroutines from that interface must be provided.
- Make registration of the listener object with the applet or other component.

The MouseListener interface specifies five different instance methods:

```
public  void  mousePressed(MouseEvent evt);
public  void  mouseReleased(MouseEvent evt);
public  void  mouseClicked(MouseEvent evt);
```

```
public  void  mouseEntered(MouseEvent evt);
public  void  mouseExited(MouseEvent evt);
```

Keyboard Events

A GUI applies the idea of input focus to find the component associated with keyboard events. At any specified time, precisely one interface element on the screen has the input focus, and that is where all the events of the keyboard are directed. If the interface element is a Java component, then the information about the keyboard event changes to a Java object of type `KeyEvent`, and it is transferred to any listener objects that are listening for `KeyEvents` associated with that component. The importance of managing input focus adds an extra complexity to working with keyboard events in Java [17].

In Java, keyboard event objects are related to a class called `KeyEvent`. The interface `KeyListener` must be implemented by an object that needs to listen for `KeyEvents`. The object must also be registered with a component by calling the component's `addKeyListener()` method. When an applet is to listen for keyboard events on itself, the registration is done with the command "`addKeyListener(this);`" in the applet's `init()` method. All this is of course directly related to the mouse events as previously discussed above. The `KeyListener` interface defines the following methods, which must be added to any class that implements `KeyListener`:

```
public  void  keyPressed(KeyEvent evt);
public  void  keyReleased(KeyEvent evt);
public  void  keyTyped(KeyEvent evt);
```

There are three types of `KeyEvent` found in Java. The types associated to pressing a key, releasing a key, and typing a character. When the user presses a key, then the `keyPressed` method is invoked, the `keyReleased` method is called when the user releases a key, and the `keyTyped` method is called when the user types a character. Note

that one user action such as pressing the "E" key can be responsible for two events, a `keyPressed` event and a `keyTyped` event.

In the case of a `keyTyped` event, the character that was typed needs to be known. This information can be gotten from the parameter `evt` in the `keyTyped` method by calling the function `evt.getKeyChar()`. This function returns a value of type `char` representing the character that was typed.

3.10 Event Handler in JBuilder

When a user interacts with a user interface such as clicking a button or selecting a menu item, an event-handling code is executed. The user can interact with any component. When a user interaction with any component happens, a message is displayed by the component. The program must listen for the component's message and respond appropriately before it can react to that interaction. The program requires a *listener* to listen for the component message, and an *event handler* to respond.

In JBuilder, all supported events for the selected component are listed by the Inspector's Events page. Each event contains a default action, out of many possible actions. For example when user double-click an event in the Inspector, JBuilder writes a listener and a stub (empty) event-handling method for the event's default action, and changes to the Source view with the cursor in the stub event-handler. The code is manually filled in thereby describing what the program should perform in response to that event.

Visual components like dialog boxes usually appear only when event-handling code is executed. (These components appear in the Default designer). For example, a dialog box is not part of the UI surface, but it's a separate UI element, which appears transiently as a result of a user operation such as a menu choice or a button press. Therefore, some of the code associated with using the dialog, such as a call to its show() method, has to be placed into the event-handling method.

3.10.1 Connecting controls and events

Event handlers are connected to their controllers by using event adapters. This can be done by using either standard event adapters or anonymous inner class adapters.

A name class is created by standard adapters. The advantage of this is that the adapter can be reused, and can be referred to later and from elsewhere in the code. On the other hand anonymous adapters create inline code. The advantage of this also is that the code is smaller and more elegant but this is single-use only [36].

When an anonymous adapter is used, the only code generated by JBuilder is the listener and the event-handling stub. When a standard event adapter is used, JBuilder generates three pieces of code:

- The event–handling stub.
- An EventAdapter.
- An EventListner.

JBuilder generates an EventAdapter class for each specific component/event connection and a name, which is associated to that particular component and event is given to it. This code is put in a new class declaration at the bottom of the file.

3.10.2 Standard event adapters

JBuilder creates an event adapter class that implements the desire interface. The class is then instantiated in the UI file and it is registered as a listener for the component [36]. For example, for a `jButton1` event, it calls `jButton1.addActionListener()`.

The advantage of standard adapter is the ability to be reused, because it is named. On the other hand its disadvantage is the limit imposed on its usefulness because it has only public and package access.

3.10.3 Anonymous inner class adapters

Inner class event adapters can be also created by JBuilder. The following are the advantages of Inner classes:

- The code is created inline, which makes the appearance of the code much simpler.
- The inner class can get access to all variables in scope where it is defined, compared with the standard event adapters that have only public and package access.

The specific types of inner class event adapters that are created by JBuilder are known as *anonymous adapters*. This style of adapter creates an adapter class with no name. The advantage is that the resulting code is compact and elegant. Whereas the disadvantage is that this adapter can only be applied for this one event, because it has not been given any name and therefore cannot be invoked from elsewhere [36].

4

The Abstract Model

4.1 Structure and Semantics

Here, a model, defined by Bachmann [50], is presented. This thesis focuses on this model and the main idea is to implement it as a framework as will be discussed in chapter 5.

Definition: A (formal) event-driven-system is a 4-tuple EDS=(S,E,A,δ),
where

- S is a finite set (of event-states),
- $E \subset S^S$ is a set (of events), where each event $e \in E$ is a total function e: $S \rightarrow S$,
- A *is* a set (of program actions) and
- δ: $S \rightarrow A^*$ is a control functions.

In this definition, it is not specified what the elements of A, are the actions. Therefore, the meaning of the model must be restricted to event-handling (the specification part) and cannot reflect the meaning of the whole event-driven system. Especially, it is ignored; that actions can raise events too and in this way change the event-state.

In order to define semantics we consider the set of ordered pairs (S, E) as a transition system where to each state s and each event e a transition s $\rightarrow$ e(s) is formed.

For any sequence $\alpha \in E^*$ of events we define the final state resulting from the execution of all events in α in the corresponding order as follows:

$$\varepsilon(s) := s, \quad e.\alpha(s) := \alpha(e(s))$$

where ε denotes the empty sequence.

The effect of a sequence $\alpha \in E^*$ of events is extended to generate all the intermediate states by considering the function $\alpha^* : S \to S^*$ defined as

$$\varepsilon^*(s) := s, \; e.\alpha^*(s) := s.\alpha^*(e(s))$$

Analogously, the control functions δ is extended to $\delta^* : S^* \to A^*$, by

$$\delta^*(\varepsilon) := \varepsilon, \quad \delta^*(s.\sigma) := \delta(s).\, \delta^*(\sigma).$$

Accordingly, the meaning of the event-driven system is defined as follows: for any sequence $\alpha \in E^*$ of events and any initial state s, a sequence $\delta^*(\alpha^*(s))$ of program actions which are executed when ε occurs are produced [50].

4.2 Refinements

The sets S and E and the function δ are precisely defined as follows:

Let X be a set of variables, then the set of states S should be $S=IN^X$, where IN denotes the set of natural numbers. Therefore, each $s \in S$ is a function $s:X \to IN$, which assigns to every variable x a number $s(x)$, the value of variable x. Obviously, in practice, not only natural numbers, but other elements, like integers or rational numbers are needed. This will be considered in the implemented framework. However, by the formal model, it was also investigated in what a way verifications can be done and therefore, the values were restricted to natural numbers.

An event e is defined by a sequence of statements of the form $C \Rightarrow t$ where the condition C can be expressed as, $C = \bigwedge_{i \in [n]} c_i$, i.e. a condition built by a conjunction of the c_i where $[n] = \{1, \ldots, n\}$.
Each c_i is a comparison that uses an operator from the set {<, <=, =, >=, >} to compare operands that are either variables from the set X or any number belonging to the set of natural numbers.

c_i and C are used to define a set of states. A comparison c_i defines all the states for which the comparison is satisfied and the condition C defines the states for which all the comparisons in C are satisfied. Therefore, the comparison c_i and the condition C are identified with their sets of states and the same symbols are used for the expressions as well as for their sets of states.

Therefore $C = \bigcap_{i \in [n]} c_i$. C is satisfied by state s if and only if $s \in C$. If $n = 0$ we have $C = S$.

Every t is a sequence of simple assignments that can be expressed as $x := e$, where x is a variable and e is an expression consisting of operands and operators. The operands are represented by variables and numbers while the operators are represented by addition and subtraction. The execution of a sequence t implements an unconditional transformation of states $t: S \rightarrow S$.

During the occurrence of an event all of its accompanying statements are analyzed and checked. If for any statement $C \Rightarrow t$ the current state s satisfies the condition C, i.e. $s \in C$, in that case the transformation t is executed. It is required that for any two statements $C \Rightarrow t$ and $C' \Rightarrow t'$ located in the same event, the set of variables are changed in t and t' must be disjoint if $C \cap C' \neq \emptyset$. As a result, the order of these statements is not important.

The control functions can also be expressed as a sequence of statements of the form $C \Rightarrow \alpha$. In this case, C is a condition and its definition is similar to that of an event, α is a

sequence of program actions. Given a current state s, the function $\delta(s)$ is defined as the concatenation of all sequences α for which the corresponding condition C is satisfied.

This implies that the effect of the control functions may depend on the order of the statements $C \Rightarrow \alpha$. However, it is permitted to rearrange these statements. Therefore, there is no assurance that the control function works according to the given order.

4.3 Optimization

4.3.1 Simplifications

The simplification of each condition $C = \Lambda_{i\in[n]} c_i$ can be done in three different ways:

- **Elimination of tautologies:**

  If for some $j \in [n] : c_j = S$, then $C = \Lambda_{i\in[n]\text{-}\{j\}} c_i$

  If n = 0, i.e. $[n] = \emptyset$, then we get $S \Rightarrow t$.

- **Elimination of contradictions:**

  If $C = \emptyset$, then a statement $C \Rightarrow t$ can be removed from the event.

- **Elimination of implications:**

  If for some $j, k \in [n] : j \neq k$ and $c_k \subseteq c_j$ then $C = \Lambda_{i\in[n]-\{j\}} c_i$

For any comparison c, it holds that c = S iff c has the form ○<=○ , ○=○ or ○>=○ for some operand ○. and, it holds that $c = \emptyset$ iff c has the form ○<○ or ○>○ for some operand ○.

4.3.2 Check of Contradictions

The tableau-method of propositional calculus is adopted as the method for checking whether C = ∅:

In a step by step manner, condition C is transformed into a set *C* of simple conditions in which all of them includes only comparisons of the form $o < o'$. This transformation is carried out in such a manner that C is satisfyable iff one of the conditions from *C* can be satisfied.

Step 1:
In C, every comparison of the form $o > o'$ is replaced by $o' < o$ and every comparison of the form $o \geq o'$ by $o' \leq o$.

Step 2:
Begin with *C* := {C}.
While there is in *C* a C which contains a comparison of form $o \leq o'$: divide C into two sets C' and C'', where the comparison $o \leq o'$ in C' is replaced by $o=o'$ but in C'' it replaced by $o < o'$.

Step 3:
While in *C* there is a set C which contains a comparison of form $o=o'$ drop it from C and perform the following:

- If o is a variable, for example x, then replace any occurrence of x in C by o',
- Otherwise, if o' is a variable, say x, then replace any occurrence of x in C by o,
- Otherwise, if both, o as well as o', are different natural numbers then remove the whole set of C from *C*.

Step 4:
While *C* is not empty do:

Take any C ∈ *C*. Let O be the set of all operands of comparisons in C and *IN* the set of natural numbers. Now, try to build a function s: O → *IN* in the following way:

- If o is a number then set s(o) := o
- Otherwise, if o is a variable then set

$$s(o) = \begin{cases} 0 & \text{if there is no comparison o}' < \text{o in C} \\ \max\{s(o') \mid o' < o \in C\}+1 & \text{otherwise} \end{cases}$$

If such a function s does not exist or for the existing s there is a comparison $o' < o$ in C such that s(o) ≤ s(o') remove C from *C*.

4.4 Reordering

A specification as explained in section 4.2 regarding events as well as control functions consists of a sequence

$$C_1 \Rightarrow x_1; \ldots C_n \Rightarrow x_n;$$

Where C_i is referred to as conditions (conjunctions of comparisons) and the x_i are either sequences of simple assignments (in the case of events) or sequences of actions (in the case of control functions).

A text of a programming language, such as Java, C, C++ or C#, can be derived from such a specification by simply transforming it into a sequence of if-statements

$$\text{if } (C_1)\ x_1; \ldots \text{if}(C_n)\ x_n;$$

In some instances it is advantageous to rearrange this sequence and to create nested if-statements. This may prolong the length of the text but improve the efficiency of execution [50]. This is done by means of the following steps:

Step 1:

We reorder the sequence $C_1 \Rightarrow x_1; \ldots C_n \Rightarrow x_n$; such that if $C_i \subseteq C_j$ then $j \leq i$.

Step 2:

We build the following sequence of nested if-statements

if (C_1) { x_1; if(C_2) x_2; . . . if (C_n) x_n; return; }

if (C_2) {x_2; if(C_3) x_2; . . . if (C_n) x_n; return; }

. . .

if (C_{n-1}) {x_{n-1}; if(C_n) x_n; return; }

if (C_n) x_n;

Step 3:

We remove each if-statement if (C_i) {. . .} if there exists a j where $C_i \subseteq C_j$.

Step 4:

In each remaining if-statement

if (C_i) {x_i; if(C_{i+1}) x_{i+1}; . . . if (C_n) x_n; return; }

We replace C_{i+k} by C'_{i+k} if C_{i+k} have the form $C_i \wedge C'_{i+k}$.

Step 5:

In each remaining if-statement

if (C_i) {x_i; if(C_{i+1}) x_{i+1}; . . . if (C_n) x_n; return; }

We remove the contained if-statement if (C_{i+k}) x_k; if $C_i \cap C_{i+k} = \emptyset$.

5

The Event-Programming-Framework *epro*

In this chapter, the implementation of the abstract model as discussed in chapter 4 is presented. Microsoft visual studio 5, C# .Net was used in the implementation of the abstract model. A detailed description about the features of the special program and how it was implemented using C# .NET is introduced. After that, an application program (Alarm-Clock Application) is given as an example in order to illustrate how the system works.

5.1 An Overview

As described in the previous chapters, the event handling will be separated from the actions of the program. Therefore, each application is divided into two parts. The first part is the *specification*, which in turn contains three parts, namely, *the state space, the event handling,* and *the control functions*. The event-programming-framework epro contains a special editor to write and manage specifications. The second part is the *hand-built program* (hbp), which contains the classes that are needed to carry out the actions. This part must be written by the developer of the application in the classical way using C#. By the event-programming-framework epro, the specification is transformed into C# classes and then glued together automatically with the *hand-built program*. In this way a C#-project is built from which an executable can be made.

The specification-editor stores all data inputs from the three parts of the specification in an XML file. A few descriptions about the XML file will be introduced in the next subsection.

In the *state space specification*, the ability is provided to enter the state-variables that are used in the system along with their types. The editor has some procedures of verification and optimization for these variables in order to avoid redundancy and inconsistency. The same procedures are applied in the *event handling specification*, for example, to handle the events that are designed to be used in the system. Each event contains conditions and state transformations. In the *control functions specification*, the editor enables one to add calls of actions. Each action-call may depend on conditions.

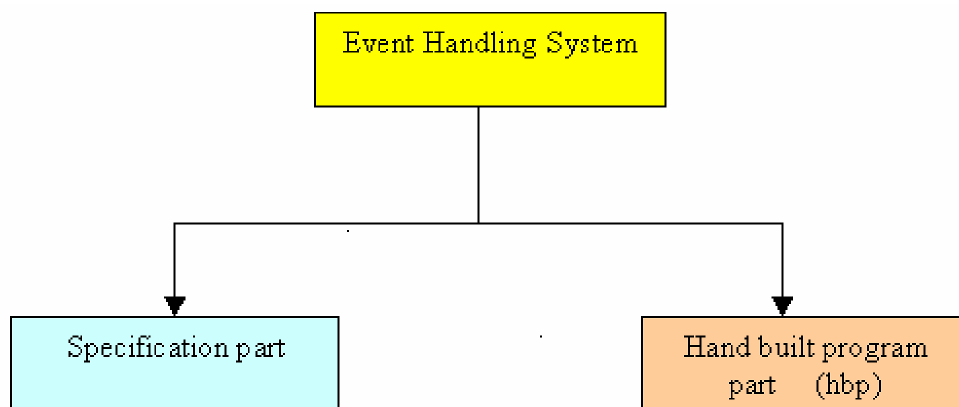

Figure 5.1: Event handling system

The specification part is transformed to the C# language as classes with corresponding fields, properties and methods. The three parts of the specification are transformed into the following classes:

state space specification	into class	`State`
event handling	into class	`Events_Handling`
control functions	into class	`Control`.

Moreover, in every application there must be a class of a certain name, say `Form`, introduced by the developer of the system or by any designer (e.g. Microsoft Visual Studio), where the GUI and the basic Components are defined. In this class the definition

of a general event handler must be inserted by hand. In a special file, called `AppFunctions.cs`, templates for all methods where corresponding actions are called in the *control functions specification* are inserted automatically. Details are explained in section 5.3.

XML File:

As previously mentioned, all the specification data of the event handling system are stored in a specified XML file. Microsoft.Net platform provides us a simple way to create XML files and it is easy to processing these files, i.e. it is easy to save, load and search the data through such files.

The XML file is a hierarchical data structure. XML tells us what kind of data we have, not how to display it. Because the markup tags identify the information and break up the data into parts [72]. So, a search program can look for state-variables, events and their contents or control functions and their characteristics. In short, because the different parts of the information have been identified, they can be used in different ways by different applications. After the declaration, every XML file defines exactly one element, known as the *root element*. Any other elements in the file are contained within that element, XML element names are case-sensitive and the end-tag must exactly match the start-tag. So, our XML file has a root element, which represents the specification and has the form `<specification> . . . </specification>`.

Since XML allows hierarchically structured data, an element can contain other elements. Therefore, the root element contains three main elements namely, statespace, events, and controls which corresponds to the three main parts of the specification part. In addition, each element contains all the related elements of its components in separate elements, for example, the statespace contains the state-variables and their characteristics such as "name", "type", "initial value", etc. and they are stored in a `statespace` element. Each state-variable is represented in a separate `tag called <variable>` and this tag

contains some attributes with their values like name="mode" and lower="0" and so on for example:

```
<variable no="1" typel="[0...3]" name="mode" initl="=0" type="3"
lower="0" upper="3" iinit="0" finit="0" binit="false" />
```

The element `events` contain the data related to the event specification where each single event is described by an `event` element. The element `event` contains all the corresponding data for this event such as "ident", "actions", "conditions", and the corresponding state transformation as elements. The elements `condition` and `trans` contain their attributes, for example:

```
<event ident="on_off">
        <action>
          <conditions>
                <condition text="mode == 1" lvar="1" oper="3" rvar="0" ival="1"
                                fval="0" />
          </conditions>
          <transformations>
                 <trans text="! alarm1on" lvar="4" oper="0" rvar="0" ival="1"
                              fval="0" bval="False" />
          </transformations>
        </action>
</event>
```

The last important element is a control which is represented by the `tag <controls>`. This element contains all the information about the *control functions specification*. It contains the element represented by the tag `<action>`. Each action contains the elements `conditions` and `calls`. The element `conditions` are represented in a separate tag `<conditions>` which also contain the element condition in which the attributes of the conditions are located. The element `calls` are also represented in a separate tag `<calls>` which contain the element call in which the attributes of the calls can be found as illustrated follows:

```
<controls>
 <action>
```

```
    <conditions>
       <condition text="updown was changed &" lvar="8" oper="7" rvar="0"
                        ival="0"   fval="0" />
       <condition text="mode != 3" lvar="1" oper="6" rvar="0" ival="0"
                        fval="0" />
     </conditions>
     <calls>
        <call text="updown_alarm(mode,fastslow,updown)" />
     </calls>
   </action>
 </controls>
```

Due to the bigger size of the “Output.xml” file of the Alarm-Clock application example, only a part of this file will be shown. The complete file can be found in the Appendix B.

```
<? xml version="1.0" encoding="utf-8"?>

<specification>
  <statespace>
    <variable no="1" typel="[0 ... 3]" name="mode" initl="= 0"
              type="3"      lower="0" upper="3" iinit="0"
              finit="0"     binit="false" />
    <variable no="2" typel="[0 ... 120]" name="rem_minutes"
              initl="= 0"   type="3" lower="0" upper="120"
              iinit="0" finit="0" binit="false"  />
    <variable no="3" typel="[0 ... 60]" name="rem_seconds"
              initl="= 0"   type="3" lower="0" upper="60"
              iinit="0" finit="0" binit="false"  />
  </statespace>
  <events>
    <event ident="mode">
      <action>
        <conditions />
        <transformations>
          <trans text="mode += 1" lvar="1" oper="2" rvar="0"
                 ival="1"  fval="0" bval="False" />
        </transformations>
      </action>
    </event>
    <event ident="on_off">
      <action>
        <conditions>
        <condition text="mode == 1" lvar="1" oper="3" rvar="0"
                       ival="1" fval="0" />
        </conditions>
        <transformations>
```

```
          <trans text=" ! alarm1on" lvar="4" oper="0" rvar="0"
                 ival="1" fval="0" bval="False" />
        </transformations>
      </action>
    </event>
  </events>
  <controls>
    <action>
      <conditions>
        <condition text="updown was changed   &amp;" lvar="8"
                   oper="7" rvar="0" ival="0" fval="0" />
        <condition text="mode != 3" lvar="1" oper="6" rvar="0"
                   ival="0" fval="0" />
      </conditions>
      <calls>
        <call text="updown_alarm(mode,fastslow,updown)" />
      </calls>
    </action>
  </controls>
</specification>
```

5.2 Specification

Specification of the *event handling system* is described in a special way. Figure 5.2 shows the description of the specification part of the *event handling system*.

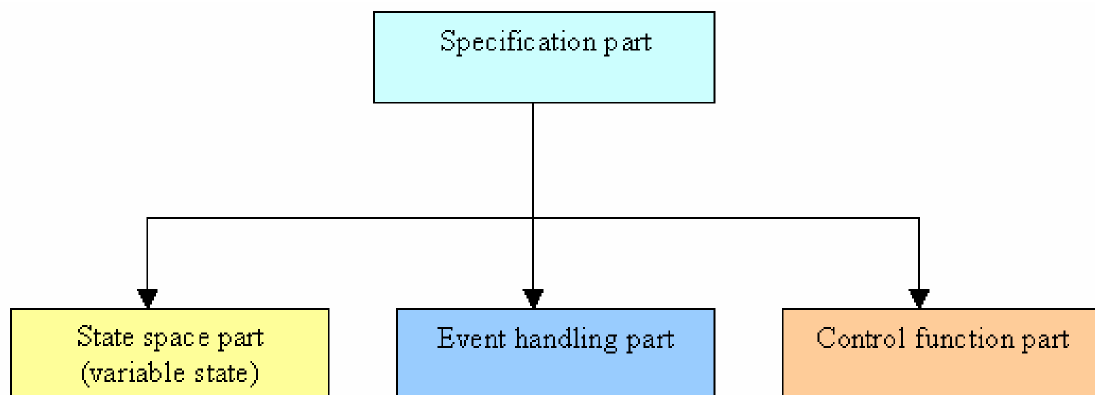

Figure 5.2: Specification part

The specification-editor provides for each of the three specification parts a special window.

5.2.1 State Space Specification

The *state space specification* contains all the state-variables, their types and initial values that are used in *event handling* and *control functions specification parts*. The following types are allowed:

- integer,
- boolean,
- range of integers with lower and upper bounds and
- float.

The *state space* specification is represented and can be managed by the form called `State Space`. Figure 5.3 shows an example of the form *StateSpace*.

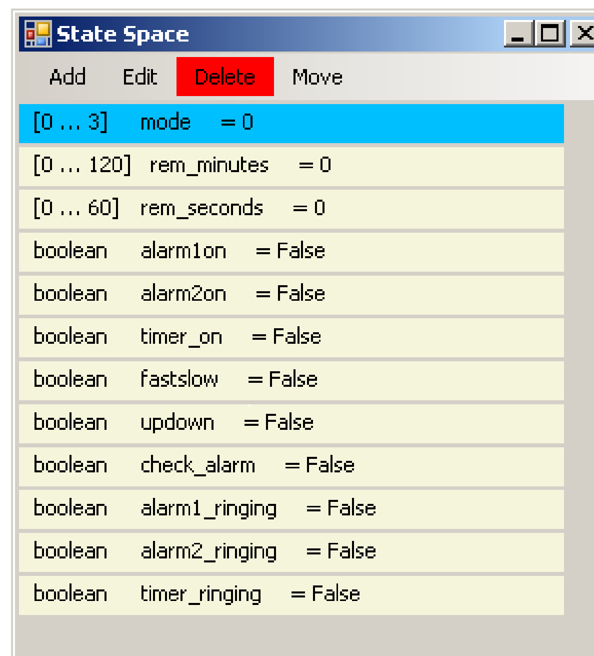

Figure 5.3: An example of the state space form

Here, the first three state-variables are of type `range`. The other state-variables are of type `boolean`.

There are some available functions such as *Add, Edit, Delete,* and *Move* to manage the state-variables. The *Edit* function is used to edit the state-variables and their characteristics. Since the state-variables are used by the *event handling specification* and *the control functions specification,* care must be taken when editing and updating the state-variables in the *state space specification*. Editing and updating a state-variable in the *state space specification* will automatically cause to edit and update this state-variable in the *event handling specification* and the *control functions specification*.

For example, in an attempt to add a new or to edit an existing state-variable, the system does not permit the generation of a duplicate state-variable name in the list of state-variables in the *state space specification*. An error message appears in the application indicating that there is a double state-variable name in the list of the state-variables. The same applies to the state-variables of `range` type. When an attempt is made to change or edit the value of a state-variable of `range` type with a single value, which is not in the range between the lower and upper bound, an error message appears indicating that the new value is out of the range.

The other situation where an error message also occurs is when an attempt is made to edit or update the value of a state-variable of a `boolean` type with an incorrect value (i.e. the value is not `true` or `false`). The error message that appears is "incorrect boolean".

An error message will appear on the system whenever an attempt is made to try to add or edit a value of a state-variable of `integer` or `float` data type, when, e.g., a character or another symbol is entered by mistake. The error message that appears is "`incorrect integer number`" for the case of `integer` type and "incorrect floating number" for the case of `float` type.

5.2.2 Event Handling Specification

The *event handling specification* encapsulates all the characteristics of all the events in the system. Each event includes one or more action panels. Each action panel consists of a condition and state transformations. Each condition can be atomic, i.e. a `boolean` variable or a comparison between variables or constants with the operators from {<, <=, ==, !=, >=, >}, or a conjunction of atomic conditions. A special condition is `always`, which is always `true`. Disjunctions are not allowed. They must be written as several conditions with the same state-transformations. When an event is raised then all its conditions are checked and, after that, all state-transformations belonging to fulfilled conditions are carried out. Figure 5.4 shows an example of the *event handling specification*.

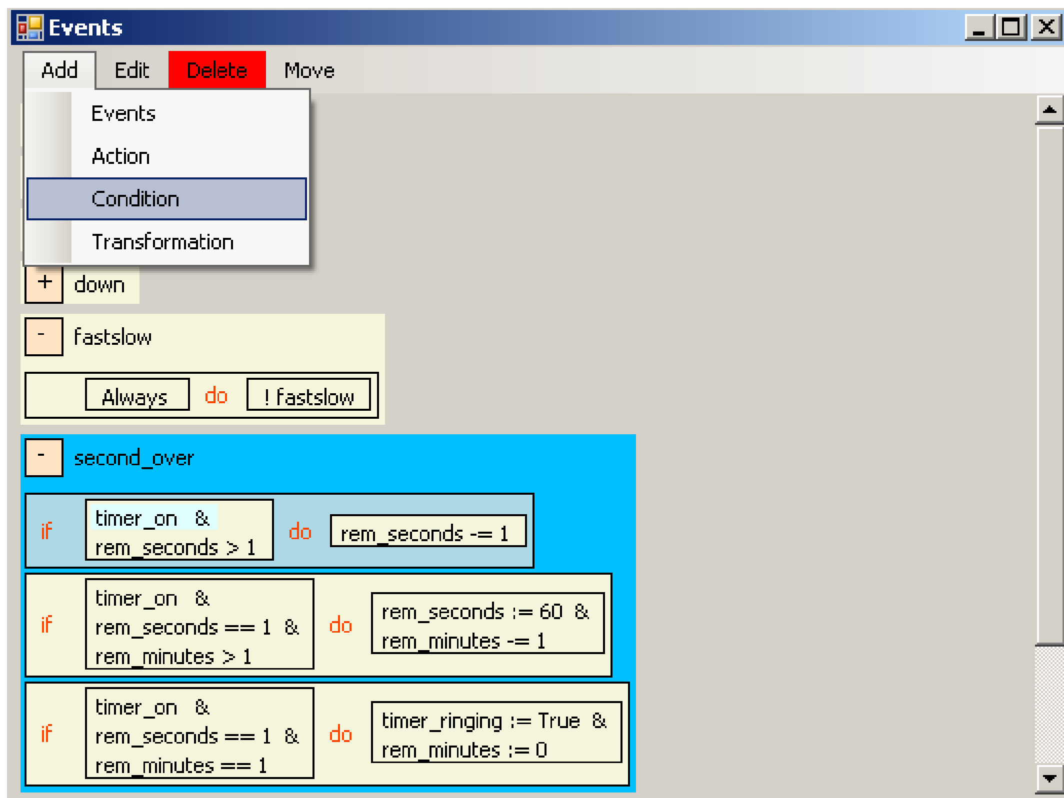

Figure 5.4: The event handling form

Each event accepts only the state-variables that are defined and introduced in the *state space specification*. To introduce a new condition, the condition form is used (cf. Figure 5.5).

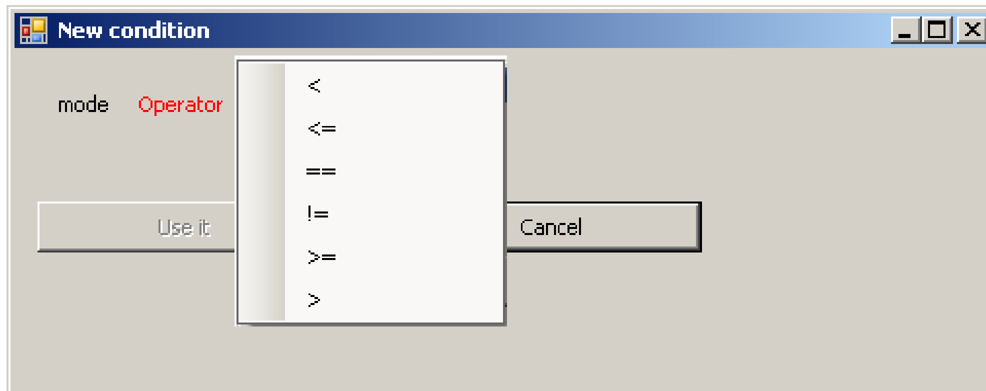

Figure 5.5: The example of condition form

Some optimization, simplification and verification steps are used and applied in order to check the conditions on the *event handling* and *control functions specification,* namely:

- **Elimination of tautologies:** if a comparison is always fulfilled, like

  ```
  A <= A
  ```

  then it is removed from the condition. In case, a condition becomes empty then it is replaced by `always`.

- **Elimination of contradictions:** A contradiction happens when a condition or a group of conditions are contradicting each other. for example:

  ```
  A < B  & B < A or
  A > 5  & A < 3 or
  A != A
  ```

In the above examples the editor will refuse the condition `B < A`, because it contradicts with the previous condition `A < B`. Similarly, `A < 3` and `A != A` are indicated as errors.

- **Elimination of implications:** This is applied if the conditions can be simplified, as in the following example:

```
A < B  & B < C  & A < C
```

The above example shows that `A is less than B and B is less than C`, so, it is clear that `A is less than C`. Hence, in this case, the editor refuses the condition `A < C` because it is an *implication* of the first both.

The main methods to deal with these optimizations and verifications are to build *transitive closures*. However, in the case of integer variables or ranged variables additional considerations are necessary in order to guarantee that valid values are possible. For instance, if we have a ranged variable `A` of type `[0..4]` and an integer variable `B` then the condition

```
A < B & B < 1
```

is a contradiction.

State transformations are defined as a sequence of simple assignments of the form

```
x:=y, x:=n, x+=y, x+=n, x-=y, x-=n
```

where `x and y` are state-variables and `n` is a number. Additionally, for `boolean` variables exists the negation. If a state transformation causes that a value of a variable is out of the range of the variable then it is automatically set to the lower or upper bound. However for variables of type `range` a value greater than the upper bound is set to the lower bound and vice versa.

As in the *state space specification*, by `add`, `edit`, `delete` and `move` the *event handling specification* and also the subparts like `action panels`, `conditions` and `state transformations` can be modified. The editor cares always that the consistency of the specification is guaranteed.

5.2.3 Control Functions Specification

The *control functions specification* is described by a set of conditional statements of the form c ⇒ α, where c is a condition as in an *event handling specification* and α is a sequence of program action calls [50].

The *control functions specification* is represented by a form called *ControlFunctions*. Every conditional statement is managed in a so called *action panel.* Also here, the functions `add`, `edit`, `delete` and `move` are available. An example for this is shown in Figure 5.6.

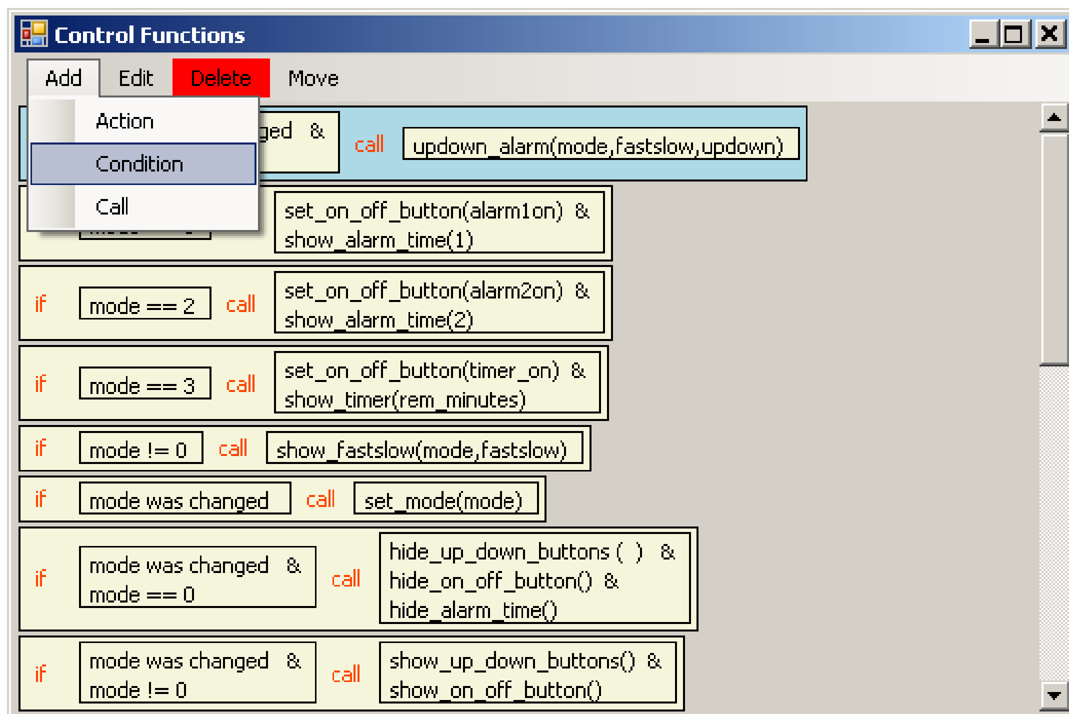

Figure 5.6: An example of control functions specification

5.3 Transformation into C#

As previously mentioned, in this section, a detailed description of the transformation process of the *event handling system* components is discussed. Section 5.3.1 presents how the *state space specification* is transformed into C# classes. It describes how to transform the state-variables into the `State` class and introduces an example concerning these state-variables and their methods and properties. Section 5.3.2 states how the *event handling specification* is transformed into `Events_Handling` class and also given, is an example of some events being used in the system. Section 5.3.3 depicts how the *control functions specification* is transformed into C# classes. It also describes two C# classes generated automatically by the system: the `Control` class and the `AppFunctions` class.

5.3.1 Transforming State Space

As mentioned before, the application program will automatically transform the *state space specification* into a C# class called `State.` The application program defines special types of state-variables in the *state space specification*. Therefore the application will transform each state-variable in the *state space specification* into the corresponding variable in the C# language. These transformations of the state-variables in the *state space specification* are implemented by means of properties. Properties are like normal variables but with more powerful and flexible characteristics. So each state-variable is converted into a property with the same name. A `get` property accessor is used to return the property value, and a `set` accessor is used to assign a new value. These accessors can have different access levels. The `value` keyword is used to define the value being assigned by the `set` indexer [30]. Moreover, each state-variable is transformed into two component variables depending on its type. One has the same data type with suffix "`_v`" and it also has an initial value. The other one has the suffix "`_changed`" which is of `boolean` type. For example, for a state-variable, say "`x`", there exists a corresponding `boolean` variable "`x_changed`", which is `true` if and only if the contents of the

variable "x" has been changed by the last handled event. The access to all of the variables is controlled by properties. The set-part of the property of variable "x" always sets the value of "x_changed" variable to true. For each available data type of state-variables in the *event driven system*, an example of a property definition will be introduced in the next subsection.

Each variable in the State class takes an initial value which is determined in the *state space specification* and its corresponding boolean variable takes an initial value equal to false.

It is known from the *state space specification* (section 5.2.1) that the special types of state-variables, which are defined in the *state space specification*, are from the following type:

- **Integer type:** In this data type, every state-variable will be transformed to the equivalent C# data type which is int. In this case the variable takes the values of type int and especially the natural numbers ***IN*** i.e. from 0 to some n.

  Example: `integer mode = 0`

  This will be transformed to the following equivalent variables and properties as illustrated below:

```
private static uint mode_v =0 ;
public static bool mode_changed  = false ;
public static uint mode
{
     get { return mode_v ; }
     set
     {
          mode_v = value ;
          mode_changed = true;
```

```
     }
}
```

- **Float type:** For this data type, the state-variable is transformed to the equivalent C# `float`, and the variables take the value of floating type

  Example: `float radius = 2.4`

  This is will be transformed to the following equivalent variables and properties as illustrated below:

```
private static float radius_v =2.4;
public static bool radius_changed  = false ;
public static float radius
{
     get { return radius_v ; }
     set
     {
          radius_v = value ;
          radius_changed = true;
     }
}
```

- **Boolean type:** In this data type, every state-variable which has a special data `boolean` type will be transformed to the equivalent C# boolean type which is `bool` data type. In this case, the variables take the value of `boolean` type so the value of variables will be `true` or `false`.

  Example: `boolean  alarm1on = false`

  This is will be transformed to the following equivalent variables and properties as illustrated below:

```
private static bool alarm1on_v =false ;
public static bool alarm1on_changed  = false ;
public static bool alarm1on
{
   get { return alarm1on_v ; }
   set
   {
        alarm1on_v = value ;
        alarm1on_changed = true;
   }
}
```

- **Range [lower .. upper] type**: In this case, every state-variable of this data `range` type must be transformed to the equivalent C# data type `int` and the variable will take its value of type `int` in the range of its lower value to its upper value.

  Example: `[0 ... 120]  rem_minutes = 0`

  The above example is transformed into C# as follows:

```
//  the type of this variable is range[lower...upper]
   private static uint rem_minutes_v =0 ;
   public static bool rem_minutes_changed  = false ;
   public static uint rem_minutes
   {
       get { return rem_minutes_v ; }
       set
       {
          rem_minutes_v = value > 120 ? 0 : value ;
          rem_minutes_changed = true;
       }
   }
```

At the end of a `State` class, there exist a function `reset_change()`. This function is invoked in order to set the values of all the "`x_changed`" variables to `false`.

The following is an example of a small part of the `C# State` class of the `Alarm-clock` application: this class is produced from the list of state-variables that are defined in the *state space specification* (cf. Figure 5.3) with their types, and it declares a function `reset_change()` at the end.

```
namespace EV_Edit
{

  static class State
 {

   //  the type of this variable is range[lower...upper]

    private static uint mode_v =0 ;
    public static bool mode_changed  = false ;
    public static uint mode
    {
        get { return mode_v ; }
        set
        {
           mode_v = value > 3 ? 0 : value ;
           mode_changed = true;
         }
    }
    private static bool alarm1on_v =false ;
    public static bool alarm1on_changed  = false ;
    public static bool alarm1on
    {
        get { return alarm1on_v ; }
        set
        {
           alarm1on_v = value ;
           alarm1on_changed = true;
        }
    }
    public static void reset_change()
    {
         mode_changed = false ;
         rem_minutes_changed = false ;
         rem_seconds_changed = false ;
         alarm1on_changed = false ;
         alarm2on_changed = false ;
         timer_on_changed = false ;
```

```
            fastslow_changed = false ;
            updown_changed = false ;
            check_alarm_changed = false ;
            alarm1_ringing_changed = false ;
            alarm2_ringing_changed = false ;
            timer_ringing_changed = false ;
        }
    }
}
```

5.3.2 Transforming Event Handling

It is known from section 5.2.2 that from the *event handling specification*, the application program automatically generates a C# class called `Events_Handling`. This class implements all the events that are defined in the *event handling specification*. Each event is implemented as a function, which includes all the contents of the event such as a condition or several conditions with corresponding state transformations.

The generated program determines the number of events and defines a queue in order to store the events in this queue. The queue has a first value and a last value. The generated program adds the first event to the queue and checks whether the queue is full or not. The generated program must be sure that the event which arises first must be handled first.

The generated program defines for each event an array for its conditions. This array is necessary to be sure that the values of the state-variables did not change. Therefore, the array contains the result of checking the conditions before executing the state transformations to avoid changes of the values of state-variables. The following example shows an array in the function `minute_over()`, which contains two indexes of conditions `cond[0]` and `cond[1]` which are used by the if-statements.

```
    public static void minute_over()
    {
       cond[0] = State.alarm1on && !State.alarm1_ringing;
       cond[1] = State.alarm2on && !State.alarm2_ringing;

   //  if (State.alarm1on && !State.alarm1_ringing)
```

```
        if (cond[0])
        {
                State.check_alarm = true ;
        }
    //  if (State.alarm2on && !State.alarm2_ringing)
        if (cond[1])
        {
                State.check_alarm = true ;
        }
     }
```

The state-conditions for each event are automatically transformed into C#-if statements. The first part of an if-statement is an element of the above mentioned array which already contains the result of the condition and the second part of the if-statement is the corresponding state transformation.

The generated program defines a function `fetch_event()` in order to select (fetch) an event from the event queue. It also defines a method `event_cycle()`. This method is used to check whether the event in the queue have the possibility of occurring. Once the event is invoked or fired, the control functions is called and the method `reset_change()` that is defined in the `State` class is also invoked.

In order to explain the relationship between these functions, this is illustrated with an example of how the event called *start* is implemented.

When the button such as “`Mode`“ in the main form of the Alarm-Clock application is clicked, the event is delegated to the event handler as shown bellow:

```
on_event += new
              eventhandler(Events_Handling.event_handling);
```

In this case, the function `event_handling()` which is located in the `Events_Handling` class will be invoked with the name of the event , in our case that referring to *start* event. In this function the event *start* will be added by invoking the

function `add_event(e)`. In the `add_event(e)` function, the event *start* will be added to the queue `evs_queue` and we will check if the index last of the queue `evs_queue` has reach its last position i.e 10 for example. After adding the event to the queue `evs_queue`, we will again return to the `event_handling()` function where the `boolean` variable `event_cycle_is_working` is found. If the value of this variable is `false` then we will invoke the `event_cycle()` function. In the `event_cycle()` function we will change the value of the `boolean` variable `event_cycle_is_working` to `true` and check if the value of the first and last index of the queue is not equal. If this is the case, then we will invoke the `fetch_event()` function. In the `fetch_event()` function we will add 1 to the first index of the queue and select its corresponding event. If the last index is reached, this implies that the first index equal to zero. We now return the selected event back to the event_cycle() function. We will use `switch` and `case` keywords to check for the selected event and its corresponding function. We know that each event in the *event handling specification* is transformed to a function with the same name , its condition and corresponding state transformation as previously mentioned. So, in our case, we will select the `start()` function which corresponds to the event *start*. Therefore, when we find the `start()` function and called it, this causes it to check all the conditions which exist in the `start()` function and stored in an array as we described before. After that the elements of the conditions are checked and if it is satisfied, then the corresponding state transformations are executed. As we introduced before, for every cycle of the `event_cycle()`function the control function is invoked. Since, we already have an instance of the `Control` class, each condition in the `Control` class is checked and if it is satisfied then the corresponding calls will be invoked and executed in the `AppFunction` class. After that, the method `reset_changed()` which exist in the `State` class will be called in order to set all the `boolean` variables with suffix "`_changed`" to `false`. Finally the `boolean` variable `event_cycle_is_working` also sets to `false`.

The following example is a part of the `Events_Handling` class of the `Alarm-clock` application. It contains information about the declaration of all events that are defined in the *event handling specification*. It also contains information about the transformation of the conditions and its corresponding state transformations into C# declarations. It also contains other information about the methods that handle the events for example `add_event(), fetch_event(), event_cycle()` and so on.

```
namespace EV_Edit
{

 enum EVS { mode , on_off , up , down , fastslow ,
           second_over , minute_over , alarm1 , alarm2 };

  static class Events_Handling
  {
     private static bool[] cond = new bool[100];
     private static EVS[] evs_queue = new EVS[100];
     private static bool event_cycle_is_working = false;
     private static uint first =0, last = 0;

     private static void add_event(EVS e)
     {
        evs_queue[last++] = e;
        if (last == 10)
          last = 0;
     }

     private static EVS fetch_event()
     {
        EVS e = evs_queue[first++];
        if (first == 10)
           first=0;
        return e ;
     }

     public static void event_handling(EVS e)
     {
        add_event(e);
        if (!event_cycle_is_working)
           event_cycle();
     }

     private static void event_cycle()
```

```
 {
    EVS e ;
    event_cycle_is_working = true ;
    while (first != last)
    {
       e= fetch_event();
       switch (e)
       {
          case EVS.mode : mode();
               break;
          case EVS.on_off : on_off();
               break;
          case EVS.up : up();
               break;
          case EVS.down : down();
               break;
       }
       ControlActions.control();
       State.reset_change();
     }
     event_cycle_is_working = false;
 }

public static void mode()
{
            State.mode += 1 ;
}

public static void on_off()
{
   cond[0] = State.mode == 1;
   cond[1] = State.mode == 1 && State.alarm1on &&
                             State.alarm1_ringing;
   cond[2] = State.mode == 2;

   if (cond[0])
   {
            State.alarm1on = !State.alarm1on ;
   }
   if (cond[1])
   {
            State.alarm1_ringing = false ;
   }
   if (cond[2])
   {
            State.alarm2on = !State.alarm2on ;
```

```
    }
  }
```

5.3.3 Transforming Control Functions

As the other two previous parts of the specification, the *control functions specification* is also transformed into the C# language. This part is implemented by the `Control` class. This class includes the implementation of conditions and corresponding methods. The implementation of methods is automatically generated in another class called `AppFunctions`.

The conditions in the *control functions specification* are automatically generated and transformed into C# if-statements. The first part of an if-statement consists of the conditions to be checked and the second part of the if-statement contains the invocation of its corresponding methods to be invoked and executed when the conditions are checked and satisfied.

In the transformation of the *control functions specification*, we do not need to store the conditions in a `boolean` array before checking such as we previously described in the transformation of the *event handling specification* because we do not have any state transformation which may change values. The conditions have only corresponding methods which do not have any values to change.

The following example is a part of the `Control` class of the `Alarm-clock` application. It contains the conditions and their corresponding methods invocations in the *control functions specification*. It also contains an instance of the `AppFunctions` class, which defines the method declarations (i.e. the bodies of the methods).

```
namespace EV_Edit
{
  static class ControlActions
  {
    public static Form2 af;
```

```
    public static void control()
    {
       if (State.updown_changed && State.mode != 3)
       {
          af.updown_alarm(State.mode,State.fastslow,
                          State.updown) ;
       }

       if (State.mode == 1)
       {
             af.set_on_off_button(State.alarm1on) ;
             af.show_alarm_time(1) ;
       }

       if (State.mode == 2)
       {
             af.set_on_off_button(State.alarm2on) ;
             af.show_alarm_time(2) ;
       }
    }
 }
}
```

The generated program automatically generates an `AppFunctions` class, which contains the bodies of methods (only the templates of the methods) without contents of its bodies. It produces empty bodies of methods that contain only the parameters. Figure 5.7 shows an example of the empty template of bodies of methods.

The user or programmer should write the contents of methods. The user should not have the ability to change the parameters of the methods. After the control functions has called, the `reset_change()` function within the `State` class must be called.

The following example is a part of the `AppFunctions` class, which contains the methods declaration.

```
namespace EV_Edit
{
     public partial class Form2
     {
```

```
        // Application specific functions, called by
        //    class Control
        // the templates (without contents of bodies) are
        //  automatically generated
        private static uint[] alarm_time = { 0, 0 };
        // manages and shows the current time
        // does not use the specification
        public void updown_alarm(uint mode, bool fs,
                                 bool ud)
        {
             int i = (int)alarm_time[mode - 1];
             int h = i / 60;
             int m = i % 60;
             if (fs)
             {
                  h = h + (ud ? 1 : -1);
                  h = (h == 24) ? 0 : (h < 0) ? 23 : h;
             }
             else
             {
                  m = m + (ud ? 1 : -1);
                  m = (m == 60) ? 0 : (m < 0) ? 59 : m;
             }

             alarm_time[mode - 1] = (uint)(h * 60 + m);

        }
        public void set_on_off_button(bool b)
        {

             on_off_button.Text = b ? "on" : "off";
        }
        public void show_alarm_time(uint mode)
        {
             uint h = alarm_time[--mode] / 60,
                  min = alarm_time[mode] % 60;
             output.Text = h.ToString() + " : " +
                           min.ToString();
             output.Visible = true;
        }
    }
}
```

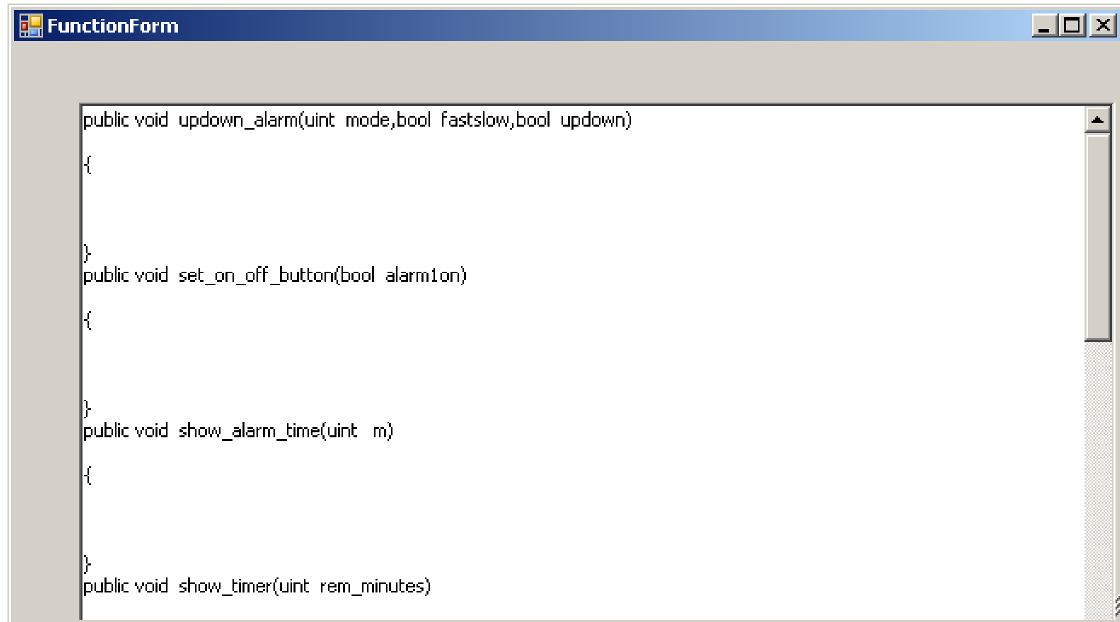

Figure 5.7: An example of empty templates of bodies of methods

5.4 Example (Alarm-Clock Application)

5.4.1 Application Description

The Alarm-Clock example is a small application. With this example, it is demonstrated how our approach works.

Usually, alarm clocks are separated devices where a small processor is included. The functions of an alarm clock are controlled by different switches and buttons. The effects of the switches and buttons can change dependent on the current situation, the alarm clock is in. Here, we simulate this by a Windows form. In Figure 5.8 a snapshot of the start of the application is given. The current time is always presented. The button Mode allows to set different modes.

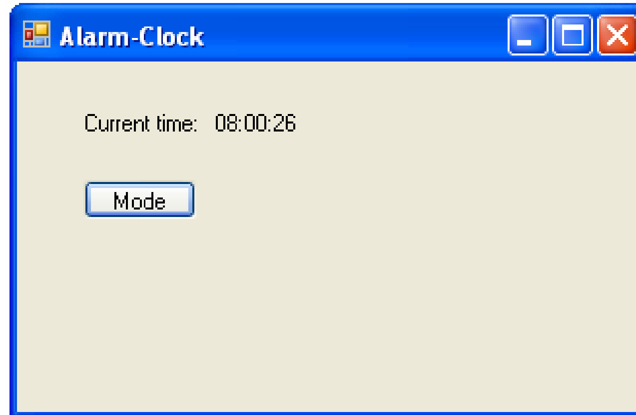

Figure 5.8: Start of the Alarm-clock Application

Pressing it, the mode changes to Alarm1 where a first alarm time can be set (Figures 5.9).

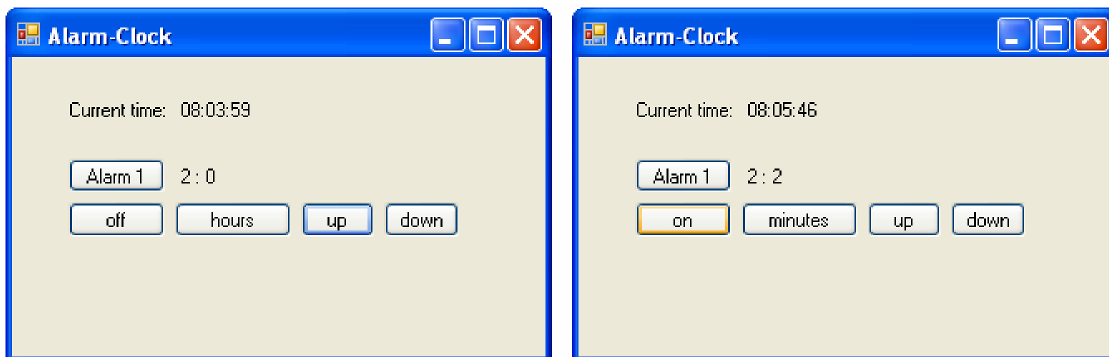

Figure 5.9 a: First alarm time (hours) **Figure 5.9 b: Second alarm time (minutes)**

By the buttons “up” and “down” the hours (a) or minutes (b) of the alarm time are set. The kind of steps are choosen by pressing the button “hours” or “minutes”. The same holds for the second alarm time. There is also a Timer (Figure 5.10), which can be set in steps of 1 minute or 10 minutes.

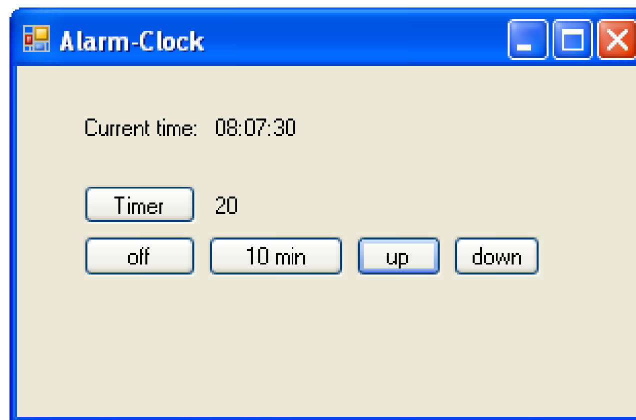

Figure 5.10: Timer alarm mode

When one of the Alarm times is reached or the timer is finished (supposed it is on) then the alarm clock rings. In our application this is simulated by showing the text “it is ringing” (Figure 5.11).

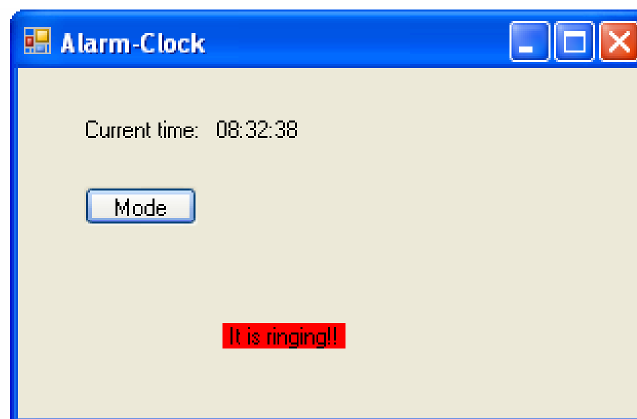

Figure 5.11: Alarm-clock ringing

In the application of Alarm-Clock example, there are some classes which are automatically generated by the application program which are releated to the specification such as `State`, `Events_handling`, `Controls` and `AppFunctions` classes.

But the other classes such as `Form2` class, is generated by the developer of the system or by any designer (e.g. Microsoft Visual Studio), where the GUI and the basic components are defined. In this class, the definition of a general event handler must be inserted by hand. Figure 5.12 shows the *main form* of the Alarm-Clock application called `Form2`.

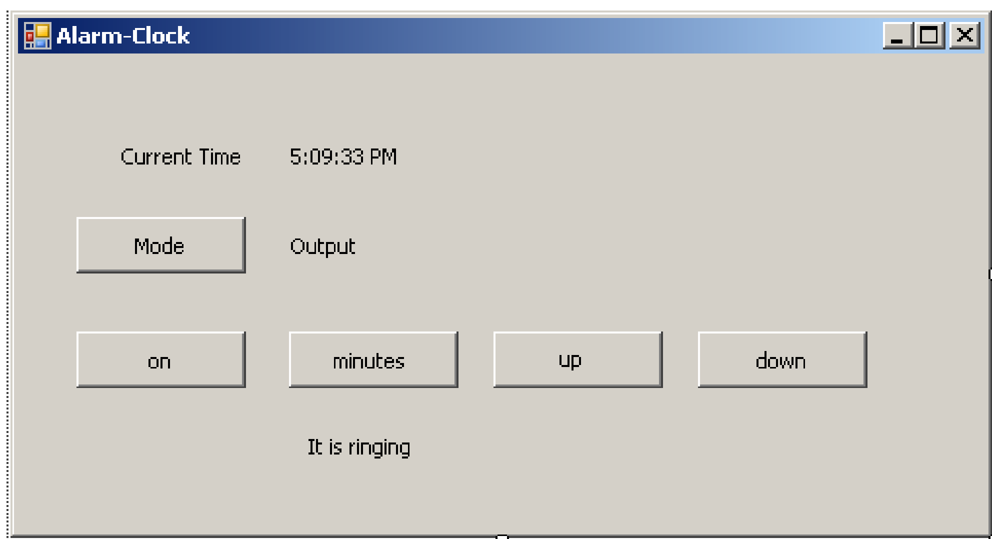

Figure 5.12: The main form of the Alarm-Clock application

This form contains some graphical user interface components such as *buttons* and *labels*. They are generated by the designer of Microsoft Visual studio 5, C#.NET. In the following paragraph, the structure of this window will be explained and the functions of its GUI components such as the labels and buttons are introduced.

The main form consists of four labels and five buttons. The label is a form of a string, which is used for the visual representation of the text. The first label is called `label1` and the function of this label is to display the message “Current time”. The second label is `ctime` and the function of this label is to display the real current time of the system.

The third label is `output` and it works as a counter for minutes and hours for the timer and the two buttons `up` and `down` are used to control this label. So with the two buttons

up and down, we can increase and decrease the value which appears on the output label.

The last label is called ringlabel. This ringlabel disappears when its visible state is false. The label appears on the screen (Window) when the visible state is true.

The mode_button takes the default text value "Mode" and when the button is clicked , it takes one of the three text values "Alarm 1" , "Alarm 2", and "Timer". It is clear that this button controls the mode state, because with regards to the text values, there are three states of the actions "Alarm 1" , "Alarm 2", and "Timer".

The on_off_button takes the text values "on" as a default value and "off" when the button is clicked. The function of this button, when the state is "on", is to set the state of the mode inactive with respect to its mode "Alarm 1" , "Alarm 2", and "Timer". The state "on" allows the user to use the buttons "up" and "down" in order to increase and decrease the time and set the counter to a fixed time say 5 minutes. The time counter will start counting from the current time of the system, and as soon as the 5 minutes are reached, the alarm starts ringing. However if the "off" button is clicked during this counting, the time counter stops.

The up_button is used to increase the time and it takes the text value "up". The down_button is used to decrease the time and also it takes the text value "down".

The mhbutton takes the two text values "minutes" and "hours". So in the case of the minutes state, the time can be incremented and decremented in minutes and in the case of hours state, the time can be incremented and decremented in hours.

The .Net Framework is used for generating and implementing the form as shown in Figure 5.12 which are represented by two files with the same name. One of them is completely generated by the designer and takes the name form2.designer.cs. This file contains all the definitions of the controls and their contents. The other file is called

form2.cs and it contains the constructor to initilize all the components of the form and their respective definitions and functions can be added by hand to it.

Furthermore we define a delegate eventhandler and an event on_event in form2 class. It delegates to the specification of the Alarm-Clock example. The following example shows the declaration of the delegate and event :

```
private delegate void eventhandler(EVS e);
private event eventhandler on_event;
public Form2()
{
  InitializeComponent();
  // on_event is Delegated to the specification ,,
  // added by hand
  on_event += new
     eventhandler(Events_Handling.event_handling);
  init_timer();
  // Control class must add by hand
  ControlActions.af = this;
 }
```

The functions of the buttons which are appear in the main form of the Alarm-Clock application (cf. Figure 5.12) are generated by the designer to generate the event on_event. For example: the following example shows the function of the mode_button button.

```
private void mode_button_Click(object sender, EventArgs
                                e)
{    // to add by hand to generate on_event
     on_event(EVS.mode);
}
```

In order to separate the event handling from the actions of the program, the event is implemented by way of calling functions such as the above example and like the following illustrations:

```
case EVS.mode : mode();
          break;
```

So in this case when the `mode_button` is clicked, the event `EVS.mode` occurs and the function mode is invoked and executed.

5.4.2 Alarm-Clock Specification

The specification of the Alarm-Clock application is an example of the specification of an event handling system. It consists of state-variables space, event handling space and control functions space.

State-Variables Space

The Alarm-clock example has a list of state-variables. Figure 5.13 shows the list of state-variables in the *state space specification* of the Alarm-Clock application.

All the available functions such as `adding, editing, deleting` and `moving` can be applied to the list of state-space variables. Each state-variable has an initial value depending on its type for example `[0…3]mode = 0`, this means that the state-variable called `mode` is of `range` type from 0 to 3 and has an initial value 0.

Each state-variable also has a corresponding variable with the suffix "`_changed`" and is of the `boolean` type. The access to variables are implemented by properties. Any time, a state-variable is changed, the corresponding variable with suffix "`_changed`" is set to `true`. There exists a function `reset_change()` which sets all the variables with suffix "`_changed`" to false. For example there is a variable `mode_changed` of the `boolean` type which corresponds to the state-variable `mode` and is defined in the list of

state-variables of Alarm-Clock application. All these state-variables will be automatically transformed to the C# class called State. In the next paragraph the representation of the state-variables will be briefly introduced.

Figure 5.13: List of state-variables in Alarm-Clock example

State-variables roles

The state-variable mode which is of the range type represents the state of the mode, i.e. if the value of the mode is 0, the mode_button text will be “Mode”, if the mode is 1 then the mode_button will take the text “Alarm1”, if the mode is 2 then the mode_button text will be “Alarm2” and if the mode is 3 then the mode_button will take the text “Timer”.

The state-variable rem_minutes is used for counting minutes while the rem_seconds state-variable is used for counting seconds and all of them are of the range type.

The state-variable alarm1on is of the boolean type and is true when we are in the mode state of "Alarm1". The state-variable alarm2on is also of the boolean type and is true when we are in the mode state of "Alarm2" . The state-variable timer_on is of the boolean type and is true when we are in the mode state of the "Timer". All the other remaining state-variables are of the boolean type.

The state-variable fastslow represents the case by which the time can be counted. In the case of mode state of "Alarm1" and "Alarm2", the time can be increase and decrease in minutes if the fastslow have the text value "minutes" using the "up" and "down" buttons and if it has the text value "hours" then the time can be increased and decreased in hours. But in the case of the mode state of "Timer", the fastslow takes the value "1min" or "10min" and it appears on the mhbutton button. The minute values can be however increased and decreased using the "up" and "down" buttons depending on the values of fastslow variable.

The state-variable updown represents the increasing and decreasing of the time. So in the case of increasing the time the value of the variable updown will be true and in the case of the decreasing the value of the variable will be false.

The state-variable check_alarm contains the mode in which we are, i.e "Alarm 1" , "Alarm 2", or "Timer".

The state-variables alarm1_ringing, alarm2_ringing, and timer_ringing represent the case when the event is fired i.e. when the conditions are satisfied and the methods are executed. Each state-variable rings with respect to its mode state i.e. for example alarm1_ringing function with respect to the mode state "Alarm 1" and so on.

Event Handling Space

In this part, the application declares all the events, which will be used in the Alarm-Clock application. Figure 5.14 shows the list of all these events.

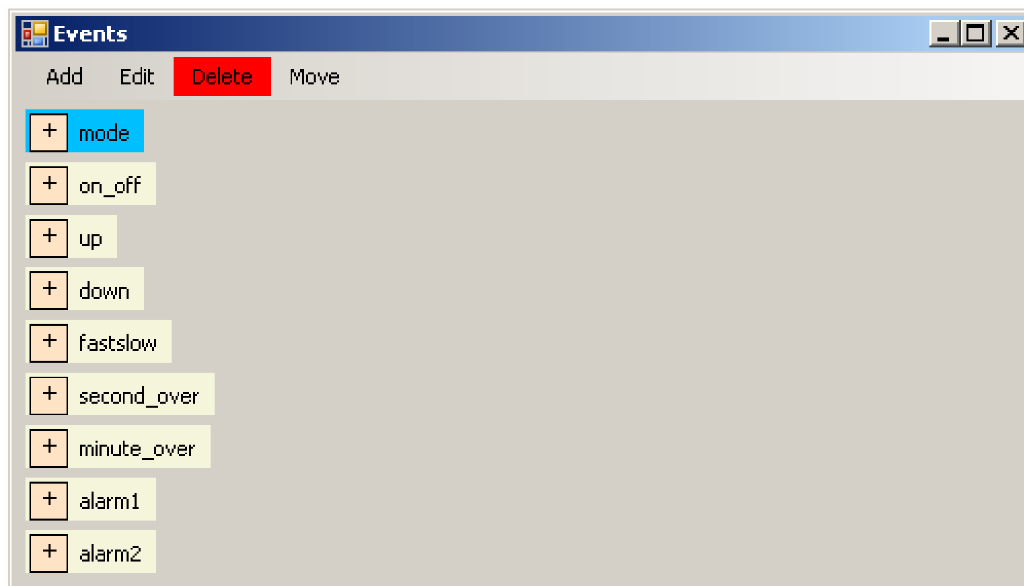

Figure 5.14: List of events in Alarm-Clock example

Of course, the event declarations include the header of the events and the properties of events i.e. the name of each event, the conditions used with this event, and the corresponding state transformations of these conditions. As will be seen later, all these events, conditions and state transformations will be transformed into the C# language. They will automatically be transformed to the `Events_Handling` class.

The form contains the following events which are transformed into methods bearing the same name. A brief descriptions of these events are introduced in the following paragraphs.

The first event is the `mode` event which is transformed into a method called `mode()`. This event is responsible for changing the state of the mode of the application, i.e.

whether the mode is in “Alarm1”, “Alarm2”, or “Timer” state. However, when the `mode_button` is clicked, the variable mode increases by 1, thereby putting the application in one of the state modes “Alarm1”, “Alarm2”, or “Timer” .

The `on_off` event is responsible for starting the count of the time of the chosen mode state. The `on_off` event has two cases, i.e “`on`” and “`off`”. In the case of “`on`”, the timer starts counting with respect to the current time of the system until it reaches the given time. On the other hand, in the case of “off”, the timer stops counting.

The `up` event is used to increase the time in minutes or hours depending on the state mode. This occurs when the user clicks on the `up_button` button. The `down` event also works in a similar manner, but decreases the time.

The `fastslow` event changes the mode of counter of the time, when the `mhbutton` button is clicked and depending on the mode whether it is “`Alarm1`” or “`Alarm2`”. The time will increase in minutes if `mhbutton` has the minute text indicated on it and similarly, if the hour text appears on it, then the time will increase or decrease in hours. If the state of the mode is “`Timer`”, then the `mhbutton` has two text values i.e. `1min` and `10min`. In the case of `1min`, the time increases by 1, whiles in the case of `10min`, the time increases by a factor of `10mins`.

The `second_over` event works with state of the “`Timer`” mode and it is used to know the remaing seconds. However, during counting, when the seconds time reachs up to 59 seconds, then the counting starts from zero. This process occurs when the time increases. For the case of the decreasing aspect, the seconds time decreases until it reaches zero and then it continues decreasing again from 59.

The event `minute_over` is used to know the remaining minutes and it is used with the mode `Alarm1` and `Alarm2` for example when the counter reached to 60 it will be returned to 0 in both operation increasing and decreasing.

The event `alarm1` is used in the case of mode "`Alarm1`" and is used when the counter of the time reaches the expected time, then in this case the conditions are satisfied and the message "`it is ringing`" will appear on the Window. The same thing for the event `alarm2` which is of course used with mode "`Alarm2`".

Control Functions Space

The control functions of the Alarm-Clock application declare all the conditions that are defined in the application and their corresponding methods. Figure 5.15 shows these conditions and their corresponding methods.

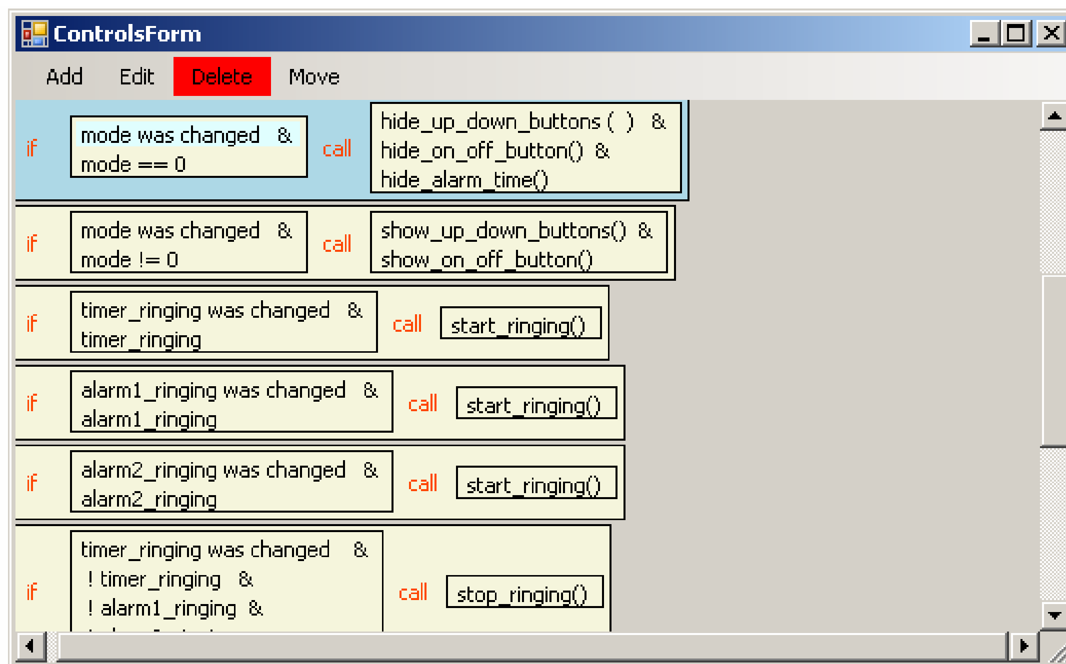

Figure 5.15: List of control functions in Alarm-Clock example

All these conditions and their corresponding program actions will be transformed into C# declarations. They will be automatically transformed into two C# classes. The first class is called `Control` which contains the transformation and implementation of conditions and program actions. The second class is called `Appfunctions` which contains the declarations of the contents of the program actions.

5.4.3 Generated Parts Of The Alarm-Clock Example

The generated program automatically generates the specification space definitions of the Alarm-Clock example into C# language classes. The *state space specification* is transformed to the `State` class, the *event handling specification* is transformed to the `Events_handling` class, and the *control functions specification* is transformed to these two classes `Control` and `AppFunctions`. A small part of each generated class will be introduced. The complete descriptions of all generated classes will be presented in Appendix A.

As known from previous secition as in the specification of the whole system, each state-variable in the *state space specification* of the Alarm-Clock example will be transformed into two corresponding components variables. One has the same data type with suffix "`_v`" and it also has an initial value. The other one has the suffix "`_changed`" which is of `boolean` type and the access to the variables are given by the properties. So `set` and `get` functions are defined for each variable. Every time, a variable is changed, the corresponding `boolean` variable with suffix "`_changed`" is set to `true`. There is also a function `reset_change()` which sets all the `boolean` variables with suffix "`_changed`" to `false`.

The following example shows a brief description of the `State` class of the Alarm-Clock application. The details of the `State` class are described in Appendix A.1.

Example : State Class

```
namespace EV_Edit
{

  static class State
 {
   //  the type of this variable is range[lower...upper]
    private static uint mode_v =0 ;
    public static bool mode_changed  = false ;
    public static uint mode
    {
        get { return mode_v ; }
        set
```

```
        {
           mode_v = value > 3 ? 0 : value ;
           mode_changed = true;
         }
    }

    private static bool alarm1on_v =false ;
    public static bool alarm1on_changed  = false ;
    public static bool alarm1on
    {
        get { return alarm1on_v ; }
        set
        {
           alarm1on_v = value ;
           alarm1on_changed = true;
        }
    }

       public static void reset_change()
    {
         mode_changed = false ;
         rem_minutes_changed = false ;
         rem_seconds_changed = false ;
         alarm1on_changed = false ;
         alarm2on_changed = false ;
         timer_on_changed = false ;
         fastslow_changed = false ;
         updown_changed = false ;
         check_alarm_changed = false ;
         alarm1_ringing_changed = false ;
         alarm2_ringing_changed = false ;
         timer_ringing_changed = false ;
    }

  }
}
```

Event handling specification of the Alarm-Clock example will be transformed to the `Events_Handling` class. For each event, the header will be transformed to a function with the same name of the event and the contents of the event will be transform into C#-if-statements. The first part of an if-statement will be the conditions of the event that will be checked and the second part of the if-statement will be the state transformations that will be executed when the conditions are satisfied.

In the `Events_Handling` class, the declaration of the events are as members of `enum` type named `EVS`. It contains all the events that are declared in the *event handling specification* of the `Alarm-clock` application.

```
enum EVS { mode , on_off , up , down , fastslow ,
           second_over, minute_over, alarm1, alarm2 };
```

In order to avoid changing the values of state-variables which may effect on the comparison and checking of the condition values, all the conditions for each event are stored in a defined *array* named `cond` as illustrated below:

```
cond[0] = State.mode == 1;
cond[1] = State.mode == 2;
cond[2] = State.mode == 3 && !State.fastslow;
cond[3] = State.mode == 3 && State.fastslow;
```

After that, the conditions which are stored in the array `cond` for each event are checked by if-statements with the corresponding state transformations of these conditions as follows:

```
if (cond[0])
 {
         State.updown = true ;
 }
if (cond[1])
 {
         State.updown = true ;
 }
if (cond[2])
 {
         State.rem_minutes += 1 ;
 }
```

```
if (cond[3])
 {
         State.rem_seconds = 10 ;
 }
```

A queue named `evs_queue` is defined in order to insert the events into the queue. There is a `boolean` variable `event_cycle_is_working` in order to manage the function `event_cycle()`. There are also two integer variables `first` and `last` for indexing the events. First variable represent the index of the first event in the queue `evs_queue` and `last` variable represent the index of the last event in the queue `evs_queue`.

There exists a function called "`event_handling(EVS e)`". It only adds a raised event into the queue `evs_queue`, and in case the `event_cycle_is_working` has value `false`, the `event_cycle()` function is invoked. The functions `add_event(EVS e)` and `fetch_event()` organize a correct management of the queue `evs_queue`.

The function `event_cycle()` checks wheather some events wait in the queue `evs_queue`. If this is the case, then these events are executed by calling the corresponding functions. In every cycle, the control functions is called, and after that, the state-variables with suffix "`_changed`" are set to `false` by calling the function `reset_change()` from the `State` class.

The following example shows a part of the transformation of the defined events and their contents in the *event handling specification* to the `Events_Handling` class of the Alarm-Clock example. The complete detail description of the `Events_Handling` class can be found in Appendix A.2.

Example : Events_Handling Class

```
namespace EV_Edit
```

```
{

  enum EVS { mode , on_off , up , down , fastslow , second_over ,
  minute_over , alarm1 , alarm2 };

  static class Events_Handling
  {
     private static bool[] cond = new bool[100];
     private static EVS[] evs_queue = new EVS[100];
     private static bool event_cycle_is_working = false;
     private static uint first =0, last = 0;
     private static void add_event(EVS e)
     {

      evs_queue[last++] = e;
      if (last == 10)
          last = 0;
     }
     private static EVS fetch_event()
     {
        EVS e = evs_queue[first++];
        if (first == 10)
           first=0;
        return e ;
     }
     public static void event_handling(EVS e)
     {
        add_event(e);
        if (!event_cycle_is_working)
           event_cycle();
     }
     private static void event_cycle()
     {
        EVS e ;
        event_cycle_is_working = true ;
        while (first != last)
        {
           e= fetch_event();
           switch (e)
           {
              case EVS.mode : mode();
                   break;
              case EVS.on_off : on_off();
                   break;
              case EVS.up : up();
                   break;
           }
           ControlActions.control();
           State.reset_change();
         }
         event_cycle_is_working = false;
     }

    public static void mode()
    {
```

```
                State.mode += 1 ;
    }

    public static void on_off()
    {
       cond[0] = State.mode == 1;
       cond[1] = State.mode == 1 && State.alarm1on &&
                 State.alarm1_ringing;
       cond[2] = State.mode == 2;

   //  if (State.mode == 1)
       if (cond[0])
       {
                State.alarm1on = !State.alarm1on ;
       }

   //  if (State.mode == 1 && State.alarm1on && State.alarm1_ringing)
       if (cond[1])
       {
                State.alarm1_ringing = false ;
       }

   //  if (State.mode == 2)
       if (cond[2])
       {
                State.alarm2on = !State.alarm2on ;
       }
    }

    public static void up()
    {
       cond[0] = State.mode == 1;
       cond[1] = State.mode == 2;

   //  if (State.mode == 1)
       if (cond[0])
       {
                State.updown = true ;
       }


   //  if (State.mode == 2)
       if (cond[1])
       {
                State.updown = true ;
       }
    }
  }
}
```

The last part of the specification of the Alarm-Clock example is the *control functions specification.* As previously mentioned, this part will automatically be transformed into

two C# classes. The first one is the `Control` class and the second one is the `AppFunctions` class.

In the `Control` class all the action panels of the *control functions specification* will be automatically transformed into C# if-statements. The first part of an if-statement is includes the conditions that needed to be checked and the second part of the if-statement is the corresponding program actions with their paramenters. Once the conditions are satisfied the corresponding methods are invoked. The `Control` class has an instance of the `AppFunctions` class in order to call the functions from it.

The automatically generated class `AppFunctions` specifies the program actions of the `Control` class. This class is invoked by the `Control` class. An empty template will appear without the contents of the function body, only the header of the function with its parameters. The user must write the contenets of the body of the function and the user should not have the ability to change the header of the function.

There are other variables for managing and showing the current time also, but these variables are not uses in the specification. There are also two functions for controling the time like:

```
System.Timers.Timer aTimer;
DateTime current_time, last_time;
private void init_timer()
{
     aTimer = new System.Timers.Timer();
     aTimer.Elapsed += new
     ElapsedEventHandler(aTimer_Elapsed);
     aTimer.Interval = 1000;
     aTimer.Start();
     last_time = DateTime.Now;
}
```

```
private void aTimer_Elapsed(object sender,
                            ElapsedEventArgs e)
{
    current_time = DateTime.Now;
    ctime.Text = current_time.ToLongTimeString();
    on_event(EVS.second_over);
    if (current_time.Minute != last_time.Minute)
    {
        last_time = current_time;
        on_event(EVS.minute_over);
    }
}
```

Below is a brief description of the two classes namely, `Control` class and `AppFunctions` class. The complete detail description of the `Control` class and `AppFunctions` class are described respectively in Appendix A.3 and A.4.

Example: Control Class:

```
namespace EV_Edit
{

  static class ControlActions
  {
    public static Form2 af;
    public static void control()
    {
       if (State.updown_changed && State.mode != 3)
       {
              af.updown_alarm(State.mode,State.fastslow,State.updown) ;
       }

       if (State.mode == 1)
       {
              af.set_on_off_button(State.alarm1on) ;
              af.show_alarm_time(1) ;
       }

       if (State.mode == 2)
       {
                af.set_on_off_button(State.alarm2on) ;
```

```
                af.show_alarm_time(2) ;
        }
    }
 }
}
```

Example: AppFunctions Class:

```
namespace EV_Edit
{
      public partial class Form2
      {
            // Application specific functions, called by class Control
            // the templates (without contents of bodies) are
            automatically generated
            private static uint[] alarm_time = { 0, 0 };
            // manages and shows the current time
            // does not use the specification

            public void updown_alarm(uint mode, bool fs, bool ud)
            {
                  int i = (int)alarm_time[mode - 1];
                  int h = i / 60;
                  int m = i % 60;
                  if (fs)
                  {
                        h = h + (ud ? 1 : -1);
                        h = (h == 24) ? 0 : (h < 0) ? 23 : h;
                  }
                  else
                  {
                        m = m + (ud ? 1 : -1);
                        m = (m == 60) ? 0 : (m < 0) ? 59 : m;
                  }

                  alarm_time[mode - 1] = (uint)(h * 60 + m);


            }
            public void set_on_off_button(bool b)
            {

                  on_off_button.Text = b ? "on" : "off";

            }
            public void show_alarm_time(uint mode)
            {
                  uint h = alarm_time[--mode] / 60, min =
                  alarm_time[mode] % 60;
                  output.Text = h.ToString() + " : " + min.ToString();
                  output.Visible = true;
            }
      }}
```

6

Conclusion and Future Work

6.1 Overview

In this thesis, the problem of distributing the event handling over the program text has been discussed. The main idea that was used to solve this problem was to clearly separate the event handling from the actions of the program. A special program responsible for the event handling alone was introduced and developed [50].

This program was divided into two important parts. The first part was the specification part, which in itself contains three parts namely, *state space specification, event handling specification,* and *control functions specification*. The second part was the *hand-built program* (hbp), which contains the classes that are needed to build the event system. This part was programmed by the developer of the application using C#.NET.

The event-programming-framework (epro) contains a special editor to write and manage specifications. The specification-editor stores all data inputs from the three parts of the specification in an XML file.

The state-variables are defined in the state space specification with their types and initial values. These state-variables are used in *event handling* and *control functions specification parts*. There are some available functions such as *Add, Edit, Delete,* and *Move* to manage the state-variables. The editor has some procedures of verification and optimization for these variables in order to avoid redundancy and inconsistency. The

same procedures are applied in *event handling specification* and *control functions specification* parts.

The events are defined in the *event handling specification part.* In this part, each event contains a condition or a group of conjunction of conditions and state transformations. Only the variables, which are defined in the state space part, are used in the conditions. The statements in the event consist of a conjunction of conditions. During the occurring of the event, all the conditions associated with it are checked. Once the state-variables satisfied the conditions, the corresponding state transformations are executed.

A set of conditional statements and corresponding actions are described in the *control functions specification* as in an *event handling specification.*

Some optimization, simplification and verification steps are used and applied in order to check the conditions on the *event handling* and *control functions specifications* such as the elimination of tautologies, elimination of contradictions, and elimination of implications. A *transitive closure* was built in order to deal with these optimizations and verifications and some additional considerations are applied in order to guarantee that valid conditions and values are possible.

The specification part is transformed into C# language as classes with corresponding fields, properties and methods. The *state space specification* is transformed into `State` class, the *event handling specification* is transformed into `Events_Handling` class, and the *control functions specification* is transformed into `Control` class. Moreover, in every application there must be a class of a certain name, say `Form`, introduced by the developer of the system or by any designer (e.g. Microsoft Visual Studio), where the GUI and the basic Components are defined. In this class the definition of a general event handler must be inserted manually. In a special file, called `AppFunctions.cs`, templates for all methods where corresponding actions are called in the *control functions specification* are inserted automatically. An application example was introduced in order to illustrate and show how the system implements the events.

6.2 Advantages and Disadvantages

The advantages of this new approach of event handling systems can be summarized as follows: the ability to separate the event handling from the control of actions gives the opportunity for the whole system to be designed by two different persons: one of whom is responsible only for managing events and the other is responsible for the effects that these events produce depending on their different states. The other advantage is that, the description of event-handling cycle as well as control of program actions can be done separately in a relatively abstract specification like manner. For this specification, no knowledge of any programming language is necessary. Nevertheless, an automatic transformation of the specification into the source code of the used programming language is possible. Moreover, the event-handling cycle can also be considered as a special transaction system. Some optimization of the transformation into programming language code and verifications of temporal assertions can be carried out.

On the other hand, the disadvantage arises when there is the need to generate and add some programming by manual means and also loading the automatically generated classes to the application. These procedures may sometimes cause mistakes to arise. The applications of some algorithms for calculating the transitive closure also may affect the rate at which the results of the program is produced i.e. it takes time for the results to be produced.

6.3 Proposed Future research work

Considering the various aspects discussed in this thesis, there are several other opportunities that need to be further exploited in order to improve the performance and the efficiency of the event handling system, the following section gives a summary of areas that need to be considered in future work :

- Due to lack of time, the parameters where not added to the part of the *event handling specification,* however it is possible to add these data to the *event*

handling specification (i.e. the event declaration such as header of the event which includes formal parameters in case there are some).

- It was planed to improve the algorithm which was used to calculate the transitive closure in order to improve some optimization, simplification and verification steps that are applied in the event handling system.

- It is proposed to develop a specified methodology that works in such a way that the user will not have to be manually adding some codes to the methods when the system is generating the classes of the event handling system

- A proposal was also suggested to consider the application of the principle of predicate transformer. The predicate transformer is an investigated method that plays a significant role for the verification of the programs. The concept refers to Sifakis [73] who applied this theory to general transition system. In this study, an event system can be considered as a transition system. Therefore, with respect to event system, the meaning of predicate transformer is the set of all states which can be transformed by some events into states which fulfill the postcondition. Predicate transformer will be useful to verify some interesting properties of event systems.

Appendix A Generated Classes

This appendix is organized as follows: section A.1 describes the State class which is containing the declaration of the state-variables, their types and properties. Section A.2 presents the Events-Handling class which contains the declarations of the events and corresponding conditions and state transformation. The Control class which is contains the declarations of the function controls, their conditions and calls are described in section A.3. The AppFunctions class is presented in section A.4.

A.1 State Class

```
using System ;
using System.Collections.Generic ;
using System.ComponentModel ;
using System.Data ;
using System.Drawing ;
using System.Text ;
using System.Windows.Forms ;
using System.IO ;
using System.Xml ;
using System.Timers ;


namespace EV_Edit
{


  static class State
 {
   //  the type of this variable is range[lower...upper]
    private static uint mode_v =0 ;
    public static bool mode_changed  = false ;
    public static uint mode
    {
        get { return mode_v ; }
        set
        {
           mode_v = value > 3 ? 0 : value ;
           mode_changed = true;
```

```
      }
    }

  //  the type of this variable is range[lower...upper]
   private static uint rem_minutes_v =0 ;
   public static bool rem_minutes_changed  = false ;
   public static uint rem_minutes
   {
       get { return rem_minutes_v ; }
       set
       {
          rem_minutes_v = value > 120 ? 0 : value ;
          rem_minutes_changed = true;
       }
   }

  //  the type of this variable is range[lower...upper]
   private static uint rem_seconds_v =0 ;
   public static bool rem_seconds_changed  = false ;
   public static uint rem_seconds
   {
       get { return rem_seconds_v ; }
       set
       {
          rem_seconds_v = value > 60 ? 0 : value ;
          rem_seconds_changed = true;
        }
   }

   private static bool alarm1on_v =false ;
   public static bool alarm1on_changed  = false ;
   public static bool alarm1on
   {
       get { return alarm1on_v ; }
       set
       {
          alarm1on_v = value ;
          alarm1on_changed = true;
       }
   }

   private static bool alarm2on_v =false ;
   public static bool alarm2on_changed  = false ;
   public static bool alarm2on
   {
       get { return alarm2on_v ; }
       set
       {
          alarm2on_v = value ;
          alarm2on_changed = true;
       }
   }

   private static bool timer_on_v =false ;
   public static bool timer_on_changed  = false ;
```

```
public static bool timer_on
{
    get { return timer_on_v ; }
    set
    {
       timer_on_v = value ;
       timer_on_changed = true;
    }
}

private static bool fastslow_v =false ;
public static bool fastslow_changed  = false ;
public static bool fastslow
{
    get { return fastslow_v ; }
    set
    {
       fastslow_v = value ;
       fastslow_changed = true;
    }
}

private static bool updown_v =false ;
public static bool updown_changed  = false ;
public static bool updown
{
    get { return updown_v ; }
    set
    {
       updown_v = value ;
       updown_changed = true;
    }
}

private static bool check_alarm_v =false ;
public static bool check_alarm_changed  = false ;
public static bool check_alarm
{
    get { return check_alarm_v ; }
    set
    {
       check_alarm_v = value ;
       check_alarm_changed = true;
    }
}

private static bool alarm1_ringing_v =false ;
public static bool alarm1_ringing_changed  = false ;
public static bool alarm1_ringing
{
    get { return alarm1_ringing_v ; }
    set
    {
       alarm1_ringing_v = value ;
       alarm1_ringing_changed = true;
```

```
        }
    }

    private static bool alarm2_ringing_v =false ;
    public static bool alarm2_ringing_changed  = false ;
    public static bool alarm2_ringing
    {
        get { return alarm2_ringing_v ; }
        set
        {
           alarm2_ringing_v = value ;
           alarm2_ringing_changed = true;
        }
    }

    private static bool timer_ringing_v =false ;
    public static bool timer_ringing_changed  = false ;
    public static bool timer_ringing
    {
        get { return timer_ringing_v ; }
        set
        {
           timer_ringing_v = value ;
           timer_ringing_changed = true;
        }
    }

    public static void reset_change()
    {
         mode_changed = false ;
         rem_minutes_changed = false ;
         rem_seconds_changed = false ;
         alarm1on_changed = false ;
         alarm2on_changed = false ;
         timer_on_changed = false ;
         fastslow_changed = false ;
         updown_changed = false ;
         check_alarm_changed = false ;
         alarm1_ringing_changed = false ;
         alarm2_ringing_changed = false ;
         timer_ringing_changed = false ;
    }


  }
}
```

A.2 Events_Handling Class

```
using System ;
using System.Collections.Generic ;
using System.ComponentModel ;
```

```
using System.Data ;
using System.Drawing ;
using System.Text ;
using System.Windows.Forms ;
using System.IO ;
using System.Xml ;
using System.Timers ;
namespace EV_Edit
{

  enum EVS { mode , on_off , up , down , fastslow , second_over ,
  minute_over , alarm1 , alarm2 };

  static class Events_Handling
  {
     private static bool[] cond = new bool[100];
     private static EVS[] evs_queue = new EVS[100];
     private static bool event_cycle_is_working = false;
     private static uint first =0, last = 0;

     private static void add_event(EVS e)
     {
      evs_queue[last++] = e;
      if (last == 10)
          last = 0;
     }

     private static EVS fetch_event()
     {
        EVS e = evs_queue[first++];
        if (first == 10)
           first=0;
        return e ;
     }

     public static void event_handling(EVS e)
     {
        add_event(e);
        if (!event_cycle_is_working)
           event_cycle();
     }

     private static void event_cycle()
     {
        EVS e ;
        event_cycle_is_working = true ;
        while (first != last)
        {
           e= fetch_event();
           switch (e)
           {
              case EVS.mode : mode();
                   break;
              case EVS.on_off : on_off();
                   break;
```

```
            case EVS.up : up();
                 break;
            case EVS.down : down();
                 break;
            case EVS.fastslow : fastslow();
                 break;
            case EVS.second_over : second_over();
                 break;
            case EVS.minute_over : minute_over();
                 break;
            case EVS.alarm1 : alarm1();
                 break;
            case EVS.alarm2 : alarm2();
                 break;
         }
         ControlActions.control();
         State.reset_change();
       }
       event_cycle_is_working = false;
   }

  public static void mode()
  {
             State.mode += 1 ;
  }

  public static void on_off()
  {
     cond[0] = State.mode == 1;
     cond[1] = State.mode == 1 && State.alarm1on &&
               State.alarm1_ringing;
     cond[2] = State.mode == 2;
     cond[3] = State.mode == 2 && State.alarm2on &&
               State.alarm2_ringing;
     cond[4] = State.mode == 3 && State.timer_on &&
               State.timer_ringing;
     cond[5] = State.mode == 3 && State.timer_on;
     cond[6] = State.mode == 3;

//   if (State.mode == 1)
     if (cond[0])
     {
             State.alarm1on = !State.alarm1on ;
     }

//   if (State.mode == 1 && State.alarm1on && State.alarm1_ringing)
     if (cond[1])
     {
             State.alarm1_ringing = false ;
     }

//   if (State.mode == 2)
     if (cond[2])
     {
             State.alarm2on = !State.alarm2on ;
```

```
        }

//      if (State.mode == 2 && State.alarm2on && State.alarm2_ringing)
        if (cond[3])
        {
                State.alarm2_ringing = false ;
        }

//      if (State.mode == 3 && State.timer_on && State.timer_ringing)
        if (cond[4])
        {
                State.timer_ringing = false ;
        }

//      if (State.mode == 3 && State.timer_on)
        if (cond[5])
        {
                State.rem_seconds = 60 ;
        }

//      if (State.mode == 3)
        if (cond[6])
        {
                State.timer_on = !State.timer_on ;
        }
    }

    public static void up()
    {
        cond[0] = State.mode == 1;
        cond[1] = State.mode == 2;
        cond[2] = State.mode == 3 && !State.fastslow;
        cond[3] = State.mode == 3 && State.fastslow;

//      if (State.mode == 1)
        if (cond[0])
        {
                State.updown = true ;
        }

//      if (State.mode == 2)
        if (cond[1])
        {
                State.updown = true ;
        }

//      if (State.mode == 3 && !State.fastslow)
        if (cond[2])
        {
                State.rem_minutes += 1 ;
        }
//      if (State.mode == 3 && State.fastslow)
        if (cond[3])
        {
                State.rem_seconds = 10 ;
```

```
        }
    }

    public static void down()
    {
        cond[0] = State.mode == 1;
        cond[1] = State.mode == 2;
        cond[2] = State.mode == 3 && !State.fastslow;
        cond[3] = State.mode == 3 && State.fastslow;

//      if (State.mode == 1)
        if (cond[0])
        {
                State.updown = false ;
        }

//      if (State.mode == 2)
        if (cond[1])
        {
                State.updown = false ;
        }

//      if (State.mode == 3 && !State.fastslow)
        if (cond[2])
        {
                State.rem_seconds = 1 ;
        }

//      if (State.mode == 3 && State.fastslow)
        if (cond[3])
        {
                State.rem_minutes = 10 ;
        }
    }

    public static void fastslow()
    {
                State.fastslow = !State.fastslow ;
    }

    public static void second_over()
    {
        cond[0] = State.timer_on && State.rem_seconds > 1;
        cond[1] = State.timer_on && State.rem_seconds == 1 &&
                  State.rem_minutes > 1;
        cond[2] = State.timer_on && State.rem_seconds == 1 &&
                  State.rem_minutes == 1;
//      if (State.timer_on && State.rem_seconds > 1)
        if (cond[0])
        {
                 State.rem_seconds -= 1 ;
        }

      //  if (State.timer_on && State.rem_seconds == 1 &&
      State.rem_minutes > 1)
```

```
       if (cond[1])
       {
                State.rem_seconds = 60 ;
                State.rem_minutes -= 1 ;
       }

      //  if (State.timer_on && State.rem_seconds == 1 &&
      State.rem_minutes == 1)
       if (cond[2])
       {
                State.timer_ringing = true ;
                State.rem_minutes = 0 ;
       }

    }

    public static void minute_over()
    {
       cond[0] = State.alarm1on && !State.alarm1_ringing;
       cond[1] = State.alarm2on && !State.alarm2_ringing;

   //  if (State.alarm1on && !State.alarm1_ringing)
       if (cond[0])
       {
                State.check_alarm = true ;
       }

   //  if (State.alarm2on && !State.alarm2_ringing)
       if (cond[1])
       {
                State.check_alarm = true ;
       }
    }

    public static void alarm1()
    {
                State.alarm1_ringing = true ;
    }

    public static void alarm2()
    {
                State.alarm2_ringing = true ;
    }
  }
}
```

A.3 Control Class:

```
using System ;
using System.Collections.Generic ;
using System.ComponentModel ;
using System.Data ;
```

```
using System.Drawing ;
using System.Text ;
using System.Windows.Forms ;
using System.IO ;
using System.Xml ;
using System.Timers ;

namespace EV_Edit
{
  static class ControlActions
  {
    public static Form2 af;
    public static void control()
    {
       if (State.updown_changed && State.mode != 3)
       {
              af.updown_alarm(State.mode,State.fastslow,State.updown) ;
       }

       if (State.mode == 1)
       {
              af.set_on_off_button(State.alarm1on) ;
              af.show_alarm_time(1) ;
       }

       if (State.mode == 2)
       {
                af.set_on_off_button(State.alarm2on) ;
                af.show_alarm_time(2) ;
       }

       if (State.mode == 3)
       {
                af.set_on_off_button(State.timer_on) ;
                af.show_timer(State.rem_minutes) ;
       }

       if (State.mode != 0)
       {
                af.show_fastslow(State.mode,State.fastslow) ;
       }

       if (State.mode_changed)
       {
                af.set_mode(State.mode) ;
       }

       if (State.mode_changed && State.mode == 0)
       {
                af.hide_up_down_buttons (  )   ;
                af.hide_on_off_button()  ;
                af.hide_alarm_time() ;
       }

       if (State.mode_changed && State.mode != 0)
```

```
        {
                af.show_up_down_buttons()  ;
                af.show_on_off_button() ;
        }

        if (State.timer_ringing_changed && State.timer_ringing)
        {
                af.start_ringing() ;
        }

        if (State.alarm1_ringing_changed && State.alarm1_ringing)
        {
                af.start_ringing() ;
        }

        if (State.alarm2_ringing_changed && State.alarm2_ringing)
        {
                af.start_ringing() ;
        }

        if (State.timer_ringing_changed && !State.timer_ringing &&
             !State.alarm1_ringing && !State.alarm2_ringing)
        {
                af.stop_ringing() ;
        }

        if (State.alarm1_ringing_changed && !State.timer_ringing &&
             !State.alarm1_ringing && !State.alarm2_ringing)
        {
                af.stop_ringing() ;
        }

        if (State.alarm2_ringing_changed && !State.timer_ringing &&
             !State.alarm1_ringing && !State.alarm2_ringing)
        {
                af.stop_ringing() ;
        }

        if (State.alarm1on && State.check_alarm_changed)
        {
                af.check_alarm1() ;
        }

        if (State.alarm2on && State.check_alarm_changed)
        {
                af.check_alarm2() ;
        }
    }
 }
}
```

A.4 AppFunctions Class:

```
using System ;
using System.Collections.Generic ;
using System.ComponentModel ;
using System.Data ;
using System.Drawing ;
using System.Text ;
using System.Windows.Forms ;
using System.IO ;
using System.Xml ;
using System.Timers ;
namespace EV_Edit
{
      public partial class Form2
      {
            // Application specific functions, called by class Control
            // the templates (without contents of bodies) are
            automatically generated
            private static uint[] alarm_time = { 0, 0 };
            // manages and shows the current time
            // does not use the specification
            System.Timers.Timer aTimer;
            DateTime current_time, last_time;
            private void init_timer()
            {
                  aTimer = new System.Timers.Timer();
                  aTimer.Elapsed += new
                  ElapsedEventHandler(aTimer_Elapsed);
                  aTimer.Interval = 1000;
                  aTimer.Start();
                  last_time = DateTime.Now;
            }
            private void aTimer_Elapsed(object sender, ElapsedEventArgs
            e)
            {
                  current_time = DateTime.Now;
                  ctime.Text = current_time.ToLongTimeString();
                  on_event(EVS.second_over);
                  if (current_time.Minute != last_time.Minute)
                  {
                        last_time = current_time;
                        on_event(EVS.minute_over);
                  }
            }
            public void updown_alarm(uint mode, bool fs, bool ud)
            {
                  int i = (int)alarm_time[mode - 1];
                  int h = i / 60;
                  int m = i % 60;
                  if (fs)
                  {
```

```
                h = h + (ud ? 1 : -1);
                h = (h == 24) ? 0 : (h < 0) ? 23 : h;
          }
          else
          {
                m = m + (ud ? 1 : -1);
                m = (m == 60) ? 0 : (m < 0) ? 59 : m;
          }
          alarm_time[mode - 1] = (uint)(h * 60 + m);
    }
    public void set_on_off_button(bool b)
    {

          on_off_button.Text = b ? "on" : "off";

    }
    public void show_alarm_time(uint mode)
    {
          uint h = alarm_time[--mode] / 60, min =
          alarm_time[mode] % 60;
          output.Text = h.ToString() + " : " + min.ToString();
          output.Visible = true;

    }
    public void show_timer(uint t)
    {
          output.Text = t.ToString();
          output.Visible = true;

    }
    public void show_fastslow(uint mode, bool fs)
    {
          mhbutton.Text = fs ? (mode == 3 ? "10 min" : "hours")
          :  (mode == 3 ? "1 min" : "minutes");

    }
    public void set_mode(uint mode)
    {
          switch (mode)
          {
                case 0: mode_button.Text = "Mode";
                      return;
                case 1: mode_button.Text = "Alarm 1";
                      return;
                case 2: mode_button.Text = "Alarm 2";
                      return;
                case 3: mode_button.Text = "Timer";
                      return;
          }
    }
    public void hide_up_down_buttons()
    {

          up_button.Visible = false;
          down_button.Visible = false;
```

```
        }
        public void hide_on_off_button()
        {
              mhbutton.Visible = false;
              on_off_button.Visible = false;

        }
        public void hide_alarm_time()
        {
              output.Visible = false;

        }
        public void show_up_down_buttons()
        {
              up_button.Visible = true;
              down_button.Visible = true;

        }
        public void show_on_off_button()
        {
              on_off_button.Visible = mhbutton.Visible = true;


        }
        public void start_ringing()
        {
              ringlabel.Visible = true;
              ringlabel.Refresh();

        }
        public void stop_ringing()
        {
              ringlabel.Visible = false;

        }
        public void check_alarm1()
        {
              if (alarm_time[0] == (current_time.Minute +
              current_time.Hour * 60))
                    on_event(EVS.alarm1);

        }
        public void check_alarm2()
        {
              if (alarm_time[1] == (current_time.Minute +
              current_time.Hour * 60))
                    on_event(EVS.alarm2);
        }
  }
}
```

Appendix B XML File

The XML file below describes a small example of all the specification data of the event handling system of the Alarm-Clock application that described in section 5.4:


```
<? xml version="1.0" encoding="utf-8" ?>

– <specification>
– <statespace>
    <variable no="1" typel="[0 ... 3]" name="mode" initl="= 0" type="3"
        lower="0"  upper="3" iinit="0" finit="0" binit="false" />
    <variable no="2" typel="[0 ... 120]" name="rem_minutes" initl="= 0"
        type="3" lower="0" upper="120" iinit="0" finit="0" binit="false" />
    <variable no="3" typel="[0 ... 60]" name="rem_seconds" initl="= 0"
    type="3" lower="0" upper="60" iinit="0" finit="0" binit="false" />
    <variable no="4" typel="boolean" name="alarm1on" initl="= False"
    type="2" lower="0" upper="60" iinit="0" finit="0" binit="false" />
    <variable no="5" typel="boolean" name="alarm2on" initl="= False"
    type="2"      lower="0" upper="60" iinit="0" finit="0" binit="false"
        />
    <variable no="6" typel="boolean" name="timer_on" initl="= False"
    type="2"      lower="0" upper="60" iinit="0" finit="0" binit="false"
        />
    <variable no="7" typel="boolean" name="fastslow" initl="= False"
    type="2"      lower="0" upper="60" iinit="0" finit="0" binit="false"
        />
    <variable no="8" typel="boolean" name="updown" initl="= False"
    type="2"      lower="0" upper="60" iinit="0" finit="0" binit="false"
        />
    <variable no="9" typel="boolean" name="check_alarm" initl="= False"
        type="2" lower="0" upper="60" iinit="0" finit="0" binit="false" />
    <variable no="10" typel="boolean" name="alarm1_ringing" initl="=
    False" type="2" lower="0" upper="0" iinit="0" finit="0" binit="false"
    />
    <variable no="11" typel="boolean" name="alarm2_ringing" initl="=
    False" type="2" lower="0" upper="0" iinit="0" finit="0" binit="false"
    />
    <variable no="12" typel="boolean" name="timer_ringing" initl="=
    False" type="2" lower="0" upper="0" iinit="0" finit="0" binit="false"
    />
  </statespace>
```

```
– <events>
– <event ident="mode">
– <action>
<conditions />
– <transformations>
        <trans text="mode += 1" lvar="1" oper="2" rvar="0" ival="1"
        fval="0" bval="False" />
</transformations>
</action>
 </event>
– <event ident="on_off">
– <action>
– <conditions>
        <condition text="mode == 1" lvar="1" oper="3" rvar="0" ival="1"
                fval="0" />
         </conditions>
– <transformations>
        <trans text="! alarm1on" lvar="4" oper="0" rvar="0"
                ival="1" fval="0" bval="False" />
 </transformations>
</action>
– <action>
– <conditions>
        <condition text="mode == 1 &" lvar="1" oper="3" rvar="0"
                ival="1" fval="0" />
         <condition text="alarm1on &" lvar="4" oper="-1" rvar="0" ival="0"
        fval="0" />
         <condition text="alarm1_ringing" lvar="10" oper="-1" rvar="0"
                ival="0" fval="0" />
 </conditions>
– <transformations>
         <trans text="alarm1_ringing := False" lvar="10" oper="1"
                rvar="0" ival="0" fval="0" bval="False" />
 </transformations>
</action>
– <action>
– <conditions>
         <condition text="mode == 2" lvar="1" oper="3" rvar="0" ival="2"
                fval="0" />
</conditions>
– <transformations>
         <trans text="! alarm2on" lvar="5" oper="0" rvar="0" ival="0"
                fval="0" bval="False" />
 </transformations>
</action>
– <action>
– <conditions>
```

```
        <condition text="mode == 2 &" lvar="1" oper="3" rvar="0"
            ival="2" fval="0" />
        <condition text="alarm2on &" lvar="5" oper="-1" rvar="0"
            ival="0" fval="0" />
        <condition text="alarm2_ringing" lvar="11" oper="-1" rvar="0"
            ival="0" fval="0" />
  </conditions>
- <transformations>
         <trans text="alarm2_ringing := False" lvar="11" oper="1"
        rvar="0" ival="0" fval="0" bval="False" />
  </transformations>
  </action>
- <action>
- <conditions>
        <condition text="mode == 3 &" lvar="1" oper="3" rvar="0"
            ival="3" fval="0" />
        <condition text="timer_on &" lvar="6" oper="-1" rvar="0" ival="0"
        fval="0" />
        <condition text="timer_ringing" lvar="12" oper="-1" rvar="0"
        ival="0" fval="0" />
</conditions>
- <transformations>
        <trans text="timer_ringing := False" lvar="12" oper="1" rvar="0"
        ival="0" fval="0" bval="False" />
  </transformations>
  </action>
- <action>
- <conditions>
        <condition text="mode == 3 &" lvar="1" oper="3" rvar="0"
            ival="3" fval="0" />
        <condition text="timer_on" lvar="6" oper="-1" rvar="0" ival="0"
            fval="0" />
  </conditions>
- <transformations>
        <trans text="rem_seconds := 60" lvar="3" oper="1" rvar="0"
            ival="60" fval="0" bval="False" />
</transformations>
  </action>
- <action>
- <conditions>
        <condition text="mode == 3" lvar="1" oper="3" rvar="0" ival="3"
            fval="0" />
  </conditions>
- <transformations>
        <trans text="! timer_on" lvar="6" oper="0" rvar="0" ival="60"
            fval="0" bval="False" />
  </transformations>
  </action>
</event>
```

```
– <event ident="up">
– <action>
– <conditions>
        <condition text="mode == 1" lvar="1" oper="3" rvar="0" ival="1"
              fval="0" />
 </conditions>
– <transformations>
        <trans text="updown := True" lvar="8" oper="1" rvar="0"
              ival="60" fval="0" bval="True" />
  </transformations>
  </action>
– <action>
– <conditions>
        <condition text="mode == 2" lvar="1" oper="3" rvar="0" ival="2"
              fval="0" />
</conditions>
– <transformations>
         <trans text="updown := True" lvar="8" oper="1" rvar="0"
              ival="60" fval="0" bval="True" />
  </transformations>
</action>
– <action>
– <conditions>
        <condition text="mode == 3 &" lvar="1" oper="3" rvar="0"
              ival="3" fval="0" />
        <condition text="! fastslow" lvar="7" oper="0" rvar="0" ival="0"
              fval="0" />
 </conditions>
– <transformations>
        <trans text="rem_minutes += 1" lvar="2" oper="2" rvar="0"
       ival="1" fval="0" bval="True" />
 </transformations>
 </action>
– <action>
– <conditions>
       <condition text="mode == 3 &" lvar="1" oper="3" rvar="0"
              ival="3" fval="0" />
        <condition text="fastslow" lvar="7" oper="-1" rvar="0" ival="0"
              fval="0"  />
 </conditions>
– <transformations>
        <trans text="rem_seconds := 10" lvar="3" oper="1" rvar="0"
              ival="10" fval="0" bval="True" />
 </transformations>
  </action>
 </event>
– <event ident="down">
– <action>
```

```
– <conditions>
        <condition text="mode == 1" lvar="1" oper="3" rvar="0" ival="1"
              fval="0" />
 </conditions>
– <transformations>
         <trans text="updown := False" lvar="8" oper="1" rvar="0"
              ival="0" fval="0" bval="False" />
</transformations>
 </action>
– <action>
– <conditions>
         <condition text="mode == 2" lvar="1" oper="3" rvar="0" ival="2"
              fval="0" />
 </conditions>
– <transformations>
        <trans text="updown := False" lvar="8" oper="1" rvar="0" ival="0"
        fval="0" bval="False" />
</transformations>
</action>
– <action>
– <conditions>
         <condition text="mode == 3 &" lvar="1" oper="3" rvar="0"
              ival="3" fval="0" />
        <condition text="! fastslow" lvar="7" oper="0" rvar="0" ival="0"
              fval="0" />
 </conditions>
– <transformations>
         <trans text="rem_seconds := 1" lvar="3" oper="1" rvar="0"
        ival="1" fval="0" bval="False" />
         </transformations>
</action>
– <action>
– <conditions>
        <condition text="mode == 3 &" lvar="1" oper="3" rvar="0"
              ival="3" fval="0" />
         <condition text="fastslow" lvar="7" oper="-1" rvar="0" ival="0"
              fval="0" />
 </conditions>
– <transformations>
         <trans text="rem_minutes := 10" lvar="2" oper="1" rvar="0"
              ival="10" fval="0" bval="False" />
  </transformations>
  </action>
  </event>
– <event ident="fastslow">
– <action>
  <conditions />
– <transformations>
```

```
<trans text="! fastslow" lvar="7" oper="0" rvar="0" ival="10"
        fval="0" bval="False" />
</transformations>
</action>
</event>
– <event ident="second_over">
– <action>
– <conditions>
      <condition text="timer_on &" lvar="6" oper="-1" rvar="0" ival="0"
        fval="0" />
      <condition text="rem_seconds > 1" lvar="3" oper="5" rvar="0"
        ival="0" fval="0" />
</conditions>
– <transformations>
      <trans text="rem_seconds -= 1" lvar="3" oper="3" rvar="0"
      ival="1" fval="0" bval="False" />
</transformations>
</action>
– <action>
– <conditions>
      <condition text="timer_on &" lvar="6" oper="-1" rvar="0" ival="0"
      fval="0" />
      <condition text="rem_seconds == 1 &" lvar="3" oper="3"
        rvar="0"  ival="0" fval="0" />
      <condition text="rem_minutes > 1" lvar="2" oper="5" rvar="0"
        ival="0" fval="0" />
</conditions>
– <transformations>
      <trans text="rem_seconds := 60 &" lvar="3" oper="1" rvar="0"
        ival="60" fval="0" bval="False" />
      <trans text="rem_minutes -= 1" lvar="2" oper="3" rvar="0"
      ival="1" fval="0" bval="False" />
</transformations>
</action>
– <action>
– <conditions>
      <condition text="timer_on &" lvar="6" oper="-1" rvar="0" ival="0"
      fval="0" />
      <condition text="rem_seconds == 1 &" lvar="3" oper="3"
        rvar="0" ival="0" fval="0" />
      <condition text="rem_minutes == 1" lvar="2" oper="3" rvar="0"
        ival="0" fval="0" />
</conditions>
– <transformations>
      <trans text="timer_ringing := True &" lvar="12" oper="1"
        rvar="0" ival="1" fval="0" bval="True" />
      <trans text="rem_minutes := 0" lvar="2" oper="1" rvar="0"
      ival="0" fval="0" bval="True" />
</transformations>
```

```
</action>
</event>
– <event ident="minute_over">
– <action>
– <conditions>
    <condition text="alarm1on &" lvar="4" oper="-1" rvar="0" ival="0"
    fval="0" />
    <condition text="! alarm1_ringing" lvar="10" oper="0" rvar="0"
        ival="0" fval="0" />
</conditions>
– <transformations>
    <trans text="check_alarm := True" lvar="9" oper="1" rvar="0"
        ival="0" fval="0" bval="True" />
</transformations>
 </action>
– <action>
– <conditions>
    <condition text="alarm2on &" lvar="5" oper="-1" rvar="0" ival="0"
        fval="0" />
    <condition text="! alarm2_ringing" lvar="11" oper="0" rvar="0"
        ival="0" fval="0" />
 </conditions>
– <transformations>
    <trans text="check_alarm := True" lvar="9" oper="1" rvar="0"
        ival="0" fval="0" bval="True" />
</transformations>
</action>
</event>
– <event ident="alarm1">
– <action>
 <conditions />
– <transformations>
     <trans text="alarm1_ringing := True" lvar="10" oper="1"
        rvar="0" ival="0" fval="0" bval="True" />
  </transformations>
  </action>
  </event>
– <event ident="alarm2">
– <action>
  <conditions />
– <transformations>
     <trans text="alarm2_ringing := True" lvar="11" oper="1"
        rvar="0" ival="0" fval="0" bval="True" />
 </transformations>
  </action>
  </event>
  </events>
– <controls>
```

```
– <action>
– <conditions>
      <condition text="updown was changed &" lvar="8" oper="7"
    rvar="0" ival="0" fval="0" />
      <condition text="mode != 3" lvar="1" oper="6" rvar="0" ival="0"
         fval="0" />
  </conditions>
– <calls>
      <call text="updown_alarm(mode,fastslow,updown)" />
  </calls>
  </action>
– <action>
– <conditions>
      <condition text="mode == 1" lvar="1" oper="3" rvar="0" ival="1"
         fval="0" />
  </conditions>
– <calls>
      <call text="set_on_off_button(alarm1on) &" />
      <call text="show_alarm_time(1)" />
  </calls>
  </action>
– <action>
– <conditions>
      <condition text="mode == 2" lvar="1" oper="3" rvar="0" ival="2"
         fval="0" />
</conditions>
– <calls>
      <call text="set_on_off_button(alarm2on) &" />
      <call text="show_alarm_time(2)" />
</calls>
  </action>
– <action>
– <conditions>
      <condition text="mode == 3" lvar="1" oper="3" rvar="0" ival="3"
         fval="0" />
  </conditions>
– <calls>
      <call text="set_on_off_button(timer_on) &" />
      <call text="show_timer(rem_minutes)" />
  </calls>
  </action>
– <action>
– <conditions>
      <condition text="mode != 0" lvar="1" oper="6" rvar="0" ival="0"
         fval="0" />
  </conditions>
– <calls>
      <call text="show_fastslow(mode,fastslow)" />
```

```
 </calls>
</action>
– <action>
– <conditions>
      <condition text="mode was changed" lvar="1" oper="7" rvar="0"
    ival="0" fval="0" />
 </conditions>
– <calls>
     <call text="set_mode(mode)" />
          </calls>
 </action>
– <action>
– <conditions>
      <condition text="mode was changed &" lvar="1" oper="7"
    rvar="0" ival="0" fval="0" />
      <condition text="mode == 0" lvar="1" oper="3" rvar="0"
         ival="0" fval="0" />
  </conditions>
– <calls>
 <call text="hide_up_down_buttons ( ) &" />
     <call text="hide_on_off_button() &" />
     <call text="hide_alarm_time()" />
  </calls>
 </action>
– <action>
– <conditions>
     <condition text="mode was changed &" lvar="1" oper="7"
         rvar="0" ival="0" fval="0" />
     <condition text="mode != 0" lvar="1" oper="6" rvar="0" ival="0"
         fval="0" />
  </conditions>
– <calls>
    <call text="show_up_down_buttons() &" />
     <call text="show_on_off_button()" />
 </calls>
 </action>
– <action>
– <conditions>
      <condition text="timer_ringing was changed &" lvar="12"
    oper="7" rvar="0" ival="0" fval="0" />
      <condition text="timer_ringing" lvar="12" oper="-1" rvar="0"
         ival="0" fval="0" />
 </conditions>
– <calls>
     <call text="start_ringing()" />
 </calls>
 </action>
– <action>
```

```
– <conditions>
      <condition text="alarm1_ringing was changed &" lvar="10"
    oper="7" rvar="0" ival="0" fval="0" />
      <condition text="alarm1_ringing" lvar="10" oper="-1" rvar="0"
          ival="0" fval="0" />
</conditions>
– <calls>
      <call text="start_ringing()" />
  </calls>
 </action>
– <action>
– <conditions>
    <condition text="alarm2_ringing was changed &" lvar="11"
    oper="7" rvar="0" ival="0" fval="0" />
    <condition text="alarm2_ringing" lvar="11" oper="-1" rvar="0"
          ival="0" fval="0" />
 </conditions>
– <calls>
    <call text="start_ringing()" />
</calls>
 </action>
– <action>
– <conditions>
     <condition text="timer_ringing was changed &" lvar="12"
          oper="7" rvar="0" ival="0" fval="0" />
     <condition text="! timer_ringing &" lvar="12" oper="0" rvar="0"
          ival="0"  fval="0" />
     <condition text="! alarm1_ringing &" lvar="10" oper="0" rvar="0"
    ival="0" fval="0" />
     <condition text="! alarm2_ringing" lvar="11" oper="0" rvar="0"
          ival="0" fval="0" />
</conditions>
– <calls>
    <call text="stop_ringing()" />
 </calls>
</action>
– <action>
– <conditions>
     <condition text="alarm1_ringing was changed &" lvar="10"
    oper="7" rvar="0" ival="0" fval="0" />
     <condition text="! timer_ringing &" lvar="12" oper="0" rvar="0"
          ival="0" fval="0" />
     <condition text="! alarm1_ringing &" lvar="10" oper="0" rvar="0"
    ival="0" fval="0" />
    <condition text="! alarm2_ringing" lvar="11" oper="0" rvar="0"
          ival="0" fval="0" />
  </conditions>
– <calls>
```

```
            <call text="stop_ringing()" />
      </calls>
      </action>
    – <action>
    – <conditions>
          <condition text="alarm2_ringing was changed &" lvar="11"
          oper="7" rvar="0" ival="0" fval="0" />
          <condition text="! timer_ringing &" lvar="12" oper="0" rvar="0"
                ival="0" fval="0" />
          <condition text="! alarm1_ringing &" lvar="10" oper="0" rvar="0"
                ival="0" fval="0" />
          <condition text="! alarm2_ringing" lvar="11" oper="0" rvar="0"
                ival="0" fval="0" />
      </conditions>
    – <calls>
      <call text="stop_ringing()" />
      </calls>
      </action>
    – <action>
    – <conditions>
           <condition text="alarm1on &" lvar="4" oper="-1" rvar="0" ival="0"
          fval="0" />
           <condition text="check_alarm was changed" lvar="9" oper="7"
                rvar="0" ival="0" fval="0" />
      </conditions>
    – <calls>
            <call text="check_alarm1()" />
       </calls>
       </action>
    – <action>
    – <conditions>
            <condition text="alarm2on &" lvar="5" oper="-1" rvar="0"
                ival="0" fval="0" />
            <condition text="check_alarm was changed" lvar="9" oper="7"
                rvar="0" ival="0" fval="0" />
      </conditions>
    – <calls>
       <call text="check_alarm2()" />
      </calls>
      </action>
      </controls>
</specification>
```

Bibliography

1. Hunt, John: Guide to C# and object orientation, Springer-Verlag London Limited 2002, First published 2002.

2. Liberty, Jesse: Learning C#, O'Reilly Media, Inc. September 2002.

3. Clocksin, William F.: Programming in Prolog 5th edition Using the ISO Standar, Springer, September 2003.

4. Robert W. Sebesta, Concepts of Programming Languages. Addison Wesley Publishing Company; 5th edition (July 31, 2001).

5. Alice E. Fischer, Frances S. Grodzinsky, The Anatomy of Programming Languages, by Prentice-Hall International Edition, 1993.

6. Simon Thompson, Haskell, The Craft of Functional Programming. Second Edition, Addison Wesley Longman Limited, First published 1999.

7. John Hughes, Why Functional Programming Matters, a tutorial paper on functional programming, Computing Science Department at Chalmers University of Technology , http://www.math.chalmers.se/~rjmh/Papers/whyfp.pdf

8. Haskell homepage, http://www.haskell.org.

9. Lawrence C. Paulson and Andrew W. Smith, Logic Programming, Functional Programming, and Inductive Definitions

10. Learning to Program by Alan Gauld , http://www.freenetpages.co.uk/hp/alan.gauld/tutwhat.htm

11. Programming in Martin-Löf's Type Theory. An Introduction, Bengt Nordström, Kent Petersson, Jan M. Smith, Oxford University Press 1990.

12. Programming in Prolog. Using the ISO Standard, Fifth Edition, William F. Clocksin and Christopher S. Mellish.

13. Visual Prolog v.6 Language Reference Glossary

14. www.mta.ca/~rrosebru/oldcourse/ 371199/prolog/history.html

15. Visual C++ Language Reference

16. Designing Applications with JBuilder. Borland Software Corporation. JBuilder help documentation.

17. Introduction to Programming Using Java Version 4.1, June 2004 Author: David J. Eck, http://www.javacommerce.com/displaypage.jsp?name=index.sql&id=18216

18. Microsoft .NET Kick Start, By Hitesh Seth. Published by Sams. Series Kick Start.

19. Logic for information technology , Antony Galton ,1990

20. http://www.webopedia.com/TERM/G/Graphical_User_Interface_GUI.html

21. Event-Driven Programming: Introduction, Tutorial, History, http://Tutorial_EventDrivenProgramming.sourceforge.net, Stephen Ferg (steve@ferg.org), Version 0.2 – 2006-02-08

22. LOGIC, PROGRAMMING AND PROLOG (2ED), Ulf Nilsson and Jan Maluszy_nski, Copyright c2000, http://www.ida.liu.se/~ulfni/lpp

23. Imperative Programming - Brief version by Stan Seibert , Oct 20 2002 , http://everything2.com/index.pl?node=imperative%20programming

24. Events Programming in C# , By Shalilesh Kumar Saha, November 24, 2003 , http://www.c-sharpcorner.com/UploadFile/sksaha/EventsinNet11152005043514AM/EventsinNet.aspx

25. Terry smith, Doing Objects in VB.NET and C# - Events and Delegates , http://www.terrysmith.net/software/dotnet_ebook/chapter4.html

26. http://en.wikipedia.org/wiki/Event

27. Event Driven programming, Andrew Gregory, http://www.scss.com.au/family/andrew/pdas/psion/toolbox/tutorial/eventdp/

28. Programming C#, 4th edition (February 22, 2005) , by Jesse Liberty. Copyright © 2005 O'Reilly Media, Inc , ISBN 0596006993

29. Sams, Teach Yourself Microsoft Visual C# in 24 Hours , Understanding Event driven programming, By James Foxall and Wendy Haro-Chun

30. C# Programmer's Reference , Events Tutorial , http://msdn2.microsoft.com/en-us/library/aa645739(VS.71).aspx

31. Delegates and Events - The Uncensored Story - Part 1, By A. Abdul Azeez. http://www.codeproject.com/csharp/delegates-part1.asp

32. Learning C# 2005 , by Jesse Liberty; Brian MacDonald

33. Delegates and Events, By Kaushik Srenevasan. 17 Aug 2003 , http://www.codeproject.com/csharp/delegatesandevents.asp

34. MSDN Magazine, C++ at Work: Event Programming , Paul DiLascia, February 2006, http://msdn.microsoft.com/msdnmag/issues/06/02/CAtWork/

35. Visual C++ Concepts: Event Handling in Visual C++, http://msdn2.microsoft.com/en-us/library/aa984459(VS.71).aspx

36. Designing Applications with JBuilder , JBuilder 2005 , Borland Software Corporation , www.borland.com

37. Software Paradigms (Lesson 1) , Introduction & Procedural Programming Paradigm, http://www.cs.nott.ac.uk/~cah/G51ISS/ExternalDocuments/lesson01.doc

38. Programming paradigm , http://en.wikipedia.org/wiki/Programming_paradigm

39. Programming Paradigms Derived From ARS , http://www.lambda-bound.com/paradigms.html

40. Programming Paradigms http://www.comp.glam.ac.uk/pages/staff/efurse/Teaching/PP/Introduction.html

41. Imperative programming, http://en.wikipedia.org/wiki/Imperative_programming

42. Frequently Asked Questions for comp.lang.functional, Edited by Graham Hutton, University of Nottingham , November 2002 , http://www.cs.nott.ac.uk/~gmh/faq.html#functional-languages

43. Why Haskell matters , http://www.haskell.org/haskellwiki/Why_Haskell_matters

44. Answers.com Technology, http://www.answers.com/topic/event-driven?cat=technology

45. Event-Driven Programming , http://www.scss.com.au/family/andrew/pdas/psion/toolbox/tutorial/eventdp/

46. Access Tutorial 13: Event-Driven Programming Using Macros , http://www.cob.ohio-state.edu/~muhanna_1/837/MSAccess/tutorials/macro.pdf

47. Programming Visual Basic .NET, 2nd Edition by Jesse Liberty , O'Reilly, April 2003, ISBN: 0-596-00438-9 , http://safari.oreilly.com/0596004389/progvbdotnet2-CHP-12-SECT-4

48. Event-Driven Architecture Overview, By Brenda M. Michelson, Sr. VP and Sr. Consultant, Patricia Seybold Group February 2, 2006

49. galsC: A Language for Event-Driven Embedded Systems , Elaine Cheong , Jie Liu , March 7-11 2005 , Munish , Germany

50. Peter Bachmann. Formal Verification of Event Driven Systems , Computer Report, Cottbus University of Technology, Department of Computer Science, October, 2006

51. State-Based Model Checking of Event-Driven System Requirements, Joanne M. Atlee & John Gannon

52. Event-Based Programming Taking Events to the Limit , Ted Faison , 2006

53. Scalable Diagnosability Checking of Event-Driven Systems , Anika Schumann and Yannick Pencol´e

54. A Practical Method for Verifying Event-Driven Software , Gerard J. Holzmann , Margaret H. Smith

55. Christian Oberschulte. Refactoring of Object-Oriented and Aspect-Oriented Software: A Refactoring Browser for AspectJ in Eclipse , Diploma Thesis Department of Business Arts, Economics, and Management Information Systems at University of Duisburg-Essen, Germany , 2003 .

56. William F. Opdyke. Refactoring Object-Oriented Frameworks. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

57. C# Concisely, Judith Bishop & Nigel Horspool, 2004

58. Shimon Rura., Refactoring Aspect-Oriented Software. Bachelor Thesis, Computer Science, Williams College, Williamstown, Massachusetts, 2003

59. Introduction to Graphical User Interface (GUI). MATLAB 6.5, by Refaat Yousef Al Ashi & Ahmed Al Ameri, UAE University, College of Engineering, Electrical Engineering Department, IEEE UAE Student Branch

60. The J2EE Tutorial for the Sun ONE Platform, Data Identification, http://java.sun.com/j2ee/1.3/docs/tutorial/doc/IntroXML2.html

61. AspectJ Team: *The AspectJ Programming Guide*, Available at http://www.eclipse.org/aspectj/ , release 1.0.6, September 2001.

62. Kiczales, G.; Hilsdale, E.; Hugunin, J.; Kersten, M.; Palm, J.; Griswold, W.G.: *An Overview of AspectJ*, In Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP), Budapest, Hungary, LNCS 2072, Springer, June 2001, pp. 327-253.

63. Kiczales, G.; Lamping, J.; Mendhekar, A.; Maeda, C.; Lopes, C.V.; Loingtier, J. M.; Irwin, J.: *Aspect-Oriented Programming*, In Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP), Jyvskyl, Finland, LNCS 1241, Springer, June 1997, pp. 220-242.

64. Aspect-Oriented Software Development Web Site, Available at http://www.aosd.net, 2003.

65. Aspect Oriented Programming By Yasser EL-Manzalawy, http://www.developer.com/design/article.php/3308941

66. Parnas, D. L.: On the criteria to be used in decomposing systems into modules, Communications of the ACM, 15(12), December 1972, pp. 1053-1058.

67. Dijkstra, E.: A Discipline of Programming, Englewood Cliffs, New Jersey, USA, Prentice Hall, 1976.

68. Hürsch, W.; Lopes, C. V.: Separation of Concerns, College of Computer Science, Northeastern University, Boston, MA, Technical Report, no. NU-CCS-95-03, February 1995.

69. Aksit, M.; Bosch, J.; van der Sterren, W.; Bergmans, L.: Real-Time Specification Inheritance Anomalies and Real-Time Filters. In Proceedings of the 8th European Conference on Object-Oriented Programming (ECOOP), Bologna, Italy, LNCS 821, Springer, July 1994, pp. 386-407.

70. Ossher, H.; Tarr, P.: Multi-Dimensional Separation of Concerns and The Hyperspace Approach, In Proceedings of the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development, ed. M. Aksit, Kluwer Academic Publishers, 2000, pp. 293-323.

71. Elrad, T.; Aksit, M.; Kiczales, G.; Lieberherr, K.; Ossher, H.: Discussing Aspects of AOP, Communications of the ACM, 44(10), October 2001, pp. 33-38.

72. Working With Java And Xml , http://www.ebooklobby.com/226/Java/Working-With-Java-And-Xml

73. J. Sifakis: A UNIFIED APPROACH FOR STUDYING THE PROPERTIES OF TRANSITION SYSTEMS, Theoretical Computer Science 18 (1982) 227-258, North-Holland Publishing Company

74. Elena Bolshakova, PROGRAMMING PARADIGMS IN COMPUTER SCIENCE EDUCATION, , International Journal "Information Theories & Applications" Vol.12, 285-290

75. Aspect-Oriented Programming and Security, Rohit Sethi 2007-10-16, http://www.securityfocus.com/infocus/1895

76. Event Library: an object-oriented library for event-driven design , Volkan Arslan, Piotr Nienaltowski, Karine Arnout, Swiss Federal Institute of Technology (ETH), Chair of Software Engineering, 8092 Zurich, Switzerland, http://se.inf.ethz.ch.