Chess Mentor: A Model of Chess Developed with Empirical Modelling Concepts

by

Cem OZTURAN

Thesis
Submitted to The University of Warwick for the degree of
Master of Science

Department of Computer Science
(September 2011)

Student ID: 1061211
Supervisor: Dr. Ranko Lazic
Additional Feedback by: Dr. Meurig Beynon
# Table of Contents

**Chapter 1 - Introduction**

1.1 Motivation .................................................................................................................. 7

1.2 Thesis Overview .......................................................................................................... 8

1.2.1 A General Overview of Empirical Modelling ....................................................... 8

1.2.2 Empirical Modelling Tools used in ‘Chess Mentor’ ............................................. 9

1.2.3 Design and Implementation of ‘Chess Mentor’ ................................................... 9

1.2.4 Conclusion and Future Directions ....................................................................... 9

**Chapter 2 - A General Overview of Empirical Modelling**

2.1 Brief Background ...................................................................................................... 10

2.2 What is Empirical Modelling? ............................................................................... 11

2.3 The Key Concepts in Empirical Modelling ............................................................ 12

2.3.1 Observable, Dependency and Agency .................................................................. 12

2.3.2 Construal ............................................................................................................. 13

2.3.3 State-As-Experienced ....................................................................................... 13

2.3.4 Definitive Systems ............................................................................................ 13

2.3.5 Summary ............................................................................................................ 15

**Chapter 3 - Empirical Modelling Tools used in ‘Chess Mentor’**

3.1 EDEN ....................................................................................................................... 16

3.1.1 Brief background to tkeden ................................................................................ 17

3.1.2 Programming in EDEN ..................................................................................... 18

3.1.3 Illustrative Examples of EDEN Windows from Chess Mentor .......................... 18

3.2 SCOUT ..................................................................................................................... 19

3.2.1 Main Window Types in SCOUT ......................................................................... 20

3.2.2 Illustrative Examples of SCOUT Windows from Chess Mentor ....................... 20

3.2.3 Data Types and Operators in SCOUT ................................................................. 21

3.3 EDDI ....................................................................................................................... 22

3.3.1 Illustrative Examples of EDDI Windows from Chess Mentor ............................ 23
3.4 Summary ........................................................................................................................................23

Chapter 4 - Designing and Implementation of 'Chess Mentor'

4.1 Stage 1: Study of an Existing Model: Pjawns ..............................................................................24
4.2 Stage 2: Creation of the Board and Moving Pieces .................................................................26
4.3 Stage 3: Implementation of Piece Images and Basic Chess Rules ...........................................28
4.4 Stage 4: Implementation of Checkmate .....................................................................................33
4.5 Stage 5: Implementation of User Interface and Time Controls ..................................................36
4.6 Stage 6: Implementation of Exceptional Rules .............................................................................40
4.7 Stage 7: Modification of the Current Match through Move Notations .......................................46
4.8 Stage 8: Implementation of Different Game Modes ....................................................................49
4.9 Summary .......................................................................................................................................50

Chapter 5 - Conclusions and Future Directions

5.1 Future Directions .........................................................................................................................52
Acknowledgements

I am greatly indebted to Dr. Meurig Beynon for his guidance at every stage of my work. This thesis would not have been possible without his contribution.

I would also like to express my sincere gratitude to Dr. Ranko Lazic for his supervision and guidance.

I am also indebted to Empirical Modelling Research Group for all the work they have done so far.

I would like to thank my parents for their endless support and their confidence in me.

I would also like to thank Arda Kilinc, Elif Dilek Cakal, Gorkem Ulkar and Rusen Aktas for their invaluable friendships and being there for me whenever I needed.

Lastly, I would like to thank my family here at the University of Warwick, most notably Rakesh Sojitra, Chris Purcell, Sebastian Schindler, Donovan Wood, Simru Delen, Pak Choknitivet, Ulpan Pshembayeva, Clara Muller, Zascha Meier Watz, Madhunika Alisha Padmanabha, Mihiri Kanjirath and Hang Tian for giving me the most unforgettable and happiest times of my life.
Declarations

This thesis is presented in accordance with regulations for the degree of Master of Science in Computer Science and Applications at the University of Warwick.

It has been composed solely by the author and has not been submitted in any previous application for any degree. All work in this thesis has been undertaken by the author unless otherwise stated.
Empirical Modelling is an ongoing research programme at the Department of Computer Science, University of Warwick. Empirical Modelling is a different approach to the concepts of computer science and aims to create a construal of real life situations based on personal understanding and experience. Compared to traditional computing methods, which are more concerned about functionality and efficiency, Empirical Modelling focuses on subjective models that can be interacted in an open-ended way and be interpreted by the users at their own will. This thesis aims to create a practical and useable product, which is named ‘Chess Mentor’, reflecting the author’s own understanding and imagination while letting its users to interact with the model in many ways most of which would not be possible with a conventional programming language’s limitations. This thesis also aims to compare traditional approaches and Empirical Modelling’s innovative solutions in a practical model rather than a discussion in complete theory.
Chapter 1
Introduction

This thesis aims to create and discuss a practical and useable product which is built within Empirical Modelling ideas and approach. The model that is the outcome of this project is named ‘Chess Mentor’, which is basically a conceptualization of chess, aiming to allow users to explore it in an open-ended way. The differences between a conventional implementation and EM method are discussed.

Since the main objective of this thesis is the actual model, it can be best appreciated through interaction with the model itself. However, all aspects of the model will be explained in detail in the following chapters as well with the aid of screenshots and code extracts from the model.

Compared to most of the previous theses on the subject, this thesis focuses on the creation a practical model based on EM ideas rather than a complete theoretical discussion of the philosophy underlying EM or technical aspects of existing EM tools.

1.1 Motivation
Since I started to learn the concepts of computer science, I have been particularly interested in programming languages and developing algorithms out of the countless number of topics. The idea of creating a product out of nothing, being able to carry out experiments with it in any way I want, knowing all the inner dynamics of it and having a chance to improve it in many aspects from complexity to memory usage has always been fascinating to me.

When I was introduced to the concept of Empirical Modelling for the first time after choosing “CS405 Introduction to Empirical Modelling” module in the first term of “MSc in Computer Science and Applications” at the University of Warwick, I was fascinated by the new ideas taught. Having learnt only traditional programming languages which all are procedural languages, the Empirical Modelling approach was a total new experience for me and it evoked an intense interest in me.

As my dissertation project, I wanted to create a practical and useable product rather than conducting a theoretical research since I have always preferred to see an observable outcome of my works. When I learnt that it was possible to study with EM concepts as a dissertation project, I decided to choose this path because it was the best way to understand EM and to compare this approach with my previous programming experiences.
After deciding to develop a project in EM, I started to think about a possible model to implement. At this point, my final decision was a model about chess for a number of reasons. First of all, I have always had a strong passion about chess since I learnt how to play it when I was 4 years old. Since then, I have always played in local, national and international tournaments. I finally became a professional chess player with a chess license from Turkish Chess Federation. Secondly, my interest in chess made me buy many resources to study, such as books and chess software. I have been busy with more than 30 different chess software. Combined with my passion about programming languages and algorithms, I have always questioned and tried to understand the inner mechanisms of chess software unlike a regular user. Finally, when I observed the other projects provided in “Projects Archive” [1] and consulted to people in Empirical Modelling Research Group, I realized that a chess model would be a good choice because it would have to cover many areas of EM concepts. So, it would give me the chance to experience this new concept in the best way together with my previous chess software experiences. It could be used as an educational tool, it could allow users to experiment in many different ways unlike a traditional chess program, and it was about building a construal of a real life situation which is one of the most important aspects of EM projects.

1.2 Thesis Overview

This thesis consists of 4 main chapters excluding the introduction chapter, which are as follows:

Chapter 2) A General Overview of Empirical Modelling
Chapter 3) Empirical Modelling Tools used in ‘Chess Mentor’
Chapter 4) Design and Implementation of ‘Chess Mentor’
Chapter 5) Conclusion and Future Directions

1.2.1 A General Overview of Empirical Modelling

Chapter 2 gives a necessary background of Empirical Modelling to understand and discuss the further content of this thesis. The history of EM is explained briefly. The main ideas underlying EM are discussed to a certain extent, relating them to ‘Chess Mentor’. Also, the key concepts in the scope of EM are introduced.

Another important discussion taking place in this chapter is how and in which aspects EM differs from the traditional ways of computing. It is illustrated with the help of additional diagrams and examples. How a real life situation is construed by a modeller in a subjective manner, which is the core aspect of EM, is described.
1.2.2 Empirical Modelling Tools used in ‘Chess Mentor’

Chapter 3 is devoted to introduction of software tools and notations which are used in the design and implementation of ‘Chess Mentor’. Each software tool and notation is introduced separately, mentioning what they are specifically designed for and how they are in relationship with each other. The idea of using those notations in the concept of EM is discussed and it is explained why they are more advantageous for building models compared to traditional programming languages. Also, small examples showing how they are used in ‘Chess Mentor’ model are provided in this chapter together with code extracts from the model and corresponding screenshots.

1.2.3 Design and Implementation of ‘Chess Mentor’

Chapter 4 is the main focus of this thesis. It is devoted to explain in detail how ‘Chess Mentor’ was designed stage-by-stage, how EM principles were applied in the model, what the methodology of the model is, and what differences there would be if the model were to be implemented in a conventional programming language. The study of an existing model is explained to show the beginning stage of the design process.

Since the main objective of this thesis was the creation and discussion of a practical and useable product within EM concepts, the algorithms to model complex nature of chess are discussed and the model itself is displayed with many screenshots both illustrating its functionality and discussing the challenges encountered. There are a lot of different functionalities of the model and they all are explained in terms of implementation and ways of interaction for the users. It is mentioned in which terms the users are restricted to the model’s limitations and in which aspects they can interpret and manipulate the model in an open-ended way.

1.2.4 Conclusion and Future Directions

The final chapter of this thesis mentions the conclusions of creating such a model within EM ideas and focuses on future directions that this model can be extended to. Since a construal in EM is the outcome of a subjective understanding of a situation, some suggestions are provided on how to modify or improve the model with a different point of view.
Chapter 2
A General Overview of Empirical Modelling

It is generally problematic for people without any Empirical Modelling background to establish the link between EM approach and its principles and tools. This chapter will focus on giving a general overview of EM and identifying its key principles.

2.1 Brief Background

Empirical Modelling has been introduced to computer science as a new approach for modelling after many years of research at the University of Warwick. It has come to life in 1983 with the development ARCA (Aid for the Realisation of Combinatorial Artefacts), which is named after Arthur Cayley (1821-1895) and is a notation for interactive Cayley diagrams (Figure 2.1) [2]. ARCA was the first one of a number of definitive notations developed at the University of Warwick. The nature of the definition-based approach introduced by ARCA led to further research at Warwick and resulted in developments of other notations like DoNaLD (A Definitive Notation for Line Drawing) in 1986, SCOUT (Definitive notation for SCreen LayOUT) and even more. One of the key moments in the history of EM is the subsequent development of EDEN (the Engine for DEfinitive Notations) in 1987, which is still the primary software tool used by EM Research Group to build models via implementation of a variety of definitive notations.

![Figure 2.1 - ARCA](image)

Modelling with such tools has allowed users to get rid of some restrictions of traditional computation methods which quite often fail to provide sufficient opportunities to users for exploring the models at their own will. This has eventually led to further research and improvements until today and EM principles have continued to evolve focusing on the definitive...
notations and the concept of dependency mainly. The main principles underlying EM will be explained in more detail in the following sections of this chapter.

2.2 What is Empirical Modelling?

Empirical Modelling took the place of the term “agent-oriented modelling”. The word ‘Empirical’ is used to emphasize that the principles involved are based on observations and experience. The word ‘Modelling’ is preferred rather than ‘programming’ because EM is about building artefacts and ways of interaction with artefacts, which differs from a traditional computer program in the fact that a conventional program limits the interaction to a functionally determined behavior whereas EM provides an open-ended environment [3].

Empirical Modelling basically aims to create artefacts to support human thinking. The idea is to produce models that can reflect the modeller’s own experience and gives users a chance to interpret the model on their own. The main difference in modelling between a traditional approach and EM is that the former one has its own limitations which restrict the user in terms of functionality and exploration, which results in a quite impersonal way of modelling. The latter one, on the other hand, is a product of imagination and experience, open to many interpretations by its users. Sense-making activities resulting from the modeller’s own construal of a real life situation are the core of EM, which makes it an innovative and personal approach, unlike a conventional interaction (Figure 2.2) [4].

![EM as construction of an experience](image)

Figure 2.2 – EM as construction of an experience

In terms of practice, principle and philosophy, what EM aims to achieve can be described as follows [3]:

**Practice:** In comparison to traditional computing theory, providing a better modern computing practice is one of EM’s main purposes. EM is about building artefacts to reflect observables, dependencies, and agencies which are results of direct personal experience of the
modeller. Those artefacts become meaningful through experience and interaction, rather than through abstract states that define traditional input-output relationships.

**Principle:** EM also aims to create better principles for software development in comparison to other approaches like object-oriented design or formal methods. Experienced realities are construed differently in a personal way depending on the modellers’ understanding although the main objective might be same. So, it provides a basis for software development as a problem-solving activity in an amethodical way.

**Philosophy:** Another aim of EM is that it wants to identify a philosophical stance to address the basic issues which rise from the new developments in computing areas. According to EM approach, knowledge is based upon personal experience and thoughts. Hence, it provides a different point of view to computing concepts which may sometimes fail to find satisfying solutions to present problems through formal language and theory alone.

2.3 The Key Concepts in Empirical Modelling

After the brief background and a general overview of EM, further information about its key concepts will be given in this section.

2.3.1 Observable, Dependency and Agency

Three main concepts in EM are observable, dependency, and agency [5].

- **Observable:** An observable is a certain feature of the environment that is being modelled. The key point about an observable is that it can be attached an identity. As a matter of fact, the main requirement of an observable is that it should have a current value or status at any time. For instance, the amount of remaining time for each player in ‘Chess Mentor’ model is an observable, such as the colors of squares.

- **Dependency:** A dependency is a relationship between observables, expressing how the corresponding observables are indivisibly linked to each other. So, it basically defines how change in an observable would affect the other observables that it is linked to through a dependency. The concept of dependency differs from a regular constraint in the point that it represents modeller’s understanding and expectation about how a change in one value would affect others instead of expressing continual relationships. An important point to mention is that dependencies together with observables form the basis of a model. More detailed explanations on this topic together with illustrative examples from ‘Chess Mentor’ model will be provided in further discussion of ‘Definitive Systems’.

- **Agency:** An agent is basically an entity which is able to change the state of a model by modifying observables or relationships between them. For instance, an agent in ‘Chess Mentor’
model is able to manipulate the observables which are used to store remaining time for each player, resulting in a change in the state of the whole model.

It is important to recognize how these concepts are consistent with the discussions of EM approach in the previous sections. They all are directly based on personal observation, understanding and interpretation. A mistake which is often made is to consider observables, dependencies, and agents as variables, constraint relations and abstractions as seen in traditional ways of creating artefacts [5]. However, this misunderstanding would lead us to isolate the model itself and what it is actually about, especially according to the modeller’s experiences.

2.3.2 Construal

A construal is a notion in EM used to describe an artefact that embodies a situation through the modeller’s understanding. It is important to note that since a construal is established as a personal understanding of a situation, it cannot be considered separately from the context in which the artefact is created and the knowledge that the user has gained by interacting with it. “The evolving insight of the modeller is reflected in coherence between an abstract explanatory model, or a construal, in the modeller’s mind, the physical embodiment of this construal in the computer artefact, and a situation in the external environment [6].”

By its nature, a construal is always open to further improvements and interpretations. ‘Chess Mentor’, for instance, can be modified in many ways to reflect a different understanding of how to conceptualize chess, which is to be discussed in further chapters.

2.3.3 State-As-Experienced

The term state-as-experienced can be considered as the most fundamental concept in EM. It is used to mention the fact that the state of a construal cannot be described and evaluated in the way that a conventional computer program is supposed to be. In the case of traditional interpretation of artefacts, it is expected to have a finite space of possible certain states which are modelled by an abstract machine. A person interacting with such a model is generally expected to have a certain amount of knowledge about the model itself to be able interact with it. In contrast, this abstract structure is not the main concern in EM. The concept of dependency here has vital importance. A user can subjectively explore the model and make his/her personal observations about the changes occurring through experiments with the construal.

2.3.4 Definitive Systems

Many programming languages provide variables and procedures. A variable is basically a storage space in which some data is hold to represent corresponding objects in the program. A procedure is a sequence of instructions manipulating the data as required. A simple program
using variables and procedures can be one which allows the user to draw a square on the screen by giving coordinates of the square as input. Such a program can easily be written using conventional procedural programming languages like C or Pascal.

However, users might like to define some relations between the objects on the screen. For example, they may wish to have a circle in the center of the square no matter where the square is drawn. Using a conventional language, the programmer will need to calculate the coordinates of the circle and draw the circle again each time the square is moved to a new position. In the example of one single object, it can be manageable but when the systems get more complex, it becomes a very error-prone and exhaustive job.

Some software packages allow users to combine several objects into a single large object. It may be helpful for simple tasks but it also has extra restrictions after a certain extent. In the example program mentioned, if the user groups the square and the circle into a single object, then any translation of the square will result in translation of the circle as well. For instance, if the user doubles the size of the square, the circle’s size will also be doubled although the user may not wish to do so.

As a consequence, it is much more sensible to use abstract definitions for objects to define them in terms of other objects. For example, Variable_1 can be defined in terms of Variable_2 as below and it will be updated whenever Variable_2 is changed:

\[
\text{Variable}_1 \equiv \text{Variable}_2 + 1; \quad (\text{Note that the keyword ‘is’ used in software tools of EM, instead of the symbol ‘≡’})
\]

A system that can manage definitions is called a definitive system or definition-based system. “A definitive system is a family of variables such that the value of each variable is either specified explicitly, or is defined by a formula in terms of constants and other variables [7].”

In a definitive system, there is only one target object in each definition, which is Variable_1 in the example given above. Usually there is at least one source object, which is used to update the value of the target object. Whenever any source object is updated, the values of the corresponding target objects are computed again. By the nature of definitive systems, cyclic definitions are not allowed since a recursive definition like “\(\text{Variable}_1 \equiv \text{Variable}_1 + 1\)” would be meaningless.

Another simple example of such a case can be the calculation of the area of a square. Assuming the length of an edge is stored in the observable \(\text{edge}\), and the area in the observable \(\text{area}\), the area of the square can be calculated as follows:

\[
\text{area} \equiv \text{edge} \times \text{edge};
\]
Since a dependency is built between those observables, whenever the value *edge* is changed, the value *area* will instantly be updated without further effort. Although it is a very simple example, it helps us understand how much help we can get by using definitive systems in the design of more complex systems. One of the main advantages of using definitive systems is that the programmer does not have to remember everything which needs to be updated after a specific data has been changed. With the accurate usage of definitions, everything will be re-computed whenever it is required. Another advantage of such systems is that it helps a lot with the applications that require human-computer interaction. It allows users to see any change on the target objects immediately after a certain data is changed by the user.

However, in spite of all advantages mentioned, it is not an easy task to implement a definitive system. *EDEN* was designed as an experimental language for this specific purpose. It is basically a hybrid of traditional programming languages, which are procedural, and definitive languages. Users can define their functions and procedures using C-like statements, and the usage of definitions are also supported.

### 2.3.5 Summary

This chapter introduced the background of Empirical Modelling, a brief overview of its history, its development and the main ideas it is based on. It was explained how EM aims to create a subjective understanding of a situation as a construal built on personal experiences.

Additionally, a general comparison of conventional computing methods and EM was made to understand better how the two approaches differ from each other. Additional diagrams were provided to illustrate this comparison.

Moreover, some of the key concepts in EM were explained in theory and examples were provided to show how they are actually applied in the model ‘Chess Mentor’. The term ‘*Definitive Systems*’ was discussed in detail to explain some of the core ideas of EM.

The next chapter will introduce the software tools and notations of EM that are used in design and implementation of ‘Chess Mentor’. As well as a brief overview of those tools and notations, a number of simple examples will be provided to illustrate their usage in ‘Chess Mentor’.
Chapter 3  
EM Tools Used in ‘Chess Mentor’

In this chapter, EM tools that have been used in the implementation of ‘Chess Mentor’ will be introduced and explained. There are many different notations developed throughout the years of research for specific purposes from 3D graphics design to agent oriented parser [7]. However, only those relevant to ‘Chess Mentor’ will be mentioned in the scope of this report.

3.1  
EDEN

EDEN is the Engine for DEfinitive Notations. It is used as the primary software tool and core notation of the Empirical Modelling research group. EDEN carries out a number of definitive notations which allow models to be built. It has been developed since 1987 and is now available for different operating systems like Windows, MAC OS X and Linux.

EDEN is syntactically similar to C language. However, it differs from C (and other similar programming languages as well) in some aspects, which eventually makes it a much more convenient choice for EM. The main differences can be summarized as follows:

- EDEN supports the implementation of dependencies so that the result of a change in one observable can be easily observed in others through their relationship. It becomes even more useful in building of a model because observables can instantly be placed in definitions without being previously initialized, or even declared.

- Another advantage of using EDEN in EM is about memory issues. Unlike C, where the programmer has to take memory allocation and management into consideration, it is automatically done in EDEN. It allows the modeller to focus on the practical aspects of the model rather than dealing with the software tool issues.

- One of most useful aspects of EDEN in terms of modelling is that it supports the usage of procedural agents which can be linked to certain observables. By doing so, the modeller simply tells the system to invoke a particular procedure, which can also be considered as a function in EM terminology, whenever a change takes place in the linked observables. In a conventional programming language like C, functions have to be called explicitly when they are needed to be triggered. This is a very important advantage of EDEN and will be discussed in more detail with an illustrative example from ‘Chess Mentor’.

EDEN package contains three tools to develop models, which are tkeden, ttyeden and dtkeden. The model proposed in this report, Chess Mentor, uses tkeden as the primary tool.
Although it is beyond the scope of this thesis to discuss *tkeden* in detail, a brief overview will be helpful to provide necessary background to interact with the model itself at user’s own will.

### 3.1.1 Brief background to tkeden

This tool allows the incorporation of main definitive notations (EDEN, Scout, DoNaLD) used for building models, together with some other specific notations depending on the aspects of the model.

![Image of tkeden environment](image)

**Figure 3.1 – tkeden environment**

When the executable *tkeden* in the EDEN package is run, the environment shown in the figure is displayed (Figure 3.1). The main elements of this interaction window are:

1) *Text entry box*: It is the box where EDEN statements are written by the modeller.
2) *Accept button*: Statements in the text entry box are executed when the accept button is clicked.
3) *Notation switches*: The modeller can switch between different notations by the provided switches. The statements entered into the text entry box must be compatible with the current selection of notation.
4) *Console window*: It appears in Windows and is used to view any text output from the environment.
3.1.2 Programming in EDEN

The two main elements of programming in EDEN are the formula definitions and action specifications.

- **Formula variables:**
  A formula variable is a variable whose value is defined in terms of other variables using an expression. For example;
  \[ t \equiv \text{exp}(a,b,c) \]
  where \( t \) is a formula variable and \( \text{exp}(a,b,c) \) is an expression which contains \( a, b \) and \( c \). Here, \( a, b \) and \( c \) are called the source variables whereas \( t \) is the target variable.

  In EDEN, ‘is’ keyword is used to define a formula variable. Such formulae are valid permanently unless they are re-defined. The main difference between using formula variables and assignment statements is that the value of the target variable will always be equal to the result of the expression on the right hand side when formula variables are used.
  \[ x \text{ is } y + 5; \]
  guarantees that no matter what the value of \( y \) is, \( x \) will always be equal to \((y+5)\).

- **Action specifications:**
  EDEN provides a way to define explicit actions as well. Therefore, certain actions can be called by just changing some other data. An action specification is an object that stores an entry point (address) of an instruction sequence – the action specification. For example;
  ```plaintext
  proc warning : var
  {
    reset(var);
    writeln(var, " has been changed!");
  }
  ``
  The keyword proc defines an action, which is warning here. The action is automatically invoked by the system whenever the value of the variable var changes. In this example, the action first calls another action, reset, and var is passed as a parameter to reset. Then it simply outputs a message to the user. Here, var is not passed as a parameter to the warning action but it can be accessed since it is a global variable. An example from ‘Chess Mentor’ is given below to understand it better.

3.1.3 Illustrative Examples of EDEN Windows from Chess Mentor

The following piece of code illustrates a function from ‘Chess Mentor’ which updates the remaining time for each player during a chess game. It is also an example of a procedural agent which is linked to a particular observable:

```plaintext
setedenclock(&tick, 1000);
```
```plaintext
proc update_time: tick
{
    if (game_started == 0 || game_over == 1)
        return;
    else if (turn == "white_to_move")
        white_time--;
    else
        black_time--;
}
```

The statement "setedenclock(&tick, 1000);" tells that the observable `tick`, which is a user defined observable like all the others, will be incremented every 1000 milliseconds.

The function `update_time` is called whenever the observable `tick` changes, so it is linked to the observable `tick`, and it is invoked by the system once every second since `tick` changes at that frequency. The observables `white_time` and `black_time` stores the remaining time for each player. When the function is invoked, it first checks if the game has already started or if it has already finished by the help of two other observables in the conditional statement. In both cases, time amounts do not need to be updated. If the game is still continuing then remaining time is decremented by one second depending on whose turn it is.

### 3.2 SCOUT

SCOUT (SCreen layOUT) is a definitive notation that is used to describe screen layout of the model. By using SCOUT, the modeller can arrange the layout of the windows defined in the model. SCOUT can be incorporated with other notations. Thus, for instance, one can use DoNaLD (another definitive notation used for drawing objects) drawings in a SCOUT window. SCOUT windows can be defined as sensitive, which gives users a chance to interact with them using mouse or keyboard. So, user interfaces can also be implemented by using SCOUT windows.

SCOUT describes displays to manage the layout of the windows as below:
```plaintext
display example = < window_1 / window_2 >;
```

This means that display `example` contains two windows to be shown on the screen, which are `window_1` and `window_2`. The order of the windows in the provided display means that `window_1` will overlay `window_2` in case their coordinates overlap.

A SCOUT window can be considered as a region with some contents and attributes depending on the window type. There are 4 main types of windows used, which are text window,
textbox window, DoNaLD window and image window. All of those 4 window types commonly have the attribute *sensitive*. This particular attribute is used to declare that corresponding windows are sensitive to mouse events and strokes on the keyboard. Other attributes are mainly used to manipulate the display of the content in many ways like alignment, size, background color, foreground color, window borders.

### 3.2.1 Main Window Types in SCOUT

1) **Text windows**: SCOUT uses text windows for simple text display on the screen. The content of this type of window is a string that is defined in SCOUT itself, which makes *string* a data type of SCOUT. A number of operators is also provided to manipulate the string data type such as string length function (strlen), sub-string function (substr) and integer-to-string conversion function (itos).

2) **Textbox windows**: Textbox windows are basically used to allow users to change the content of the window. A key point here is that the attribute *sensitive* must be set to ON so that user can change the content by simply choosing the window on the screen.

3) **Image windows**: Although the functionality of image windows is very limited currently, images can be displayed by using the image file as the source.

4) **DoNaLD windows**: DoNaLD windows are very similar to image windows but their content is DoNaLD drawings instead of images.

### 3.2.2 Illustrative Examples of SCOUT Windows from Chess Mentor

An illustrative example showing two text windows and a textbox window from ‘Chess Mentor’ is shown in the figure below (Figure 3.2):

![Figure 3.2](image.png)

Figure 3.2 – SCOUT window examples from ‘Chess Mentor’

The implementations of these windows in SCOUT will be provided to observe the differences as clearly as possible. The first text window in the figure is the one telling the user to enter a number. It is a simple window with just constant string as the content. The implementation in SCOUT is as follows:

```plaintext
window question_no_win = {
    type: TEXT
    frame: ([promote_tl + [-20,150], promote_tl + [251, 170]])

Enter the question number (1-23): [23]

Done
```
The second window is the Done button. This window gives the user a chance to actively interact with the model, so it is sensitive.

```plaintext
window done_q_btn = {
    type: TEXT
    frame: ([promote_tl + {115,195}, promote_tl + {175, 212}])
    string: "Done"
    alignment: CENTRE,
    border: 3  ## border size of the window
    bgcolor: "grey"
    relief: "raised"
    sensitive: ON
};
```

The only textbox window here is used to get input from the user, where ‘21’ is entered in the figure. Since an input is taken from it, it is also defined as sensitive:

```plaintext
window question_no_inp = {
    type: TEXTBOX
    frame: ([promote_tl + {255,150}, {3,1}])
    alignment: LEFT,
    border: 1
    bdcolor: "black"  ## border color of the window
    bgcolor: "white"  ## background color
    sensitive: ON
};
```

It is worth mentioning that since Done button is used to accept the input from the textbox window, there is also a corresponding function in the model, which is invoked automatically when a click on the button occurs.

Note how the orientation of such similar windows differs with small changes in attributes.

### 3.2.3 Data Types and Operators in SCOUT

SCOUT has two more essential data types in addition to window and display, which are integer and point. They are basically used to define the regions of windows. Also, SCOUT
provides a range of basic operators to manipulate those data types. Some of the basic operators are used for integer arithmetic, construction of points, windows and displays, selectors to select a specific field of a window or a certain axis coordinate of a point.

3.3 EDDI

EDDI stands for Eden Definition Database Interpreter. It is a definitive database notation. Although it is contained in EDEN package by default, it has to be enabled in an EDEN tool once the tool is started.

In EDDI, there is a number of data manipulation and data definition commands. Five basic data definitions are used for creating relation tables, adding or deleting tuples from tables, and dropping or truncating tables.

EDDI also supports six relational operators to be used in relational expressions, which are union (+), intersection (.), difference (-), projection (%), selection (:), and join (*). The operators union, intersection, and difference are used on two tables to get the tuples that take place in either of the tables, in both tables, or in only one of the tables. The projection operator is used to select certain columns from a table. The selection operator is used with a predicate and returns the tuples that satisfy the provided predicate. The last operator, join, returns tuples that are pairs of tuples from two different tables and have common attributes.

EDDI supports both assignment operator (=) and definition (is) which can be used to define tables as below:

\[
\text{table}\_\text{name} = \text{relational expression};
\]

or

\[
\text{table}\_\text{name}\ \text{is} \ \text{relational expression};
\]

As the nature of empirical modelling suggests, the value of the variable on the left-hand side in the assignment case will be assigned to the current value of the right-hand side at the time of assignment. However, the value of the left-hand side in the case of a definition will always be maintained to be up-to-date value of relational expression on the right-hand side. An important point here is that cyclic definitions are not allowed in EDDI.

In the current versions, there are a few bugs and limitations of EDDI. The most basic limitation is that it does not have any support for defining keys or constraints in relations as many database systems do. Also, it may sometimes fail to give users feedback through the dos console in Windows.
An example usage of EDDI from ‘Chess Mentor’ is given in the next section to show it more clearly.

### 3.3.1 Illustrative Examples of EDDI Windows from Chess Mentor

The following piece of code is written in EDEN tool so, as mentioned previously in this chapter, EDDI has to be enabled manually although it is included in the package by default. A number of statements are *executed* in the order they are written. First of all, EDDI is enabled. Then, we move to EDDI notation from EDEN notation so that we can create a table to store the data. The database table here is *table*, which has 4 columns for each tuple. 3 columns store integer values, *move_no, from*, and *to*. Last column, *captured*, stores a string. After the creation of database table, EDEN notation is chosen again.

```plaintext
execute("installeddi();\n" //
"%eddi\n" //
"table (move_no INT, from INT, to INT, captured CHAR);\n" //
"%eden");
```

Note that it is required to go back to EDDI notation to store tuples in the database table.

### 3.4 Summary

This chapter was devoted to introduction of EM tools that are used in the design and implementation of ‘Chess Mentor’. A brief overview of each notation and tool is given, explaining their importance in the process of building a model. The main differences between a traditional programming language, C, and EDEN, the primary notation used by Empirical Modelling Research Group for creating models are discussed.

The next chapter will be the main focus of this report, which is the design and implementation of ‘Chess Mentor’ model. It will be analyzed as a practical and useable product developed with an Empirical Modelling approach.
Chapter 4
Design and Implementation

This chapter discusses the design and implementation of ‘Chess Mentor’ model in detail. A stage-by-stage approach is carried out through the chapter, so it can also be considered as a timetable of the development progress.

Each and every part of the model is illustrated with screenshots and pieces of code and is explained in terms of methodology, algorithms, and different parts’ relationships with each other.

It is worth emphasizing that the model can be best appreciated through interaction with the model itself together with the necessary background provided in this thesis report.

4.1 Stage 1: Study of an Existing Model: Pjawns

First stage of the development process was to study of an existing model to get familiar with the practical applications of EM. For this purpose, the model Pjawns was chosen. ‘Pjawns’ (Figure 4.1) was developed as a coursework assignment 2002-2003 'Introduction of Empirical Modelling' course by John Martin (4 year MEng, 1999-2003) [9]. It is designed as the model of a fictional board game which is similar to draughts but containing some ideas from chess as well. The idea is to place pieces on the board, and move them until the last rank of the board like pawns in chess.

![Figure 4.1 – Pjawns by John Martin](image)
The study of this model was helpful for a number of reasons. The most basic reason is that it is a practical example of a board game, which is the basis of ‘Chess Mentor’ model. Additionally, it contains some similarities to ‘Chess Mentor’ especially in terms of mouse triggered events and user interface. Choosing and moving a piece by clicking and interacting with the model via user interface are the most common aspects of ‘Pjawns’ and ‘Chess Mentor’.

However, ‘Pjawns’ actually has much less concerns than ‘Chess Mentor’ in terms of implementation. The most obvious reason is that it has a single type of piece, which can move just a single square in only one direction. The only thing to take into consideration is that pieces can move two squares in their first move. Another uniform aspect of ‘Pjawns’ is that there is only one game type, which is always played with exactly same rules and without time controls for players. Since there is only one type of piece in ‘Pjawns’, the deeper analysis of the model showed that there are actually pieces already drawn on each square, and only the color of the piece is changed so it is visible to the user when required. It is a very neat trick to visualize the arrangements of the pieces on the board for such a model. Nonetheless, since there are many types of pieces in chess, it is not applicable to any chess model and requires further complexities.

Moreover, special conditions like checkmate, castling, move repetition, removing a threat on the king, pinned pieces are of no concern to ‘Pjawns’, which are of vital importance in a chess game. There is no position on the board that requires the game to end in a draw in the case of ‘Pjawns’, while a chess model has to consistently check if a rule to end the game in a draw has already been satisfied or not. The modeller implemented a very good feature taking advantage of fairly simple rules of this fictional game, which is a basic AI player. It is a very well thought feature of this model, and allows users to further explore the model. Unfortunately, implementation of AI requires an extensive amount of work to be applied to a chess game. It is, of course, always possible to easily implement an AI player that makes a random move out of possible moves, but it is an extremely challenging task to take it a step forward.

Another simpler aspect of ‘Pjawns’ compared to ‘Chess Mentor’ is that its interface does not give users much chance to manipulate the game through the interface. Players cannot go back to a certain position and modify the game at that point. It can always be achieved by interacting with the _tkeden_ console, but it requires users to have knowledge about the observables of the model.

It should be emphasized here that although ‘Pjawns’ is a simple model of a fictional board game, it is a very well designed model reflecting the modeller’s own imagination and stream of thought. It is a quite robust model which is also consistent with EM principles discussed previously and focuses on a practical outcome allowing users to conduct their own experiments with it.
4.2 Stage 2: Creation of the Board and Moving Pieces

After the study of ‘Pjawns’, next step was the drawing of the board and triggering of mouse events on the pieces.

An important aspect of the model is that squares are numbered from 1 to 64 on the board, as depicted in the figure (Figure 4.2) below. It is important to note this to understand the further discussions in the following sections more clearly which are about the implementation of the model. Also, row (1-2-3-4-5-6-7-8) and column (A-B-C-D-E-F-G-H) labels are to be noted because they are used to describe the moves in a chess game.

![Figure 4.2 – Numbering of squares in ‘Chess Mentor’](image)

The drawing of the board was done using SCOUT notation. Since the user is supposed to click on the squares to choose a piece and move it, and since each square can be interacted individually, 64 different SCOUT windows are used for board. An example of a single square’s implementation is given below to see the content and attributes of each square:

```javascript
window square_11 = {
  type: TEXT  # The content is a string which is the piece’s name.
  frame: ([o11, [o11.1 + square_width, o11.2 + square_height]])
  border: 1
  bgcolor: bg11  # Color of the square
  sensitive: ON  # Must be sensitive to trigger actions
  string: p11  # Stores the piece name, which is on this square.
  ""  # "" is stored if the square is empty.
};
```
*Note that the color of a square is not a constant string, but a user-defined observable although a square will always stay as a white or black square. The reason is that when a square is chosen or if it is a possible square for the chosen piece to move, it is in some conditions highlighted particularly to give user a better chance of interaction. The color of the square and the name of piece are actually EDEN observables. So, they are evaluated through EDEN functions but can easily be used in SCOUT definitions thanks to the compatibility of the two notations.

At this stage, I was unable to show the pieces visually because of a bug that the current version of EDEN package has. The problem was that the size of chess piece images was not fitting into the squares and it would not look proper if squares were to be made any bigger, so I had to scale the images using ImageScale function provided by SCOUT. After spending some time on the issue, it was realized that scaling images in SCOUT caused telden to crush regardless of the scaling value. So, until figuring that out, instead of piece images, the model had abbreviations of chess pieces’ names on the corresponding squares.

After drawing the board, I started to implement the user interface as well. A button to start a new game and a button to rotate the board were inserted into the model. These initial steps made me familiar with how to manipulate the model itself through the interface and how EDEN and SCOUT notation are related to each other.

As the last step of this stage, I implemented mouse events for all 64 squares. So, a click on a square highlighted it as the chosen square, and a second click on another square moved the content of the initial square to the content of the destination square. This was the basis of a move which took me to the next stage of development process. The final product of this initial stage is depicted in the next figure (Figure 4.3).
This figure shows that a white pawn (W.P) is moved to f4 square from f2 square, and the square having white king (W.K) is chosen by the user.

4.3 Stage 3: Implementation of Piece Images and Basic Chess Rules

The next stage was the most critical one which is the implementation of chess rules. However, since there are a lot of chess rules, the first priority was to move each piece according to its own regulations.

Since squares are numbered from 1 to 64, I stored the current position on the chess board in a list observable, current_position. It has 64 elements, one for each square, and stores the name of the piece on each square whereas an empty string stored if there is no piece on the square. Each piece name consists of 3 characters. The first character denotes the color of the piece, second character is an underscore, and the last character denotes the type of the piece.

Before implementing the game rules, I first converted the piece names into images. Since I could not re-scale the images using SCOUT, I re-scaled them using an external tool before inserting them into the model. Then I linked piece names to the image files in SCOUT, so they were automatically converted to images in the model after changing window types to image windows instead of text windows.

The algorithm followed here to move a piece is that the name of the piece is found from the current_position list, using the number of the clicked square. Then the square number is given as parameter to a function, find_possible_square. The piece name does not be passed as a
parameter since it is a global value. Only local value that needs to be considered is the initial square of the piece. Then the function works for each piece as follows (Refer to figure 4.2 for square numbers):

**Pawns:** If a pawn is on its initial square, then it can move one square forward (meaning that destination square number = initial square number + 8 for white player) or two squares forward (destination square number = initial square number + 16 for white player). If there is a piece on any destination square, then the pawn will be blocked and will not be able to move to that particular square. Note that rules are exactly for both players, so switching the ‘+’ operator used for white player to the ‘-’ operator will give the equivalent calculations for black player. Unlike it actually moves, a pawn captures opponent pieces diagonally, one square forward and to the left or to the right. Here it gets tricky and another function is required. The problem here actually applies to all other pieces except for rooks, so it is worth explaining in detail. The trick in such a position is that when a pawn is said to capture a piece, it is expected to have following calculations for white player:

- Destination square number = Initial square number + 7 (Diagonal on the left side)
- Destination square number = Initial square number + 9 (Diagonal on the right side)

However, if the pawn is on the rightmost (H column) or the leftmost column (A column), it ends up on a wrong square. For instance, a pawn on square h5 (square number 40) would be expected to capture opponent pieces on squares 47 and 49 according to the mentioned calculations but square 49 is actually somewhere impossible for the pawn to move (Figure 4.4).

![Figure 4.4 – Column problem in piece movements](image)
In the figure 4.4, the pawn on d2 (square 12) can capture on squares 19 and 21 as expected, but the pawn on h5 (square 40) should not be able to move to a7 (square 49).

To overcome this situation, an additional function was implemented for pawns’ captures so a pawn can capture only in one diagonal direction if it is on column A or column H. Other pieces except for rooks also suffer from the same problem with little in differences in calculations and they will be explained in their own section.

At this stage, although a pawn was able to move properly, I did not implement all chess rules regarding pawns like en passant (which is a rare way of capturing which can be done only in certain positions with no other pieces but pawns), or pawn promotion (meaning that a pawn can be promoted to any piece other than king if it can be moved to the last row of the board) because they required many additional functions and would be difficult to implement at this stage. The main difference between the modelling of pawn movements and other pieces is that pawn is the only piece that moves and captures in different ways.

**Knights:** The movement of knights was fairly easy compared to the other pieces because knights can jump over other pieces and cannot be blocked. This unique feature gives a chance to simply calculate the set possible destination squares by mathematical operations. However, as in the pawn case, knights at the edges of the board require an additional function. Unlike pawns, knights have more than one possible square that they can go to, so instead eliminating false destination squares exhaustively one-by-one, a helper function is implemented, which takes the column of the knight as input and returns the possible columns that it can move to. Since the distance that a knight moves is always constant in chess, this simple approach works fine for knights.

**Bishops:** Bishops can move diagonally in all 4 diagonal directions. Although it seems quite easy to do, it gets tricky when it is actually implemented. One problem is that in each direction, the number of the current square has to be updated in a different way. When figure 4.2 is referred, the following calculations are required for each direction:

- Destination square number = Initial square number + 7 // Forward and Left
- Destination square number = Initial square number + 9 // Forward and Right
- Destination square number = Initial square number - 9 // Backward and Left
- Destination square number = Initial square number - 7 // Backward and Right

Note that these calculations are valid for just one square in each direction. They are repeated until the edges of the board are reached.

Another problem with bishops is the one that was mentioned for pawns and knights. When the edges of the board are reached, calculations begin to result in invalid destination
squares. Moreover, the solution that worked for pawns and knights does not work for bishops. The reason is that a bishop on a given column can move to any column on the board since it moves as many squares as it wants in a certain direction. So, the previous solution of eliminating invalid columns needed to be modified for bishops. The modified version uses the initial square number and recursively checks if the edges of the board are reached in any of the 4 diagonal directions. It moves the bishop just one square in each direction and adds reached squares to the set of possible squares. If it encounters a piece or reaches to the edge of the board for a particular direction, then it stops processing that direction and goes on with the others, which allows us to find all squares that a bishop can safely move to.

**Rooks:** Rook is the only piece that cannot end in a wrong column since it moves horizontally and vertically, as many squares as it wants until it encounters a piece or board edges. Only trick when moving a rook is that, like bishops, it moves in 4 different directions and each direction requires a different calculation in terms of square number. By making sure that it stays in the same row, or the same column, rook movement is easily implemented.

A unique effect of moving a rook is that after a rook is moved, it cannot be used in castling, which is a special of move that can be done only once in a game with the king and one of rooks. This will be discussed later in further stages.

**Queens:** A queen can move in any direction, diagonally, vertically or horizontally, and as many squares as it wants until a piece or edges are reached. Although it might seem more difficult than other pieces, it is simply a combination of bishop and rook movements, with no effect on any other rules of the game, so it was very easily implemented.

**Kings:** King’s movement was by far the most difficult one to implement for a number of reasons. The most important reason is that a chess game is won when the opponent’s king is checkmated. Checkmating the opponent’s king means that there must be a present threat on the king, and no matter what move your opponent makes, his/her king must still be threatened. In chess, it is not allowed to intentionally move your king into an already threatened square. So, unlike any other piece, the set of possible squares for a king to move must be processed by another function and the squares that are threatened by the opponent player must be removed from this set. This is a quite challenging task since we need to check if a particular square is currently threatened by any one of the opponent’s pieces, so the helper function must be invoked for each opponent piece.

Since king moves one square in any direction, it also has the problem of ending in a wrong column but the solution for knight works fine with king as well since both have a limited move range.
- **Castling:** Also, a king can initiate the castling move. Castling is a special move in which the king moves two squares towards one of the rooks, and the corresponding rook is moved to the square next to the king on the other side. The thing which needs attention is castling can be done only once by a player in a game, it cannot be done if the king has been moved before, it cannot be done with a rook that has been previously moved. Also, all the squares between king and rook must be empty, and most importantly king’s initial square, destination square, and the square that it crosses over must be threatened by opponent. This move requires many helper functions to implement. First of all, three observables are required to see if the king or either of the rooks has been moved before. Then we need to check if there is any piece between king and rook. Finally, we need to check if any of the squares used by the king in this move is threatened or not. What I did here is that, I had two observables for each player (can_white_do_short_castling and can_white_do_long_castling) representing each one of the two directions that the king can do castling (these directions are called as long castling and short castling in chess terminology) to check if castling is allowed in a particular direction or not. All previously mentioned factors are processed in a single function at each move, and the function changes the values of can_white_do_short_castling and can_white_do_long_castling accordingly.

An advantage of using these two observables is that they are stored in a list after each move so that when the user wants to manipulate a game by going back to a position, or saving/loading a game, castling will work properly. It is needed because we cannot tell by just looking at the board position if a player has lost his chance to castle or not, so the values need to be saved for each move.

A screenshot illustrating the concept of castling is given in the figure below (Figure 4.5):
The figure on the left shows that the king was chosen by the white player, and possible squares are highlighted in pink. As it can be observed, the king can move to c1 from e1, which is the castling move. However, it cannot move to g1 for castling since it is threatened by the black bishop on d4. After white does long castling, we have the position depicted in the figure on the right. Also note that f2 is also not highlighted on the left since it is threatened by the bishop as well. Showing the possible squares in a different color provides users a chance to explore the model without prior knowledge and guides them to come up with their own personal experiences.

4.4 Stage 4: Implementation of Checkmate

After the implementation of basic rules, I preferred to implement the concept of checkmate rather than additional functionalities through user interface or exceptional cases like en passant, stalemate, insufficient material, etc.

As mentioned previously, checkmate decides the end of a game in favor of one player. It means that no matter what losing side plays in the next move, the king will still be on a threatened square, available to be captured by the winning side.

Detecting the Threat on the King

The first step here was the recognition of a threat on the king, which is expressed as check in chess terminology. Although it is a tricky task, I wrote a function which takes a single square number as input, and outputs if it is threatened by a certain player or not. Since I had to find all threatened squares by a player in the previous stage to move a king, I could easily detect the threat on a king.

Removing the Threat on the King

After recognizing a check without difficulty, the challenging part was next, which is how to remove the threat. There are 3 ways of removing a threat on the king and each of them is implemented in the model:

1) Moving the king out of threat: Since king is currently under attack, a possible way is to simply move it to a safe square.

2) Capturing the checking piece: Another way is to capture the attacking piece. It can be captured by any piece, including the king.
3) **Blocking the checking piece:** The last and most complicated way is to place a piece to a square between the king and the attacking piece.

To move the king to safety, I simply used my function that I previously implemented for king movement. For the second and the third ways mentioned above, I had to use some helper functions. I wrote a function, `find_path_to_king`, which finds all the squares between the attacking piece and the king. It returns a list, `path_to_king`, containing set of squares on the path to king, which are the squares between the king and attacking piece. Then I added the attacking piece’s own square to this list as well, so the last two ways mentioned were combined together. Then, whenever threatened player chooses a piece, all possible squares for that particular piece is found first as discussed previously. After that, the set of those possible squares are intersected with the list, `path_to_king`, so that we are only left with the squares that can either capture the attacking piece or be placed in between blocking the threat (Figure 4.6). Note that `path_to_king` does not need to include the squares in between if the attacking piece is a knight since it can jump over other pieces.

![Figure 4.6 – Getting rid of check](image)

In positions shown above, white queen on e5 threatens the black king on e8. The figure on the left shows that the black queen can either capture the white queen or move to e6 to block the threat. `path_to_king` here is \{e5, e6, e7\} so all other squares for black queen to move are eliminated. The figure on the right shows the safe squares for the king to move.

Although this method works perfectly on a single threat on the king, it is insufficient in the rare case of two threats at the same time. In such a case, we are left with only one way to get rid
of threat, which is to move the king to safety because two threats mean two completely different paths to the king, so no other piece can block both threats at the same time (Figure 4.7).

![Figure 4.7 – Multiple threats on the king by a discovered attack](image)

The figure on the left shows a certain position reached in the game. White moves his bishop on e2 to h5 and threatens the black king on e8. Also a discovered attack takes place by the white queen on e1 since the bishop is now out of the way. Since it is not possible to block threats on two different paths with no common square, the only way is to move the king to a safe square (d7 or f8 in this case) away from both threats.

There is one last tricky part while creating the set of threatened squares by the checking piece. Normally, we determine the possible squares using simple mathematical operations depending on the piece type. We add squares to the list of possible squares until we reach another piece or the board edges. However, if the piece that we encounter while moving in a particular direction is our opponent’s king, then we have to add the square behind the king to the set of possible squares as if the king were not in our way at all (Figure 4.8).
Figure 4.8 – Expanding the set of possible squares on the other side of the king

Figure 4.8 shows a position where the white king on e1 is checked by black rook on h1. When calculating possible squares for the rook to move to, we move towards left one square at a time until we reach a piece or the board edge. After adding g1 and f1 to the set of possible squares, we reach the white king. Since it is an opponent piece, we add e1 to the set as well. If it was a piece other than king, we would be done with this particular direction to move the rook and square d1 would be out of reach for the rook. However, since the piece we reached is the king, we should move one more square in our direction and add d1 to the set. Otherwise, the king would consider d1 as a safe square to move because it would not be in the list of threatened squares by black player.

After being able to detect and escape check, I finally coded the checkmate situation. This part was a straightforward implementation since the necessary conditions for checkmate are satisfied if king has no safe square to move to and if no other piece can move to a square on the path_to_king list.

4.5 Stage 5: Implementation of User Interface and Time Controls

After implementation of checkmate, the rules for a chess game to be played properly were almost completed. Before moving to exceptional rules of the game, which would be more challenging compared to previously completed parts, I decided to finish user interface’s implementation.

Since I knew what kind of functionalities I had planned to add to the model, this stage was done without getting into further complexities although it took a long time to implement because of large numbers of windows needed (Figure 4.9).
Figure 4.9 – User interface

Figure 4.9 shows how the model looked after user interface is completed. Each section of the interface is described below:

**Captured Pieces:** Total numbers of captured pieces are shown for each piece type under the chess board. I actually decided to show the captured pieces in this way due to the nature of Empirical Modelling. Most conventional chess programs normally show each captured piece separately. So, for instance, the user is shown three pawn images next to each other if three pawns are captured in total. It works fine for a traditional way of modelling because the number of pieces in a chess game is always 32 and since the user is limited to functionalities of this conventional modelling approach, the modeller can simply implement a constant number of image windows to display captured pieces. However, this solution would not work for an Empirical Modelling approach. Note that one of the core ideas of EM is ‘gaining knowledge through personal exploration and observation’. Users should have a chance to manipulate the models without limitations. In case of ‘Chess Mentor’, for instance, a user can modify the position on the board at any time via `tkeden`. Remember that the observable `current_position` was defined as a list of 64 elements, containing the pieces on each square. So, a user can edit the contents of `current_position` and add more and more pieces to the board (Figure 4.10).
Figure 4.10 shows how a user can manipulate the model’s state. In the initial position of a chess game, the user enters some EDEN statements in tkeden, which add white pawns (“w_p”) to the squares from a3 (square 17) to h3 (square 24). As soon as the user clicks “Accept” button, 8 pawns are added to the game and the state of the model changes.

This example clearly shows how EM differs from a conventional way of modelling in the aspect of personal exploration. Because of such limitless interactions and the impossibility of having an infinite number of image windows, the captured pieces are shown in terms of numbers under the corresponding piece type in ‘Chess Mentor’. So, no matter how many pieces a user adds to the model, all captured pieces can be shown properly.

**Game Status:** This window helps users to keep track of the current game. Player names for each color are shown, which will be taken as input in the further stages of implementation. Time controls for each player are also displayed here. As briefly explained in discussion of EDEN in Chapter 3, an observable is used here together with EDEN clock to trigger a function once every second. The function decreases the time of the next player to move by 1 second if the game is on.

In the middle part of ‘Game Status’ window, the move notations are displayed. This type of move notation is used worldwide for chess games. Each move is shown in a separate SCOUT text window. By the help of these windows, the users can follow the game and understand how the current position on the board was reached at any time.

One problematic part here is that since a chess game can end quickly or go on for a large number of moves, we cannot predict how many windows we need. I could not succeed to dynamically create new text windows whenever needed and use a scroll bar to reach them. Instead, I implemented a total of 30 text windows, which can display 15 moves for each player. When the
game keeps going on, the content of the windows are shifted and always last 15 moves of each player are displayed.

‘Game Status’ window will be discussed in further stages again since it allows users to change the game by interacting with move notations in the final version of ‘Chess Mentor’.

**New Game:** This button (Note that each button is actually a SCOUT window) simply resets everything in the model and starts a new game at this stage. Later, it will be modified in terms of functionality when I implement two different game modes, which are “Training Mode” and “Official Mode”.

**Chess Academy:** This section contains 23 chess problems. Some of the problems are my own design while some are taken from external sources [10]. Users choose the question number that they want to solve, and are given 5 minutes to solve it (Figure 4.11).

![Figure 4.11 – Chess Academy Section](image)

Some of the questions are quite challenging even for experienced players, so some hints are provided if users fail to solve them at their first trial (Figure 4.12).
4.6 Stage 6: Implementation of Exceptional Rules

In this stage, I implemented a number of exceptional rules to the model. They are discussed in more detail throughout this chapter.

**En Passant:** ‘En Passant’ is a special pawn capture which can occur immediately after a player moves a pawn two squares forward from its starting position, and an opposing pawn could have captured it, had it moved only one square forward [11].

En Passant must be played immediately after the pawn has moved two squares. If a turn passes, En Passant can no longer be played on that pawn [12].

The rule itself is fairly easy to implement for a single move. However, the fact that it has to be played on the next move when an opportunity occurs makes the implementation problematic. By analyzing the position on the board, it is not possible if there is a possible En Passant move or not. We need to know the last move of the opponent player for this. So, I used an observable to
store the square number on which En Passant is possible. If there is an opponent pawn next to the
En Passant square in adjacent columns, then En Passant is possible. After each move, the
observable storing En Passant becomes 0 so the opportunity is lost in the next move if En Passant
move is not played (Figure 4.13).

In the Figure 4.13, black played his pawn from e7 to e5, and white has a chance to capture it by
moving his pawn on d5 to e6 as if the black pawn were on e6 (En Passant). However, this
opportunity is lost if white chooses to play another move (Figure 4.14).

Figure 4.13 – En Passant (1)

Figure 4.14 shows the position after white refuses to make En Passant move, and simply moves
the pawn on a2 to a3. When black replies with a7-a6, then white does not have the chance to
capture the pawn on e5 anymore as it can be observed by the difference of highlighted squares in the figures above.

**Stalemate:** ‘Stalemate’ is a special position in chess where the player to move has no legal move and his/her king is not threatened. In this case, the player cannot make any move and the game ends in a draw [13].

I used the function which was discussed previously and is used to find possible squares for a certain piece to move to. I called the function for each piece of the player who is supposed to make the next move in the game, and appended all possible squares for each piece to a single list. Then I check the number of elements (which are square numbers for the possible moves) in the list and if there is a threat on the king or not. If there is no element in the list, which means no piece has a legal move, and if the king is not attacked, the game ends in ‘Stalemate’ (Figure 4.15).

![Figure 4.15 - Stalemate](image)

In Figure 4.15, white king is not checked but all squares around the king are threatened by black rooks. White’s only other piece, pawn on h8 is also blocked by black king and has no legal move. The game ends as a draw.

**Pawn Promotion:** A pawn can be promoted to any piece other than king if it can reach the last row of the board. This is a unique case in the model because after the user moves the pawn to its destination square, he/she needs to choose which piece to promote to and the list for threatened squares or any check condition has to be checked after the promotion (Figure 4.16).
Figure 4.16 shows how pawn promotion is modeled in ‘Chess Mentor’. White player moves his pawn from h7 to h8 and a menu pops up to allow the player to choose which piece to promote to. After the choice is made, pawn is promoted and the model’s state is updated accordingly.

One issue to take into consideration in pawn promotion concept is that if the promoted piece is captured at some point, it must be considered as a pawn capture in chess. So, for instance, if a player promotes two pawns to queens and if the opponent captures them, the opponent is said to capture two pawns rather than two queens. The effect of this situation in the model is about ‘Captured Pieces’ window. To show correct number of captured pieces, I changed the names of promoted pieces in such a way that I could always know which pieces were originally pawns in the initial position. Assuming a pawn was promoted to a queen, the change in piece name is that “w_p” becomes “wrq”. As discussed earlier, the first character and the last character represent the color and the type of the piece, respectively. The second character is an underscore for regular pieces and an ‘r’ character for promoted ones.

Also, the move notation shown on ‘Game Status’ window is changed accordingly. Instead of “h7-h8”, the notation is updated to “h7-h8=Q” indicating the details of the promotion.

**Insufficient Material:** The game ends in a draw if there is not enough material on the board to make a checkmate possible. For this, I check the number of pieces on the board after each move. Since kings have to be on the board all the time, at least one of the two players has to have more than 2 pieces on the board, including the king. In some cases, 2 pieces are also sufficient if the second piece is queen, rook, or pawn.

Some endgames (for instance, one pawn and king against a single king) are proven to end in draw against correct defense. However, it does not satisfy the requirements for an insufficient material draw since a checkmate is mathematically possible.
**3-Move Repetition:** The game ends in a draw when the same position occurs 3 times on the board. In most of the simple chess applications like the ones running on mobile phones, this rule is simplified as “the game ends in a draw when the same moves are repeated 3 times in a row by both players”. In ‘Chess Mentor’, I also followed the same strategy and the rule is satisfied when the same moves are repeated 3 times in a row in the final version of the model. The problem with this solution is that the exact same position can occur at different times after different sequences of moves. The model, as well as many simple chess applications, fails to detect this situation. In “Future Directions” chapter, some suggestions on the implementation of this rule will be discussed.

**Pinned Pieces:** Implementation of the effects of pinned pieces was the most challenging one among all the exceptional cases mentioned in this chapter. It is also the most important one as well because unlike other exceptional rules, the condition of pinned pieces occurs very frequently from the opening until the end of the game.

A piece in chess is said to be *pinned* if it is attacked by the opponent and if it exposes a more valuable piece to the attack when the pinned piece moves to safety (Figure 4.17).

![Figure 4.17 – Pinned Pieces (1)](image)

In the depicted figure, the white bishop on f6 is pinned by the black rook on f8 because if it moved, the white king would be exposed to the attack of the rook. Since it is pinned, there is no highlighted square in pink on the board as it cannot move anywhere. Also note that the white pawn on e4 pinned by the black bishop on c6 as well.
Despite exposing a more valuable piece is considered enough to call a piece \textit{pinned}, pinned pieces are allowed to move if the exposed piece is not king. However, they cannot expose the king under any circumstances.

It might seem easy to implement this condition. The modeller can simply check if the king is on the other side of the pinned piece and decide that the pinned piece cannot move. Nevertheless, the most problematic part is that, in some particular cases, pinned pieces can actually move without exposing the king (Figure 4.18).

As it can be seen in the figure above, the white rook on f1 stands between the white king on d1 and the black rook on h1. If the white rook was not there, then the king would be exposed to the attack so the rook is pinned. However, the rook can still block the threat so long as it moves horizontally.

This example can be applied to bishops diagonally, to rooks both vertically and horizontally, and to queens in all directions. My algorithm here was to process all possible squares of the pinned piece as if it were not pinned at first. So, I found all the squares that it could move to if there was no pin. Then I created a temporary board with the exact same position as the original one, and placed the pinned piece on each one of its possible squares. The squares exposing the king were eliminated from the set of possible squares, so the correct set was found.

There are many possible ways to implement a solution for this case. There might be more efficient solutions which can be based on the types of pinned piece and the attacking piece. Still, I preferred my algorithm because same functions worked correctly for all piece types.
4.7 Stage 7: Modification of the Current Match through Move Notations

Next thing in the development process was to allow users to modify a current match. I wanted to provide a chance to users of the model to go to any position which occurred in the game and change their moves if they want to.

For this purpose, instead of implementing additional buttons to the user interface, I made move notation windows in ‘Game Status’ sensitive. So, in ‘Chess Mentor’, a mouse click on a move notation displayed on the screen transforms the game into the position at the particular moment that the clicked move was made (Figures 4.19 – 4.20).

![Figure 4.19 – Going to a previous position in the game (1)](image1)

![Figure 4.20 - Going to a previous position in the game (2)](image2)
In the figures shown, the user goes back to a previous position (after the third move of black player) from the current position (after the sixth move of white player). The user can go to any position in the game by clicking on the moves listed in the move notation list on the screen. So, it is possible to go back to a previous, and then come back to the current position, or any position in between.

When the user changes position on the board through such kind of interaction, he/she can also change the next move, which will result in a different game. Since a change in a previous move will change the remaining of the game, move list will be modified so that the moves played after that particular position will be excluded (Figure 4.21).

![Figure 4.21 – Changing a move in a previous position](image)

So, the user decided to change the fourth move of the white player, game and the move list is updated accordingly to reflect the new state of the game.

Implementation of this feature was quite difficult than expected. The most problematic issue here is that, in a chess game, there are some very important factors which cannot be evaluated by only examining the position on the board.

By storing the initial squares and destination squares for each move, I could easily manipulate the position on the board at first. Then, the issue of captured pieces had to be dealt with. For instance, assume that a piece was moved from square 10 to square 15. So, 10 and 15 are stored in initial squares list and destination squares list, respectively. If the user wants to take this move back, we need to know whether there was an opponent piece on the destination square or not. If there was a capture, then the captured piece must be placed back on square 15 when the user takes this move back. That’s why, I had to store captured piece for each move in another list.
The unique case of castling, where two pieces are moved in a single move, was a tricky part that needed attention. To solve it, I simply stored square numbers greater than 64 in move lists that I used to manipulate the game. Since no initial or destination square can have a number greater than 64, I could easily detect both long castling and short castling.

Up to this point, everything about the implementation of game manipulation was about getting to the correct position on the board at a particular moment. The following figure (Figure 4.22) shows why getting to the correct position on the board does not help us manipulate the game properly at all:

Figure 4.22 – Analysis a previous position

Assume that this figure (Figure 4.22) shows us a previous position in the current game, and the user came back to this position by clicking a move notation on the screen. Also assume that the position on the board was exactly like this before so the model correctly created this previous position. Now, let’s try to answer the following questions:

1) Whose turn is it? (We cannot tell without any other information than the position on the board.)
2) Which queen is a promoted one? The one on e2 or the one on a3? (Note that they both can actually be promoted! Moreover, any piece which is not a pawn or king can be a promoted piece!)
3) Has the white knight on b8 just been promoted? (If that is the case, then it must be converted to a pawn for any position which occurred before this one. In the position that occurred after this one, it must be a knight again.)

4) Can the black pawn on h4 capture the white pawn on g4 by En Passant? (Note that if it’s black turn and if white’s last move was moving the pawn from g2 to g4, then En Passant is possible, but we do not know what the last move is and whose turn it is.)

5) Can any player castle? Has anyone of the rooks or the kings ever been moved in the game? Or has the chance for both short and long castling already been lost? (Although kings and all rooks of both players are in correct locations for castling and the castling squares are not attacked by opponent and empty except for black’s long castling path, we are still unable to answer if kings or rooks have ever been moved or not.)

6) How much time do the players have on their clocks? (We cannot tell.)

As it can be understood, we need to store many values for each move to modify the game properly. Solution to this problem was to decide and store each and every observable needed to be able to answer the questions asked previously. Explaining each of those observables would be unnecessarily repetitive since the solution is same for all them. For instance, I have an observable in my model, which is named can_white_do_short_castle, which stores a value to check if the white player has a chance for short castling or not for the rest of the game. So, I simply store it in a list after each move, and the model can know whether white can make a short castling move at a certain point in the game. When this idea is applied to all other important observables, the game can properly be transformed to any position that occurred.

For saving and loading a game, I apply the same idea since the nature of the problem is completely same. All the important observables are saved and loaded to get to the exact same state. The only difference is that when a game is loaded, the move list in the ‘Game Status’ window is completely changed, while it may be enough to modify it partly when the game is manipulated through move notations on the screen.

4.8 Stage 8: Implementation of Different Game Modes

This was the last stage of my implementation. Having implemented all the rules and interface functionalities I had in my mind, I wanted to create two different game modes so the users can experience and compare the differences.

The first mode is “Training Mode”. The screenshots provided so far were all taken from this mode. The characteristics of this mode are as follows:

- When a piece is chosen, the possible squares for the piece to move to are highlighted in pink.
- The user can modify the game by clicking on the move notations as discussed in the previous section.
The user can deselect a piece after selecting it.

The user is notified through the ‘Additional Information’ window when his/her king is threatened.

When the user decides to play a training game, he/she will have a chance the change the time settings in any way he/she wants. White and black players can be given different amounts of time. Also, the user can set a bonus time per move, separately for each player, so that a player’s remaining time will be increased by the bonus time amount that the user set for the corresponding player (Figure 4.23).

The second mode is “Official Mode”, which is designed like an official chess game. The features of this mode are as follows:

- Each player is given 60 minutes per game.
- Once a piece is chosen, it has to be moved if it has at least one legal move. If it has no legal move, then the opponent’s time is increased by 2 minutes as a penalty.
- The user cannot go to a previous position on the board.

4.9 Summary

This chapter was devoted to the detailed explanations of implementation process of ‘Chess Mentor’. It is worth mentioning once again that the model can be best appreciated through personal interaction but still everything implemented from user interface to inner algorithms of the model was
discussed and illustrated with screenshots in this chapter. The differences between a traditional approach and EM were mentioned in the corresponding sections.

Next chapter will be about the conclusions of this thesis and suggestions about future work on ‘Chess Mentor’.
Chapter 5
Conclusions and Future Directions

This thesis aimed to create a practical and useable product, which can be considered as software and even an educational tool. A real life situation, which is a chess game in the current case, was modeled without any restrictions and based on its modeller’s personal understanding.

Another aim of this thesis was to analyze how EM differed from conventional programming approaches. It was a fascinating and quite helpful experience for me to observe the differences between the two approaches by producing a practical work rather than conducting just a theoretical research.

‘Chess Mentor’ is a model which reflects many aspects of Empirical Modelling. The users will observe and realize the fact that it is a very open-ended model, allowing users to interact with it using their own imagination, which was the main idea behind all this effort.

5.1 Future Directions

As suggestions regarding the future of ‘Chess Mentor’, I would encourage the use of dependencies as extensively as possible. Right now, although the model is built on EM principles, there are procedural parts as well, which can be modified to be based on dependency relationships. A very clear usage of dependency in ‘Chess Mentor’ can be seen in the next three figures (Figures 5.1, 5.2, 5.3):

![Figure 5.1 – Example of Dependency (1)](image)
By default, the width of each square on the board is assigned a value of 49 units in ‘Chess Mentor’. This results in the default screen layout depicted in Figure 5.1. Assume that the user enters a statement in *tkeden* and changes the value of the width of each square as follows:

![Figure 5.2 – Example of Dependency (2)](image)

Then, all SCOUT drawings are re-arranged to reflect a proper user interface with the new value for square width (Figure 5.3):

![Figure 5.3 – Example of Dependency (3)](image)

We have wider squares but the board and all windows adapt to the change of one single observable. This is very simple and effective example to show how powerful Empirical Modelling can be
compared to a traditional programming language. The strength of EM can be better understood when we consider that the total number of SCOUT windows on the last figure provided is around a few hundred. The programmer would have to deal with a very large number of calculations to re-organize the screen layout. However, using a single observable in dependencies was enough here to manipulate so many windows even when the model was running. However, it should be noted that the user will be able to interact with the model much more freely if he/she has some information about the observables in the model. In this example, for instance, the user needs to know that the observable storing the width value of each square is \textit{square\_width}.

In terms of chess rules, the most interesting improvement would be the rule which says that the game ends in a draw when the same position occurs 3 times on the board. The most straightforward approach would be to store each position in a list, \textit{current\_position} observable in ‘Chess Mentor’, and then compare the last element on the list (which is the last position reached on the board) with the rest of the list to see if the same position was reached two more times previously. However, this approach would require many positions to be stored in the list, especially in longer games. Also, considering that a single position consists of 64 squares, it is not a good solution. Still, a very practical trick here is that whenever a piece is captured, we can delete every position we had stored so far because the same position can never be reached without the captured piece. Another very useful trick is we can again delete every position we stored whenever a pawn is moved. Since pawns have to move to the next move even if they just move on the same column or capture diagonally, they can never go back to their previous rows, which means that the previous positions can never be reached. These two tricks will probably make the task much easier to implement.

To improve it even more, game status window can be modified to include a scroll bar, so the user can see all moves, not just last 15 moves for each player. For this purpose, “The \%angel notation” project [14], which is a PhD thesis by Antony Harfield, can be studied.

Another improvement which was beyond the scope of this project can be the implementation of ‘Chess Mentor’ in Cadence, and even inclusion of network connection so that it can be run on two computers.

This thesis has taken EM principles into consideration to model the understanding of a personal experience and analyzed how different EM is in all aspects compared to the classical computer science approaches.

The author hopes that this thesis and its final product, ‘Chess Mentor’ will be helpful in the future modelling studies in Empirical Modelling.
Bibliography


[4] Going beyond classical programming, CS405 Lecture Notes, Online, (Accessible via CS405 website: http://www2.warwick.ac.uk/fac/sci/dcs/research/em/teaching/cs405/)


