

Faculdade de Engenharia da Universidade do Porto



# **Generic Interface for Developing Abstract Strategy Games**

**Ivo Miguel da Paz dos Reis Gomes**

Master in Informatics and Computing Engineering

Supervisor: Luís Paulo Reis (PhD)

February 2011



# **Generic Interface for Developing Abstract Strategy Games**

**Ivo Miguel da Paz dos Reis Gomes**

Master in Informatics and Computing Engineering

Approved in oral examination by the committee:

Chair: António Augusto de Sousa (PhD)

External Examiner: Pedro Miguel Moreira (PhD)

Supervisor: Luís Paulo Reis (PhD)

---

11<sup>th</sup> February 2011



# Abstract

Board games have always been a part of human societies, being excellent tools for conveying knowledge, express cultural ideals and train thought processes. To understand their cultural and intellectual significance is not only an important part of anthropological studies but also a pivotal catalyst in advances in computer sciences. In the field of Artificial Intelligence in particular, the study of board games provides ways to understand the relations between rule sets and playing strategies. As board games encompass an enormous amount of different types of game mechanics, the focus shall be put on abstract strategy games to narrow the study and to circumvent the problems derived from uncertainty.

The aim of this work is to describe the research, planning and development stages for the project entitled “Generic Interface for Developing Abstract Strategy Games”. The project consists in the development of a generic platform capable of creating any kind of abstract strategy game, and is essentially divided in three phases: research on board games and related topics, design and specification of a system that could generate such games and lastly the development of a working prototype.

The research undertaken for this project covers several topics, starting with an extended study on abstract strategy games and board games to a smaller extent. This research provided the means to classify and categorize these games in terms of mechanics and similarities between rules, a necessity for the later specification of a generic language that could comprehend all abstract games. Artificial Intelligence approaches to board games were also researched, to study what methodologies would be better suited to create an artificial intelligence agent-based system capable of playing multiple games. The final research topic covered was the study of general game playing systems that could be used in conjunction with the project, in order to test and validate the games created.

The second stage of the project was the specification and design of the Generic Interface system, defining how abstract games should be created and what manner of options should be presented to the users in order to simplify the game creation process.

The final stage was the development of a prototype of the system that could be used to verify and test the project’s design and specification. The prototype is a software tool that allows users to create several distinct abstract games through a series of options, which define the games’ rules. This project opens up the possibility to easily create and test games, without the need to program them. It is a step forward in abstract game creation in general and it represents a development for Artificial Intelligence research.



# Resumo

Os jogos de tabuleiro sempre fizeram parte das sociedades humanas, sendo excelentes ferramentas para a transmissão de conhecimentos, expressar ideais culturais e treinar processos de pensamento. Entender o seu significado cultural e intelectual não é apenas uma parte importante dos estudos antropológicos, mas também um catalisador fundamental no avanço das ciências da computação. Em particular no campo da Inteligência Artificial, onde o estudo dos jogos de tabuleiro oferece maneiras de compreender as relações entre conjuntos de regras e estratégias de jogo. Como os jogos de tabuleiro englobam uma enorme quantidade de diferentes tipos de mecânica de jogo, o foco será colocado em jogos de estratégia abstractos para reduzir o estudo e para contornar problemas derivados da incerteza.

O objectivo deste trabalho é descrever as fases de pesquisa, planeamento e desenvolvimento do projecto intitulado "Interface Genérica para o Desenvolvimento de Jogos de Estratégia Abstractos". O projecto consiste no desenvolvimento de uma plataforma genérica capaz de criar qualquer tipo de jogo de estratégia abstracto, e é essencialmente dividido em três etapas: pesquisa em mecânicas de jogos de tabuleiro e tópicos relacionados, desenho e especificação de um sistema capaz de gerar tais jogos e finalmente o desenvolvimento de um protótipo funcional.

A pesquisa realizada para este projecto cobre diversos tópicos, começando com um estudo aprofundado em jogos de estratégia abstractos e, com menor profundidade, em jogos de tabuleiro. Esta pesquisa forneceu os meios necessários para classificar e categorizar estes jogos em termos de mecânica e de semelhanças entre regras, algo necessário para a posterior especificação de uma linguagem genérica capaz de compreender todos os jogos abstractos. Também foram estudadas abordagens de Inteligência Artificial a jogos de tabuleiro, para compreender que metodologias seriam mais apropriadas à criação de um sistema com agentes inteligentes capaz de jogar vários jogos distintos. O ultimo tópico da pesquisa efectuada cobre o estudo de sistemas de *General Game Playing* que poderiam ser usados em conjunto com o projecto, quer para testar como para validar os jogos gerados.

A segunda etapa do projecto consiste na especificação e desenho da Interface Genérica, definindo como é que os jogos abstractos devem ser criados e que tipo de opções devem ser apresentadas aos utilizadores de forma a simplificar o processo de criação de jogos.

A etapa final consiste no desenvolvimento de um protótipo do sistema com o intuito de verificar e testar o desenho e especificações do projecto. O protótipo é uma ferramenta de *software* que permite aos utilizadores criar diversos jogos abstractos distintos a partir de uma série de opções, que definem as regras dos jogos. Este projecto põe em aberto a possibilidade de

facilmente criar e testar jogos, ser que para isso seja necessário programá-los. É um avanço na criação de jogos abstractos e como tal representa um desenvolvimento para o estudo de Inteligência Artificial.



# Acknowledgements

First of all I would like to thank my supervisor, Luís Paulo Reis, for all the support, guidance and especially for allowing me total freedom throughout all the steps of this project.

I would also like to thank the organization of the conference Videojogos 2010 for their interest in this project, and for allowing me the opportunity to present it.

My family also deserves a heartfelt thank you for always believing in me and for never allowing my confidence to waver.

To my girlfriend, without whom I doubt I could have found the strength to overcome all the trials and obstacles, my deepest thank you.

To my colleagues and brothers in arms that accompanied me in this journey through all those sleepless nights at the university. My sincere gratitude for all the time we spent discussing our projects and laughing at our mistakes.

Last but definitely not least, I would like to extend my gratitude to all my friends that even though not fully understanding the problems I was faced with never for a moment doubted my ability to conquer them.

Ivo Paz dos Reis



# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Context .....	1
1.2	Motivation .....	4
1.3	Objectives .....	5
1.4	Summary of Contents .....	5
<b>2</b>	<b>State of the Art</b>	<b>7</b>
2.1	Board Games .....	7
2.1.1	Abstract Strategy Games .....	8
2.1.2	Game Mechanics .....	10
2.1.3	Classification Systems .....	12
2.1.4	Conclusions .....	17
2.2	Artificial Intelligence Approaches .....	18
2.2.1	Specific Knowledge Based Systems .....	19
2.2.2	Evaluation Functions .....	20
2.2.3	Machine Learning .....	21
2.2.4	Evolutionary Algorithms .....	23
2.2.5	Conclusions .....	23
2.3	General Game Playing Systems .....	24
2.3.1	Zillions of Games .....	25
2.3.2	The LUDÆ Project .....	26

2.4	Conclusions .....	24
<b>3</b>	<b>Project Overview</b>	<b>29</b>
3.1	Creating Abstract Strategy Games .....	30
3.1.1	Defining an Abstract Strategy Game .....	30
3.1.2	Board Definition .....	32
3.1.3	Movement Definition .....	36
3.1.4	Piece Definition .....	41
3.1.5	Goal Definition .....	42
3.1.6	Rules Definition .....	46
3.1.7	Conclusions .....	48
3.2	Zillions of Games Platform Analysis .....	48
3.2.1	Goal Analysis .....	48
3.3	Generic Language for ASG .....	51
3.4	Improving the AI .....	52
3.4.1	Heuristics Layer .....	53
3.4.2	Learning and Evolutionary Algorithms .....	54
3.5	Conclusions .....	52
<b>4</b>	<b>Generic Interface Development</b>	<b>57</b>
4.1	Specification .....	57
4.1.1	Use Cases .....	57
4.1.2	Features .....	58
4.1.3	Non-Functional Requirements .....	64
4.1.4	System Architecture .....	65

4.2	Conclusions .....	66
<b>5</b>	<b>System Implementation</b>	<b>67</b>
5.1	Prototype Development.....	67
5.1.1	Game Data Classes and Objects .....	69
5.1.2	XML Language .....	70
5.1.3	ZRF Code Generation.....	73
5.1.4	Graphical User Interface.....	76
5.2	Implemented Features .....	77
5.3	Conclusions .....	80
<b>6</b>	<b>Results</b>	<b>82</b>
6.1	Game Creation Analysis.....	82
6.1.1	Examples of Games Created.....	83
6.1.2	Game List .....	90
<b>7</b>	<b>Conclusion and Future Work</b>	<b>91</b>
7.1	Work Summary .....	91
7.2	Future Work .....	92
	<b>References</b>	<b>94</b>
	<b>Glossary</b>	<b>100</b>
	<b>Appendix A – Interface Design Mock-ups</b>	<b>102</b>
	<b>Appendix B – Prototype Screenshots</b>	<b>108</b>
	<b>Appendix C – Game Creation Example: Maze Game</b>	<b>114</b>
	<b>Appendix D – Examples of generated ZRF code</b>	<b>115</b>

# List of Figures

Figure 1.1: Kasparov vs. Deep Blue (1997) [AP, 1998] .....	2
Figure 1.2: CPU Transistor Counts 1971-2008 and Moore's Law [EETimes, 2009].....	3
Figure 2.1: Portrait of a game of <i>Senet</i> , displayed in Merknara's tomb [Ro��, 2011] .....	8
Figure 2.2: Zillions of Games Start-up Menu [ZoG, 2009] .....	26
Figure 3.1: ASG Component Diagram.....	31
Figure 3.2: Board Definition Diagram .....	32
Figure 3.3: Position Diagram .....	34
Figure 3.4: Grid lines in a <i>Go</i> board.....	35
Figure 3.5: Torus Geometrical Surface .....	35
Figure 3.6: Movement Definition Diagram.....	37
Figure 3.7: Knight Move with Radius 3 (left) and with Radius 4 (right).....	39
Figure 3.8: Capture Diagram.....	40
Figure 3.9: Piece Definition Diagram .....	41
Figure 3.10: Goal Definition Diagram .....	43
Figure 3.11: Rules Definition Diagram.....	47
Figure 3.12: Heuristics applied to Zillions' AI Engine.....	53
Figure 3.13: Pawn Structure favouring the White player.....	54
Figure 3.14: Zillions' AI improvement plan diagram.....	55
Figure 4.1: General System Usage Use Case Scenario .....	58
Figure 4.2: Game Options Editing Use Case Scenario .....	58

Figure 4.3: Overall System Architecture.....	65
Figure 5.1: Prototype’s Class Diagram (generated via Microsoft Visual Studio 2010).....	68
Figure 5.2: Game Data Sub-Class Relationships Conceptual Diagram .....	69
Figure 5.3: Generic Interface, XML and ZRF relationship.....	71
Figure 5.4: Excerpt of the generated XML Document.....	72
Figure 5.5: Promotion Macro Example (simplified) .....	71
Figure 5.6: Move Code Generation Diagram.....	71
Figure 5.7: Implemented Features Fulfilment Chart .....	80
Figure A.1: Start-up Game Tab Design Mock-up.....	102
Figure A.2: Board Tab Design Mock-up .....	103
Figure A.3: Moves Tab Design Mock-up .....	104
Figure A.4: Pieces Tab Design Mock-up .....	105
Figure A.5: Setup Tab Design Mock-up .....	106
Figure A.6: Goals Tab Design Mock-up.....	107
Figure B.1: Game Tab Prototype Screenshot.....	108
Figure B.2: Board Tab Prototype Screenshot.....	109
Figure B.3: Moves Tab Prototype Screenshot.....	110
Figure B.4: Pieces Tab Prototype Screenshot .....	111
Figure B.5: Setup Tab Prototype Screenshot .....	112
Figure B.6: Goals Tab Prototype Screenshot .....	113
Figure C.1: Screenshot of the Maze Game running on Zillions.....	114
Figure D.1: Generated ZRF code: Tic-Tac-Toe.....	115
Figure D.2: Generated ZRF code: Chess.....	116

# List of Tables

Table 2.1: Examples of Strategy and Tactics in <i>Chess</i> and <i>Go</i> .....	9
Table 2.2: Rodeffer's Proposed Categorization.....	13
Table 2.3: IAGO and Zillions of Games' ASG Classification.....	14
Table 2.4: World of Abstract Games Classification System.....	17
Table 3.1: Jump Over Options .....	38
Table 3.2: Move Distance Options.....	39
Table 3.3: IAGO Goal Types and Generic Interface Comparison .....	44
Table 3.4: Goal Types and Conditions.....	45
Table 3.5: ZRF Goal Notation Analysis.....	50
Table 3.6: ZRF Goal Options Organized by Goal Types.....	51
Table 4.1: File Management Feature List .....	59
Table 4.2: Interface Feature List .....	59
Table 4.3: Game General Parameters Feature List.....	60
Table 4.4: Board Configuration Feature List .....	60
Table 4.5: Movement Configuration Feature List.....	61
Table 4.6: Piece Configuration Feature List .....	62
Table 4.7: Board Initial Setup Feature List .....	62
Table 4.8: Goal Configuration Feature List .....	63
Table 4.9: Additional Rules Configuration Feature List.....	63
Table 4.10: Zillions Additional Features List .....	64



Table 5.1: Code Generation Construct Sequence.....	73
Table 5.2: Feature Fulfilment.....	78
Table 6.1: Movement Possibilities Permutations .....	82
Table 6.2: Generic Interface input options for <i>Tic-TacToe</i> .....	84
Table 6.3: Generic Interface input options for <i>Chess</i> .....	86
Table 6.4: Generic Interface input options for <i>Connect 4</i> .....	88
Table 6.5: Generic Interface simplified input options for a maze-like puzzle .....	89
Table 6.6: Well-known ASG list and Prototype limitations .....	90

# Abbreviations

AI	Artificial Intelligence
ASG	Abstract Strategy Games
BGG	BoardGameGeek website
FEUP	Faculty of Engineering of the University of Porto
IAGO	International Abstract Games Organization
GUI	Graphical User Interface
WPF	Windows Presentation Foundation
XML	Extensible Markup Language
ZoG	Zillions of Games
ZRF	Zillions Rules File

# Chapter 1

## Introduction

The aim of this work is to describe the research, planning and development stages for the project entitled “Generic Interface for Developing Abstract Strategy Games”. The project consists in the development of a generic platform capable of creating any kind of abstract strategy game.

The research undertaken for this project covers several topics, starting with an extended study on abstract strategy games and board games to a smaller extent. This research provided the means to classify and categorize these games in terms of mechanics and similarities between rules, a necessity for the later specification of a generic language that could comprehend all abstract games. Artificial Intelligence approaches to board games were also researched, to study what methodologies would be better suited to create an artificial intelligence agent-based system capable of playing multiple games. The final research topic covered was the study of general game playing systems that could be used in conjunction with the project, in order to test and validate the games created.

The second stage of the project was the specification and design of the Generic Interface system, defining how abstract games should be created and what manner of options should be presented to the users in order to simplify the game creation process.

The final stage was the development of a prototype of the system that could be used to verify and test the project’s design and specification. The prototype is a software tool that allows users to create several distinct abstract games through a series of options, which define the games’ rules. This project opens up the possibility to easily create and test games, without the need to program them. It is a step forward in abstract game creation in general and it represents a development for Artificial Intelligence research.

### 1.1 Context

The Greek philosopher Aristotle once said: *“If every tool, when ordered, or even of its own accord, could do the work that befits it then there would be no need either of apprentices for the*

*masters or of slaves for the lords*” [Isom, 2005]. Despite the tools and technology of the time being far behind of what exists today, it is clear that the creation of autonomous thinking machines is one of humanity’s long sought desires.

An essential step in creating machines that rival humans in terms of rational thinking is the development of Artificial Intelligence (AI). The advances in computers in the mid 20<sup>th</sup> century lead to the possibility of making a reality of these expectations, eventually leading to the creation of AI as a field of study in computer sciences [McCorduck, 2004]. The term “Artificial Intelligence” was first coined in the conference “The Dartmouth Summer Research Project on Artificial Intelligence”, held in 1955 [McCarthy et al., 1955].

In 1952 the first working AI program of *Checkers* was written by Christopher Strachey, ran on the Ferranti Mark I in the Manchester Computing Machine Laboratory. Strachey said that his program could “play a complete game of draughts at a reasonable speed”. The first chess-playing program by Dietrich Prinz was also written for the Manchester Ferranti. It was designed for solving simple problems of the mate-in-two variety, examining every possible move until a solution was found [Copeland, 2000]. These two programs created an unprecedented event in computer history, by utilizing the computational power of a machine to solve simple game problems, it was established that AI applied to games was essential for improvements in Artificial Intelligence as a whole. Game AI has thus been seen as a way to measure the progress throughout the developments in Artificial Intelligence.

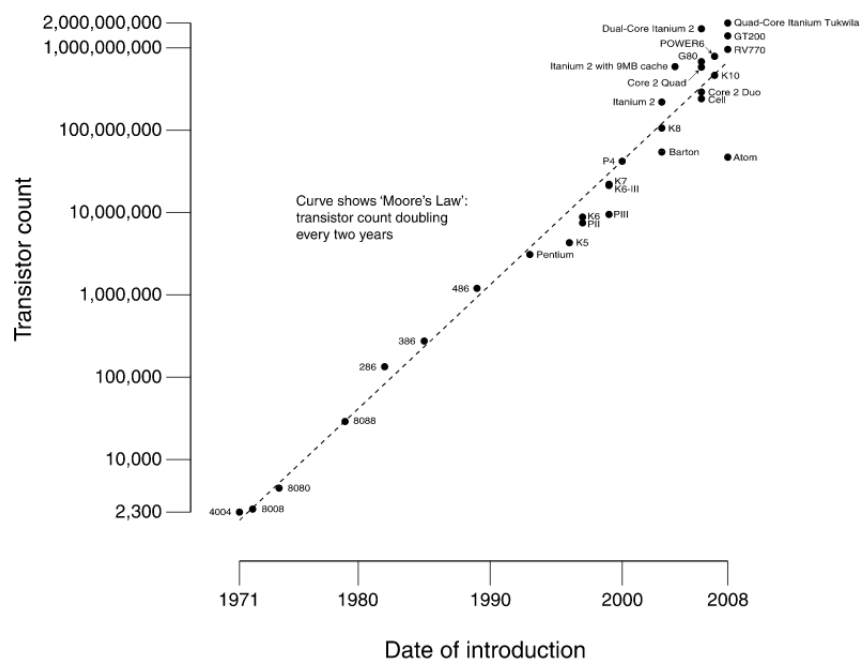


**Figure 1.1: Kasparov vs. Deep Blue (1997) [AP, 1998]**

Figure 1.1 portrays the defeat of Garry Kasparov, the World Chess Champion at the time, by IBM’s Deep Blue computer in 1997. This event is seen as a remarkable feat of AI engineering, proving that advancements in technology made it possible to rival the human intellect in a specific game. Deep Blue was designed to perform a brute force recursive calculation of possible moves, with the aid of extensive knowledge of chess board structures, such as openings [Archi-Plans-4u, 1998][IBM Research, 1997][Krauthammer, 1998].

Although the utopian view of a sentient machine is far from becoming a reality, slowly progresses such as the Deep Blue's victory are taking this field towards the next step. In a prophetic presentation from 2003, robotics expert Rodney Brooks explains how robots, with different levels of Artificial Intelligence, are going to work their way into society in the next 50 years, starting with toys and moving into household chores and beyond [Brooks, 2003].

Moore's Law [Moore, 1965] dictates that for each two years the approximate number of transistors that can be placed inexpensively in an integrated circuit doubles [Intel Corporation, 2005], and so far it has proved to be correct as it can be seen in Figure 1.2. Futurologist Ray Kurzweil has used Moore's law to predict that desktop computers will have the same processing power of that of a human brain by the year 2029, and he also claims that by 2045 Artificial Intelligence will reach a point where it is able to improve itself at a rate that far exceeds anything conceivable in the past [Kurzweil, 2005].



**Figure 1.2: CPU Transistor Counts 1971-2008 and Moore's Law [EETimes, 2009]**

Current trends in philosophy distinguish “strong AI” – the ability of a computer to think as a human – from “weak AI”, bent on problem solving in a seamlessly intelligent fashion [PO, 2008]. Presently the goals are set to developed “weak AI”, as technology has yet to be sufficiently advanced to simulate a human brain. Nonetheless, progress in AI has a lot to gain by focusing on abstract games, as study in this area provides ways to understand the relations between rule sets and playing strategies, enhancing the manner in which a program can effectively play a board game [Luger, 2009].

In order for Artificial Intelligence to advance and evolve, and to bring these predictions and long-sought desires to fruition, projects such as this one are necessary in the step by step improvement of this field of study. Therefore, creating a tool that enables users to effortlessly develop new games in order to test and improve the solving capabilities of Artificial Intelligence software applications may prove to be useful.

## 1.2 Motivation

There are two main reasons that justify research on games. First, human fascination with game playing is long-standing and pervasive [AAAI, 2010]. Anthropologists have catalogued popular games in almost every culture. Games are intriguing because they address important cognitive functions. The second reason is that most of the difficult games remain to be completely solved, and some of those games people play very well but computers are not able to play them at a decent level. These games set challenges that demonstrate that the current approaches in AI still require further developments. Intensifying the effort to meet these challenges can nevertheless provide significant advancements in this field of research [Epstein, 1999].

Board games in particular have played an important role as research objects in the sciences of the 20<sup>th</sup> century. At first, games and board games were studied from a historical perspective. In 1944, Von Neumann and Morgenstern provided a basis for using games and board games in the computer sciences and in economics, such as in the field of game theory [Neumann & Morgenstern, 1944]. Research on board games accelerated with research on *Chess*, which has proved fundamental in the cognitive sciences [Gobet, 2006] [Newell & Simon, 1972]. Chess is still dominant in most fields but slowly other championship games are finding their way as examples or tools in research [Voogt, 1998].

The role of board game study in computer sciences is an essential part of the progress in Artificial Intelligence. Board games are easily replicable environments where the concepts of tactics, strategy, searching, learning and other relevant features of AI can be tested in their purest state. Their conceptual flexibility allows for an open testing field for many different ideas concerning AI approaches [Neto, 2003].

Existing board game software applications only allow for the playing of a restricted set of games. The development of a tool that enables users to create their own games and provide them with AI techniques for playing them is an excellent way of further developing this field of research. In order to accomplish this, the idea of designing an interface for creating abstract strategy games was born. If given a set of options a user could easily define the rules that govern a board game, and subsequently the interface is tasked with the generation of the game in a format that makes it playable in a general game playing system.

## 1.3 Objectives

This project is based in the idea that given a set of options almost any abstract strategy game can be created. Thus, the main goal of this project is the development of a generic platform that supports the creation of most types of abstract strategy games. It is also required that games created this way have the possibility of being translated to a general game playing system capable of playing them with Artificial Intelligence techniques. [Reis, 2009].

The breakdown of the objectives of the project is as follows:

- Research on board game characteristics, showing similarities between rules, board types and pieces
- Develop a language capable of describing abstract strategy games
- Develop a platform with a simple graphical user interface that enables users to create their own board games
- Generate the user defined abstract strategy games in a general game playing format

## 1.4 Summary of Contents

This document is divided in seven chapters.

In the first chapter, ‘Introduction’, the context for the project is presented, along with the motivation that lead to its creation and the definition of its general goals.

The second chapter, ‘State of the Art’, is further divided in three sections, one concerning the research on board games, a section dedicated to the Artificial Intelligence approach to board games and the third describing general game playing systems.

The third chapter, “Project Overview”, describes the full intent of the project and the ideas that define it given the research made in chapter two.

The fourth chapter describes the stages of the ‘Generic Interface Development’ for the project, explaining in detail the most important decisions taken during implementation, covering topics from abstract game definition to ideas on how to improve the AI approach in such games.

The fifth chapter details the “System Implementation”, detailing the system specifications in terms of use cases, features to implement, non-functional requirements and architecture..

The sixth chapter depicts the ‘Results’ achieved in this work.

The seven and final chapter presents the ‘Conclusions and Future Work’, where an overview of all the work done is made, as well as giving focus to the features that could be implemented in the future to enhance the project’s capabilities.



## Chapter 2

# State of the Art

This chapter focuses on the research undertaken in this thesis, and is divided in three topics: Board Games, AI approaches and General Game Playing Systems.

The Board Games section is an overview of board games in general, giving particular focus to abstract strategy games, their mechanics and categorization. The following section is concerning AI methodologies used in ASG, ranging from specific knowledge based systems, evaluation functions, machine learning and evolutionary algorithms. The third section gives an overview of general game playing systems applied to ASG.

### 2.1 Board Games

A game played on a pre-marked surface is considered a board game. Besides the board itself there are usually pieces that are placed on it, removed from, or moved across. Simple board games are often seen as ideal "family entertainment" as they can provide entertainment for all ages. Some board games, such as *Chess*, *Go*, *Chinese Chess*, *Shogi*, or *Oware*, have intense strategic value and have become classics [AllExperts, 2010].

Board games are a complex form of games. They consist of boards and various kinds of pieces, a system of rules, and most importantly players. Taking it all into perspective, it appears that the players, boards, pieces and rules, are elements that cannot be separated for a complete understanding of a board game. The rules may influence the board and vice versa. The players may determine the shape and kind of boards and the specificity of the rules. Board games in their complexity present researchers with various questions, for instance: the relationship of the aspects, such as board types and pieces with rules, of a board game are little understood [Voogt, 1998].

Board games have a long history and have been played in most cultures and societies; some even pre-date literacy skill development in the earliest civilizations. A number of important historical sites, artefacts and documents exist which shed light on early board games [AllEx-

perts, 2010]. The oldest board game discovered is *Senet* (Figure 2.1), found in Pre-dynastic and First Dynasty burials of Egypt, 3500 BC and 3100 BC respectively [Piccione, 1980].



Figure 2.1: Portrait of a game of *Senet*, displayed in Merknera's tomb [Ro  , 2011]

There are many different types and classifications of board games. Some games are simplified simulations of real life. These are popular as they can combine make-believe and role playing along with the game. Classic popular games of this type include *Monopoly*, which is a rough simulation of the real estate market; *Carcassonne*, in which players have to deploy the board tiles in order to simulate the building of a city; *Puerto Rico*, which features economic, resource gathering and city building elements; *Power Grid*, that simulates the energy industry by having players bid in a market and expanding the range of their power grids to reach several cities; *Settlers of Catan*, which is one of the best known of thousands of games attempting to simulate warfare and geo-politics. Other games only loosely attempt to imitate reality. These include abstract strategy games like *Chess* and *Checkers*; word games, such as *Scrabble*; and trivia games, such as *Trivial Pursuit* [AllExperts, 2010].

The website BGG contains one of the most complete assortments of board games, detailing rules and ratings for each individual game, as well as classifying them in terms of gameplay and theme. They have divided board games in several categories, including: abstract games, customizable games (with role-playing elements), family games (quizzes and trivia), party games (dexterity, drawing and other skills), strategy games and war games [BGG, 2011].

### 2.1.1 Abstract Strategy Games

Board games are a broad genre, where diplomacy, luck and strategy are common if not even required in the majority of these games. Nevertheless, the purest of board games are those without randomness and negotiation between players, focusing purely on strategy and tactics.

Most of these games are also devoid of a theme, thus the term ‘abstract’ applies [Neto, 2003] [Thompson, 2000].

The key point in abstract strategy games is perfect information, which means that all players have complete knowledge of all the information of the game at all times, there is no hidden information like cards or dice rolls. This requirement makes abstract strategy games an excellent research subject for computer sciences and specially Artificial Intelligence, since the only constraints are combinatorial in nature [Neto, 2003] [Thompson, 2000] [Thompson, 2010] [BGG, 2011].

Ideally, a computer with infinite processing power can calculate all the possible states of the board from the initial state to every possible endgame situation. However, the combinatorial possibilities of such games are usually far beyond the order of billions. It is estimated that *Checkers* has a game-tree complexity of  $10^{31}$  possible positions, while *Chess* has around  $10^{123}$ . As for *Go*, the possible legal game positions range in the magnitude of  $10^{170}$  [Thompson, 2000] [Eppstein, 2007].

Other characteristics of abstract strategy games are the fact that they are built on simple and straightforward design and mechanics, and that they should also promote one player overtaking their opponent, or opponents in the case of a multi-player game [BGG, 2011] [Thompson, 2010].

As strategical gameplay is the main focus of these games, a distinction between essential concepts such as Strategy and Tactics is necessary. Strategies are long term plans to win a game, whereas a tactic is an immediate action, typically planned to advance your strategy. Tactics often are available as a result of a particular strategy. It is common that the winner of an abstract strategy game is the one that better coordinated tactics with strategies, and typically strategic goals are accomplished through a combination of tactics. Another interesting perspective is the possibility of allowing for a tactical loss but nevertheless gaining a strategic advantage. If the question is “*what to do to win*” the focus is on strategy, tactics is when the question is “*how does one win*” [Reynolds, 2009].

**Table 2.1: Examples of Strategy and Tactics in *Chess* and *Go***

	<i>Chess</i>	<i>Go</i>
<i>Strategic</i>	Pawn Structure King Safety Centre	Amashi Shinogi Reduction
<i>Tactical</i>	Forks Pins Discovered Attacks	Ladder Net Snapback

It is common for players to be referred to as ‘strategical players’ or ‘tactical players’. For most abstract strategy games, a player needs to understand both strategy and tactics to be well-

rounded. Every game has a natural balance of tactics and strategy, and may reward more tactical or strategic gameplay. A player needs to know how to balance the two levels [Reynolds, 2009].

## **2.1.2 Game Mechanics**

Abstract strategy games include an astounding variety of games, each with specific sets of rules and strategies. In order to describe the game mechanics in a general manner it is necessary to broaden the scope of the analysis, pinpointing the most usual similarities between rule sets without restricting the possibility of rules not covered by this generalization.

There have been many attempts to fully describe a generalization of board game mechanics, but the most comprehensive one found is the document “Definition of Abstract”, by the International Abstract Game Association and is described as follows [IAGO, 2009].

### **2.1.2.1 Key elements**

There are multiple key elements that are common to all abstract strategy games:

- Turns: A turn is defined an action or a series of actions that a player can perform that change the state of pieces in a game
- Players: Actors who change the states of pieces in the game
- Pieces: These are items the players control that are used to define the state of game
- Board (play area): Board is a term used to describe area where pieces are positioned and interact with one another. A board may also represent a Board with no spaces
- Null (off play area): Used to describe where pieces are located when they are not on the board. Null can be defined and configured many different ways. It can consist of a queue or multiple queues. For example: captured pieces in *Chess* and unplaced pieces in *Go*.

### **2.1.2.2 Turn Sequencing**

Only one player may perform actions at a time. Rules govern the order of player turns.

It is typically required some random way of deciding who the first player is, in order to begin the game. Despite the fact that this adds some sort of probability to the game, it is not considered as such as the decision is taken before the actual start of the game. Games like *Chess*,

however, have rules specifically defining that the white player always goes first. Nevertheless, it is still necessary to decide what player is going to play white, before the game starts.

### **2.1.2.3 Players**

Several distinct types of players can be categorized, depending on their ‘intelligence factor’. This is useful for defining different Artificial Intelligence players:

- Intelligent: aware of game conditions and acts upon information in game to adjust plans
- Unintelligent: acts deterministically, reactive or random. Can also act as a mix of all three.
- Deterministic: unaware of game conditions; non-reactive but goal driven
- Reactive: acts based upon game conditions, deterministic solely by the rules; not goal-driven
- Random: unaware of game conditions and acts unpredictably.

### **2.1.2.4 Piece Attributes**

- Ownership (who owns the piece): Piece may be owned by one or more players; ownership is defined as a player being able to manipulate a piece during their turn.
- Traits: States of a piece that determines how it is governed by rules.

### **2.1.2.5 Board Attributes**

- Discreteness of spaces:
  - Discrete (finite): Multiple spaces that fit distinct pieces on each other
  - Non-Discrete (Infinite): Can be argued it is a “space less” board; boundaries are defined by pieces, not spaces on the board
- Boundness of board:
  - Board is finite: Board has boundaries
  - Board has infinite number of spaces (non-discrete): Board is expandable, and has boundaries extended
- Traits: Attributes of board spaces (and board) that are governed by rules, and impact how rules govern pieces.
- Static vs. Dynamic: Attributes don't change / Attributes can change

### 2.1.2.6 Relationship of Board to Pieces

Rules govern if and how pieces may be grouped together on which spaces.

- Uni-space: Each space on the board may contain only one piece.
- Multi-space: Each space on the board may contain multiple pieces. Rules regarding limits to quantities are required (even if unlimited).
  - Queued (stack): Pieces enter and leave in a sequence from one another.
  - Multi-queued: Each player has their own queue for spaces on the board.
  - Pile: pieces freely leave or gather independent of each other, not in sequence. A pile can contain multiple player pieces in it.

### 2.1.2.7 Null Types

The null are is the off board area where pieces are sent when captured or before being placed in the game. Several distinctions can be made within the available null types:

- Void: Pieces into Null never return to game (eliminated).
- Single space: Has capacity for one piece only. Prior pieces are sent to equivalent of Void (described above)
- Pool: Pieces can enter back into play in any order
- Queue: Pieces enter back into play following a sequence.
- Multiple Queues: Each player has his own queue.

## 2.1.3 Classification Systems

There is much debate when it concerns to classification of the numerous types of board games. Usually the majority of board games can be classified by themes, such as economics (*Monopoly*), war-like (*Risk*), word games (*Scrabble*), knowledge (*Trivial Pursuit*), diplomacy and politics (*Settlers of Catan*), dexterity (*Jenga*), etc. [BGDF, 2008]

However, since abstract strategy games are, by definition, unrelated to any specific theme, their categorization must follow different standards. One can argue that games like chess have a war-like theme, but as it does not affect the game in the way the game is played it is considered merely a symbolic theme [Neto, 2003] [Thompson, 2000].

Although there is no consensus even in the international board games associations in this matter [IAGO, 2009] [BGS, 2010] [BGG, 2011], a selected assortment of classification systems

for abstract strategy games are described next, presented in an ascending fashion in terms of complexity.

### 2.1.3.1 Rodeffer's Proposed Categorization

This classification, proposed by C. Rodeffer of the International Abstract Games Organization, takes into account game mechanics, specifically rules that concern actions such as claims, occupations, connections, captures, immobilizations, eliminations and checkmates. This division has four categories: Board Target, Board Majority, Piece Target and Piece Majority. It is completely focused on goals to achieve victory conditions [IAGO, 2009]. Table 2.2 presents a simplified version of Rodeffer's categorization.

**Table 2.2: Rodeffer's Proposed Categorization**

<i>Actions</i> \ <i>Target/Majority</i>	<i>Board</i>	<i>Piece</i>
Claim Capture	✓	✓
Occupy Connect	✓	✗
Immobilize Eliminate Checkmate	✗	✓

The difference between Board/Piece Target and Majority is that in Target there are absolute specific goal piece(s) or goal space(s), whereas in Majority the actions must be performed on the relatively most (or more important) goal piece(s) or space(s).

### 2.1.3.2 Open Directory Project

This approach tries to categorize abstract strategy games in terms of mechanics, in a way very similar to the previous classification. However, it is important to note the creation of a category for “Unequal Forces”, which are games that may have different sets of pieces or even rules for each player [ODP, 2008].

The categories in this classification are self explanatory: Alignment, Battle, Capturing, Connection, *Mancala* type games, Race, Territory and Unequal Forces.

### 2.1.3.3 International Abstract Games Organization and Zillions of Games

The official classification of the International Abstract Games Organization is based on the Zillions of Games commercial application. It follows the principles of rule analysis for describing games, focusing on objectives for victory (Table 2.3) [IAGO, 2009] [ZoG, 2009].

Table 2.3: IAGO and Zillions of Games' ASG Classification

Types	Explanation	Sub-Types	Objectives	Example
<i>Escape</i>	The location of pieces to positions on the board matters; players try to get a piece or pieces onto key positions on board; win positions for pieces on board is absolute; victory is occupation of all critical areas	<i>Bi-sided</i> ( <i>race</i> )	Move piece(s) to goal positions, which can be different for each player	<i>Arimaa</i>
		<i>Uni-sided</i> ( <i>break-through</i> )	One player tries to take piece(s) to goal positions while the other tries to stop them from reaching their goal	<i>Tablut</i>
<i>Territory</i>	A player tries to control/own most spaces on board; ownership of board spaces determines winner; victory is a control/own majority of all critical areas	<i>Occupy</i> ( <i>absolute</i> )	Direct occupation of spaces	<i>Othello</i>
		<i>Claim</i> ( <i>relative</i> )	Players try to own spaces indirectly, based on position of pieces on board	<i>Go</i>
<i>Positioning</i>	Players configure groups of pieces on the board in some manner; win positions for pieces on board is relative, meaning that pieces could end up partly being on absolute spots, however the relative positions are governed by the rules	<i>Connection</i> ( <i>chain</i> )	Players try to connect two or more spaces on the board with a chain of their pieces	<i>Hex</i>
		<i>Arrangement</i> ( <i>cluster</i> )	Players try to get their pieces in a certain configuration	<i>Lines of Action</i>
<i>Elimination</i>	The quantity of usable pieces on board matters; players seek to capture or immobilize one or more of opponent's pieces; win condition are pieces not on board or pieces that are immobilized	<i>Royal</i> ( <i>capture / checkmate</i> )	Players try to capture or checkmate a specific opponent's piece or all of a specific type of pieces	<i>Chess</i> ( <i>and variants like Ex-tinction</i> )
		<i>Multipiece</i> ( <i>capture / im-mobilize</i> )	Players try to eliminate or immobilize all, or a number, of the opponent's pieces	<i>Checkers</i>



#### 2.1.3.4 The LUDÆ Project

In the article “*Defining the Abstract*”, Thompson considers four qualities an abstract game must possess to have lasting merit: depth, clarity, drama, and decisiveness [Thompson, 2000]. These properties are described below with further detail. The LUDÆ project follows this approach, dividing these properties between quantitative and qualitative measures and at the same time adding a few, more simplistic, properties as well [Neto, 2003].

##### *Quantitative measures*

- **Size**: the dimension of the search space, or in other words the number of possible positions inside the game tree. This can be used to define the potential game complexity.
- **Width**: The branching factor of the game. A measure of the average expected number of legal moves of a valid position.
- **Tempo**: The timing between actual interactions (the possibility of capture or entering enemy influenced territory) between reasonable adversaries. A measure of the average number of moves to piece interaction.
- **Mobility**: The overall piece mobility. A measure of the average number of moves per piece in relation to the percentage of empty cells.
- **Clearness**: The simplicity of the rules may also be a measure of the quality of a game.
- **Depth**: The potential number of increasing levels of expertise that can differentiate players.

This last property, *Depth*, gives the game lasting interest because the player continues to learn how to improve his play for a long time. If players in the first level all lose regularly to the players in the second one, and those to the ones in the third one and so on, up to class  $n$ , then the value of  $n$  measures the depth of the game. *Go* is considered the ‘deepest’ amongst the classic abstract games. The number of titles for ranking players in *Chess* gives a good example of how the deep this game is [FIDE, 2010].

##### *Qualitative measures*

These measures are not quantifiable in more complex games, given that an entire game tree analysis is not feasible.

- **Decisiveness**: The average percentage of moves with positional advantage for which there is a winning strategy.

A positional advantage is defined by a positive evaluation using the strategy of the best known player. If decisiveness approaches zero, it means that the game tree endgame states are densely covered with draws. This implies that a winning state can only be achieved by the op-

ponent committing a mistake, meaning that if both players are experts the game is likely to end in a draw, and is thus considered a flawed game in terms of strategic significance.

A game with good decisiveness should easily be able to reach pre-endgame positions, where the positional or material advantage in those positions is enough to win the game with reasonable playing. Decisiveness in a game is important as it should be possible for one player to achieve strategic advantage from which the other player cannot recover. A game with poor decisiveness like *Abalone*, as there is a defensive strategy that a weaker player can adopt (group most of the pieces in the centre and never extend them), making it impossible for the stronger player to win. In this situation the stronger player is faced with typical puzzle “*How can I push my advantage to a victory?*” but he can’t ever find a solution. *Tic-Tac-Toe* has zero decisiveness for the reason that between perfect players the game always ends in a draw.

- **Drama:** It measures the chance of the losing player recovering the opponent’s advantage.

Games with high drama make it possible for a player to recover from a weaker position and still win, and victory should not be attained by a single move. In a game with poor drama a disadvantage in the beginning may cause the rest of the game to be uninteresting. Again a similar question is posed to the losing player “*How can I change my disadvantage to a victory?*” with no viable answer.

Even though *Chess* is considered to be a dramatic game, between expert players the drama tends to decline, as most high ranking games end in a draw. Usually, good *Chess* players resign when winning is no longer a possibility (when the game is no longer dramatic). Masters of the game resign when it becomes clear they must lose a piece without gaining in exchange either an enemy piece, or a positional advantage; grand masters may even resign at the loss of a pawn. The drama of *Chess*, for them, must consist of the alternation of very delicate shades of positional advantage.

- **Clarity:** Defines how easy it is to define a tactical and strategic plan.

Anyone with knowledge of the rules should easily find a decisively winning move in an endgame situation. It should even be relatively easy to find a move that leads to advantage in midgame. The popular chess variant *Ultima*, with its free moving pieces and its many ways of capturing, lacks clarity. The author, Robert Abbott, says that is almost impossible to see more than two steps ahead in the game tree and because of that fact most losses are caused by surprise attacks [Abbott, 1975][Abbott, 1988].

### 2.1.3.5 World of Abstract Games

Besides the LUDÆ Project, João Neto has also created an extensive list of abstract games in the website “*World of Abstract Games*” [Neto, 2010]. For each game there is a list of features related to the game’s mechanics. If available, there is also a ZRF file for playing that game in the Zillions of Games platform. Games present in the site are divided into three main groups:

- ***Games of Soldiers***: games which use only one type of piece (the soldier) for each player
- ***Games of Kings and Soldiers***: games which use two types of piece (the soldier and the king) for each player
- ***Games of Towers***: games which use stackable pieces

According to this classification methodology, the many features that classify abstract games are divided in four major categories, as can be seen in Table 2.4.

**Table 2.4: World of Abstract Games Classification System**

Category	Feature	Example
<b><i>Board Type</i></b>	<b><i>Square Grid</i></b>	<i>Go</i>
	<b><i>Hexagonal Grid</i></b>	<i>Hex</i>
	<b><i>1-D Board</i></b>	<i>Welter's Game</i>
<b><i>Movement</i></b>	<b><i>Pieces are Dropped</i></b>	<i>Go</i>
	<b><i>Pieces may Move</i></b>	<i>Chess</i>
	<b><i>Pieces may Capture</i></b>	<i>Chess</i>
	<b><i>Pieces may Jump</i></b>	<i>Checkers</i>
	<b><i>Pieces may Stack each other</i></b>	<i>Stacks</i>
<b><i>Board Destruction</i></b>	<b><i>Cells may be destroyed</i></b>	<i>Blocks</i>
<b><i>Goal</i></b>	<b><i>Stalemate Opponent</i></b>	<i>Blockades</i>
	<b><i>Pattern Oriented</i></b>	<i>Qubic</i>
	<b><i>Capture Oriented</i></b>	<i>Checkers</i>
	<b><i>Reaching a Cell Oriented</i></b>	<i>Pompeii</i>
	<b><i>Territory Oriented</i></b>	<i>Go</i>
	<b><i>Connection Oriented</i></b>	<i>Dara</i>

A game may have only one Board Type, but as many Movement and Goal features desired, and the Board Destruction feature as well (although this feature is not very common).

#### 2.1.4 Conclusions

This section shows that there are many ways to categorize and analyse ASG. There are even some games that, despite most consider them to belong to a category, share goals and features present in another (like *Chess* which can end in victory with a checkmate or draw with a stalemate). The amount of different views serves to show that it is no easy task to generalize and put game creation in abstract or undefined terms.

To fully understand the mechanics and differences in ASG one has to research, categorize and analyse the games themselves, as well as to take into account what the experts say about these fundamental concepts of game mechanics and game classification. This study was undertaken in order to give a broader view on ASG and to facilitate the designing and implementation of an application that, through the knowledge collected herein, be able to supply the user with the tools necessary for generating playable abstract games.

Several sources state that for analysing an abstract game one cannot leave out of the picture factors such as playability and aesthetics, but that analysis, although important, is not one of the goals of this project. The importance is figuring out what makes a game, and not what makes a good game from a human point of view.

Another important factor in ASG is game balance, or in other words what makes a game equally fair to both sides. There is much difficulty in having a purely balanced game, because these games are played in turns and one player will always play before the other in the beginning of the game. This obviously conditions the second player's choices and strategies to make up for that slight disadvantage (or advantage as some games may prove to be easier to win for the second player). The game balance evaluation also falls in the category of game playability, and is therefore discarded as an influential factor in generic ASG creation. It is important, however, to ensure that the games created this way do not have conflicting rules and are, in fact, games solvable (to some extent) by a sufficiently powerful AI engine.

## 2.2 Artificial Intelligence Approaches

Applying Artificial Intelligence to solve the puzzles posed in abstract games has been one of the key focuses of research. These problems, and more importantly the solutions found, are vital progresses in Artificial Intelligence. Alan Turing proposed the programming of a game of *Chess* playing as a good project for studying computers' ability to reason. In many ways, games have provided simple proving grounds for many of AI's powerful ideas [AAAI, 2010]. The diversity of ways that scientist and engineers have developed to solve these problems are proof that this is a vital area of research in computer sciences, possibly opening the way to ever more complex advances in Artificial Intelligence as a whole [Munakata, 2008] [Nilsson, 1999].

The essential AI methodologies applied to abstract games are listed in this section, starting with specific knowledge based systems, evaluation functions, machine learning and evolutionary techniques.

### **2.2.1 Specific Knowledge Based Systems**

Also referred to as Expert Systems, Specific Knowledge Based Systems are, as the name implies, systems with domain-specific knowledge. The primary purpose of expert systems research is to make expertise quickly available to decision makers and technicians, bringing them decade's worth of knowledge instantly to aid in problem solving. The same systems can assist supervisors and managers with situation assessment and long-range planning. These knowledge-based applications of Artificial Intelligence have enhanced productivity in business, science, engineering, and the military [Ignizio, 1991] [AAAI, 2010].

Realizing the importance of having domain-specific knowledge was one of the most important insights gained from the early days of research in problem solving. For example, the ability for doctors to successfully diagnose a patient's illness is based in his expertise in the field of medicine, and not his some innate general problem-solving skill. Similarly, a geologist is effective at discovering mineral deposits because he is able to apply a good deal of theoretical and empirical knowledge about geology to the problem at hand. Expert knowledge is a combination of a theoretical understanding of the problem and a collection of heuristic problem-solving rules that experience has shown to be effective in the domain. In 1956, Alan Newell and Herbert Simon created the Logic Theorist, the first "expert system", used to help solve difficult math problems. Expert systems are constructed by obtaining this knowledge from a human expert and coding it into a form that a computer may apply to similar problems [Luger, 2009]. This reliance on the knowledge of a human domain expert for the system's problem solving strategies is a major feature of expert systems, and the means to obtain this knowledge and implement it on expert systems is usually attained throughout interviews to experts in specific fields [Piazza & Helsabeck, 1990].

The usual process that building an expert system requires is goes through the following phases: the domain expert provides the necessary knowledge of the problem domain through a general discussion of his problem-solving methods and by demonstrating those skills on a carefully chosen set of sample problems; the AI specialist, or knowledge engineer, as expert systems designers are often known, is responsible for implementing this knowledge in a program that is both effective and seemingly intelligent in its behaviour. Once such a program has been written, it is necessary to refine its expertise through a process of giving it example problems to solve, letting the domain expert criticize its behaviour, and making any required changes or modifications to the program's knowledge. This process is repeated until the program has achieved the desired level of performance [Luger, 2009].

It is interesting to note that most expert systems have been written for relatively specialized, expert level domains. These domains are generally well studied and have clearly defined problem-solving strategies. Problems that depend on a more loosely defined notion of "common sense" are much more difficult to solve by these means. In spite of the promise of expert systems, it would be a mistake to overestimate the ability of this technology. Current deficiencies include [Luger, 2009]:

- Difficulty in capturing "deep" knowledge of the problem domain. Specific Knowledge Based Systems only know what they were programmed with, and cannot infer new information by analyzing the data they have
- Lack of robustness and flexibility. If humans are presented with a problem instance that they cannot solve immediately, they can generally return to an examination of first principles and come up with some strategy for attacking the problem. Expert systems generally lack this ability.
- Inability to provide deep explanations. Because expert systems lack deep knowledge of their problem domains, their explanations are generally restricted to a description of the steps they took in finding a solution. For example, they often cannot tell "why" a certain approach was taken.
- Difficulties in verification. Though the correctness of any large computer system is difficult to prove, expert systems are particularly difficult to verify. This is a serious problem, as expert systems technology is being applied to critical applications such as air traffic control, nuclear reactor operations, and weapons systems.
- Little learning from experience. Current expert systems are handcrafted; once the system is completed, its performance will not improve without further attention from its programmers, leading to doubts about the intelligence of such systems.

### **2.2.2 Evaluation Functions**

An evaluation function, also known as a heuristic evaluation function or static evaluation function, is a function used by game-playing programs to estimate the value or goodness of a position in the Minimax and related algorithms. The evaluation function is typically designed to be fast and accuracy is not a concern (therefore heuristic); the function looks only at the current position and does not explore possible moves (therefore static) [Russel & Norvig, 1995].

There can be any number of different scoring mechanisms all working at the same time. Each can look for different strategic features of the game. One scoring mechanism may look at the number of units each side controls, another may look at patterns for territory control, and yet another might look for specific traps and danger areas. There can be tens of scoring mechanisms in complex games [Millington, 2006].

Each separate scoring mechanism is then combined into one overall score. This can be as simple as adding the scores together with a fixed weight for each. One popular strategy for constructing evaluation functions is as a weighted sum of various factors that are thought to influence the value of a position. For example, an evaluation function for *Chess* might take the form:  $c_1 \times \text{material} + c_2 \times \text{mobility} + c_3 \times \text{king safety} + c_4 \times \text{centre control} + \dots$  Where  $c_n$  represents values to be multiplied to each of the individual heuristics, components of the evaluation function as a whole [Millington, 2006].

*Chess* beginners, as well as the simplest of *Chess* programs, evaluate the position taking only "material" into account. They assign a numerical score for each piece and sum up the score over all the pieces on the board. On the whole, computer evaluation functions of even advanced programs tend to be more materialistic than human evaluations. This is compensated by the increased speed of evaluation, which allows most of the game tree to be examined. As a result, some chess programs may rely too much on tactics at the expense of strategy [Newell & Simon, 1976].

For every game the evaluation function differs, as it is dependent on a mathematical translation of strategy and tactics specific to each game. The game's objective (like reaching a specific space in the board) must have the highest value among all the portions of the evaluation function, in order to assure that a possible winning move is rated highest in the search.

The use of heuristics is common in problem-solving search techniques, especially in abstract game's Artificial Intelligence. Algorithms like *Minimax* are heavily dependent on evaluation functions, as they evaluate the board with heuristics, trying to find the best possible move and for that move they apply the same evaluation function to try and find the best counter move the opponent can play. This search is done until a pre-determined depth in the game tree and eventually the best combination of player move / opponent counter move is found.

### 2.2.3 Machine Learning

One of the greatest challenges in Artificial Intelligence is enabling a program to learn. Machine learning refers to a system capable of the autonomous acquisition and integration of knowledge. This capacity to learn from experience, analytical observation, and other means, results in a system that can continuously self-improve and thereby offer increased efficiency and effectiveness [AAAI, 2010].

Over the past 50 years the study of machine learning has grown from the efforts of a handful of computer engineers exploring whether computers could learn to play games, and a field of statistics that largely ignored computational considerations, to a broad discipline that has produced fundamental statistical-computational theories of learning processes, has designed learning algorithms that are routinely used in commercial systems from speech recognition to com-

puter vision, and has spun off an industry in data mining to discover hidden regularities in the growing volume of online data [Mitchell, 2007].

The field of machine learning can be organized around three primary research fields [Michalski et al., 1983]:

- Task-Oriented Studies: the development and analysis of learning systems oriented toward solving a pre-determined set of tasks
- Cognitive Simulation: the investigation and computer simulation of human learning processes
- Theoretical Analysis: the theoretical exploration of the space of possible learning methods and algorithms independent of application domain

Although many research efforts strive primarily towards one of these objectives, progress in one objective often leads to progress in another. For instance, in order to investigate the space of possible learning methods, a reasonably starting point, may be to consider the only known example of robust learning behaviour, namely humans (and perhaps other biological systems). Similarly, psychological investigations of human learning may be helped by theoretical analysis that may suggest various plausible learning methods. The need to acquire a particular form of knowledge in some task-oriented study may itself spawn new theoretical analysis or pose the question: “*How do humans acquire this specific skill (or knowledge)?*” The existence of these mutually supportive objectives reflects the entire field of Artificial Intelligence, where expert systems research, cognitive simulation, and theoretical studies provide some mixture of problems and exchange of ideas [Carbonell et al., 1983].

Currently, Machine Learning techniques rely on heuristic search algorithms, neural networks, genetic algorithms, temporal differences, and other methods. It is still a field with much work to be done, and the future of AI stands heavily on the shoulders of research in this area [Scott, 2002].

Board games such as *Chess* and *Go* have been an important area of AI research since the pioneering work of Turing and Samuel. Turing realised that a computer could be programmed to play a game, and Samuel investigated how a program could learn to play *Checkers*. Whilst much research has been done on modelling how performance improves in playing a single game, little research has been done on developing a general model which can learn a variety of board games. Board games are a good domain to do machine learning research in as other domains are much too complex to learn. A good model of how people learn to play games would have important implications in both education and in the development of more general machine learning systems [Furse, 1995].



## **2.2.4 Evolutionary Algorithms**

In Artificial Intelligence, an evolutionary algorithm is a generic population-based meta-heuristic optimization algorithm. An evolution algorithm uses some mechanisms inspired by biological evolution: reproduction, mutation, recombination, and selection. Candidate solutions to the optimization problem play the role of individuals in a population, and the fitness function determines the environment within which the solutions "live". Evolution of the population then takes place after the repeated application of the above operators. Artificial evolution describes a process involving individual evolutionary algorithms; evolutionary algorithms are individual components that participate in an artificial evolution [Bäck et al., 1997] [Poli et al., 2008] [Ashlock, 2006].

Evolutionary algorithms often perform well approximating solutions to all types of problems because they ideally do not make any assumption about the underlying fitness landscape; this generality is shown by successes in fields as diverse as engineering, art, biology, economics, marketing, genetics, operations research, robotics, social sciences, physics, politics and chemistry [Bäck, 1996] [Browne & Maire, 2010].

## **2.2.5 Conclusions**

Artificial Intelligence and games share a deep connection. Many advances in AI, and its first practical uses, are linked to board games and abstract games in particular. There is no denying the fact that by researching games and developing AI methods capable of efficiently playing games this field of study is bound to advance.

This section focused on four distinct AI approaches that can be applied to ASG. This study is necessary to distinguish what approaches may be better focused than others to apply to a generic abstract game application.

Specific Knowledge Base systems for instance are a poor choice because the whole point of a system of this type is to understand and compile as much information as possible about a specific theme (in this case game), which would make it impractical to use on a general game playing system but perfect for playing a single game.

Evaluation functions, on the other hand, can be designed in a way as to analyse and evaluate the rule set of a given game, attributing values to the game tree and re-evaluating each step with the new information discovered as it goes deeper in the tree. The drawback of this type of approach is that it depends on the way it performs its evaluation, and this evaluation will always be made the same way independently of how many games the application has played. Some randomness may be added to add some variety but the overall result will be the same.

Both machine learning and evolutionary algorithms are focused on improving the way an AI engine plays a game over time. Theoretically, the more iterations and the more games it plays the better the engine will be at playing the same game. The downside is that usually approaches like these take a long time and processing power to perfect themselves to be able to provide a decent challenge to a human being.

The ideal way of having the best possible AI engine is a combination of evaluation functions, machine learning and genetic algorithms. The idea of the LUDÆ project was similar, having an arena where AI agents would play games against each other, then sorted out according to their performance and applying a genetic algorithm to produce the next batch of AI agents.

## **2.3 General Game Playing Systems**

A General Game Playing System is one that can accept a formal description of a game and play the game effectively without human intervention. General Game Playing is the design of Artificial Intelligence programs to be able to play more than one game successfully. Unlike specialized game players, such as Deep Blue, general game players do not rely on algorithms designed in advance for specific games; and, unlike Deep Blue, they are able to play different kinds of games. For many games like chess, computers are programmed to play using a specially designed algorithm, which cannot be transferred to another context. For example, a chess playing computer program cannot play checkers. A General Game Playing system, if well designed, would be able to help in other areas, such as in providing intelligence for search and rescue missions [Genesereth & Love, 2005].

Several projects concerning General Game Playing Systems exist, but few have been completed and there is still no standard definition of abstract games, as described in section 2.1.3 [Love et al., 2008].

This section will focus on the Zillions of Games platform, as it is the most successful application of this kind for running abstract strategy games; and the LUDÆ project, because of its simplicity in describing what a General Game Playing Systems applied to board games should be like.

### 2.3.1 Zillions of Games

*“The first infinitely expandable PC gaming system.”* – Zillions of Games promotional tag-line [ZoG, 2009].

Zillions of Games is a commercial General Game Playing system developed by Jeff Mallett and Mark Lefler in 1998. Zillions of Games is a game package with a ‘universal gaming engine’ for board games. This allows players to play nearly any abstract (two-dimensional) board game or puzzle. This engine takes as input games written in a specific language for board games [Dobbe, 2007]. The game rules are specified with S expressions, Zillions Rule Language (saved in the file format ZRF – Zillions Rules File). It was designed to handle mostly abstract strategy board games or puzzles. After parsing the rules of the game, the system's Artificial Intelligence can automatically play one or more players. It treats puzzles as solitaire games and its AI can be used to solve them [ZoG, 2009].

A game defined using the Zillions of Games language format (ZRF) consists of four basic parts. Each of these parts will be discussed in turn. The first part is the game description, which contains the title of the game, a description of the game, some history, and some strategy information. This allows the inclusion of interesting meta-information, but this information contributes with nothing for the games’ creation and as such it can be considered irrelevant [Dobbe, 2007] [ZoG, 2009].

The second part of the ZRF file contains a definition of the objects in the world: the pieces and the players. Multiple pieces can be defined, including their look for each player, some meta-information, and the moves allowed for that piece.

The pieces must then be placed on the board, which is the third part of the ZRF file. The board is given a shape, dimensions and a look. The initial layout is specified using the pieces defined before. The last part of the ZRF file is used to determine the win-or-lose condition, this is a rule defining when a certain player has won or lost the game.

Zillions of Games recognized the need for a different language to define AI modules in. Conventional programming languages are better suited for this purpose than the Zillions of Games language. Therefore a plug-in framework has been created in which authors of a game can provide a special AI module to perform the AI for their specific game, while allowing the game to be specified in the Zillions of Games format.

The language also has support for macros that ease development. Shortcuts can be defined up-front to prevent repetitive typing when using the same concept; in effect this extends the existing language with more specific purpose constructs. [Dobbe, 2007] [ZoG, 2009].

Figure 2.2 shows the start-up menu of the application, where users are invited to choose from a selection of pre-defined popular games.



Figure 2.2: Zillions of Games Start-up Menu [ZoG, 2009]

### 2.3.2 The LUDÆ Project

The LUDÆ project, for instance, is an attempt to create such a General Game Playing system with learning capabilities, utilizing multi-agent systems and genetic algorithms for their learning capabilities. This project, as well as defining a new way of categorizing abstract games, has also defined a generic way of describing these games with set theory and on more programming-friendly terms [Neto, 2003].

A game consists of a set of gaming material (the board and the piece set), the rule set (which defines the set of valid moves for a given game state), the endgame function (a function that defines when the game ends) and an initial setup (the initial state for a game). A game state is defined by a certain board position, the next player and some extra information (e.g., the number of off-board pieces, the actual game phase, number of captured pieces, etc.) visible to both players. More formally:

The board is a set of labels, or cells, and a set of ordered pairs of cells, or links:

- Each cell is associated with a value, which defines if it's empty or occupied by a certain piece (white/black; soldier/king)
- A board may, optionally, have labels associated with set of cells, called sectors.
- A position is a board with a set of values associated for each cell

A state consists of: a board position, information on who the next player is and an optional set of registers defining extra information.

The set  $S$  is the set of all states for game  $G$ ,  $S_0$  being a special state called the game setup.

The rule set  $R$  is a relation  $S \times S$

- Each pair  $(S_i, S_j)$  in  $R$  is called a legal move
- $S$  is a valid state if and only if  $S = R^n(S_0)$ , i.e., if there is a set of  $n \geq 0$  legal moves that reaches  $S$  from the setup
- All valid states  $S_F$  from which there are no  $(S_F, S)$  in  $R$  are called final states.
- A game allows passes if  $R$  is reflexive
- A game is invertible if  $R$  is symmetric
- A game has a super-KO rule if  $R$  is transitive

The endgame function:  $S \rightarrow N \times N$  is defined as:

- $\text{endgame}(S) = \langle 0, 0 \rangle$  if  $S$  is not a final state
- $\text{endgame}(S) = \langle 1, 1 \rangle$  if  $S$  is a final state and a win for the first player
- $\text{endgame}(S) = \langle 1, 0 \rangle$  if  $S$  is a final state and a draw for both player
- $\text{endgame}(S) = \langle 1, -1 \rangle$  if  $S$  is a final state and a win for the second player

The rule set  $R$  is also called the game tree

- The set of legal moves starting at  $S_0$  which are known and considered as good or bad moves, is called the opening theory of the game
- The set of valid states from which it is known all legal moves until some final state is reached, is called the endgame theory of the game
- The set of all other valid states is called the middle game.

An action is a function  $S \rightarrow S$ .

- An action is an atomic change on the game board (e.g., insert/remove a piece or cell, change a game register, etc.).
- A set of actions  $A_1, A_2, \dots, A_n$  define a legal move for a given state  $S$ , if and only if  $(S, A_1 \circ A_2 \circ \dots \circ A_n(S))$  belongs to  $R$

A game policy is a function  $P: S \rightarrow A^n$  that decides for each valid state, what should be the next actions.

- A tactic is a utility function returning a real for each valid state.
- A strategy is a weighted sequence of tactics.
- The policy then uses one or more strategies to maximize the expected value of the next state, and so to decide which will be the next move, i.e.,  $P(\text{state}) = \arg \max_{\text{actions}} (\text{strategy}(\text{actions}(\text{state})))$

In most games, the rule set  $R$  cannot be stated explicitly (the game tree may be infinite or extremely large), so there must be an implicit description to define  $R$ . A low-level description focuses on coding the procedure *validMoves(state)*, which gives the set of valid moves from a given legal state. A high-level description focuses on defining a language to represent rules (Zillions' ZRF is the most used and successful example) [Neto, 2003].

## 2.4 Conclusions

The research topics covered in this chapter try to offer different views on board game analysis, AI approaches and general game playing systems. The focus was to understand the requirements for developing a system capable of describing such games, as well as allowing both users and computer AI engines to test them.

Board games vary tremendously in theme and mechanics, and the distinctions from each other are what make them fun to play as well as making them interesting study cases for advancements in AI. Abstract games provide an excellent environment to apply AI techniques, as they are perfect information games, where there are no probabilities or hidden information involved. To better understand this vast field of games the research was focused on different views on how to define their mechanics and how to classify and categorize them.

Four different AI approaches to board games were covered as well, explaining the methodologies of each one and how they apply to abstract games. The AI approaches covered in the research are Specific Knowledge Based Systems, Evaluation Functions, Machine Learning and Evolutionary Algorithms.

Lastly, a study of the most significant general game playing systems for abstract games was performed. This study revealed that the Zillions of Games platform already featured, although partially, some of the main objectives of the project, and was consequentially chosen to be the testing ground for the games generated with the Generic Interface.

## Chapter 3

# Project Overview

The idea of being able to create any game via a simplified application and without the need to have any particular programming skills is not new. There are already some platforms that allow users to do that, mostly for platform games or role playing games. However, in the case of abstract strategy games, such applications do not yet exist.

The capabilities of the General Game Playing platform Zillions of Games are remarkable, they enable the creation of any abstract game or puzzle via the ZRF language and the application is then able to play the games or solve the puzzles applying a powerful rule analysing AI engine [ZoG, 2009].

The aim of this thesis is the creation of a software platform capable of doing just that, with a game creation graphical interface more suitable for the general public. Thus, instead of designing a completely new system, which would probably prove to be an impossible task in the very constrained time available, the development of a generic interface capable of generating abstract strategy games in the Zillions of Games format became the main goal of this thesis.

The first section of this chapter focuses on describing abstract games in terms of board, movements, pieces, goals and rules. This ASG analysis and decomposition is a necessary step because it is essential to designing and implementing the Generic Interface.

Although Zillions of Games is currently the only suitable application for creating and running playable ASG, defining the games in an intermediate language that could later be understood by a different platform became another goal of this project. The expandability and generalization provided by this methodology may prove to be an essential step in the further development of this project. The second section of this chapter further details this idea.

The final step of the project is the improvement of existing AI techniques to solve abstract strategy games, analysing the potential of methodologies like machine learning and genetic algorithms with AI agents. The third and final section of this chapter describes how this AI improvement can be done.

## 3.1 Creating Abstract Strategy Games

The main problem in trying to generalize the creation of ASG is that each game is unique for a reason, usually in the form of a particular rule that makes a game different from all the others, changing strategy and tactics in wholly distinct ways. For example, if taking a game of Checkers and removing the rule that forces the capture of an enemy checker the way the game is played, the way a player must think to achieve victory is changed. The tactical sacrifice of pieces is no longer necessary to compel the opponent to play along to one's strategy.

Another key issue to solve is the categorization of ASG by similarities in rule sets and game characteristics. This categorization is necessary for understanding what is or is not commonplace in one game category, what rules can be generalized and what rules make a game intrinsically unique.

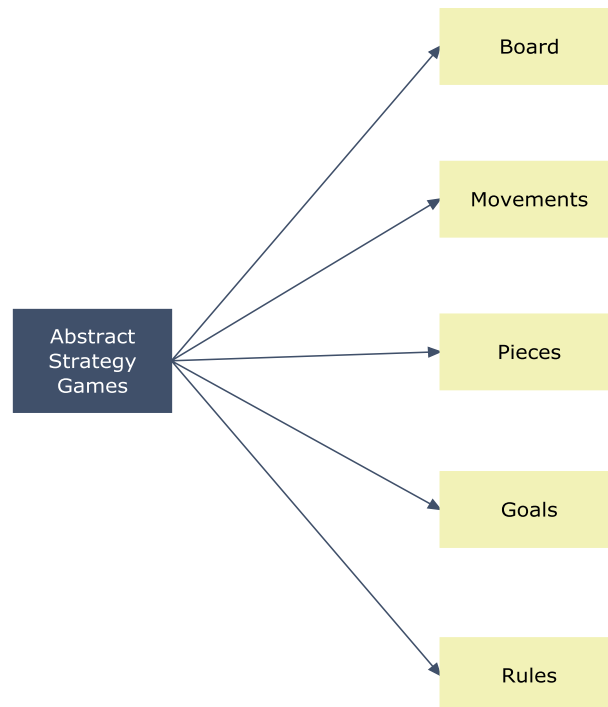
### 3.1.1 Defining an Abstract Strategy Game

In order to generalize ASG it is necessary to divide a game in key components. This division must ensure that all components are common to all abstract games and in a way that they can be easily recognizable and defined by the user, keeping in mind that the ultimate goal is ASG creation.

According to what is defined in section 2.1.2 (Game Mechanics) and the different categorization methodologies presented in section 2.1.3 (Classification Systems), a new scheme for describing ASG was created specifically for this project. The way ZoG defines games was also considered, on one hand because one of the goals of this project is to generate Zillions game code and on the other hand to heed the recommendations set by IAGO on ASG categorization.

Every abstract strategy game can be decomposed in the five components described in Figure 3.1. A game is played on a board, where players control the movements of pieces in order to reach a set of goals and following a set of rules that govern gameplay. All these components are related, and a game cannot be described with their absence.





**Figure 3.1: ASG Component Diagram**

The board is the play area, or in other words where the action of a game occurs. Pieces move through the board, goals can be defined by set positions, movements are restricted by the board dimensions and positions.

A movement can be considered an abstraction of how a specific piece interacts in the board and with other pieces. Instead of having movements defined solely for a piece or a type of piece one can describe a generic move such as ‘drop in an empty board space’ and then associate a piece with this movement. This way it is possible to separate a piece and its attributes from the movements it is able to do. For instance, considering the movements ‘slide across the board orthogonally’ and ‘slide across the board diagonally’ the movements of three chess pieces can be described: the rook, the bishop and the queen.

Pieces are the tokens that players control and own on an abstract game. They represent each player and the way a game develops is solely due to interactions between pieces and the board. The object of any game is to allow each player the opportunity to achieve a goal by ‘playing’ with the pieces, which means that a player must use the abilities (movements or other properties) of his pieces each turn to win the game.

Despite the different attempts to categorize ASG one aspect seems to be essential for differentiating one game from the next – the goal. Abstract games are games, and games imply that there should be a way to win, a set of steps necessary to undertake in order to emerge victorious

in the game. In an ASG one player must try and overtake his opponent, and the goals define how that can be done. A game can be won by removing all of the opponent's pieces from the game, or occupying a specific position in the board, or not allowing the other player to move at all.

Rules are what govern a game's universe. They define what a turn is, what the players can and cannot do, and how the board, piece and goals interact with one another. They are basically what cannot be implicitly declared in the other components, as they affect the game as a whole.

### 3.1.2 Board Definition

Abstract games are board games, and as the name suggests the board is a mandatory part of any board game. In ASG boards can have a multitude of different configurations, and they are not even restricted to the three dimensions of space, keeping in mind that an abstract game is always a logical puzzle. Figure 3.2 shows the components that are part of the board definition.

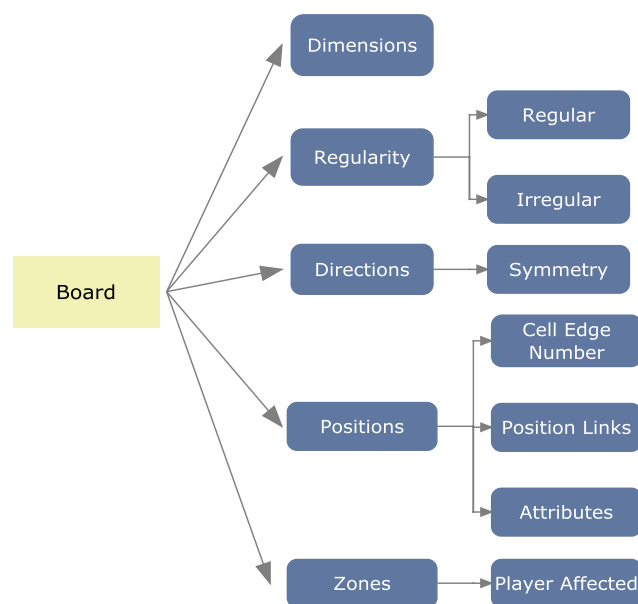


Figure 3.2: Board Definition Diagram

#### 3.1.2.1 Board Dimensions

Board dimensions define how many dimensions affect the board. There can be boards with only one dimension (games played along a single line of cells), the typical two-dimensional boards, three-dimensions and the more far-fetched and impossible to represent in the real world

four-dimensional boards and beyond. A game played in a one dimensional board is bound to be simplistic; the number of interactions between cells is largely reduced (each cell can only have two neighbouring cells at most). ASG in general are played in two dimensions, but some like *3D Tic-Tac-Toe* represent a challenge by extending the board to the third dimension, multiplying the number of cells and cell interaction in the board. Games with more than three dimensions escape the grasp of human reasoning, but computer applications with AI have no problem as they are handled as purely logical problems.

### **3.1.2.2 Board Regularity**

Most ASG have regular boards built up by connected positions of equal size and shape, forming a regular polygon shaped board (a square in *Chess* or a hexagon in *Alquerque*). Nevertheless, there are some games whose board definition must be more complex to allow for different sized and shaped positions, and this feature of some boards makes it necessary to have an Irregular Board type in board definition, so as to differentiate from simpler, more easily defined boards.

A regular board is a set of equal size and shape positions. For rectangular boards it is even possible to use a Column x Row number notation. An irregular board on the other hand is built up by different sets of positions, and each set can have any number of positions and form any kind of shape.

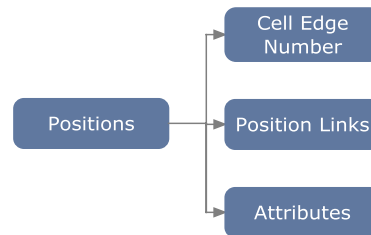
### **3.1.2.3 Board Directions**

In order to define how a movement will be performed on the board it is necessary to represent the links between positions with directions. In a 2x2 board with links between all neighbouring cells it is possible to define directions to represent the movement from one cell to the other. For this example one can use either the coordinate system with 'North', 'South', 'East' and 'West' to represent the directions of movement or simply 'Up', 'Down', 'Left' or 'Right'. It doesn't matter what names are chosen to represent the links, provided they are not ambiguous and self-explanatory. In 3D boards more directions are necessary, like In and Out for instance.

Symmetry is also an aspect to consider in board definition, essentially due to the fact that most games' boards are symmetric and pieces behave or are positioned symmetrically on the board. Saying that a board has vertical symmetry means for both players their pieces are moving forward but they are going in opposite directions nonetheless. Essentially symmetry is represented by a pair of opposite directions, like 'Forward' and 'Back', and it means that through one player's perspective his movements are going forward but to his opponent the same movements are going back. This is easily understood by picturing the two players sitting in front of each other, playing a game where symmetry exists (like *Chess*) and understanding that both players

are thinking in the same directions of movement (advancing in the board). To fully describe the game it is necessary to define and clarify this distinction on direction symmetry.

#### 3.1.2.4 Board Positions



**Figure 3.3: Position Diagram**

As shown in Figure 3.3 a board can be built up by any number of cells (positions) linked together in some way. A cell is the atomic part of a board, indivisible. A cell must also be described in number of edges, whose purpose is to define a polygon. This is required otherwise the board cannot be drawn, or the links between cells become hard to visualize.

An edge is important because it represents the boundary of a cell, and it can be the link between two different cells in a board. So, the number of sides each cell of the board has directly affects the number of physical links possible between cells. However, most ASG fall in two categories, square cells and hexagon cells.

There are games where the pieces are placed on vertexes or move along the edges of board cells, instead of within cells. This particularity of some boards could represent the need for further detail in board definitions, but a closer look on these boards and on the placement of pieces in these games can provide a different perspective on how to represent these types of games. Figure 3.4 shows that despite what the fact that the pieces are indeed placed upon the vertexes of the board grid; nevertheless it is possible to create typical square board with the pieces moving inside cells just by outlining the gridlines around the original vertexes. In truth what matters is not if the pieces move inside a cell or in its perimeter, but how to represent any game board in a way similar to all other boards, making board definition generic.

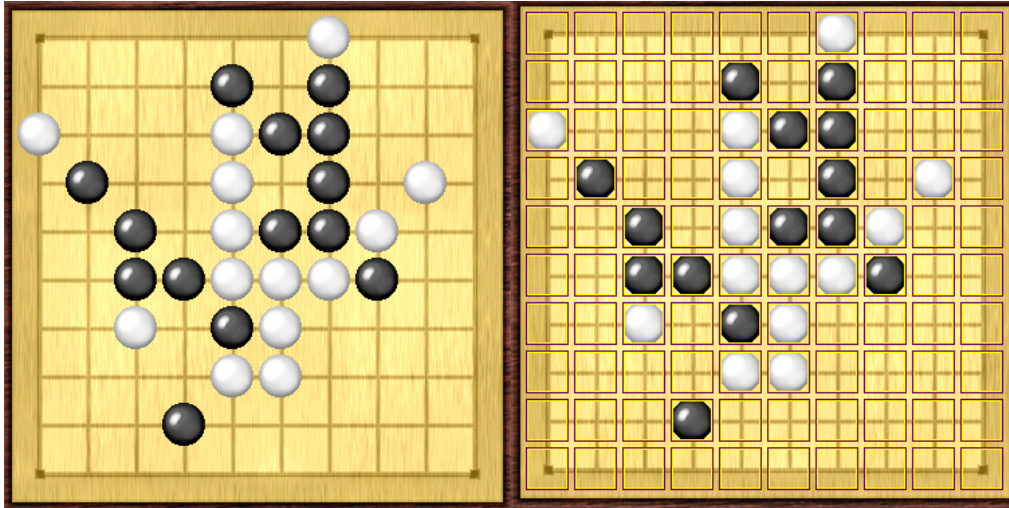


Figure 3.4: Grid lines in a Go board [ZoG, 2009]

Each position has a set of links to other positions, usually its neighbouring positions. It is also possible to have a link to any position in the board. This enables the creation of ‘worm-holes’ in the game, where a piece can travel from a position of the board to another unrelated position. Extending this idea to the sides of a square board, linking the top side with the bottom one and the left side with the right, it is possible to create a completely linked board. In two-dimensional space it is impossible to represent such links in a board, but in three dimensions the board is represented as a torus, the doughnut shaped geometrical surface presented in Figure 3.5. As a result, boards may come in a variety of ways, often difficult to represent geometrically, but the board definitions must be generic enough to allow for such positional links to exist.

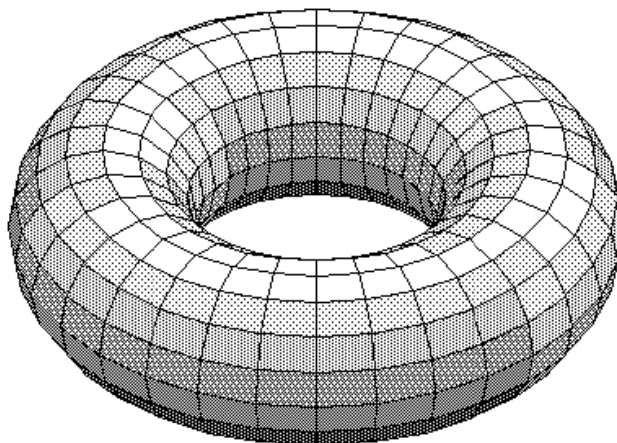


Figure 3.5: Torus Geometrical Surface [Leung, 2009]

Each position in the board can have a set of attributes associated with it, like its ability to support more than one piece (creating stacks of pieces) or simple properties that signal if the

position was occupied during the course of the game or not. For example, a position can be marked as a kill position, or a trap in other words, and all the pieces that would be placed upon it are automatically removed from the game. The nomenclature used for naming positions is also part of a position's attributes, as well as the coordinates in the board and all the pertinent information relating to a singular position.

#### **3.1.2.5 Board Zones**

Some rules of the game can only apply to a discrete number of positions in the board. A zone is simply a grouping of these positions in order to simplify game specification. The need for associating a player to a defined Zone can be explained for instance with piece promotion on the last row, the last row being different for each player. So a promotion zone for one player can mean the last row of the board but to his opponent that promoting zone is his first row.

#### **3.1.3 Movement Definition**

Pieces must be placed in the board or moved across it so that a game's action unfolds. The purpose of movements is to define how a piece interacts with the board and with other pieces (by capturing or flipping). The movement definition diagram is presented in Figure 3.6.

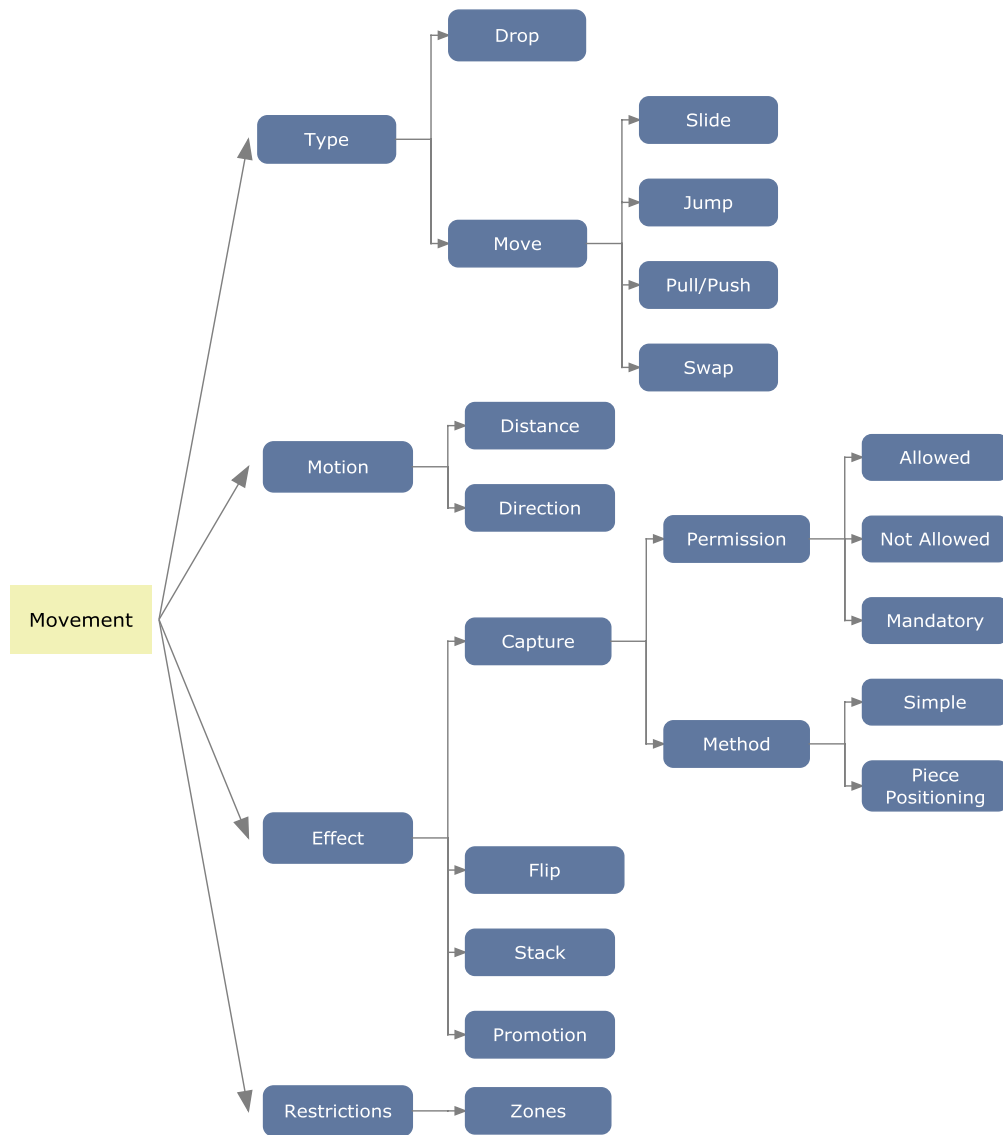


Figure 3.6: Movement Definition Diagram

### 3.1.3.1 Movement Types

The analysis of several distinct ASG revealed that there are numerous ways that pieces move across the board, each with different restrictions and possibilities. The basic distinction is when a piece is placed on the board from the off play area (like in *Tic-Tac-Toe*) and when a piece is already in the board and is required to move across it. These two major movement types are 'Drop' and 'Move' (for the lack of a better term).

In the case of dropping a piece in the board there are a few restrictions that can be applied, like if the position is empty or not, and if the piece can be dropped on top of other pieces (creat-

ing stacks of pieces). This way a drop can be restricted to empty spaces, spaces occupied by friendly, enemy or nay kind of pieces. There is also room for defining drops by means of on board piece positioning (like only being able to drop a piece in a row where another piece is) or with concepts like gravity, where a piece can only be dropped to the last available row (*Connect 4*).

The basic forms of moves available in most ASG are ‘slide’, ‘jump’, ‘push’, ‘pull’ and ‘swap’. There are others, but with some degree of freedom it is possible to create more move types using these five basic ones.

A ‘slide’ is when a piece moves across the board with a direction, not being able to move across obstacles (like friendly pieces). The capturing option may or may not allow the piece to move to positions occupied by enemy pieces. The distance of a slide defines how many spaces in the board the piece shall move across.

A ‘jump’ is a move where a piece goes from its origin position to a target position without passing through any of the connecting positions. However, it is necessary to explicitly declare over what kind of spaces this type of move can jump over. It is possible to jump over spaces occupied by friendly pieces, enemy pieces or empty spaces, and any combination of these three, as can be more thoroughly understood in Table 3.1. Extending this idea it is also possible to have jumps over specific types of pieces, further enhancing the Jump Over options.

**Table 3.1: Jump Over Options**

Jump Over					
Enemy Pieces	Friendly Pieces	Enemy and Friendly Pieces	Enemy Pieces and Empty Spaces	Friendly Pieces and Empty Spaces	Anything

A jump can also be defined by a direction or by a radius value. Directional jumps function in a way similar to slides, given a direction the piece shall jump across the board in that direction (like the checker diagonal jump in *Checkers*). Radius jumps function differently: given a radius value the jump will be the sum of all the possible two direction trajectories leading to the target position. For instance, a jump with radius 3 can symbolize the Knight leap in *Chess* (jumping in ‘L’) by moving two positions in one direction and one in the other, or moving one position in one direction and then two in the other (see example in Figure 3.7), or simply 2+1 and 1+2. The radius value number is the sum of the distances travelled in both directions, so by having a jump radius of 4 the number of possible jumps is increased as the possible choices of directions are 3+1, 2+2 and 1+3, with radius 5 the jumps are 4+1, 3+2, 2+3, 1+4 and so on.



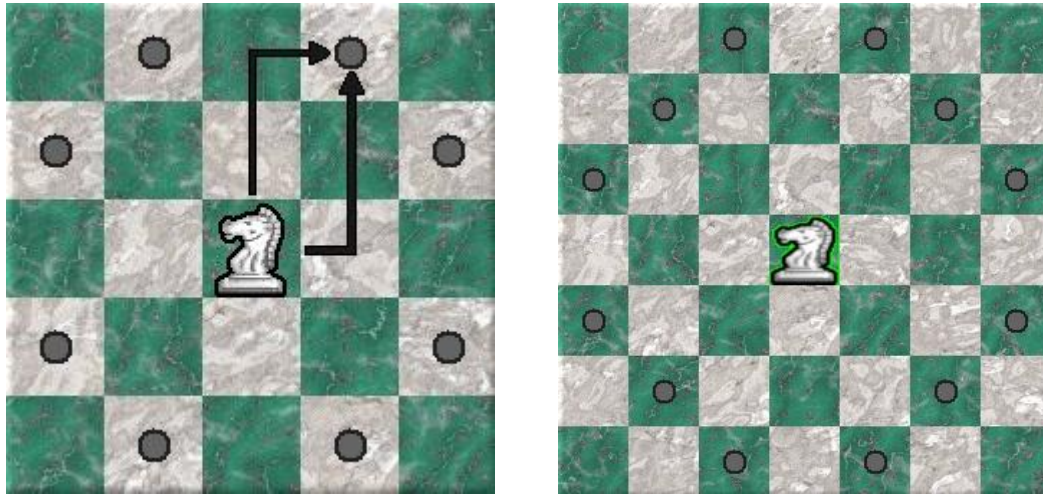


Figure 3.7: Knight Move with Radius 3 (left) and with Radius 4 (right)

Pushing and Pulling are movements required by some games like *Alquerque*, where moving a piece requires it to move adjacent pieces along with it. It is necessary to define what types of pieces are to be moved this way (enemy, friendly or specific piece types).

Swapping is a movement that, and as the name suggests forces two pieces to swap places with each other. It can be argued if this is really a movement *per se*, but as some games use this feature it was decided to include it in the movement definitions. Swapping places with an empty square it is possible to define a ‘teleporting’ type movement, where a piece is free to move to any space in the board. Therefore, a ‘swap’ move can be distance and direction independent. Another possibility is not having the piece move at all, but using this movement to create a capture by distance type movement, enabling moves that represent shooting an arrow or a laser (like in the game *Amazons*).

### 3.1.3.2 Movement Motion

The motion attribute of a movement can be either described in terms of direction and in terms of movement. A piece can move any number of positions depending on the distance factor, and in any number of directions depending on the direction factor. Table 3.2 shows the possible distance options for movement definition.

Table 3.2: Move Distance Options

Distance			
Any	Up To	Exactly	Furthest

In the case of jump type moves it is also possible to have a radius distance, as described above.

### 3.1.3.3 Movement Effect

When a movement occurs several actions can be performed on the pieces involved in the movement, such as capturing pieces, flipping pieces (changing owners), creating or increasing stacks of pieces or promoting to different piece types.

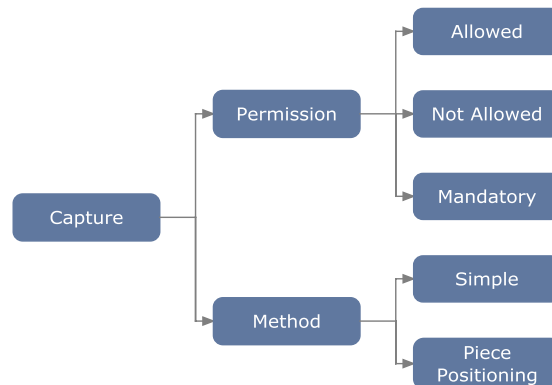


Figure 3.8: Capture Diagram

Expressing what capturing options a move has is required because it can alter the way the movement is going to be performed. If a slide movement can end in a capture of an enemy piece then that means that it is necessary to verify if the target position is occupied by an enemy piece. Capture permission defines if a capture can occur or not, and if that capture is mandatory (the piece cannot move unless for capturing purposes). Another characteristic of capturing is the method used for removing enemy pieces from the game. The simple method is when a movement ends in a position occupied by the enemy piece, thus capturing it. During jump type moves it is also possible to capture pieces during the jump, not only when the piece lands on its target position. Piece Positioning Capturing refers to the possibility of capturing pieces only when certain conditions of piece positions on the board are met (like in *Tablut* where capture occurs when a piece is between two of the opponent's pieces). These capturing components are shown in Figure 3.8.

Other noteworthy effects a movement can have are flipping, stacking and promoting. When a piece flips it changes ownership. Some abstract games revolve around the exchange of piece control between players until one of them meets the criteria to win the game. In games where it is possible to have multiple pieces in the same position of the board these pieces form stacks. Stacks can have properties (sometimes represented as unique pieces altogether) or pieces in a stack become unable to move until they are the first on the stack. Promotion on the other hand removes the current piece from the game and replaces it by another type of piece when the movement meets a certain criteria (reaching a zone for instance). A generalization of the promotion attribute without actually moving the piece may represent adding pieces to the game (as opposite to the capture by distance concept presented above). Yet another use from promotion

can be when it is necessary to replace a piece in the game with another piece exactly like the original, but with its attributes reset (like flags signalling if the piece has been previously moved).

### 3.1.3.4 Restrictions

Movement restrictions apply when it is necessary to restrict a movement to a zone on the board, either by only allowing the movement to be performed if the piece is within the designated zone or forbidding the move altogether. Other options may include the verification of the target position belonging or not to a zone or even if the pieces in between the trajectory of the movement are in or out of designated zones.

### 3.1.4 Piece Definition

The whole point of an abstract strategy game is to have the players interact with their pieces, trying to somehow fulfil the game's objective in order to win. Pieces represent an essential role as they are the tokens that the players control and move in the game board. The diagram that shows their components is shown in Figure 3.9.

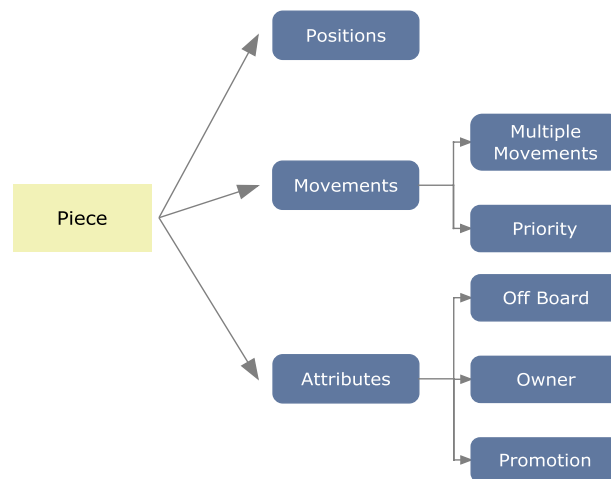


Figure 3.9: Piece Definition Diagram

#### 3.1.4.1 Piece Positions

The initial setup of a board is required in some games to define where the pieces are positioned at the game's start. Therefore, 'Piece Positions' represents the set of positions that con-

tain the piece. During the course of the game this set is changed as the pieces move around the board, are captured or dropped, and so on.

#### **3.1.4.2 Piece Movements**

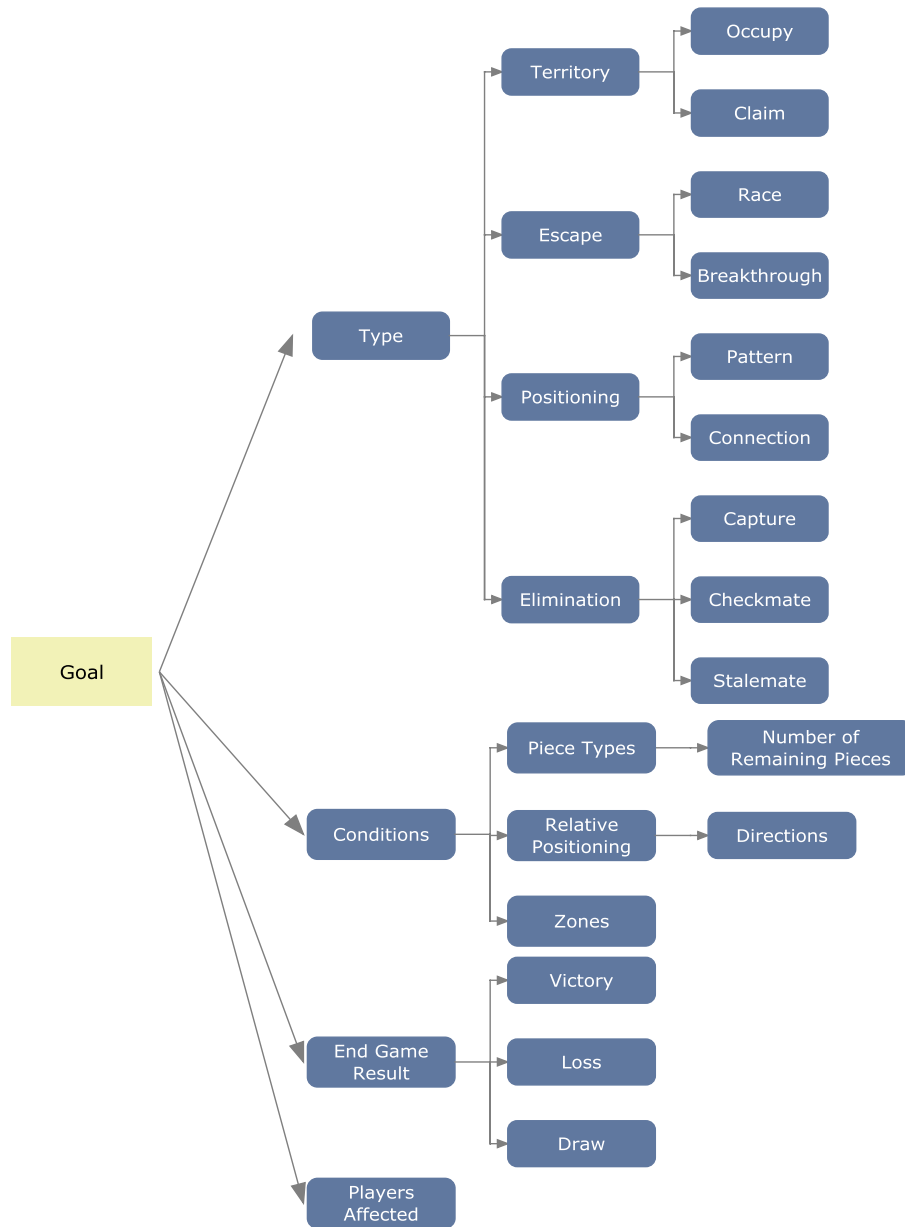
Each piece has a set of movements that allow it to be moved across the board. However, there are games in which a piece can have multiple movements, or movements that when performed allow the piece to move again (like a capture in *Checkers*). For these multiple movements it is possible to define a new set of movements that represent the sequence of moves that generate the final multiple movements. It is also possible to attribute a priority level to a movement of a piece, which in turn forces the piece to only perform low priority movements when the higher priority ones are unavailable.

#### **3.1.4.3 Piece Attributes**

This section of Piece Definition covers the number of off board pieces, that is to say the quantity of pieces that are not placed in the board at the beginning of the game, and can only be placed if the piece in question has a movement of the type ‘drop’. An additional attribute of a piece defines who its owner is at a given point in the game (given that this can change if a flip occurs). The final attribute is promotion, is a set of piece types that define to what kind of piece it can promote.

### **3.1.5 Goal Definition**

Goals define how an abstract strategy game ends and what are the conditions needed to be fulfilled for one of the players to be declared the winner (or the loser). A diagram of the different aspects of goals is shown in Figure 3.10.



**Figure 3.10: Goal Definition Diagram**

### 3.1.5.1 Goal Types

The ASG categorization of IAGO shows that games can be divided in four categories, Escape, Territory, Positioning and Elimination. This categorization looks at goals as the main distinction point between games, as the goal of a game dictates every action of the players to achieve a means to an end. So, these categories provide an excellent clue as to how the game flow will be like as well as providing some strategical insight. The only difference in this pro-

posed categorization from the one proposed by IAGO is the distinction of 3 subcategories of Elimination: Capture, Checkmate and Stalemate.

In IAGO's point of view the subcategories are Royal Capture, Royal Checkmate, Multi-Piece Capture and Multi-Piece Immobilization. For instance, IAGO considers that only one piece is liable to be checkmated, but that is not necessarily true as one can design a *Chess* variation with more than one King. Another difference is that it is simpler to remove the distinction between royal (one) and multi-piece (several) and create a condition verifying the number of pieces involved in a goal of this type.

A more thorough explanation of each goal type can be found in Table 2.3 of section 2.1.3.3, keeping in mind the slight name changes present in Table 3.3 and the Elimination category alteration aforementioned.

**Table 3.3: IAGO Goal Types and Generic Interface Comparison**

Original Goal Type Subcategories	Generic Interface Goal Type Definitions
Bi-Sided	Race
Uni-Sided	Breakthrough
Arrangement	Pattern
Royal	Capture Checkmate Stalemate
Multi-Piece	

### 3.1.5.2 Goal Conditions

Defining a goal type will only alter the way the rest of the goal definition conditions are perceived and used, as some of them might even prove unnecessary with some goal types.

Table 3.4 shows what goal types make use of each condition.

Table 3.4: Goal Types and Conditions

	Piece Types	Zones	Number of Re- maining Pieces	Relative Piece Positioning
<i>Occupy</i>	✓	✓	✗	✗
<i>Claim</i>	✓	✗	✓	✓
<i>Race</i>	✓	✓	✗	✗
<i>Breakthrough</i>	✓	✓	✓	✗
<i>Connection</i>	✓	✗	✗	✓
<i>Pattern</i>	✓	✗	✗	✓
<i>Capture</i>	✓	✗	✓	✗
<i>Checkmate</i>	✓	✗	✗	✗
<i>Stalemate</i>	✗	✗	✗	✗

Piece Types is a set of pieces that are affected by the goal, so if a game has numerous piece types only the ones existing in piece types influence the game's end. For instance the definition of *Chess*'s checkmate goal requires the King as the only piece in Piece Types.

The condition regarding Zones represents a set of positions (defined by the Zones in the board, as mentioned in section 3.1.2.5) in which the pieces in Piece Types are required to be in order for the goal condition to be met. Depending on the goal type the way Zones affect the goal may differ: in Race and Breakthrough the objective is to get all of the pieces in Piece Types reach positions defined in Zones, but in Occupy it's the other way around as all the positions in Zones need to be occupied by pieces in Piece Types.

The Number of Pieces Remaining condition is necessary for Capture and Breakthrough type goals, indicating how many pieces a player must have in order not to lose (if his remaining pieces fall behind the number defined it means his opponent has captured the rest of his pieces). In Claim type goals this counter is used in conjunction with the Relative Piece Positioning condition to define how many pieces are required for determining the winner, as this type of goal can have a player 'own' positions in the board without actually occupying them with his pieces but through relative piece positioning (like owning an NxN square in the board by having only the vertexes of the square occupied).

Relative Piece Positioning defines how pieces must be ordered in relation to each other, either to form patterns (Pattern), or a chain of pieces (Connection) or even for drawing shapes in the board (Claim).

It is worth mentioning that Stalemate doesn't require any particular condition, given that a stalemate occurs when one player has no more moves left. The only particularity of this goal is

that it is not possible to have a Stalemate goal in games where players can pass their turn, not moving pieces at all. The game goal would never be verified because the player could always pass his turn.

#### **3.1.5.3 Goal End Game Result**

There are three possible outcomes of a goal fulfilment: winning the game, losing it or ending the game in a draw. By separating the result from the goal type definitions it is possible to have any number of combinations, like winning the game if the King is checkmated instead of losing it.

Most ASG have more than one goal defined, mostly for defining what qualifies as a victory and what represents a draw. Taking yet again the example of *Chess*: King is checkmated represents a loss, but a stalemate represents a draw. *Chess* therefore has two distinct goals and two different end game results for each goal. Other games can have a multitude of combinations for victory, and this possibility of multiple endings is what makes a game's strategies and therefore its study more intriguing.

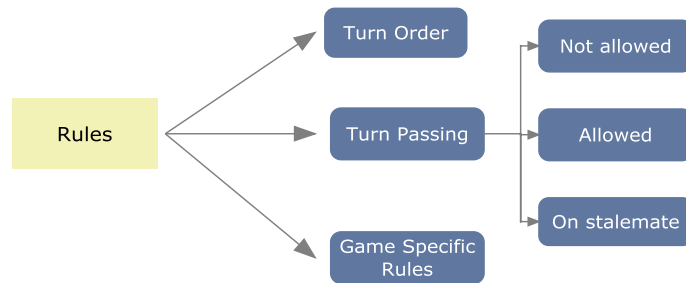
#### **3.1.5.4 Goal Player Affected**

By separating the End Game Result from the Goal Types it became obvious that the player affected by the goal could also be subject to a similar distinction. One player can therefore be affected by a series of goals that mean nothing to his opponent, and vice-versa. This is particularly important for the Breakthrough type goals, where one player needs to get to the designated goal positions while his opponent tries to prevent him from achieving his objective.

### **3.1.6 Rules Definition**

Game rules that cannot be defined in terms of piece and their movements, or in terms of goals or in the board definition fall into this category. Rules Definition are therefore used to represent turns in an ASG and also to define game specific rules that cannot be declared in the previous ASG Definition categories. Figure 3.11 shows the rules' components.





**Figure 3.11: Rules Definition Diagram**

### 3.1.6.1 Turn Order

Abstract strategy games are played in turns, where one player performs some sort of action to the game state and then the other player has a chance to do the same. However, it is possible for players to move more than one piece in the same turn or even have games. To encompass this feature of some ASG it is necessary to be able to define how turn ordering occurs in a given game. Some games even have an increasing number of moves for each player, where the first to play has the right to move one piece, the second player then moves two pieces, then the first player moves three and so on. Turn Order helps define these possibilities.

### 3.1.6.2 Turn Passing

Some games have the possibility of a player to pass his turn instead of performing a movement, while others force the player to pass his turn when he has no more moves available.

The three possibilities for Turn Passing consequently are: no turn passing allowed (like in most ASG); turn passing allowed (a player can choose to give up his turn, not altering the game) and turn passing on stalemate (the turn is passed automatically when one player is on a stalemate situation).

### 3.1.6.3 Game Specific Rules

What makes an ASG unique is the set of rules that differentiate it from the other games. The idea of this project is to make ASG creation the more generic as possible, but there are always situations where the generalization falls short and it is necessary to define a rule that doesn't belong to any specific category.

A good example of this is Castling in *Chess*: if the positions between the King and one of the Rooks are free and not being attacked by enemy pieces, if neither the King or the Rook has moved in the game, the King can slide two spaces closer to the Rook and the Rook in turn can jump over the King and land on the position on his side. The jump can be of two spaces to the

left in the case of short Castling (right side Rook) or in the case of long Castling three spaces to the right (left side Rook).

There are so many particularities in this rule that it is almost impossible to make its creation generic. It is possible to restrict movements if the spaces in between the original position and the target position are being attacked; it is also possible to verify if a piece has already been moved; it is possible for a player to be allowed more than one move per turn and to only give that player the option to move another piece if a certain move is chosen. However, putting together all these conditions to create a single movement with all the implied restrictions is counterproductive to trying to generalize movement creation.

### **3.1.7 Conclusions**

The diagrams and definitions presented in this section are meant to help the design and implementation of an interface capable of offering users a discrete set of options with which to generate numerous different abstract strategy games. There are bound to be cases where that is impossible, and the Zillions of Games creators do not guarantee that the ZRF language is capable of generating all ASG [ZoG, 2009].

The point of this project is generalization, and if this generalization means sacrificing a bit of the particular features of some games then on the overall picture the cost is not so great. However, implementing ever more features mean that the number of rules that the Generic Interface will be unable to create will decrease, and the ultimate goal is to give the users complete freedom on abstract game creation.

By dividing ASG in the five components described above (Board, Movements, Pieces, Goals and Rules) the methodologies necessary for developing a prototype for the Generic Interface become clearer. By analysing different game mechanics and different views on how to categorize and classify ASG it was possible to distinguish these basic components, in the hope that their accuracy will prove fundamental in generic game creation.

## **3.2 Zillions of Games Platform Analysis**

The project intends to develop an application that can work as an intermediate step between users and the ZRF language file that Zillions needs to understand game rules. The idea is to present users with a multitude of choices in a visual interface and then translate these choices, herein the new game itself, to a ZRF format readable by the Zillions of Games platform.

In order to create a code generation module that generates games in the ZRF format it is necessary to analyse the language Zillions uses. This language was designed keeping in mind the main objective of the application, board game creation, and several useful keyword functions were included to facilitate game programming. The language uses S-expressions to define each of the game's components, using a specialized parser to identify keywords, defining what type of tokens can be associated with each keyword. For instance, pieces can have two types of movements: drops (requiring off board pieces) and regular moves. However, the most important aspect to consider in the Zillions language is how to define the game goals, and especially how to adapt the keywords and game logic capabilities of the Zillions language to the Generic Interface.

### 3.2.1 Goal Analysis

Pieces and boards in the ZRF are fairly simple to define, as they require specific attributes and follow a restrict template. For example, a piece must have a name, images and movements all included within the piece construct. Abstract games have an enormous variety of endgame possibilities, and a standard format for goal definition does not exist in the Zillions language, as each goal type has very specific options.

Instead of having goal keywords to define each goal type as defined by IAGO (section 2.1.3.3), Zillions has four top-level expressions to define end game situations, all in the form of conditions: count, win, draw and loss-condition. Win, draw and loss all function in a similar way, the only variation being affecting the game outcome as their name implies. Count-condition however uses the number of pieces in the game to determine the winner. Each of these conditions requires further expressions to be declared, such as goal keywords (like stalemate or checkmate), the players affected by the condition, what type of pieces to consider, and so on.

Table 3.5 contains an analysis of all the possible combinations of the goal condition expressions and the possible goal keywords associations, as well as other attributes necessary for each condition to be verified in the game's logic. The effect of each pair (condition and keyword with attributes) represents a goal type definition, and its end game effect is also explained. For simplification all the effects consider regarding the win, draw, loss conditions are explained as though it is a win game situation if the condition is verified. The possible attributes for each goal are defined between '<' and '>', and depending on the goal they can contain player names, piece types, board positions or zones, a specific numeric value or a direction.

The ZRF notation present in Table 3.5 is useful for the next step of the development process, where these goals must be implemented in order to generate the ZRF files via the Generic Interface. Some of these goals, like Claim for instance, have so many possible combinations and

outcomes that their definition in a generic fashion may prove too complex to implement, requiring very specific options to fully implement all the possibilities.

**Table 3.5: ZRF Goal Notation Analysis**

Condition	Goal Keyword and ZRF Notation	Goal Type and Effect
count-condition	<i>&lt;players&gt; stalemated</i>	<b>Claim:</b> when no more moves are available for either player, the player with the most pieces on board wins
	<i>(total-piece-count &lt;number&gt;)</i>	<b>Claim:</b> wins the player with more pieces when the total number of on board pieces reaches a given value
	<i>(total-piece-count &lt;number&gt; &lt;piece-type&gt;)</i>	<b>Claim:</b> wins the player with more pieces of a specified type when the total number of on board pieces of that type reaches a given value
win-condition	<i>(total-piece-count &lt;number&gt;)</i>	wins the last player that moved when the total number of on board pieces reaches a given value
	<i>(total-piece-count &lt;number&gt; &lt;piece-type&gt;)</i>	wins the last player that moved when the total number of on board pieces of a specified type reaches a given value
	<i>(&lt;player&gt;) (total-piece-count &lt;number&gt;)</i>	<b>Claim:</b> the specified player wins when the total number of pieces reaches the specified value
	<i>(&lt;player&gt;) (total-piece-count &lt;number&gt; &lt;piece-type&gt;)</i>	<b>Claim:</b> the specified player wins when the total number of pieces of a given type reaches the specified value
	<i>(&lt;players&gt;) (pieces-remaining &lt;number&gt;)</i>	<b>Capture, Breakthrough:</b> wins the player whose total number of on board pieces reaches a given value
	<i>(&lt;players&gt;) (pieces-remaining &lt;number&gt; &lt;piece-type&gt;)</i>	<b>Capture, Breakthrough:</b> wins the player who has a specified number of pieces of a given type on board
	<i>(&lt;players&gt;) (absolute-config &lt;check-occupant&gt;...&lt;check-occupant&gt; (&lt;position-arg&gt;...&lt;position-arg&gt;))</i>	<b>Breakthrough, Race, Occupy:</b> wins the player who has a piece of the specified types at the given positions or zones
	<i>(&lt;players&gt;)(relative-config &lt;check-occupant&gt; &lt;direction&gt; &lt;check-occupant&gt;...)</i>	<b>Claim, Connection, Pattern:</b> wins the player who has pieces of the specified types in the direction of another piece, forming a pattern
	<i>(&lt;players&gt;) stalemated</i>	<b>Stalemate:</b> wins the player who is stalemated
	<i>(&lt;players&gt;) repetition</i>	<b>Stalemate:</b> wins the player who repeated the same move 3 times in a row
draw-condition	<i>(&lt;players&gt;) (captured &lt;piece-type&gt;...&lt;piece-type&gt;)</i>	<b>Capture:</b> wins the player when a piece of the specified type is captured
	<i>(&lt;players&gt;) (checkmated &lt;piece-type&gt;...&lt;piece-type&gt;)</i>	<b>Checkmate:</b> wins the player when a piece of the specified type is checkmated
loss-condition		

To simplify the process of defining the goals in the implementation phase, organizing all the end game goal possibilities present Table 3.5 to fit the requirements set forth in the goal definitions of section 3.1.5.2. Table 3.6 contains the relation between the goal options Zillions has and the goal types defined for this project, described in a less restrictive format than the

ZRF notation of Table 3.5. Again, to simplify, all the effects consider the game ending in a win situation.

**Table 3.6: ZRF Goal options organized by Goal Types**

Goal Type	Options	Effect
Claim	<i>no more moves available</i>	wins the player with the most pieces on board when there are no more moves available
	<i>total amount</i>	wins the player with the most pieces on board when a specified amount of pieces are in play
	<i>total amount &amp; piece type</i>	wins the player with the most pieces on board of a specific type when a specified amount of pieces are in play
	<i>player pieces total</i>	wins the player that has a specified number of on board pieces
	<i>player pieces total &amp; piece type</i>	wins the player that has a specified number of on board pieces of a specific type
	<i>relative piece positioning</i>	Wins the player that owns the most spaces based on position of pieces on board
Occupy	<i>absolute positions</i>	wins the player who has more pieces (or pieces of a specific type) in the designated goal positions (zones)
Race	<i>absolute positions</i>	wins the player has his pieces in all of the goal positions
Breakthrough	<i>absolute positions</i>	one player wins if he has some or all of his pieces reach goal positions
	<i>pieces remaining</i>	one player wins if the opponent's pieces (or pieces of a specific type) are less than the designated number
Connection	<i>relative piece positioning</i>	wins the player that arranges his pieces in a type of chain designated beforehand
Pattern	<i>relative piece positioning</i>	wins the player that arranges his pieces in a pattern designated beforehand
Stalemate	<i>stalemated</i>	wins the player that has no more moves left
Checkmate	<i>checkmate</i>	wins the player that has some or all of his pieces checkmated
Capture	<i>captured</i>	wins the player that has captured a specified opponent 's piece
	<i>pieces remaining</i>	wins the player who has reduced his opponent's pieces to the designated number

### 3.3 Generic Language for ASG

The research undertaken in section 2.3, General Game Playing Systems, shows that each attempt to generalize ASG creation resulted in the specification of a language capable of describing these games. Some languages like the one created for the LUDAE project were designed at the top-level and are highly abstract, giving little room in terms of concrete rule definition.

In contrast, the Zillions' language is too specific. It is after all a programming language that one needs to fully understand in order to successfully implement a game. Its capabilities are unquestionably remarkable, but the programming skills required for developing games in the ZRF format turned it to a tool that only experts and adepts use.

One of the goals of this project is to facilitate ASG creation without knowledge of any programming language, hiding the low-level operations and algorithms necessary for implementing a game's logic.

Therefore having an additional step between the generic interface and code generation may bring significant value to the project. Creating a bridge between the two fundamental pillars of the project, the interface and the games themselves, may even provide the means for the project's future expansion, effectively dislodging it from a particular General Game Playing language and allowing for the eventual generation of games in other formats other than the Zillions' language.

Another use for this intermediate language is related to usability issues. Game creation can be complex, and most of the times the game created will not satisfy the user's initial intent. When the interface generates the code representing a game in the Zillions language this generation is not bi-directional. Loading a game already written in the ZRF format and parsing all the rules and information contained in the file is not a feature required for the generic interface, the point is to create new games in a generic fashion, not parsing a specific language for abstract games. It would be an interesting feature for future iterations of the project, but the focus should be given to game generation first. In terms of usability the use of this intermediate language means that it is possible to save the user's progress in the interface, or having the user create a game and saving it in both the Zillions format and the intermediate language, and then being able to access the full game specification using the file generated in the intermediate language.

### **3.4 Improving the AI**

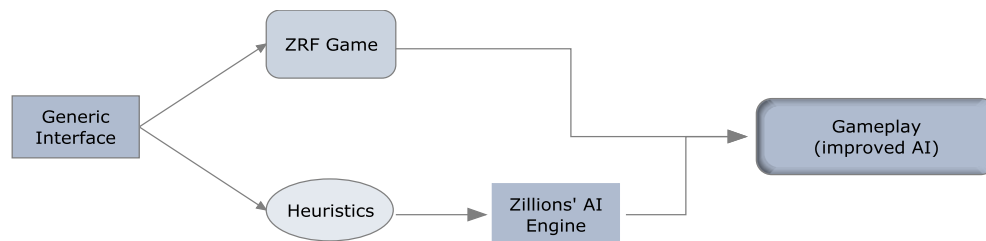
Despite having a lot of success in solving puzzles, the Zillions of Games application has been criticized for not having game specific strategies, which would greatly improve the AI engine's capabilities. The developers knew of this and made it possible to improve the AI engine by the allowing users to develop additional functional libraries and deploy them as *.dll* files over the engine, thus enabling the more experienced users to create their own AIs for their games. However, it is not possible to imbue a ZRF file with heuristics or any other form of Artificial Intelligence algorithms, it is necessary to program in a different language and then reference the new Artificial Intelligence module that will either overwrite the way Zillions thinks or improve it.

### 3.4.1 Heuristics Layer

Adding strategies and simple tactics by means of evaluation functions, with several heuristics pertaining different positional evaluations, is the most obvious step to take in order to improve Zillions's AI.

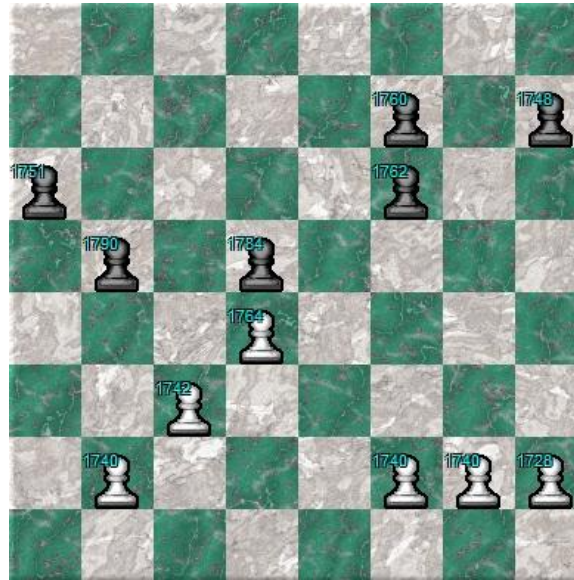
Currently the evaluation function of the game state ZoG uses for each game is dependent solely on rule analysis, which has proved to be a viable methodology for generic game play but for more complex games this analysis falls short when compared to programs specifically designed for each game.

The solution to this may be the creation of a heuristics interface layer that would function as a bridge between a game and Zillions' AI engine, providing an external game evaluation function that the AI would use to improve its own evaluation method (as seen in the schematic diagram in Figure 3.12). By implementing a heuristics layer it would be possible to define strategies for specific ASG, forcing the AI engine to consider a movement or a piece to be more valuable than what the original evaluation function attributed to it.



**Figure 3.12: Heuristics applied to Zillions' AI Engine**

For example, Zillions has no knowledge of the influence of pawn structures in *Chess*, but a heuristic that gave priority on a defensive pawn structure over others would effectively alter the way Zillions plays *Chess*, and with enough well thought alterations the engine could become a very effective player.



**Figure 3.13: Pawn Structure favouring the White player**

Figure 3.13 is an example of Zillions' lack of pawn structure analysis in *Chess*. In this case the White's pawns clearly provide a more solid defensive structure than Black's [Soltis, 1995]. Zillions' evaluation, however, looks at the positions of the pawns in the board, disregarding their strategic value. The evaluation seems to favour pawns closer to the promoting line and pawns with more overall mobility as opposed to evaluating the defensive lines they form that have more strategic value in *Chess*.

### 3.4.2 Learning and Evolutionary Algorithms

As described in the LUDÆ project, a learning approach to abstract games might be the most interesting solution, as having the AI engine learn just by playing the game might prove to be the most rewarding way of increasing its effectiveness. There are several ways of doing this, analysing how human players play the games, having agents battle it out in an ever evolving evolutionary agent gene pool or statistically analysing large quantities of game data and trying to figure out patterns in the most successful victories.

The concept best suited for the project is having an arena where AI agents could play games created through the Generic Interface, evaluating their performance and using this evaluation to select those that are more apt for playing several distinct games. Over time the generations of agents would eventually improve their effectiveness.

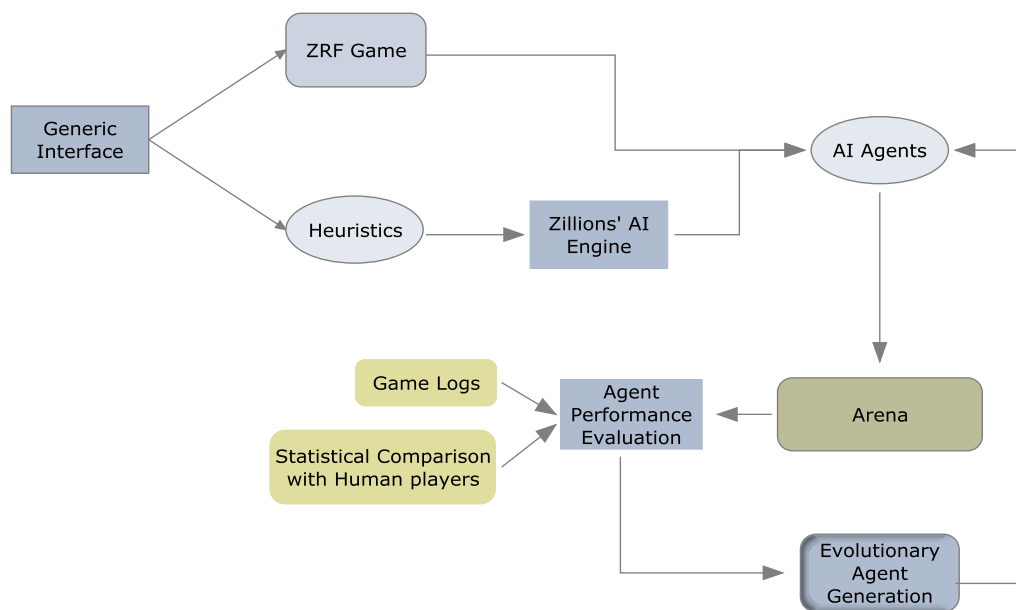


Evaluating AI agents by how well they play a game needs an approach that can be either by comparing result with real live human players or by statistically analysing success rates (winning streaks) by using agent gameplay logs.

In order to fully implement this arena the Zillions platform would need additional alterations so that the agents could interact with the engine, taking advantage of its features, and creating game logs that could then be analysed to evaluate the agents' performance.

By further developing ZoG AI capabilities it could be possible to generate the first batch of AI agents based on the AI module already present in the platform, possibly enhancing it with heuristics as described in the previous section.

Figure 3.14 shows the schematics of the complete AI improvement methods that can be applied to the Zillions engine, and their connections to one another.



**Figure 3.14: Zillions' AI improvement plan diagram**

### 3.5 Conclusions

This chapter featured several different aspects of the project that, when combined, help define the project as a whole as well as providing the guidelines for the development processes of the next stage of the project.

A thorough definition of abstract games is detailed, containing all the possible aspects of these games organized in a way that provides this project with a specific view on ASG creation. An ASG is defined by having a board, movements, pieces, goals and rules. Each of these crucial components is further decomposed in attributes, and an abstract game must be defined by all of these components put together.

An analysis of the Zillions language is also detailed in this chapter, focusing on goal definitions. The way Zillions allows users to define the game goals is somewhat ambiguous, and in order to relate all the possible goal combinations a thorough analysis of Zillions' goals and the goals defined for this project was necessary. This analysis combines the ZRF code notation for each goal, its end game effect and the goal types that it relates to. A further simplification organizes the results in terms of goal types, to simplify the development stage of the project.

The need to define a Generic Language apart from the one ZoG uses is also given special focus in this chapter, justifying in detail the reasons for creating such a language and providing some insight as to how to define it.

The final section of this chapter provides an overview of AI improvement ideas for the Zillions platform, in order to enhance the capabilities of Zillions AI engine. Heuristics play a special role in ASG, as they are clear ways of defining tactical and strategical game evaluation processes that an AI engine can use to improve its gameplay capabilities. The next step proposed for AI improvement in ASG is a combination of learning methodologies, evolutionary algorithms and heuristics. The idea is to create an engine that analyses game rules with heuristics and then produces a batch of AI agents capable of playing any game. Then the agents compete in an arena where the most effective players are evaluated and used to produce the next batch of AI agents.

## Chapter 4

# Generic Interface Development

This chapter is divided in two sections, one regarding the specification of the Generic Interface as a whole and the second one describing the Prototype that was developed in order to try and prove the concept of generalizing abstract strategy game creation.

The project's specification contains the list of features and requirements, as well as their respective priorities. The section also details the Use Cases of the Generic Interface and the overall architecture of the application.

### 4.1 Specification

The system to be implemented requires a core module that will function as both the interface and as a code generation utility. The interface should enable users to choose from a variety of options that together define an ASG, allowing them to generate a game file that the Zillions of Games application can read and play. It is also required that the Generic Interface allows users to save their progress for future use, as well as to access the games saved this way.

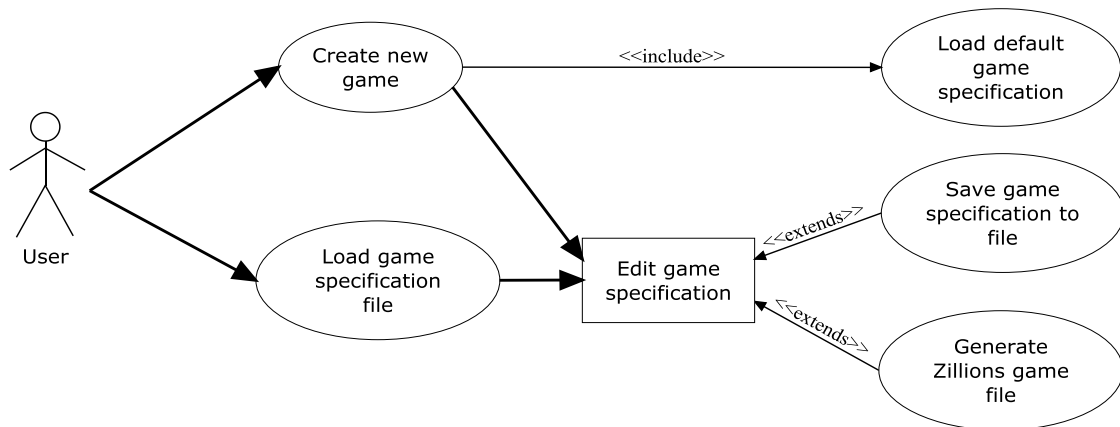
The options the user should have at his disposal regarding the creation and definition of rules for ASG must be in conformance to what is describe in section 3.1 regarding abstract games definitions. Some game options specific to the Zillions language should also be included in the Generic Interface.

This section therefore features the Use Cases for the system, the complete list of features required, the non-functional requirements of the system and its overall architecture design.

#### 4.1.1 Use Cases

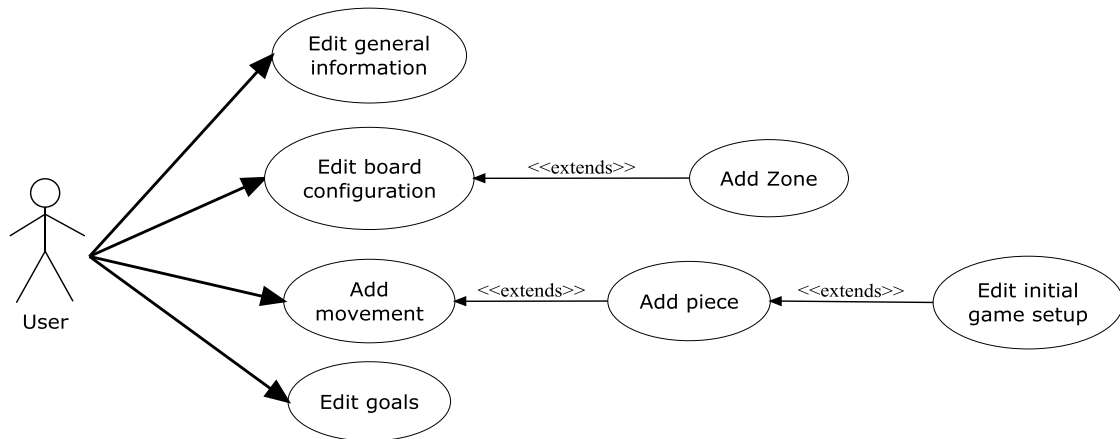
The basic interaction with the system should allow the user to create a new game based in a default configuration file, load an existing game from a previously created file and edit game

options and rules. The system should always enable the user to generate a game in the ZRF format, as well as saving his progress in game creation to a file for later use. The general system use case diagram is showed in Figure 4.1.



**Figure 4.1: General System Usage Use Case Scenario**

The specification of a game's options demands further detail, as it encompasses all the possible use case scenarios that the ASG definition in section 3.1 describes. However, due to the extent of the options available only a general overview of these options is shown in Figure 4.2.



**Figure 4.2: Game Options Editing Use Case Scenario**

#### 4.1.2 Features

This section contains the complete features required for the Generic Interface. These features were based on the analysis of the definitions of ASG creation and on the capabilities of the Zillions language. To each feature is assigned a unique Id and a qualitative priority value, ranging from **Low**, **Medium** and **High**. The colour code used varies from yellow (low), orange (me-

dium) and red (high) to symbolize the importance of the different degrees of priorities. The features are further divided in categories, grouping them in terms of functionality.

#### 4.1.2.1 File Management

**Table 4.1: File Management Feature List**

<b>Id</b>	<b>Feature</b>	<b>Priority</b>
F1.1	Save game creation process	High
F1.2	Choose file directory paths	Medium
F1.3	Load an existing game specification file to the interface	High
F1.4	Generate ZRF game code to specified folder	High
F1.5	Create board and pieces images on game folder	High

Table 4.1 details the features necessary for File Management. F1.1 refers to the capability of the system to create a file in a generic language that contains all the data inserted/modified by the user. F1.3 acts as the opposite, loading game rules from a file and updating all the necessary fields in the interface. The main goal of the application is to be able to generate code in the ZRF format, which is implied in F1.4. F1.2 and F1.5 are necessary for the good functioning of the application (Zillions cannot run a game whose images are undefined).

#### 4.1.2.2 Interface

**Table 4.2: Interface Feature List**

<b>Id</b>	<b>Feature</b>	<b>Priority</b>
F2.1	Verify validity of inserted data	Medium
F2.2	Help tooltips / messages	Medium
F2.3	Check validity of game specification on Load	Low
F2.4	Avoid errors by restricting user inputs	Medium
F2.5	Allow for manual ZRF code alterations	Low

The feature list of Table 4.2 is focused on the interface's usability, trying to detect and correct user mistakes and restricting the user's options depending on the case (F2.1 and F2.4). Help messages and tooltips are useful for improving user-interface interaction (F2.2). If a file containing a game specification in the generic language has been modified the system should be capable of detecting the errors and notifying the user for missing/wrong data (F2.3). The possibility of altering the generated code allows users that know the ZRF syntax to manually alter the code (F2.5).

#### 4.1.2.3 Game Creation General Parameters

Table 4.3: Game General Parameters Feature List

<b>Id</b>	<b>Feature</b>	<b>Priority</b>
F3.1	Change game name	High
F3.2	Change player names	Low
F3.3	Change game optional fields (description, strategy and history)	Low

The Game Creation General Parameters feature list is presented in Table 4.3. F3.1 is a high priority feature, as it serves to give a new name to a game. The other features in this list are purely for enhancing the game playing experience, by adding different optional information on a given game and modifying the names of players.

#### 4.1.2.4 Board Configuration

Table 4.4: Board Configuration Feature List

<b>Id</b>	<b>Feature</b>	<b>Priority</b>
F4.1	Spatial dimensions	Low
F4.2	Change dimension values (length, width, etc.)	High
F4.3	Symmetry	Medium
F4.4	Cell polygon definition by edge number value	Medium
F4.5	Automatic board drawing	High
F4.6	Manual board drawing (irregular boards)	Low
F4.7	Automatic directions definition	High
F4.8	Link/Unlink positions	Medium
F4.9	Kill positions	Low
F4.10	Choose board image	Low
F4.11	Create board bitmap image	High
F4.12	Zone creation (name, player affected and positions list)	High

Features related to Board Configuration are presented in Table 4.4. It was decided that two-dimensional boards have higher priority, so defining more dimensions became a low priority feature (F4.1). Depending on the dimension number chosen, the values that can be altered vary (F4.2), which is crucial for having different sized boards. F4.3 represents board symmetry, that is the possible symmetry lines in which to divide a board so that for one player's point of view a direction is opposite for his opponent's point of view. Most ASG have square cells or hexagon cells that make up the game board, but a board cell can be of any regular shape formed by its edges (F4.4). For irregular boards the system should enable users to manually add positions (F4.6) but the option to draw a regular board automatically should have a higher priority (F4.5). Movements need to correlate to directions that link the positions in the board and this definition can be made automatically with F4.7. The user should also be able to link or unlink any position to another position to create all kinds of connections in a board or even setting kill positions, or traps (F4.8 and F4.9). Choosing and creating a board image to the specified game folder are also

requisites of the system, but the image creation takes precedence because of the automatic board drawing feature (F4.10 and F4.11). Finally, adding zones to the board by selecting a set of positions, a name and what players are affected is also a high priority feature for most games to function correctly (F4.12).

#### 4.1.2.5 Movement Configuration

Table 4.5: Movement Configuration Feature List

Id	Feature	Priority
F5.1	Choose move types	High
F5.2	Capturing	High
F5.3	Flipping	Medium
F5.4	Priorities	Medium
F5.5	Drop options	High
F5.6	Directions	High
F5.7	Distances (up to, exactly, any, furthest)	High
F5.8	Jump radius	High
F5.9	Jump over	High
F5.10	Jump capturing	High
F5.11	Swap options	Medium
F5.12	Push/Pull options	Medium
F5.13	Advanced capturing (positional, firing, pushing over the edge the board, etc)	Low
F5.14	Zone restriction (outside/inside/indifferent)	Medium
F5.15	Zone restriction (start, during and end of movement)	Low
F5.16	Create multiple move as a sequence of movements	Medium
F5.17	Create multiple move as repeated move while a condition is verified	Medium

Adding a new movement to a game can involve several different features and steps (Table 4.5). The higher priority features are related to the drop, slide and jump type movements that most ASG have (F5.1, F5.2 and F5.5 through F5.10). Flipping, prioritizing movements (what moves take precedence over others), swap and pull/push move type options and restricting a move with zones all fall in the medium priority category (F5.3, F5.4, F5.11, F5.12, F5.14). Advanced Capturing options and further zone restrictions depending on the piece movement origin and target position are Low priority features, as they represent specialized game options for concrete ASG. Multiple movements have Medium priority, as they are necessary for some games but not as essential as the actual basic moves themselves (F5.16 and F5.17)

#### 4.1.2.6 Piece Configuration

Table 4.6: Piece Configuration Feature List

Id	Feature	Priority
F6.1	Name and optional information (help and description)	High
F6.2	List available moves for pieces	High
F6.3	Check for incompatibilities between moves	Low
F6.4	Choose images	High
F6.5	Define promotion zones	Medium
F6.6	Define promoting movements	Medium
F6.6	Promoting capabilities: advanced options	Low
F6.8	Capture Hierarchy	Medium

The features regarding Piece Configuration are shown in Table 4.6. A piece must have a name and some optional information regarding it (F6.1). The moves list created in the previous section should be made available for linking movements to pieces (F6.2). Images are also important to define otherwise Zillions won't be able to play a game (F6.4). Promotion capabilities and organizing a capture hierarchy (similar to a food chain between the pieces of a game) are all Medium priority features (F6.5, F6.6 and F6.8). F6.3 is marked with low priority for it is another error preventing tactic that can prevent games to be created with mistakes in their logic. The feature pertaining advanced promoting capabilities like for instance adding pieces to the board is also set as a low priority feature (F6.6).

#### 4.1.2.7 Board Initial Setup

Table 4.7: Board Initial Setup Feature List

Id	Feature	Priority
F7.1	Pieces' initial positions	High
F7.2	Pieces' off board count	High
F7.3	Symmetry in piece positioning	Low

The three features of the Board Initial Setup are presented in Table 4.7. Some games must start with their pieces in specific positions, while others require pieces to be dropped onto the board; F7.1 and F7.2 represent these crucial features. F7.3 serves to simplify this initial setup it is possible to use the board symmetry for piece positioning (saving the user's time and effort).



#### 4.1.2.8 Goal Configuration

Table 4.8: Goal Configuration Feature List

Id	Feature	Priority
F8.1	Choose goal type (enable specific goal options)	High
F8.2	End game result	High
F8.3	Relative directions	Medium
F8.4	Remaining number of pieces / number of pieces to capture	High
F8.5	Choose piece types	High
F8.6	Choose zones	High
F8.7	Prioritize goals	Medium
F8.8	Advanced specific goal options	Medium
F8.9	Advanced end game options (turn counter, movement repetition, etc)	Low

The Goal Configuration feature list is shown in Table 4.8. Basic goal options like type, pieces remaining, zones, end game result (who wins and who loses) and piece types all have maximum priority (F8.1, F8.2 and from F8.4 to F8.6). More advanced options like ordering goals depending on priority, creating relative piece configurations by choosing a set of directions and even more specialized goal options for goal types like Claim fall in the medium priority category (F8.3, F8.7 and F8.8). The only low priority feature are the advanced end game options, as they fall out of the scope of the typical ASG goal types but are nevertheless interesting end game options (F8.9).

#### 4.1.2.9 Additional Rules Configuration

Table 4.9: Additional Rules Configuration Feature List

Id	Feature	Priority
F9.1	Define turn order	Low
F9.2	Turn options (specific mover per turn, multiple turns)	Low
F9.3	Turn passing	Medium
F9.4	Game specific rules (castling, en passant, firing arrows, special captures, etc)	Medium

Table 4.9 shows the Additional Rules Configuration features. Defining the turn order can be important to some games that allow for varying turns for each player, as well as having a player move only specific pieces or perform specific movements in his turn, but these are advanced rules and as such are marked as low priority features (F9.1, F9.2). Turn passing and having specific rules for well known games however are by far more interesting features to add to the system, and as such are given a medium priority (F9.3, F9.4).

#### 4.1.2.10 Zillions Additional Features

Table 4.10: Zillions Additional Features List

Id	Feature	Priority
F10.1	Game sounds	Low
F10.2	Multiple players	Low
F10.3	Dummy pieces/positions	Medium
F10.4	Boolean Logic applied to rules	Medium

The Zillions platform allows for many different options, but the only the most relevant should be considered for this project (Table 4.10). Adding sounds to the game or having more than two players are complementary features, so they are marked as low priority (F10.1, F10.2). Dummy positions and piece can be used as markers or place holders, and are an interesting feature to consider (F10.3). F10.4 makes use of Zillions Boolean Logic and applies it to the rules, which can largely increase the combinatorial power of previously implemented features by enabling for instance their negation (doing the opposite instead).

#### 4.1.3 Non-Functional Requirements

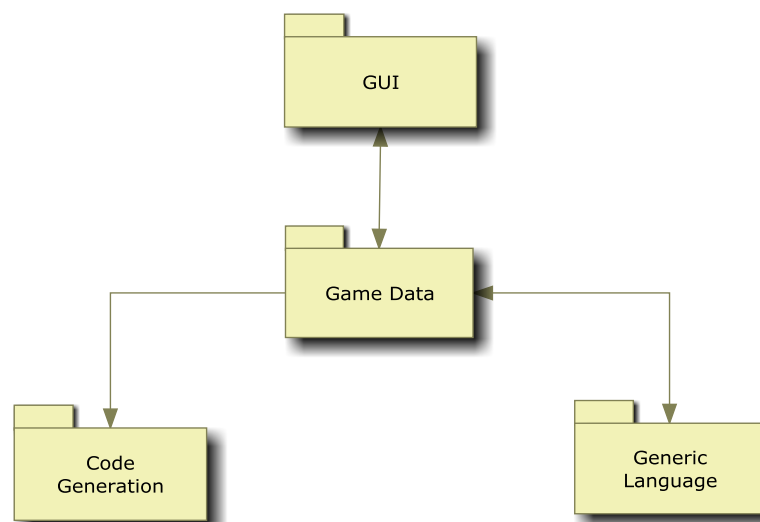
Non-functional requirements are requirements that specify criteria that can be used to judge the operation of a system, rather than specific behaviours. The amount of non-functional requirements that can be associated with projects of this kind are numerous, so focus shall be given only to the most relevant.

- Emotional factors: the application should absorb the user and its use should be considered an entertaining activity
- Extensibility: the system should be built considering further expansion in the future, by the development of new features and modules
- Fault-tolerance: the system should be able to continue performing in the case of a component failure or individual user problem
- Legal Issues: despite the fact that the application is designed to generate files to be used in another system, such feature must avoid patent-infringement
- Platform: the system must run in the Operating System Microsoft Windows (XP/Vista/7)
- Usability: this functional requirement is a qualitative attribute that assesses how easy user interfaces are to use, and can be further divided in the following aspects:
- Learnability: the system should avoid having a steep learning curve, it should be relatively easy for users to perform basic tasks the first time they interact with the system
- Efficiency: users that have already learned the basic structure of the system's design should be able to perform tasks more swiftly

- Memorability: users that interact with the system after a period of time after their last interaction should be able to easily re-establish proficiency
- Errors: the systems should be built in order to minimize the amount of user error and their severity, as well as allow users to recover from such situations
- Satisfaction: the user should feel that using the system's capabilities was a pleasant experience

#### 4.1.4 System Architecture

The system architecture contains four main modules: the user interface (GUI), Game Data, Generic Language and Code Generation (Figure 4.3).



**Figure 4.3: Overall System Architecture**

The GUI is responsible for all human-computer interaction; all the choices users can make must be represented in the interface. This module communicates with Game Data to translate the user choices into game specifics, or load game specifics to the interface (modifying the options in the interface to fit the game specification).

Game Data represents the core of the application. It embodies everything that is possible to define and attribute to an ASG through the user interface. It is responsible for the storage in runtime of all the game information data, as well as functioning as a bridge between the generic language files (for saving and loading games). Game Data is also responsible for providing the necessary game information to the Code Generation module.

Code Generation is the module that is responsible for generating the game code in a format readable and playable by a General Game Playing System (the Zillions of Games platform). The game data is obtained through the Game Data module.

The Generic Language module creates and reads from files written in the generic language created for this system, allowing users the option to save and load their game creation process.

## **4.2 Conclusions**

This chapter focused on the Generic Interface system as a whole, defining what is necessary to include in the system, delineating the boundaries of the projects usefulness. The top-level requirements for the project are also defined recurring to use cases, a feature list, non-functional requirements for the system and overall system architecture.

The system's specification is the second part of the work predicted for the project, after research and before actual development of a working prototype one is required to define and design with some degree of detail what is required of the system. A deeper definition of what is required of the system is not always a good idea, especially in software development projects such as this one where the requirements and the development process tend to be hard to separate. Therefore, this chapter focuses on an over the top view of the project, drawing the lines that define the shape of the project but not going too deep as to force the development project in a particular way rather than another. It is a set of guidelines to be followed, but they are not necessarily strict nor do they represent the ultimate view over what can be done in the project.

## Chapter 5

# System Implementation

To assert the validity of the claim proposed by this project – that creating ASG through a generic interface by choosing a set of predefined options is possible, and the more option one has at his disposal the more diverse are the games one is able to create – a prototype of had to be implemented.

This chapter goes into further detail over what was actually developed, explaining what was implemented and how, giving insight to some of the decisions taken during the development process as well as describing what technologies were used.

The prototype, named ‘*Abstract Games Creation Kit*’, was developed according to the features listed in section 4.1.2, trying to implement as many of the high priority features as possible. It was also designed to be expandable and according to the more general system characteristics described in chapters 3 and 4.

The prototype was developed in the *C#* programming language and the interface uses WPF (Windows Presentation Foundation).

This chapter describes the development process, featuring the most relevant elements of the prototype. Some algorithms used are explained, as well as some explanations on critical decisions taken.

### 5.1 Prototype Development

The development was done in stages, first by designing the interface and studying what options should be available to users and more importantly how to display this information without having too many interface components clogging the main window. Actual development was initiated only after a few initial mock-ups were designed and an overall activity schematic was created. Appendix A shows initial mock-ups of the interface, and Appendix B shows the prototype’s current appearance. It is interesting to note that despite the fact that most of the interface

was programmed according to the initial design, some features either evolved to others or even in some cases removed.

The modules (or classes) that are part of the system follow the architecture scheme present in section 4.1.3. The main interface class is a WPF form, containing all the interface options and the methods necessary to link the other classes. Game Data is a class that defines what an ASG object is, with several inner classes specifying different game characteristics (moves, pieces, zones, etc). There is also the XmlManager class and the ZRFCreator class, each responsible for creating files containing a game in their specific formats. The XmlManager class is also able to read from an XML game document and passing a game object to the main form interface.

Figure 5.1 shows a summarized class diagram of the prototype.

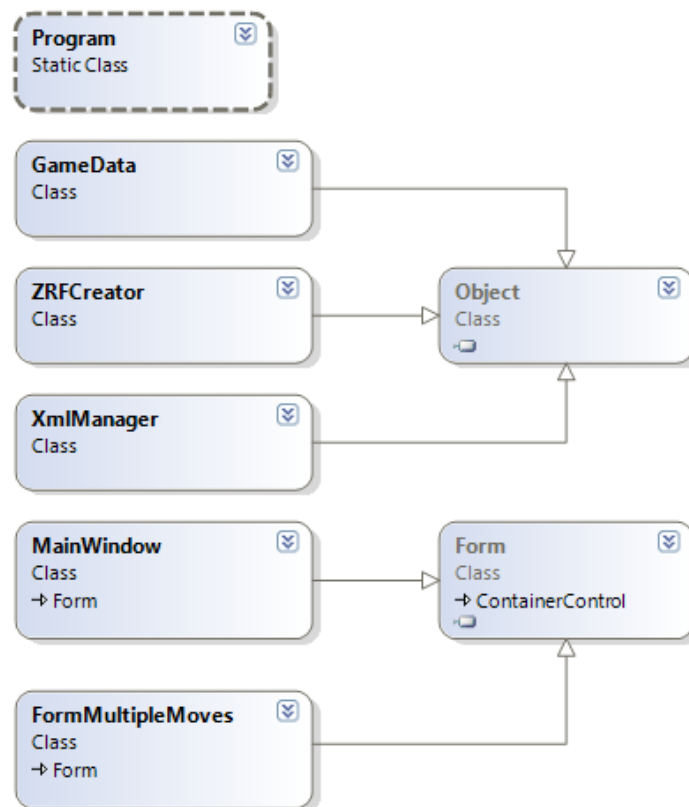
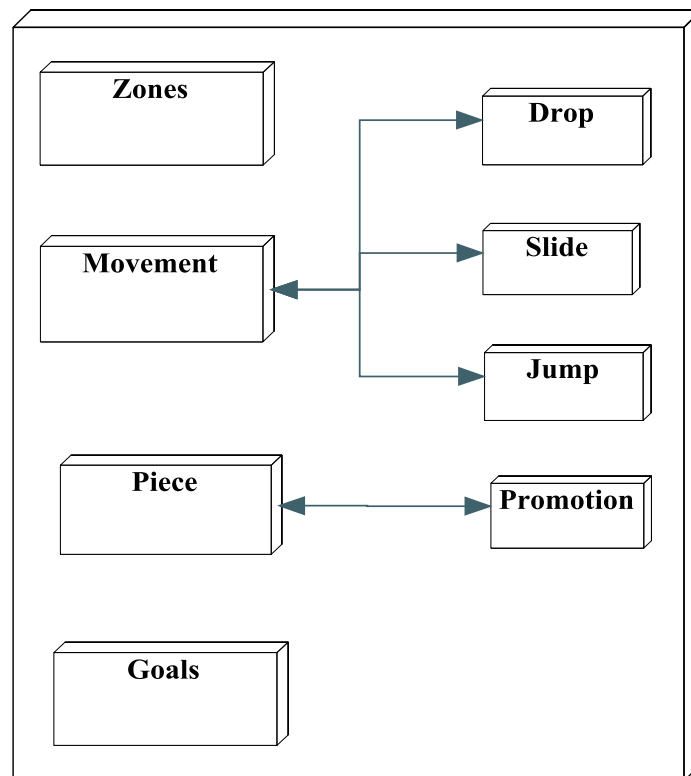


Figure 5.1: Prototype's Class Diagram (generated via Microsoft Visual Studio 2010)

The program starts by launching the interface (Main Window) and then the user's choices are reflected in what data will be saved in a game object, defined by the Game Data class. This game object is accessed and used by the file management classes, XmlManager and ZRFCreator to read and write XML documents and generate code, respectively.

### 5.1.1 Game Data Classes and Objects

As mentioned above, the Game Data class is the core of the Abstract Game Creation Kit. It allows the program to store game data in an organized fashion, by a series of interlinked objects that represent zones, pieces, the board, movements, goals and the game itself. Figure 5.2 presents a diagram that further illustrates what sub-classes are part of Game Data.



**Figure 5.2: Game Data Sub-Class Relationships Conceptual Diagram**

Besides these classes Game Data also stores information such as board definitions (size, image path, etc), the game and players' name, as well as conditions related to the board image definition and a few flags regarding automatic board image generation and how to describe turn passing.

The Zones class can create zone object, which have a name, a players (or players) affected attribute and a set of board positions. Zones are used later on in Movements and to define some of the goals.

Movement creates move objects, containing the name of the movement, its type, a list of zones that may affect it and priority and capture privileges. A move doesn't contain all the in-

formation, each move object is further split into an object of its type that are defined by subclasses of move and contain type specific info (like radius for jump).

A piece object must contain a list of movements, a name and images. It can also store information regarding the initial board positions, how many pieces are on board and can be even further expanded by adding a promotion object. A promotion is comprised of a set of pieces to which the original piece can promote and a list of zones in which that can happen.

Goals have list of zones, directions and pieces, as well as attribute that define the effects that fulfilling a goal has on the game.

Data structures in this case are fairly simple; each object has sets of names pointing to other objects, the interface and the other classes access the data by a number of *Find* methods that search the object lists of a game and return the target object given an index or a string to search. Other noteworthy features are related to particularities in naming: goals are named automatically according to the number of goals existing of the same type; zones can have the same name if and only if the rest of their attributes are distinct, which means that it is possible to have two promotion zones that affect different players and have distinct promoting positions but have the same name, the game's logic implies that such zones are indeed but one zone affecting each player differently.

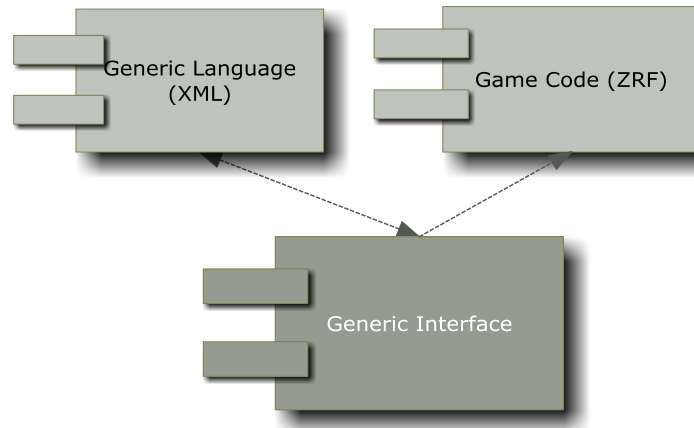
This subdivision of a game object is closely related, although simplified, to the description present in 3.1.

### **5.1.2 XML Language**

Design and specification of a generic language for ASG became a requirement for the project, and the Extensible Markup Language (XML) was chosen for its extensibility features and well-known use in similar situations. XML documents are text files written according to a grammar that can be defined any way necessary, representing a game in the form of a tree of tags and attributes.

The extensibility of XML also allows for more rules and features to be added to the language without the need to re-structure the whole language, with care as not to alter the basic format that defined the grammar in the first place.





**Figure 5.3: Generic Interface, XML and ZRF relationship**

Figure 5.3 shows the relationships between the Generic Interface and the two file types it is able to handle: XML (for the generic language) and ZRF (for game code). The double pointed arrow in the relation between the interface and XML represent the ability of data being transferred both ways, towards the interface to load a game specification and towards the XML to symbolize saving a game specification to an XML document.

The XML generic language grammar is strongly influenced by the way Zillions of Games defines ASG, as well as by the definitions on how to create an ASG in detailed in section 3.1. A game is therefore structured as a tree with nodes and leaves, each node representing a category or subcategory of a game rule and each leaf being an indivisible attribute. For example, a node can symbolize a piece, which in turn has nodes that represent its movements and promotion, and attributes that contain atomic data like the name of the piece and the number of off board pieces the game initially has.

The generic interface needed a bridge between the interface and the game code generated to store data for later use, as loading (parsing) game code is complex task with little return value compared to other features of the project. The class XmlManager basically has two main methods, one to write an XML document with a game's specification and another to read from such file and translate it to a game object.

For each attribute, rule, list, and object contained in a game object created through the interface the writer creates a file with tags containing such information, and each tag ends with an atomic node with the specific value of a game attribute (which can range from strings to integer numbers). The reader works an exact opposite procedure, by creating a game object, reading a file and incrementally adding structured data to the said object, which is in turn passed on to the interface.

The versatility of an XML document only allows for some data storing, but for saving colour information for instance instead of writing a colour's name it was decided to store its RGBA values instead, mainly because most colours in the RGB format do not have a predefined system name.

Examples of a piece and a goal written in the generic language created for these XML documents are shown in the figure 5.4.

```
<piece>
  <pieceName>checkerblack</pieceName>
  <ImageP1Path>c:\zillions\Connect4d\checkerblack_black.bmp</ImageP1Path>
  <ImageP2Path>c:\zillions\Connect4d\checkerblack_red.bmp</ImageP2Path>
  <OffBoardP1>30</OffBoardP1>
  <OffBoardP2>0</OffBoardP2>
  <pieceMoves>
    <pieceMove>blackslide</pieceMove>
    <pieceMove>blackdrop</pieceMove>
  </pieceMoves>
</piece>
<goal>
  <goalName>Pattern1</goalName>
  <goalType>Pattern</goalType>
  <Value>0</Value>
  <WinDrawLoss>0</WinDrawLoss>
  <goalPlayersAffected>0</goalPlayersAffected>
  <goalPieceType>checkerred</goalPieceType>
  <goalPieceType>checkerblack</goalPieceType>
  <goalDirection>N</goalDirection>
  <goalDirection>N</goalDirection>
  <goalDirection>N</goalDirection>
```

Figure 5.4: Excerpt of the generated XML Document

This example features a piece named “*checkerblack*”, two image paths for its visualization, 30 of board tokens of the piece that can be dropped in the game board during the game and two moves (that are previously detailed on the same file in their respective sections). The goal presented here is of the pattern type, the value 0 in the *WinDrawLoss* tag represents a victory if the goal is achieved, affects both the black and the red checker and the directions define that whomever manages to arrange 4 of his pieces in a vertical line (each one directly north of the other). This code is a sample taken from the game *Connect 4*, and the goal presented is one of the four necessary to define the winning conditions of the game (the other three being east, northwest and northeast).

### 5.1.3 ZRF Code Generation


In order to translate the user's options in the generic interface to game code the class ZRFCreator is imbued with a number of methods that take the game object and progressively generate the ZRF code.

Zillions' language grammar itself is built based on the lisp programming language, so different tags are needed in order to specify what part of a game one is trying to program. Most of a game's code is written inside the game tab, but useful macros can be defined out of the game's scope, and can later be accessed from inside the game to simplify movements or other verifications. ZRF language has several key words that allow for the definition of almost anything on an ASG, but the 'add' keyword plays a special role: it defines when a move is generated for calculating a piece's available moves in the game tree.

Macros are useful because they allow for a general definition of a rule, without the need to specify what piece or goal it affects. In the prototype macros are used for two main game specifications: promotions and moves.

Code generation is done in steps, further detailed in table 5.1.

**Table 5.1: Code Generation Construct Sequence**

<b>Macros</b>	Promotions	
	Moves	
<b>Game Construct</b>	Rules	
	Board	
	Pieces	
	Board Setup	
	Goals	

The code is written according to constructs of game rules, by the order defined in the above. The following sections will detail each of these constructs to explain how they function.

#### 5.1.3.1 Macro Code Generation

A promotion implies that a piece, when moving, can reach a position defined by a zone where it can promote to another type of piece. The method responsible for these macros takes the list of pieces of a game and for the ones that have promoting capabilities simply creates a macro that checks whether a movement has ended in the promoting zone or not. If it hasn't, the move is performed as usual, if it has then the method checks to what types of pieces the one in question can promote to and the move ends by prompting the player to choose what type of new piece he desires. The resulting macro is written following the format depicted in Figure 5.5.

```

define [promotingPiece.Name-add]
  if (not-in-zone? zone)
    add
  else
    add pieceTypes

```

**Figure 5.5: Promotion Macro Example (simplified)**

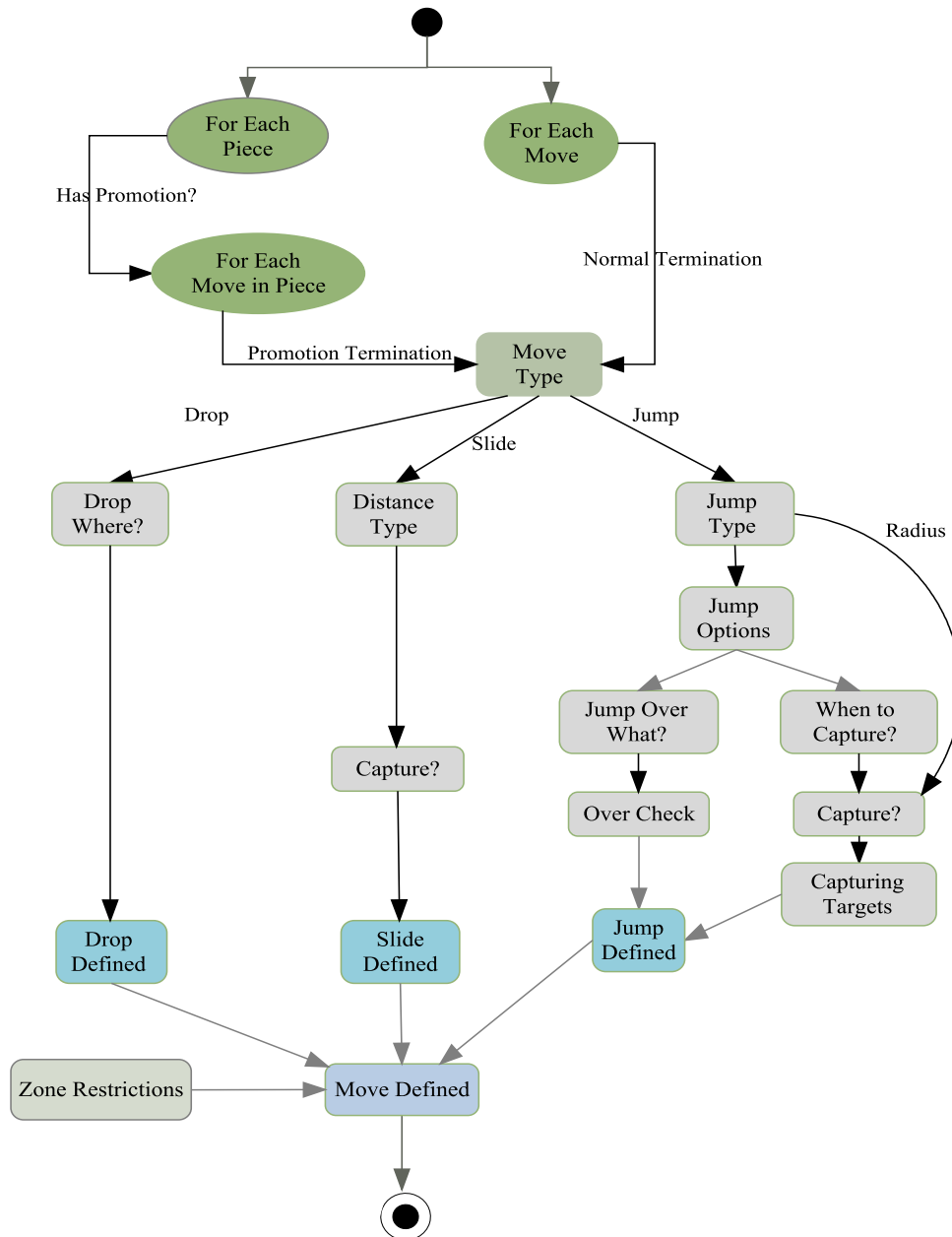
A movement can be of many different types and have multiple options regarding how it will be performed. The following diagram (Figure 5.6 below) shows a compact version of what constitutes a move definition in the code generating module.

The diagram shows how the code generation is done, depending on the move type and its attributes. The first part of the diagram represents the need to verify if it is a normal move or a move specific to a piece that can promote, because the termination of the move will be different, as it may be necessary to call the macro defined above in section 5.1.3.1: Promotion Code Generation. Skipping ahead of the move types section of the diagram until a move is defined it is necessary to verify the zones affecting the move and only after that is the move truly defined. This zone verification only checks if the move is starting from inside or from outside each zone affecting the move.

If a move is of the type drop, which means that a piece will be dropped from out of the board into it, it is only necessary to verify where it can be dropped. The options are self-explanatory and they are ‘anywhere’, ‘empty spaces’ and ‘on enemies’.

For moves of the slide type the case is different, as it is necessary to check the distance type selected, which can be ‘any’, ‘exactly’, ‘furthest’ and ‘up to’. Each type represents a completely different way of how the move is generated and also how the code will turn out to be. The only similarity is the ending of a movement, which needs to check the capturing capabilities of a move, ranging from ‘allowed’, ‘not allowed’ and ‘mandatory’. Depending on the capturing method the target destination will be ‘not-empty’, ‘empty’ or ‘enemy occupied’ respectively.

The most complex move types are the jumps. A jump can be distinguished into the following categories: ‘any’, ‘exactly’ and ‘up to’ that function in much the same way as in slide type moves, and ‘radius’ which is a specialized jump type (explained in detail in section 3.1.3.1). A jump implies that the piece will “fly” over other pieces, and this affects two aspects of the jump: when to capture its targets and over what spaces in the board it can jump. Capturing in a jump can be either when the piece lands, during its flight or both methods, defining the capturing targets of a jump. Jumping Over options are detailed in Table 3.1 of section 3.1.3.1 and define the Over Check. These verifications result in a jump being composed of an ‘over what to jump’ check, following the specific code depending on the jump type which can feature the ‘during flight’ capture target and ending in the landing target capturing check. The only exception is the radius jump which only checks for the landing target and nothing else.



**Figure 5.6: Move Code Generation Diagram**

Directions are conveniently left out of the moves definition, given that they are macros the idea is to call them from within a piece construct and only there the directions are featured, calling a move by its macro name followed by the desired direction. This efficiently generalized move creation, both simplifying the code generated as well as reducing the number of calculations the Zillions AI engine is required to perform.

### 5.1.3.2 Game Construct Code Generation

In reality there is no ‘rules’ construct on the code, but several minor methods that define simple rules and information regarding the game. Things like the game’s name, its players’ names, optional information, turn order and game options (including if passing one’ turn is allowed) and the ordering priority of moves all fall into this category.

A board is defined with its image (that points to the generated image created by the interface), a set of directions that define links between the spaces in the board, another method is responsible for attributing names for each position according to the board’s dimensions (in columns and rows), symmetry between directions is also generated and finally a set of zone construct are written.

The main simplification between board creation with the generic interface and writing the code the traditional way is the automatic definition of pixels in the board image are related to the positions in the board.

Board symmetry varies from ‘none’, ‘vertical’, ‘horizontal’ and ‘all’. This means that for vertical symmetry the north and south directions are symmetric, as well as northwest and southwest, northeast becomes symmetric with southeast. Horizontal applies the same idea but with east-west symmetry and ‘all’ includes all of the former symmetries.

For each piece a piece construct is created, including the basic attributes like the name, images, optional information and the drops and moves construct. Depending on the moves a piece has, these movement construct may be empty. For each move defined this way the method calls the corresponding macro after organizing the moves in terms of priority (each priority level is attributed a specific name) with a direction if one is required.

The board setup construct is used to define what positions are occupied with each piece, and how many pieces of each type start the game outside the board.

Finally, the goals are defined individually, by their order of priority defined in the interface. For each goal its type is verified, and depending on the options specific to each goal, the code is generated. This is done according to what is defined in section 3.2.1: Goal Analysis.

### 5.1.4 Graphical User Interface

The GUI is an obviously a very important component of the prototype. Aside from providing the means of interaction between the user and the program, it also has some built in features to prevent human error and to facilitate ASG creation.

There are two basic types of verification associated to each of the GUI's components: prevention and correction.

Prevention methods disable and enable options according to the user's choices, for instance, if a user chooses a jump type movement and the jump over friendly pieces options the capturing methods of the jump become locked and only 'on landing' is permitted. This makes sense because if a piece can only jump over its owner's pieces it is impossible to capture enemy pieces during the jump. This restricts the user's choices as well as preventing errors from reaching the other modules of the system (they require that all the information in the game object be accurate and unambiguous).

Correction occurs when the user is asked to write a string (naming something or adding optional information) and if an invalid name is inserted an error message is shown describing what the user did wrong and how to correct it. Another message of the sort is shown when the user tries to add something (a move, a piece or a goal) and mandatory fields are still empty. Input string analysis is done according to regular expressions that search for the occurrence of numbers, empty spaces, invalid characters and ZRF specific keywords.

While reading an XML document containing a game specification the interface is also tasked with filling all the GUI's components with the game data contained in the document, which the user can then modify or delete and create another game based on the loaded one.

## 5.2 Implemented Features

The complete feature list of the Generic Interface present in section 4.1.2 represents all of the possible rule permutations and requirements for the project. The prototype's development however, only implemented some of those features. Table 5.2 shows what features were implemented and to what degree. The decisions that lead to some features being partially or not at all implemented are explained next. A completeness attribute was added to each feature, which can vary between **Full** and **Partial**, with a new colour code for these attributes as well: green for full completeness and orange for partial completeness. The unimplemented features absent from the table are explained either due to time constraints for the project or to having been assigned a lower priority.

Table 5.2: Feature Fulfilment

Id	Feature	Priority	Completeness
F1	<b>File Management Features</b>		Full
<b>Interface</b>			
F2.1	Verify validity of inserted data	Medium	Full
F2.2	Help tooltips / messages	Medium	Partial
F2.4	Avoid errors by restricting user inputs	Medium	Full
F3	<b>Game Creation General Parameters</b>		Full
<b>Board</b>			
F4.2	Change dimension values	High	Full
F4.3	Symmetry	Medium	Partial
F4.5	Automatic board drawing	High	Partial
F4.7	Automatic directions definition	High	Partial
F4.10	Choose board image	Low	Full
F4.11	Create board bitmap image	High	Full
F4.12	Zone creation	High	Full
<b>Movements</b>			
F5.1	Choose move types	High	Partial
F5.2	Capturing	High	Full
F5.4	Priorities	Medium	Full
F5.5	Drop options	High	Partial
F5.6	Directions	High	Full
F5.7	Distances	High	Partial
F5.8	Jump radius	High	Full
F5.9	Jump over	High	Full
F5.10	Jump capturing	High	Full
F5.14	Zone restriction (outside/inside/indifferent)	Medium	Full
F5.16	Create multi-move as a sequence of movements	Medium	Partial (GUI)
F5.17	Create multi-move as repeated move	Medium	Partial (GUI)
<b>Pieces</b>			
F6.1	Name and optional information	High	Full
F6.2	List available moves for pieces	High	Full
F6.4	Choose images	High	Full
F6.5	Define promotion zones	Medium	Full
<b>Initial Setup</b>			
F7.1	Pieces' initial positions	High	Full
F7.2	Pieces' off board count	High	Full
<b>Goals</b>			
F8.1	Choose goal type	High	Partial
F8.2	End game result	High	Full
F8.3	Relative directions	Medium	Partial
F8.4	Remaining pieces / Pieces to capture	High	Full
F8.5	Choose piece types	High	Full
F8.6	Choose zones	High	Full
F8.7	Prioritize goals	Medium	Full
<b>Additional Rules</b>			
F9.3	Turn passing	Medium	Full



The decision to not implement more than two spatial dimensions was due to the fact that most ASG are played in 2-D boards and the actual gain in playable games able to create would be minimal (compared to the sum of all other features), this in turn lead to the partial implementation of feature 4.7 (inward and outward directions). Diagonal and specific symmetries were discarded due to their low gameplay value (F4.3).

Not allowing for the creation of different polygons for cells by edge number (F4.4) had major consequences to the overall project, has it was essential for defining hexagon and triangular boards (which are quite common in ASG), as well as restricting feature 4.5 and prohibiting the implementation of feature 4.6 (which also needed a wizard like interface to allow the user to create a board at will).

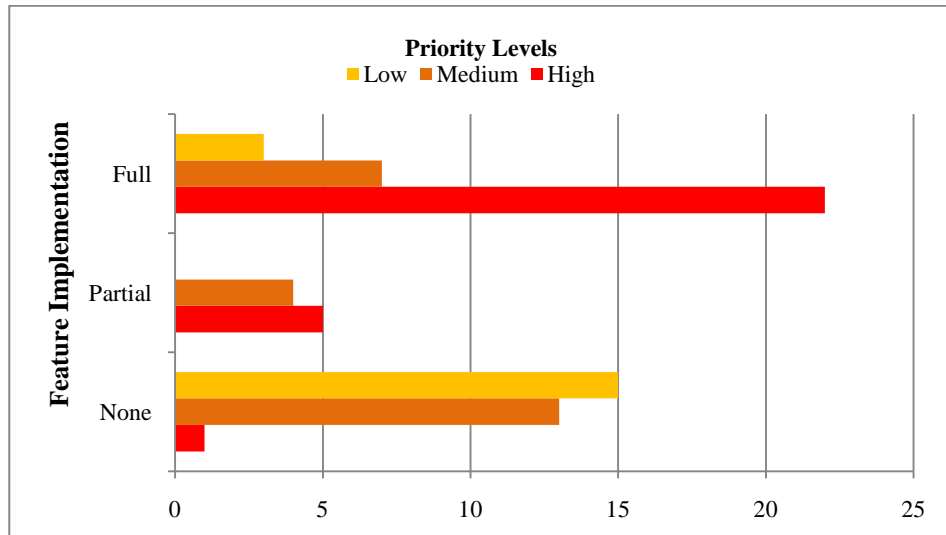
Movement priority was given to Drops, Slides and Jumps, and Swaps, Pulls and Pushes were dismissed because of their lower occurrence in ASG. This decision prevented features F5.11 and F5.12 from being implemented. In a similar fashion, priority was given to some capturing options over flipping, but even some more advanced capturing situations were discarded (like positioning capture). The only relevant decision taken in the Movement category was the decision to dismiss furthest jumps, as they represent a rare situation in ASG and they're implementation was completely different from the other jump types (which could be derived from one another). Multiple moves are present in the interface, but their translation into ZRF code proved problematic due to the previous implementation of move priorities, and was consequently discarded because move priorities had a higher priority.

Some games like *Jungle Game* pose an interesting challenge by having pieces capture according to a hierarchy. This feature was nonetheless dismissed due to a difficulty in accessing pieces lists inside a move definition (mainly due to the fact that moves were made generically with macros and have no information of the pieces that have them).

Regrettably, many goal types were not fully implemented ('Claim', 'Race', 'Breakthrough' and 'Connection'), and some needed more work (Occupy). Most are explained due to the lack of time, but 'Claim' and 'Connection' for instance required very particular goal specific options and a deeper study of their design as well. More options for the goals that were implemented would be a great addition to the project as well.

A major decision was the complete disregard for game specific rules. This decision was pondered and it was decided that the goal of having a generic interface capable of creating numerous (although unknown or unbalanced) games far outweighed the importance of creating all the rules of well known and famous games like *Chess*, *Checkers* and *Go*. One cannot hope to generalize to a point that unique game specific rules are definable in this generalization, unless even more specific options are included in the Generic Interface. A more complete explanation of this problem can be found in the example present in Section 3.1.6.3.

The chart in Figure 5.7 summarizes the prototype’s feature fulfilment in comparison to the project specification.



**Figure 5.7: Implemented Features Fulfilment Chart**

From the 28 features tagged with high priority, 22 of them were fully implemented, while 5 were only partially implemented. In what regards the 24 medium-level features only 7 were fully implemented and 4 partially implemented. Finally, from the total of 18 low priority features only 3 were fully implemented.

This feature fulfilment analysis demonstrates that the priority attribution to each feature was seriously taken in consideration in the development process of this project, respecting the decisions that led to some features having lesser priority than others.

### 5.3 Conclusions

This chapter described the prototype developed for this project with the purpose of testing the claim that ASG creation could be generalized and that the more options available the more complex the amount of games possible to create become.

Focus is given to different aspects of the development process, first by giving a summarized overview of what the prototype is capable of and of its design, followed by the description of the various modules that are part of the system. The prototype is developed in modules, and each of these components is detailed in this chapter, often giving examples of important decisions taken during the development stages and demonstrating some of the features with generated code or diagrams.

The Graphical User Interface is the connecting bridge between the system and its users, serving the purpose of sending the different choices pertaining game creation to the Game Data module. It is also responsible for restricting the user's choices depending on the data inserted and to avoid inaccuracies in a game specification to pass to the other modules.

The different modules interact through the Game Data, which is responsible for storing all the information regarding a game. It is basically a mixture of game objects, each of them symbolizing a key component of an ASG. There are movements, the board, pieces (which in turn have associated movements), promotions to other pieces, board zones and goals.

The Code Generation module is the one responsible for taking the game information and translating it to a playable Zillions of Games game file. It is imbued with a set of methods that allow for the correct generation of Zillions' LISP like language.

The final module is the Generic Language that basically reads and writes games specified in the Generic Interface by means of XML documents. These documents are created according to a generic language designed specifically for this project.

The last section of this chapter, Implemented Features, serves two purposes: detailing what features were in fact implemented and to what degree, and giving a comparative oversight with the project's specification of the previous chapter.

# Chapter 6

## Results

This chapter presents an evaluation of the results achieved by the system specification and by the implemented prototype. In order to test the prototype's ability to create and generate abstract games two different approaches were used. First by analysing game creation with a set of example games created through the interface, and then by testing the prototype with a number of users to validate the system's performance and overall usability.

### 6.1 Game Creation Analysis

Although many features and game mechanics of some ASG were not implemented in the prototype, the amount of possible games to create is astounding nonetheless. Even by adding up all the implemented ASG rules and defining a combinatorial interaction between them it is still impossible to derive the approximate number of games the interface is able to create, even taking in account the necessity to disregard immeasurable aspects like board symmetry or zone definition that fall out of place in this type of analysis. However, what can be done is a rough demonstration of the number of possible combinations of one of the aspects of game creation: movement definition.

Table 6.1: Movement Possibilities Permutations

	Type	Zones	Capturing	Direction	Method	Jump Over	Total
Drop	3	3	1	1	1	1	9
Slide	4	3	3	8	1	1	288
Jump	3	3	3	8	3	6	3888
Radius Jump	1	3	3	1	1	1	9
Total	11	12	10	18	6	9	4194

Table 6.1 shows the potential of the prototype to generate a vast amount of movements, but the real number of possible combinations is even greater if one considers aspects such as the board dimensions and zones that can affect each movement. The prototype's ability to generate distinct games is further expanded by the possibility of having multiple goals in a game, each

with different restrictions. It is possible, for instance, to have a game that ends with the following combination of goals: when all the pieces of a specified type are captured, when a piece reaches a designated goal zone and if a player has no more moves available

The point is that despite several crucial ASG characteristics were intentionally left out (like hexagonal boards) the prototype is capable of generating an immeasurable amount of different games, serving its purpose: generalizing ASG creation.

This section features examples of two games created with the prototype (*Tic-Tac-Toe* and *Chess*), explaining what options are required to generate them and why they are needed. This section also features two games that the prototype was not designed to be able to create but that are possible nonetheless (*Connect 4* and a maze-like game).

### **6.1.1 Examples of Games Created**

This section shall focus a few examples of generated games, what options were required for their definition and a comparison of the final result with ‘hand-made’ ZRF code versus the prototype’s generated game code.

#### **6.1.1.1 Tic-Tac-Toe**

*Tic-Tac-Toe* is considered the simplest abstract game, two players play in turns on a 3x3 board trying to position their pieces in a three piece line. This game’s complexity is negligible since that with perfect play the game always ends in a draw.

Table 6.2 shows all the necessary options a user is required to input in the prototype for the creation of *Tic-Tac-Toe*.

**Table 6.2: Generic Interface input options for *Tic-Tac-Toe***

Game	Name	P1Name	P2Name	
	TicTacToe	circle	cross	
Board	Columns	Rows		
	3	3		
Moves	Name	Type	Drop Type	
	dropped	drop	empty spaces	
Pieces	Name	Moves		
	stone	dropped		
Setup	Piece	P1OffBoard	P1OffBoard	
	stone	5	5	
Goals	Name	Type	Piece Types	Relative Directions
	Pattern 1	Pattern	stone	N, N
	Pattern 2	Pattern	stone	E, E
	Pattern 3	Pattern	stone	NW, NW
	Pattern 4	Pattern	stone	NE, NE

The game itself is fairly simple, only one piece type and one movement available (drop a piece in an empty space). It is also important to note that five off board pieces for each player are necessary to indicate how many pieces each player has to play.

Defining the goals of the game is where the Generic Interface's capabilities become more evident. By choosing the goal type 'Pattern' and then selecting a set of directions one is able to define a relative configuration of pieces on the board. This way, having four goals that define the directions of three piece lines, the goals of *Tic-Tac-Toe* is achieved. The prototype takes each direction selected and generates game code with the selected piece (in this case the only piece type available) and the directions in the following manner:

[piece type] + [direction] + [piece type] + [...] + [direction] + [piece type]

Depending on the number of directions inserted the final goal can have any number of parameters, and it is even possible to create goals with different shaped polygons, like a square if North, East, South and West directions are selected.

#### 6.1.1.2 Chess

*Chess* is by far the most famous abstract game, and is well known for its complexity and strategical gameplay. As such, the rules are well known, but their definition in any programming language requires some expertise. The prototype can generate a simplified version of *Chess*, in which only the *Castling* and the *En Passant* rules were discarded (as explained in section 3.1.6.3).

The game comprises a variety of pieces with different movements, and most notably the pawn that can promote and has three distinct ways of moving (diagonal one space capture, first rank two space movement and normal one space advancement). All the movements have the

‘capturing allowed’ option except for the pawn’s movements, which has ‘not allowed’ for moving forward and ‘mandatory’ for its diagonal capture (it can only move diagonally if capturing). By defining orthogonal (vertical and horizontal) and diagonal movements it is then possible to associate these movements to the Rook, Bishop and Queen pieces. It is also required to define zones in the board that affect when a Pawn promotes and when it can move two steps forward; different zones can have the same name if they affect different players, which in turn simplifies the game’s logic. The goal definition is made simply by choosing which piece needs to be checkmated, and what happens when that occurs (the owner loses the game). Since the pieces in *Chess* all have fixed initial positions it is also necessary to specify these positions for each piece and player in the Generic Interface.

With the prototype’s capabilities creating *Chess* variants is straightforward. *Extinction Chess* can be generated by changing the goal to ‘Capture’ and choosing what piece types are to be targeted; *Losing Chess* only requires a slight change to win in the end game condition of the checkmate goal; *Grand Chess* can be defined by extending the board to 10x10 and adding two new pieces, the Cardinal and the Marshall which combine the rook and bishop’s abilities and the rook and knight’s abilities, respectively. The generalization possibilities are numerous and generating these games is as simple as changing a few options in the prototype.

Table 6.3 shows the set of options the Generic Interface requires to create this version of *Chess*.

**Table 6.3: Generic Interface input options for Chess**

Game	Name	P1Name	P2Name				
	Chess	White	Black				
Board	Columns	Rows	Board Symmetry				
	8	8	All				
Zones	Name	Players Affected	Positions				
	promotion	White	a8,b8,c8,d8,e8,f8,g8,h8				
	promotion	Black	a1,b1,c1,d1,e1,f1,g1,h1				
	rank2	White	a2,b2,c2,d2,e2,f2,g2,h2				
	rank2	Black	a7,b7,c7,d7,e7,f7,g7,h7				
Moves	Name	Type	Distance	Directions	Capture	Zones Restriction	Zones
	slide_orth	slide	any	N,S,E,W	allowed	-	-
	slide_diag	slide	any	NE,NW,SE,SW	allowed	-	-
	pawn_slide	slide	exactly 1	N	not allowed	-	-
	pawn_first	slide	exactly 2	N	not allowed	inside	rank2
	pawn_capture	slide	exactly 1	NW,NE	mandatory	-	-
	king_slide	slide	exactly 1	All	allowed	-	-
	knight_leap	jump	radius 3	-	allowed	-	-
Pieces	Name	Moves					
	Pawn	pawn_slide	pawn_first	pawn_capture			
	Rook	slide_orth	-	-			
	Bishop	slide_diag	-	-			
	Queen	slide_orth	slide_diag	-			
	King	king_slide	-	-			
	Knight	knight_leap	-	-			
Setup	Piece	P1 Positions		P2 Positions		Promoting Zone	Promote To
	Pawn	a2,b2,c2,d2,e2,f2,g2,h2		a7,b7,c7,d7,e7,f7,g7,h7		promotion	Queen, Rook, Bishop, Knight
	Rook	a1,h1		a8,h8		-	-
	Bishop	c1,f1		c8,f8		-	-
	Knight	b1,g1		b8,g8		-	-
	Queen	e1		e8		-	-
	King	d1		d8		-	-
Goals	Name	Type	Piece Types	Players Affected	End Game Condition		
	Checkmate1	Checkmate	King	Both	Loss		



### 6.1.1.3 Connect 4

*Connect 4* is a game played in a vertical board, where pieces are dropped in the top row and they fall to the last free space in the column they were inserted. The goal is to connect four pieces in lines similar to *Tic-Tac-Toe*. Although this may seem simple enough, it requires a specific drop move to simulate gravity (pieces falling down) that was not contemplated in the prototype's development. Another possibility that wasn't implemented was having multiple moves, one to drop a piece in the top row and then another to slide down the piece to the furthest available space. However, with some ingenuity and it is still possible to create this game with the prototype's limited options.

By adding two zones on the first rows of the board and restricting the movements of the pieces to their respective zones (the first player uses the second row and the second player uses the first row) it is possible to simulate the drop and fall of pieces in two turns and to maintain game balance. First a player drops his piece in an available position on his row and on his next turn he is forced to slide down the piece. This is achieved by assigning the drop movements a lesser priority than the fall movements (if a no fall move is available then a drop can be performed).

This example shows how the prototype, despite some crucial features not being implemented, is still versatile enough to allow the creation of some ASG if the user can work around the available options.

Table 6.4 shows the options required for defining this game.

**Table 6.4: Generic Interface input options for *Connect 4***

Game	Name	P1Name	P2Name					
	Connect 4	red	black					
Board	Columns	Rows	Board Symmetry					
	6	9	None					
Zones	Name	Players Affected		Positions				
	row2	red		a8,b8,c8,d8,e8,f8				
	row1	black		a9,b9,c9,d9,e9,f9				
Moves	Name	Type	Type Option	Directions	Capture	Zones Restriction	Zones	Priority
	red_drop	drop	empty spaces	-	-	inside	row1	1
	black_drop	drop	empty spaces	-	-	inside	row2	1
	red_fall	slide	furthest	S	not allowed	inside	row1	0
	black_fall	slide	furthest	S	not allowed	inside	row2	0
Pieces	Name	Moves						
	red_stone	red_drop		red_fall				
	black_stone	black_drop		black_fall				
Setup	Piece	P1OffBoard		P1OffBoard				
	red_stone	21		0				
	black_stone	0		21				
Goals	Name	Type	Piece Types			Relative Directions		
	Pattern 1	Pattern	red_stone		black_stone	N,N,N		
	Pattern 2	Pattern	red_stone		black_stone	E,E,E		
	Pattern 3	Pattern	red_stone		black_stone	NW,NW,NW		
	Pattern 4	Pattern	red_stone		black_stone	NE,NE,NE		

#### 6.1.1.4 Maze

Abstract strategy games are games played between two players, but with some alterations and craftiness one can adapt the prototype's options to serve another purpose, such as creating solitaire puzzle games.

The idea is to force the second player to pass his turn, enabling the 'Pass on stalemate' option, and give his pieces no valid movements. The first player can have a piece that moves one square in each orthogonal direction and the goal of the puzzle is to reach a specific space in the board (which is achieved with a goal type 'Occupy' and a goal zone with that one space). A labyrinth of a different piece type can set, forcing the only moving piece to find its way across the board (no capturing allowed).

Table 6.5 shows a summarized version of the options needed to define a game such as the one described above. Appendix C shows a screenshot of this game running on the Zillions application.

**Table 6.5: Generic Interface simplified input options for a maze-like puzzle**

Game	Name	P1Name	P2Name		
	Maze	P1	P2		
Board	Columns	Rows			
	12	12			
Zones	Name	Players Affected	Positions		
	goal_zone	P1	i12		
Moves	Name	Type	Type	Directions	Capture
	slide	slide	exactly 1	N, S, E, W	not allowed
Pieces	Name	Moves	P1 Positions		
	Token	slide	a1		
	Wall	-	[any]		
Goals	Name	Type	Piece Types	Goal Zones	
	Occupy1	Occupy	Token	goal_zone	

Generalizing ASG is the main goal of the project, but this generalization gave forth new possibilities not previously thought possible with the current state of the prototype. Ingenuity and creativity play a vital role, but this capability of the Generic Interface is noteworthy nonetheless.

### 6.1.2 Game List

The Generic Interface is a tool that allows users the possibility of creating any kind of ASG. Table 6.6 shows some well-known games that are possible, to some extent, to define with the prototype. For each game the unsupported rules or features are also detailed.

**Table 6.6: Well-known ASG list and Prototype limitations**

Games	Unimplemented Rules / Features			
Alquerque	Multiple captures			
Amazons	Special drop requirements	Multiple moves	Adv. promotion options (add pieces)	
Arimaa	Special move distance requirements	Multiple moves	Push/Pull moves	Adv. turn Order
Attaxx	Flipping moves			
Checkers	Multiple captures			
Chess	Castling		En passant	
Connect 4				
Go	Piece positioning capture		Claim goal type	
Hex	Hexagonal Board			
Jungle Game	Hiararchy capturing			
Lines of Action	Special move distance requirements			
Nine Mens Morris	Advanced capturing	Manual board configuration		
Reversi	Flipping moves	Special drop requirements		
Pente	Piece positioning capture			
Quixo	Special drop requirements	Board edge links		
Surakarta	Piece positioning capture			
Tablut	Piece positioning capture	Advanced zone restrictions		
Tic Tac Toe				

As explained in section 5.2 (Implemented Features), the most generic and common rules of ASG were given a higher priority than more complex and unfamiliar ones.

## Chapter 7

# Conclusion and Future Work

### 7.1 Work Summary

The goals of this dissertation were to specify and develop a prototype of a system capable of generalizing abstract game creation. The system should be able to generate game code in two outputs, one to be used and tested a general game playing system and another in a generic abstract game language.

The project was divided in three steps. Firstly an extended research on board games and ASG in particular was done. Secondly, design and specification of a system that could generate such games and lastly the development of a working prototype of the project to assert the research done and to validate the claim that generalizing ASG creation is possible.

In order to achieve the proposed goals several steps were undertaken, starting with an extended research on abstract strategy games and board games to a smaller extent. This research provided the means to classify and categorize these games in terms of mechanics and similarities between rules, a necessity for the later specification of a generic language that could comprehend all abstract games. Understanding the way abstract games work and more importantly, how they relate to one another was an essential step in preparing the groundwork for developing a generic platform for ASG.

Artificial Intelligence techniques applied to ASG were also researched, to study what methodologies would be better suited to create an AI agent-based system capable of playing multiple games. The AI approaches covered, such as machine learning and evaluation functions, allowed for a broader view of the way problem-solving can be done in abstract games. The solution for designing an AI engine capable of playing ASG should not be focused on a singular AI approach, as it is usually done, but instead it should combine ideas from several approaches.

The final research topic covered was the study of general game playing systems that could be used in conjunction with the project, in order to test and validate the games created with the developed system. The result of this research was the definition of the Zillions of Games commercial application as the cornerstone of the project, mainly due to the fact that it does already

support the creation of almost any kind of abstract strategy game and is also equipped with a somewhat powerful rule analysing AI engine.

The second stage of the project was the specification and design of the Generic Interface system, defining how ASG should be created and what manner of options should be presented to the users in order to simplify the game creation process.

During the design and specification stage of the project an opportunity came up to present the work done so far in the conference “Videojogos 2010 – Conference of Sciences and Arts in Video Games”, held in Lisbon at IST (Instituto Superior Técnico) during 15, 16 and 17 of September 2010 [Videojogos, 2010]. A Work in Progress paper was submitted and accepted for this conference and is currently awaiting publication in the information and communication sciences magazine “Prisma.com” [Prisma, 2011] [Reis & Reis, 2010].

The final stage was the development of a prototype of the system that could be used to verify and test the project’s design and specification. The prototype is a software tool that allows users to create several distinct abstract games through a series of options, which define the games’ rules.

Despite the fact that the prototype developed is still not capable of generating all kinds of different games, it is important to note that the main question of the project is answered: it is possible to generalize abstract strategy game creation. Furthermore, it was also made clear that the more options are featured in the Generic Interface the more complex the games possible to create will become. It is regretful that some famous games like *Chess* can be generated but with the exception of particular rules (like castling) that are so specific to each game that their generalization becomes too complex. Priority was therefore given to generalization over specificity.

This project opens up the possibility to easily create and test games, without the need to program them. It is a step forward in ASG creation in general and it represents a development for AI research.

## 7.2 Future Work

Regardless of having a complete system specification and a functional prototype that implements most of the proposed features, the project could benefit from further developments in order to become a more mature system:

- Fully implement all the features defined in the project’s specification (sections 4.1.2 and 5.2 of this document). Further development of the project in this area would allow for a more complete generalization of ASG, making it possible to create games with rules that deviate from typical ASG mechanics.

- The prototype is a software application that works locally, and turning it to a web-service would be an interesting development. The system was implemented using WPF technology, but it could be straightforwardly rewritten in Silverlight to accommodate for a web environment.
- The most restricting fact about the project is that it requires the Zillions of games platform to play and test the games created. It would be an advantageous advancement to the project to build a new general game playing platform capable of playing the games created through the Generic Interface.
- Alongside the development of a new general game playing system is the development of a generic AI engine similar to the one Zillions uses, but with the learning and user-input heuristics capabilities described in section 3.4. This new development of the project is by far the most ambitious one, but its concretization could mean a revolutionary step forward for ASG creation and the AI methodologies implied herein.

# References

- [AAAI, 2010] *Association for the Advancement of Artificial Intelligence*. USA, 2010. Website available from: [www.aaai.org/home.html](http://www.aaai.org/home.html) (Cited 20/02/2010)
- [Abbott, 1975] Abbott, R. *Under the Strategy Tree*. Games & Puzzles Issue 36, England, UK, 1975. Available from: [www.logicmazes.com/games/tree.html](http://www.logicmazes.com/games/tree.html). (Cited 18/02/2010)
- [Abbott, 1988] Abbott, R. *What's Wrong With Ultima*. World Game Review, Issue 8, July 1988. Available from: [www.logicmazes.com/games/wgr.html](http://www.logicmazes.com/games/wgr.html). (Cited 8/02/2010)
- [AllExperts, 2010] *Board game: Encyclopedia*. AllExperts, 2010. Website available from: [www.associatepublisher.com/e/b/bo/board\\_game.htm](http://www.associatepublisher.com/e/b/bo/board_game.htm). (Cited 15/02/2010)
- [AP, 1998] *Artificial Intelligence: Recursive Search – How AI Works to Calculate Load-Points & Stresses in Stretched-Domestic & Multi-level Building & Structures*. Archi-Plans-4u, 2008. Website available from: [archi-plans-4u.com/rec.htm](http://archi-plans-4u.com/rec.htm). (Cited 13/02/2010)
- [Ashlock, 2006] Ashlock, D. *Evolutionary Computation for Modeling and Optimization*. Springer, New York, USA, 2006
- [Bäck et al., 1997] Bäck, T., Fogel, D., Michalewicz, Z. *Handbook of Evolutionary Computation*, Oxford University Press, Oxford, UK, 1997
- [BGDF, 2008] *Game Types*. Board Game Designers Forum, August 2008. Website available from: [www.bgdf.com/node/276](http://www.bgdf.com/node/276). (Cited 17/02/2010)
- [BGS, 2010] *Board Game Studies*. International Society for Board Game Studies, 2010. Website available from: [www.boardgamestudies.info](http://www.boardgamestudies.info). (Cited 17/02/2010)
- [BGG, 2011] *BordGameGeek*, 2011. Website available from: [www.boardgamegeek.com](http://www.boardgamegeek.com). (Cited 10/01/2011)
- [Boden, 1996] Boden, M. *Artificial Intelligence – Handbook of Perception and Cognition 2<sup>nd</sup> Edition*. University of Cognitive and Computing Sciences, University of Sussex, Academic Press, Brighton, England, UK, 1996
- [Brooks, 2003] Brooks, R. *Robots will invade our lives*. TED 2003. Available from: [www.ted.com/talks/rodney\\_brooks\\_on\\_robots.html](http://www.ted.com/talks/rodney_brooks_on_robots.html)



- [Browne & Maire, 2010] Browne, C., Maire, F. *Evolutionary Game Design*. IEEE Transactions on Computational Intelligence and AI in Games, Vol. 2, Issue 1, pp 1-16, March 2010.
- [Carbonell et al., 1983] Carbonell, J., Michalski, R., Mitchell, T. *Machine Learning – A Historical and Methodological Analysis*. The AI Magazine, Vol. 4, No. 3, pp. 69-79, 1983. Available from: [www.aaai.org/ojs/index.php/aimagazine/article/view/406/342](http://www.aaai.org/ojs/index.php/aimagazine/article/view/406/342)
- [Copeland, 2000] Copeland, J. *A Brief History of Computing - The Manchester Machine*. AlanTuring.net, June 2000. Available from: [www.alanturing.net/turing\\_archive/pages/Reference%20Articles/BriefHistofComp.html](http://www.alanturing.net/turing_archive/pages/Reference%20Articles/BriefHistofComp.html). (Cited 13/02/2010)
- [Piazza & Helsabeck, 1990] Piazza J., Helsabeck, A. *Laps: Cases to Models to Complete Expert Systems*. AI Magazine Vol 11, No 3, pp. 80-107, 1990. Available from: [www.aaai.org/ojs/index.php/aimagazine/article/view/844/762](http://www.aaai.org/ojs/index.php/aimagazine/article/view/844/762)
- [Dobbe, 2007] Dobbe, J., *A Domain-Specific Language for Computer Games*. Delft University of Technology, Delft, the Netherlands, July 2007
- [EETimes, 2009] EE Times Group, *Using multi-core processors in embedded systems*. EE Times, October 2009. Available from: [www.eetimes.com/design/industrial-control/4019514/Using-multicore-procesors-in-embedded-systems--part-1](http://www.eetimes.com/design/industrial-control/4019514/Using-multicore-procesors-in-embedded-systems--part-1). (Cited 13/02/2010)
- [Eppstein, 2007] Eppstein, D. *Combinatorial Game Theory*. University of California, Irvine, California, USA, 2007. Available from: [www.ics.uci.edu/~eppstein/cgt](http://www.ics.uci.edu/~eppstein/cgt). (Cited 17/02/2010)
- [Epstein, 1999] Epstein, S. *Game Playing: The Next Moves*. Proceedings of the Sixteenth National Conference on Artificial Intelligence. AAAI Press, Menlo Park, California, USA, pp. 987-993, 1999.
- [FIDE, 2010] FIDE – World Chess Federation. Website available from: [www.fide.com](http://www.fide.com). (Cited 18/02/2010)
- [Furse, 1995] Furse, E. *Learning Board Games*. University of Glamorgan, Wales, UK, 1995. Available from: [www.comp.glam.ac.uk/pages/research/ai-cognitive-science/learning-board-games.html](http://www.comp.glam.ac.uk/pages/research/ai-cognitive-science/learning-board-games.html) (Cited 19/02/2010)
- [Genesereth & Love, 2005] Genesereth, M., Love, N. *General Game Playing: Overview of the AAAI Competition*. Stanford University, California, USA, 2005
- [Gobet, 2006] Gobet, F. *Adriaan De Groot: Marriage of Two Passions*. ICGA Journal, Uxbridge, UK, December 2006. Available from: [people.brunel.ac.uk/~hsstffg/preprints/De%20Groot%27s%20obituary%20--%20ICGA.pdf](http://people.brunel.ac.uk/~hsstffg/preprints/De%20Groot%27s%20obituary%20--%20ICGA.pdf) (Cited 20/02/2010)

- [Handerson, 2007] Handerson, H. *Artificial Intelligence – Mirror for the Mind*. Chelsea House Publishers, England, UK, 2007
- [IAGO, 2009] *Definitions of Abstracts*. International Abstract Games Organization, 2009 Available from: [abstractgamers.org/wiki/definitions-of-abstracts](http://abstractgamers.org/wiki/definitions-of-abstracts). (Cited 17/02/2010)
- [IBM Research, 1997] *Kasparov vs Deep Blue – the Rematch*. IBM Research, 1997. Available from: [www.research.ibm.com/deepblue/home/html/b.shtml](http://www.research.ibm.com/deepblue/home/html/b.shtml) (Cited 13/02/2010)
- [Ignizio, 1991] Ignizio, J. *An Introduction To Expert Systems – The Development and Implementation of Rule-Based Expert Systems*. Mcgraw-Hill College, Boston, Massachusetts, USA, January 1991.
- [Intel Corporation, 2005] *Excerpts from a Conversation with Gordon Moore: Moore's Law*. Intel Corporation. 2005. Available from: [ftp://download.intel.com/museum/Moores\\_Law/Video-Transcripts/Excepts\\_A\\_Conversation\\_with\\_Gordon\\_Moore.pdf](http://download.intel.com/museum/Moores_Law/Video-Transcripts/Excepts_A_Conversation_with_Gordon_Moore.pdf) (Cited 13/02/2010)
- [Isom, 2005] Isom, J. *A Brief History of Robotics*. MegaGiant Robotics, 2005. Available from: [robotics.megagiant.com/history.html](http://robotics.megagiant.com/history.html). (Cited 13/02/2010)
- [Konar, 1999] Konar, A. *Artificial Intelligence and Soft Computing: Behavioral and Cognitive Modeling of the Human Brain*. CRC Press, Inc. Boca Raton, Florida, USA, December 1999
- [Krauthammer, 1998] Krauthammer, C. *BE AFRAID. The Meaning of Deep Blue's Victory*. 1998. Available from: [wright.chebucto.net/AI.html](http://wright.chebucto.net/AI.html). (Cited 13/02/2010)
- [Krishnamoorthy & Rajeev, 1996] Krishnamoorthy, C., Rajeev, S. *Artificial Intelligence and Expert Systems for Engineers*. CRC Press, Inc., Boca Raton, Florida, USA, September 1996
- [Kurzweill, 2005] Kurzweill, R. *The Singularity Is Near: When Humans Transcend Biology*. Viking Press, 2005
- [Leung, 2009] Leung, H. *Knot Theory*. Department of Mathematics, Cornell University, New York, USA. 2009. Available from: [www.math.cornell.edu/~mec/2008-2009/HoHonLeung/page6\\_knots.htm](http://www.math.cornell.edu/~mec/2008-2009/HoHonLeung/page6_knots.htm). (Cited 4/01/2011)
- [Levy, 2006] Levy, D. *Robots Unlimited: Life in a Virtual Age*. A K Peters, Ltd., USA, 2006

- [Love et al., 2008] Love, N., Hinrichs, T., Haley, D., Genesereth, M. *General Game Playing: Game Description Language Specification*. Stanford Logic Group Computer Science Department Stanford University, California, USA, 2008
- [Luckin et al., 2007] Luckin, R., Koedinger, K., Greer, J. *Artificial Intelligence in Education – Building Technology Rich Learning Contexts That Work*. Frontiers in Artificial Intelligence Applications Series, IOS Press, Amsterdam, the Netherlands, 2007
- [Luger, 2009] Luger, R. *Artificial Intelligence: Structures and Strategies for Complex Problem Solving*. 6<sup>th</sup> Edition. Addison-Wesley, Reading, Massachusetts, USA, 2009
- [McCarthy et al., 1955] McCarthy, J., Minsky, M., Rochester, N., Shannon, C. *A Proposal for the Dartmouth Summer Research Project on Artificial Intelligence*. Dartmouth College, New Hampshire, USA, 1955. Available from: [www-formal.stanford.edu/jmc/history/dartmouth/dartmouth.html](http://www-formal.stanford.edu/jmc/history/dartmouth/dartmouth.html). (Cited 13/02/2010)
- [McCorduck, 2004] McCorduck, P. *Machines Who Think: A Personal Inquiry into the History and Prospects of Artificial Intelligence – 25<sup>th</sup> Anniversary Edition*. A K Peters Ltd., Natick, Massachusetts, USA, 2004
- [Michalski et al., 1983] Michalski, R., Carbonell, J., & Mitchell, T. *Machine Learning: An Artificial Intelligence Approach*. Morgan Kaufmann Publishers, San Francisco, California, USA, pp. 1-10, 1983.
- [Millington, 2006] Millington, I. *Artificial Intelligence for Games*. Series in Interactive 3D Technology. Morgan Kaufmann Publishers, San Francisco, California, USA, 2006
- [Mitchell, 2007] Mitchell, T. *The Discipline and Future of Machine Learning*. Seminar Talk, Carnegie Mellon University School of Computer Science's Machine Learning Department, 2007. Available from: [www.ml.cmu.edu/seminars/Mitchell\\_lecture.3.07.mov](http://www.ml.cmu.edu/seminars/Mitchell_lecture.3.07.mov). (Cited 21/02/2010)
- [Moore, 1965] Moore, G. *Cramming more components onto integrated circuits*. Electronics, Vol. 38, No. 8, pp.114-117, April 1965. Available from: [ftp://download.intel.com/research/silicon/moorespaper.pdf](http://download.intel.com/research/silicon/moorespaper.pdf) (Cited 12/02/2011)
- [Neumann & Morgenstern, 1944] Neumann, J., Morgenstern, O. *Theory of Games and Economic Behavior*. Princeton University Press, New Jersey, USA, 1944. Available from: [www.archive.org/stream/theoryofgamesand030098mbp#page/n9/mod/e/2up](http://www.archive.org/stream/theoryofgamesand030098mbp#page/n9/mod/e/2up) (Cited 20/02/2010)

- [Neto, 2003] Neto, J. *The LUDÆ Project*. 2003. Website available from: [homepages.di.fc.ul.pt/~jpn/ludae/index.htm](http://homepages.di.fc.ul.pt/~jpn/ludae/index.htm). (Cited 02/02/2010)
- [Neto, 2010] Neto, J. *World of Abstract Games*. 2010. Website available from: [homepages.di.fc.ul.pt/~jpn/gv](http://homepages.di.fc.ul.pt/~jpn/gv) (Cited 02/02/2010)
- [Newell & Simon, 1972] Newell, A., Simon, H. *Human problem solving*. Prentice-Hall, Englewood Cliffs, New Jersey, USA, 1972
- [Newell & Simon, 1976] Newell, A., Simon, H. *Computer science as empirical inquiry: symbols and search*. Communications of the ACM, Vol. 19, No. 3, pp. 113-126, March 1976.
- [Nilsson, 1999] Nilsson, N. *Artificial Intelligence – A new Synthesis*. China Machine Press, China, 1999
- [ODP, 2008] *Board Games: Abstract*. Open Directory Project, 2008. Available from: [www.dmoz.org/Games/Board\\_Games/Abstract](http://www.dmoz.org/Games/Board_Games/Abstract). (Cited 17/02/2010)
- [PO, 2008] *Philosophy of Mind - Functionalism - Strong and Weak AI*. Philosophy Online, 2008. Available from: [www.philosophyonline.co.uk/pom/pom\\_functionalism\\_AI.htm](http://www.philosophyonline.co.uk/pom/pom_functionalism_AI.htm). (Cited 15/02/2010)
- [Piccione, 1980] Piccione, P. *In Search of the Meaning of Senet*. Archaeology 33, pp. 55-58, 1980.
- [Poli et al., 2008] Poli, R., Langdon, W., McPhee, N. *A Field Guide to Genetic Programming*. UK, March 2008.
- [Prisma, 2011] *Prisma.com – Revista de Ciências da Informação e de Comunicação do CETAC*. Portugal, 2011. Available from: [prisma.cetac.up.pt](http://prisma.cetac.up.pt)
- [Rasskin-Gutman, 2009] Rasskin-Gutman, D. *Chess Metaphors – Artificial Intelligence and the Human Mind*. The MIT Press, Cambridge, Massachusetts, USA, 2009.
- [Raynor, 1999] Raynor, W. *The International Dictionary of Artificial Intelligence*. Glenlake Publishing Company, Ltd., USA, 1999
- [Reis, 2009] Reis, L. *Proposta de Dissertação: Inteligência Artificial em Jogos de Tabuleiro*, 2009. Available from: [www.fe.up.pt/si/ESTAGIOS\\_EMPRESAS.VER\\_DADOS\\_PROPOSTA?p\\_id=31439](http://www.fe.up.pt/si/ESTAGIOS_EMPRESAS.VER_DADOS_PROPOSTA?p_id=31439)
- [Reis & Reis, 2010] Reis, I., Reis, L. *Generic Interface for Developing Abstract Strategy Games*. FEUP, Porto, Portugal, 2010. Available from: [ivopazdosreis-thesis.weebly.com](http://ivopazdosreis-thesis.weebly.com)

- [Reynolds, 2009] Reynolds, S. *Strategy and Tactics: What's the Difference?* Abstract Strategy Games – Anything and everything about abstract strategy games, August 2009. Website available from: <http://abstractstrategygames.com/?p=118>. (Cited 17/02/2010)
- [Rising, 2007] Rising, G. *Inside your Calculator – From Simple Programs to Significant Insights*. John Wiley & Sons, Inc., New Jersey, USA, July 2007
- [Roé, 2011] Roé, J. *Ancient Egyptian Game: Senet*. Ancient Sacred Arts, 2011. Available from: [www.freewebs.com/javasroe/senettblajtk.htm](http://www.freewebs.com/javasroe/senettblajtk.htm) (Cited 18/02/2011)
- [Russel & Norvig, 1995] Russel, S., Norvig, P. *Artificial Intelligence – A Modern Approach*. Prentice Hall Series in Artificial Intelligence, Prentice Hall, Inc., New Jersey, USA, 1995
- [Schwab, 2004] Schwab, B. *AI Game Engine Programming*. 2<sup>nd</sup> Edition, Game Development Series. Charles River Media, Inc. Rockland, Massachussets, USA, 2004
- [Scott, 2002] Scott, J. *Machine Learning in Games*. 2002. Available from: [satirist.org/learn-game/](http://satirist.org/learn-game/). (Cited 21/02/2010)
- [Setup Group, 2010] *Abstract board games information center*. Setup Group, Inc. 2010. Website available at: [www.setupgroup.com/xo](http://www.setupgroup.com/xo). (Cited 18/02/2010)
- [Soltis, 1995] Soltis, A. *Pawn Structure Chess*. McKay Chess Library, 1995.
- [Thompson, 2000] Thompson, J. *Defining the Abstract*. The Games Journal – A Magazine About Boardgames, 2000. Available from: [www.thegamesjournal.com/articles/DefiningtheAbstract.shtml](http://www.thegamesjournal.com/articles/DefiningtheAbstract.shtml)
- [Thompson, 2010] Thompson, J. *Abstract Games*. 2010 Available from: [home.flash.net/~markthom/html/abstract\\_games.html](http://home.flash.net/~markthom/html/abstract_games.html) (Cited 17/02/2010)
- [Videojogos, 2010] *Videojogos 2010 – 3<sup>a</sup> Conferência Anual em Ciência e Artes dos Video Jogos*. Lisbon, Portugal, September 2010. Available from: <http://gaips.inesc-id.pt/videojogos2010/>
- [Voogt, 1998] Voogt, A. *Board Game Studies*. International Journal for the Study of Board Games, Vol. 1, No. 3, 1998. Available from: [www.boardgamestudies.info/pdf/issue1/BGS1-complete.pdf](http://www.boardgamestudies.info/pdf/issue1/BGS1-complete.pdf) (Cited 20/02/2010)
- [ZoG, 2009] *Zillions of Games*. Zillions Development Corporation, 2009. Available from: [www.zillionsogames.com/](http://www.zillionsogames.com/)

# Glossary

<b>capture</b>	In board games, a capture occurs when an opponent piece is removed from the board by a move of a piece of the current player
<b>checkmate</b>	A piece is considered checkmated if it is being attacked and all of its movements point to spaces on the board attacked by enemy pieces
<b>claims</b>	In board games, a claim is when part of the territory (of the board) possession is transferred to a player
<b>connection</b>	In board games, a connection represents a line or geometric figure composed by pieces occupying spaces of the board. For example, a line of three pieces, hence connected, to win
<b>elimination</b>	Elimination occurs when a piece or a space in the board is removed from the game, it does not necessarily mean it is a capture
<b>flipping</b>	When a piece changes owner during the course of a game it is said that the piece is flipped (like flipping a two-sided coin)
<b>immobilization</b>	When a piece is unable to perform any kind of movement it is considered immobilized
<b>jump</b>	If it is possible to jump over a piece in a movement, that it, land in a space without going through all spaces in between
<b>castling</b>	A specific move for <i>Chess</i> and chess-like games, where the king and a rook can both move in the same turn and not according to the rules that define their normal movement
<b>combinatorial explosion</b>	A problem (such as finding a chess move) that grows rapidly in complexity because of the exponentially increasing number of possibilities
<b>consciousness</b>	Something that humans feel is part of their essence, but very hard to pin down. Elements of consciousness may include self-awareness, perception of being a “subject,” of one’s experience, and the

	ability to experience emotions as opposed to simple neural stimulation
<b>Deep Blue</b>	An IBM program that defeated world chess champion Garry Kasparov in a match in 1997
<b>depth</b>	The number of levels to be examined in a tree-type search. In chess, for example, the depth is the number of moves (or half moves) to look ahead
<b>en passant</b>	A <i>Chess</i> specific rule, where a pawn can capture an opponent's pawn without touching it but only if certain criteria are met
<b>expert system</b>	Software that examines a set of rules or assertions in order to draw useful conclusions, such as for diagnosing faults in an engine.
<b>knowledge base</b>	A database of facts, descriptions, or rules used by an expert system
<b>knowledge engineer</b>	A person who creates knowledge bases for expert systems, usually after extensively interviewing a human expert in the appropriate field
<b>Lisp</b>	One of the most popular languages used for a variety of AI projects because of its ability to manipulate lists and create powerful functions
<b>neural network</b>	A system in which a grid of processing units (nodes) are arranged similar to neurons in the human brain. Successful nodes are reinforced, creating a learning effect.
<b>occupation</b>	When a piece is in a space of the board it is occupying it
<b>stack</b>	In board games, if pieces are able to be placed upon each other they form a stack
<b>stalemate</b>	A player is stalemated if he has no more available moves
<b>Strong AI</b>	The belief that sufficiently powerful Artificial Intelligence can achieve humanlike mental states, including emotions and consciousness
<b>Weak AI</b>	The belief that intelligent human behaviour can be modelled or simulated by a machine, without making claims about the machine's consciousness or inner state.

# Appendix A

## Interface Design Mock-ups

The mock-up shows a window titled "Abstract Games Creation Kit" with a menu bar (File, Edit, Tools, Help) and a tabbed interface. The "Game" tab is active, while "Board", "Moves", "Pieces", "Setup", and "Goals" are grayed out. A yellow callout box explains that all tabs are grayed out until necessary information is inserted, with "Title" being mandatory. The "Game" tab contains a "Title" field with the placeholder "insert game title here". Below this is an "Optional Information" section with two columns for "Player 1" and "Player 2", each with a name field and a "color chooser" button. A "Description:" field with a text area and scrollbars contains the placeholder "write a short text explaining the game's rules and goals". Below this is a "History:" field with a text area and scrollbars containing "add a little flavour by providing the game's history". At the bottom of the "Optional Information" section is a "Strategy:" field with a text area and scrollbars containing "briefly explain the basic strategies and tactics that provide a good chance of winning the game". At the bottom of the window are "Save", "Edit Code", and "Next" buttons. A footer bar states "This area is reserved for explaining functionalities of the program".

Abstract Games Creation Kit

File Edit Tools Help

Game Board Moves Pieces Setup Goals

Title insert game title here

Optional Information

Player 1 insert player name color chooser

Player 2 insert player name

Description: write a short text explaining the game's rules and goals

History: add a little flavour by providing the game's history

Strategy: briefly explain the basic strategies and tactics that provide a good chance of winning the game

Save Edit Code Next

This area is reserved for explaining functionalities of the program

All the tabs are grayed out until the necessary information is inserted, in this case only "Title" is mandatory

Figure A.1: Start-up Game Tab Design Mock-up



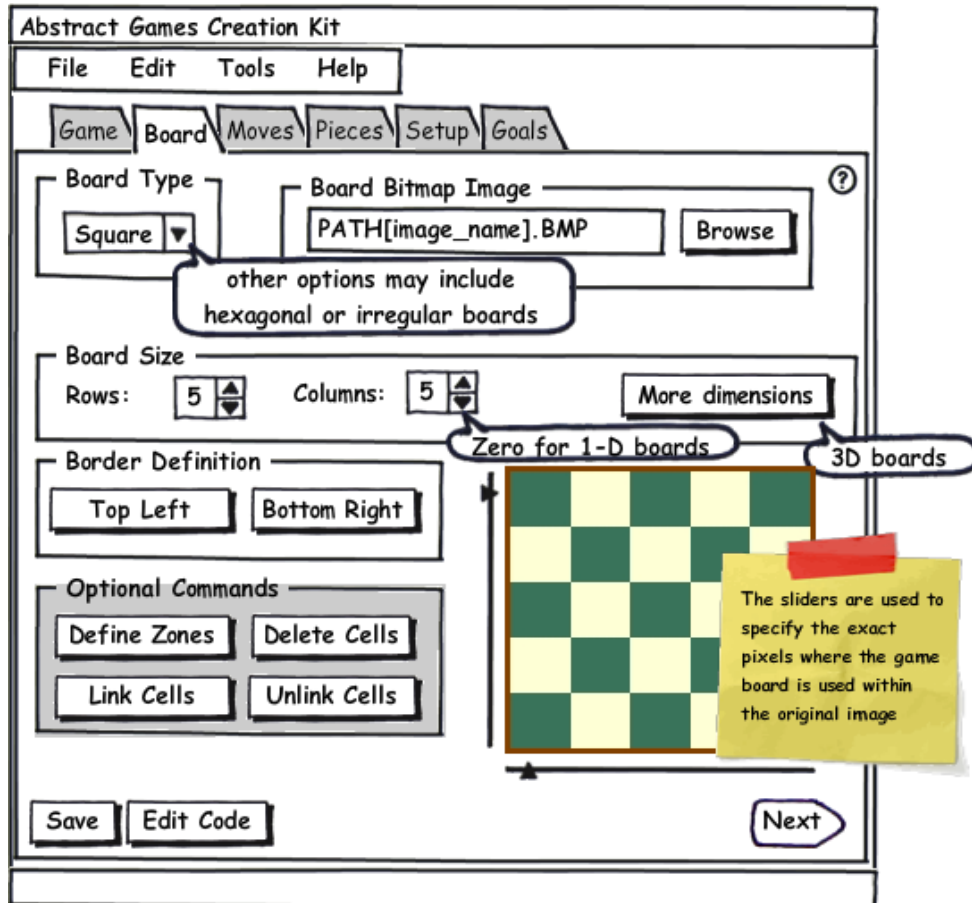


Figure A.2: Board Tab Design Mock-up

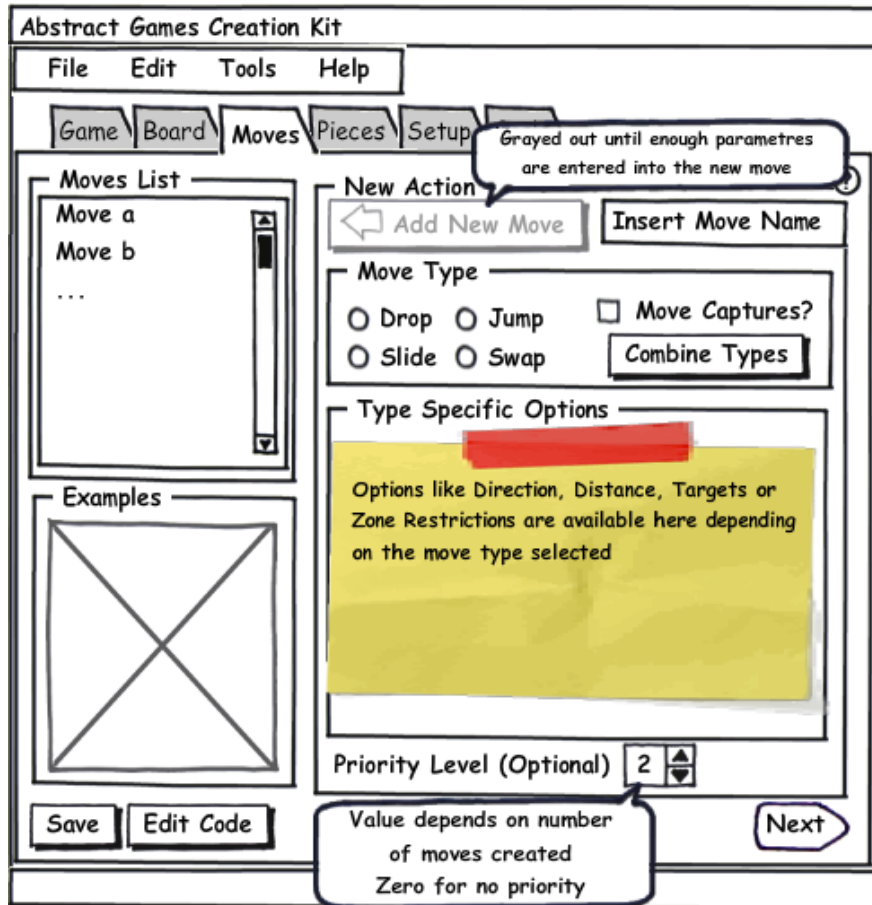


Figure A.3: Moves Tab Design Mock-up

Abstract Games Creation Kit

File
Edit
Tools
Help

Game
Board
Moves
Pieces
Setup
Goals

Create a Piece Type

Name: insert name

Help: short description

Description: insert a full description of the piece and its abilities (optional)

Moves List

Move a

Move b

Move c

...

→ Add

← Remove

Add All Moves

Piece Bitmap Images

P1 PATH[image\_name].BMP Browse

P2 PATH[image\_name].BMP Browse

Piece Moves

Move b

Browse Piece List

Add Piece to Piece List

Create New Piece

Save

Edit Code

Next

Figure A.4: Pieces Tab Design Mock-up

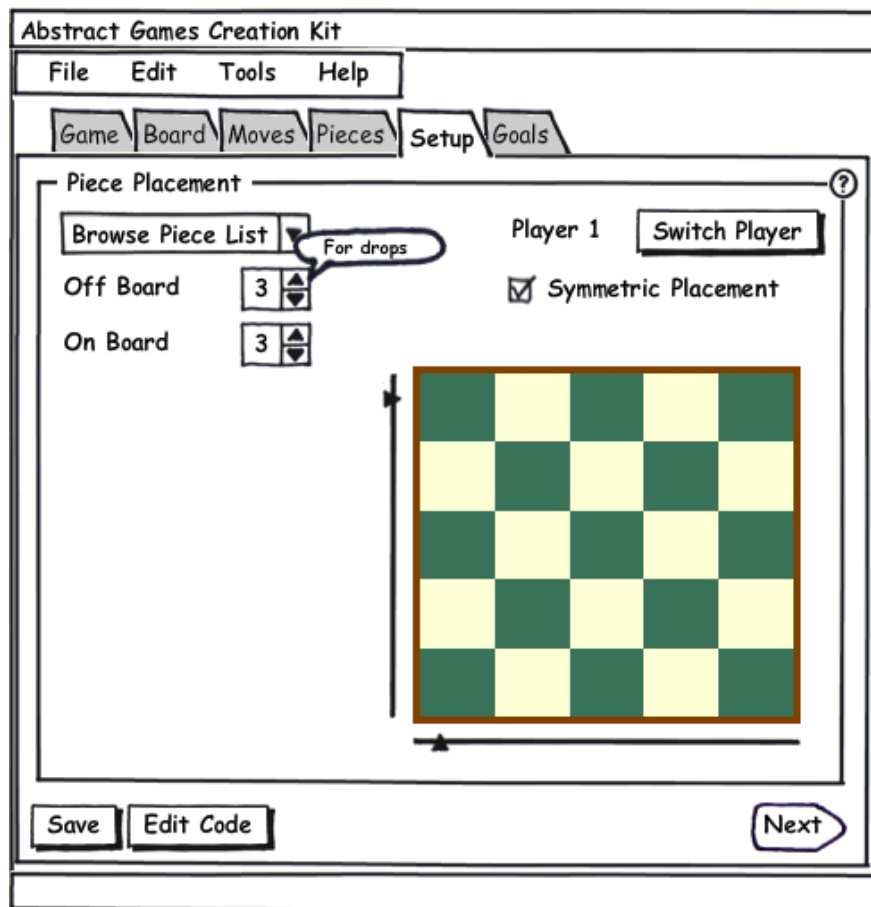


Figure A.5: Setup Tab Design Mock-up

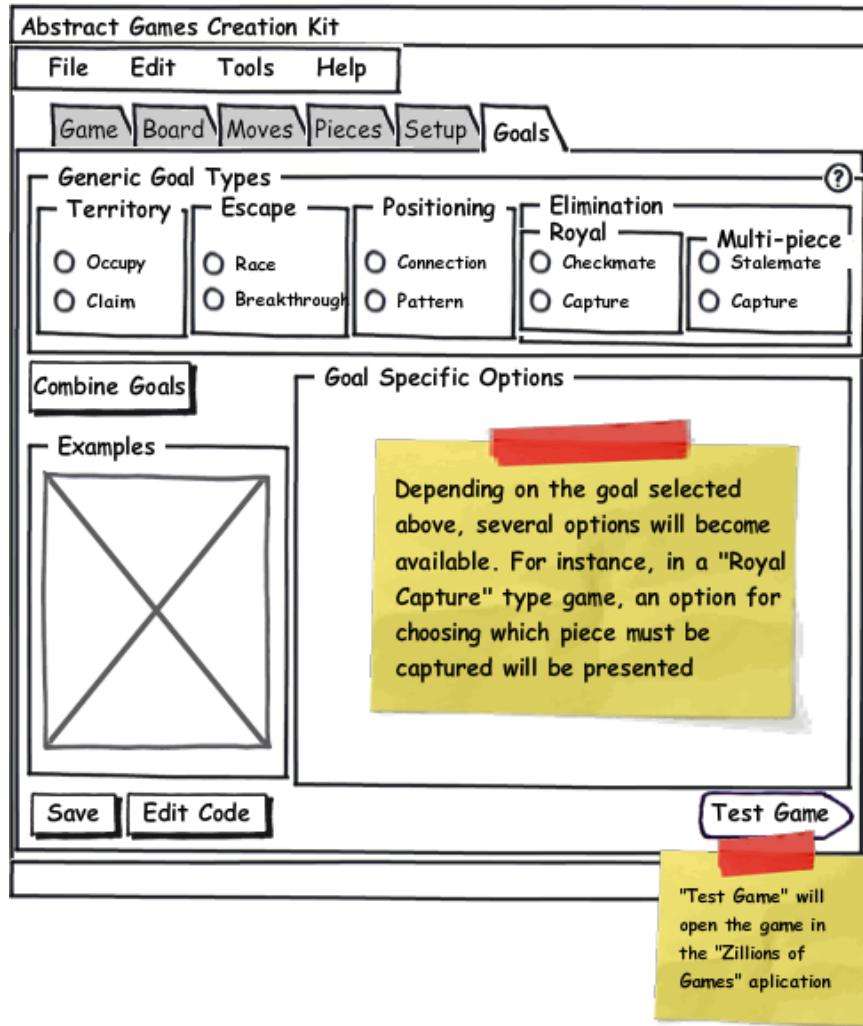
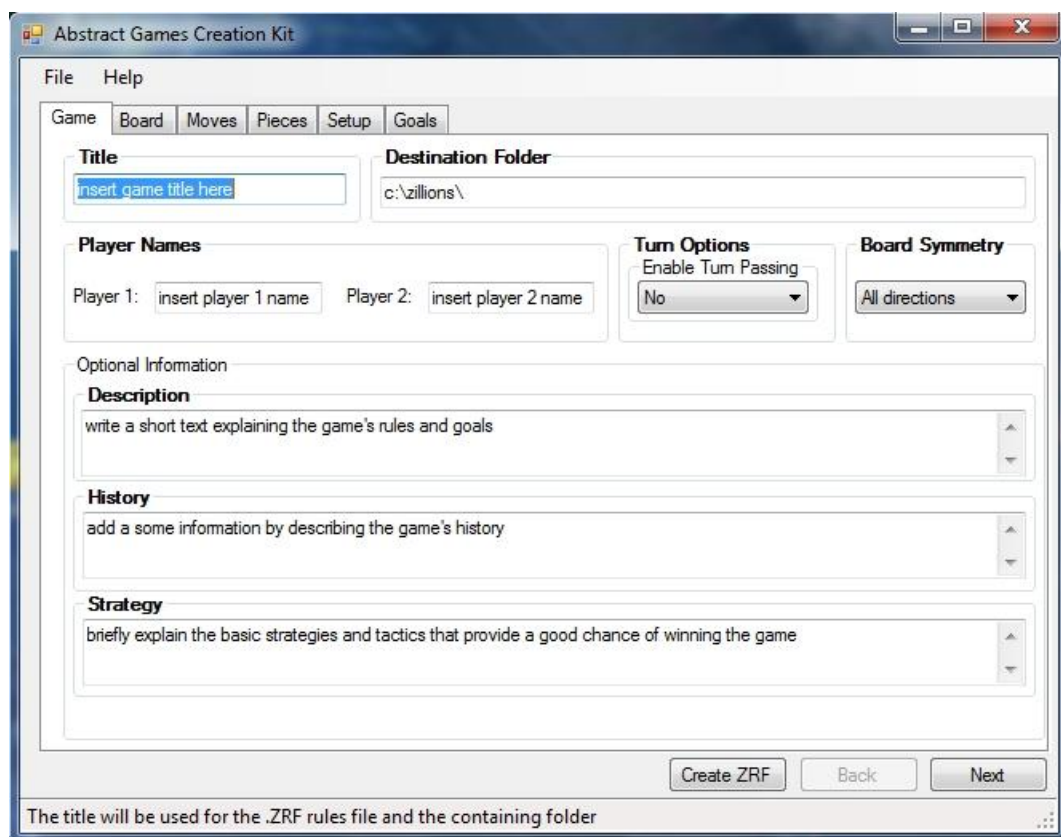


Figure A.6: Goal Tab Design Mock-up

## Appendix B

### Prototype Screenshots



The screenshot shows a software window titled "Abstract Games Creation Kit" with a standard Windows interface. Inside, there's a menu bar with "File" and "Help". Below it is a tabbed interface with "Game", "Board", "Moves", "Pieces", "Setup", and "Goals". The "Game" tab is active, displaying several input fields and options. At the top, there's a "Title" field with placeholder text "insert game title here" and a "Destination Folder" field with "c:\zillions\". Below these are three sections: "Player Names" with fields for "Player 1" and "Player 2", "Turn Options" with a dropdown for "Enable Turn Passing" (set to "No"), and "Board Symmetry" with a dropdown for "All directions". A section titled "Optional Information" contains three text areas: "Description" (placeholder: "write a short text explaining the game's rules and goals"), "History" (placeholder: "add a some information by describing the game's history"), and "Strategy" (placeholder: "briefly explain the basic strategies and tactics that provide a good chance of winning the game"). At the bottom right are "Create ZRF", "Back", and "Next" buttons. A status bar at the very bottom states: "The title will be used for the .ZRF rules file and the containing folder".

Figure B.1: Game Tab Prototype Screenshot

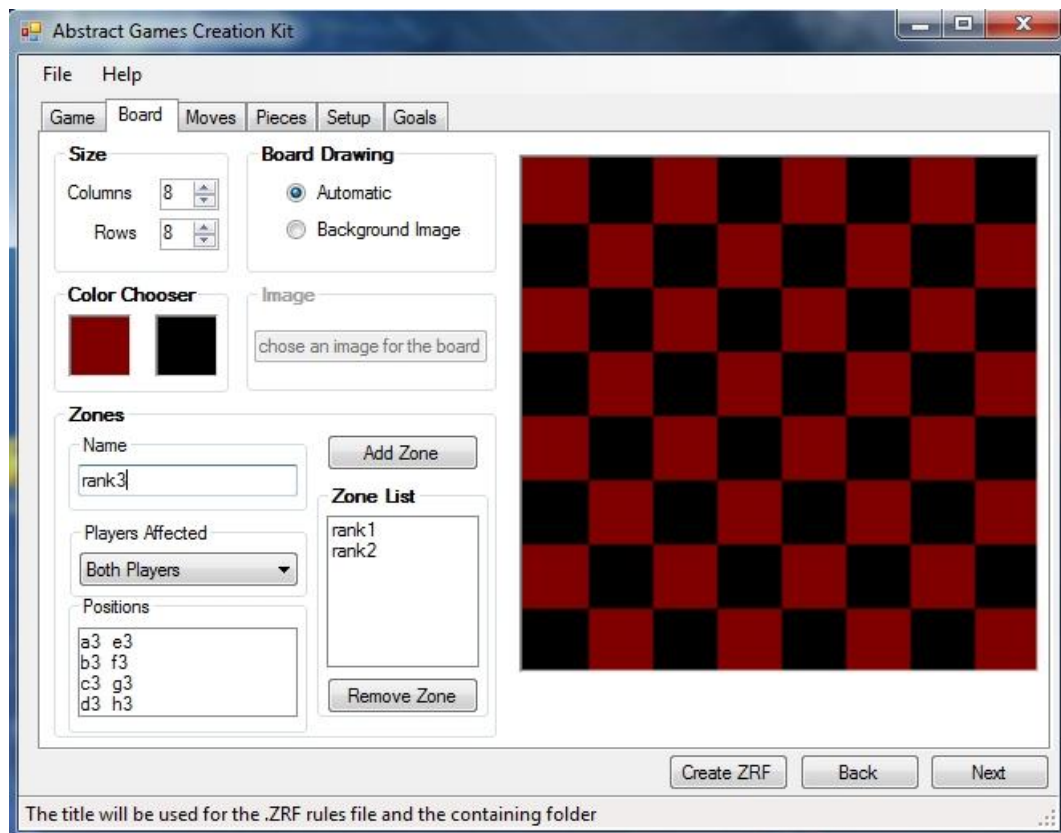


Figure B.2: Board Tab Prototype Screenshot

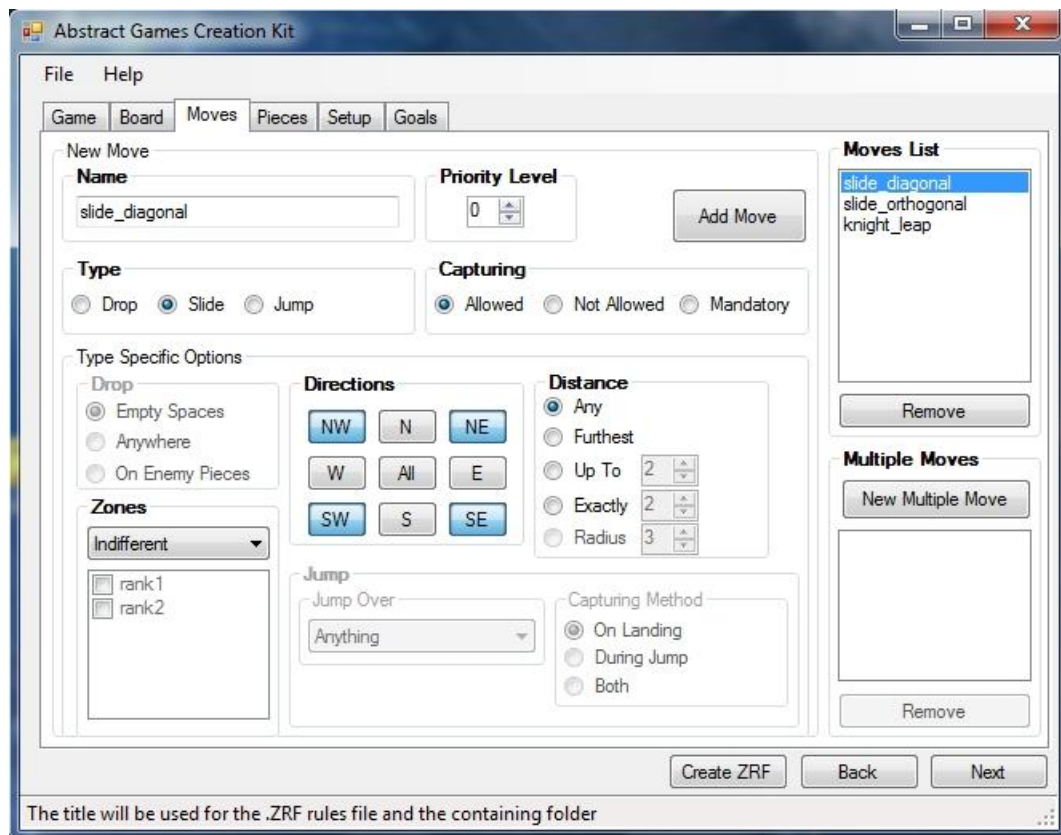


Figure B.3: Moves Tab Prototype Screenshot



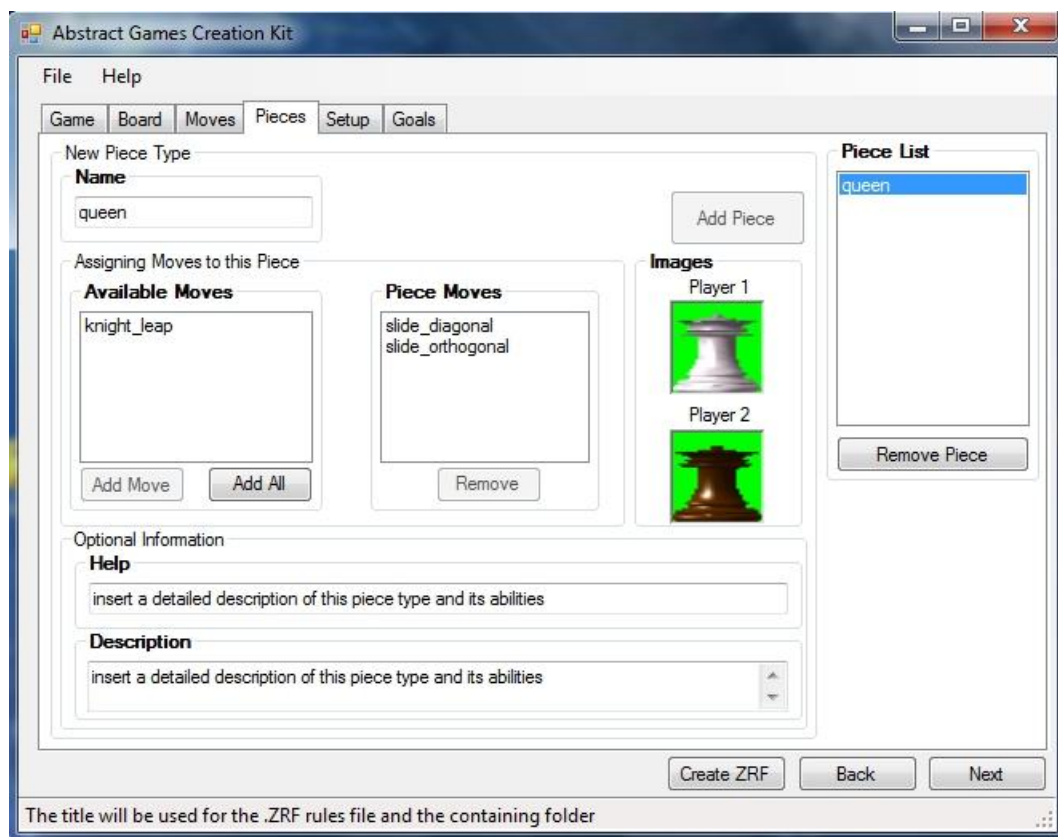


Figure B.4: Pieces Tab Prototype Screenshot

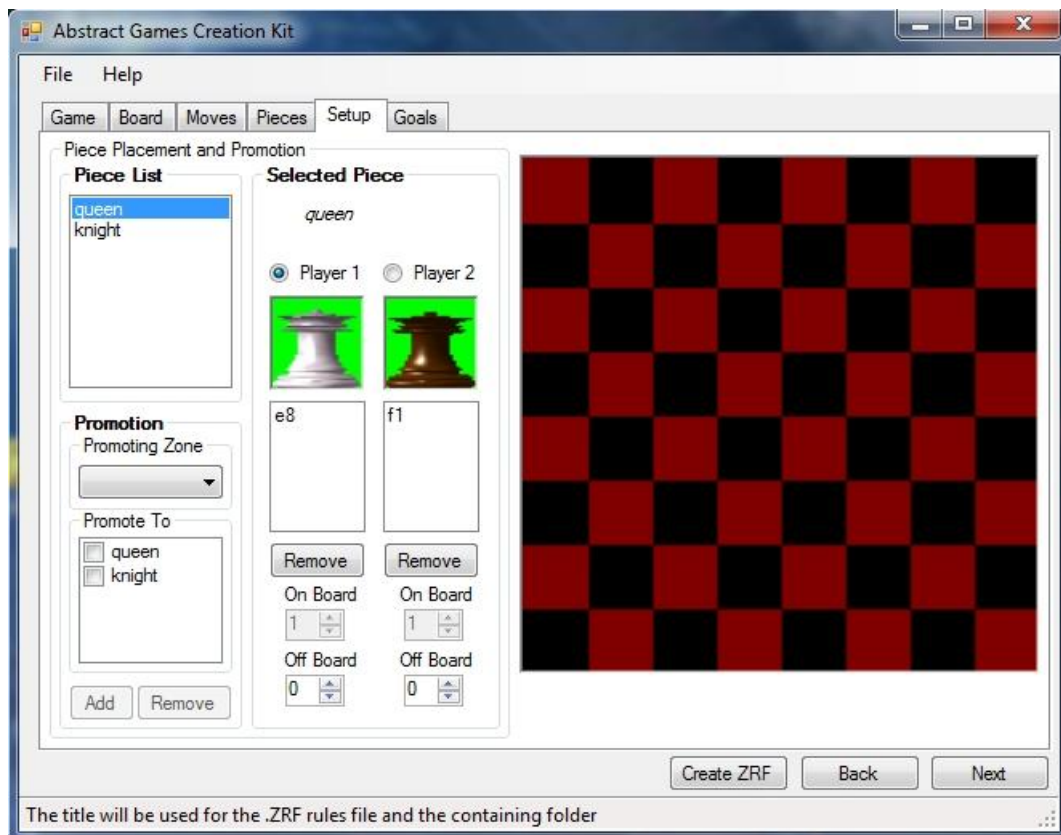


Figure B.5: Setup Tab Prototype Screenshot

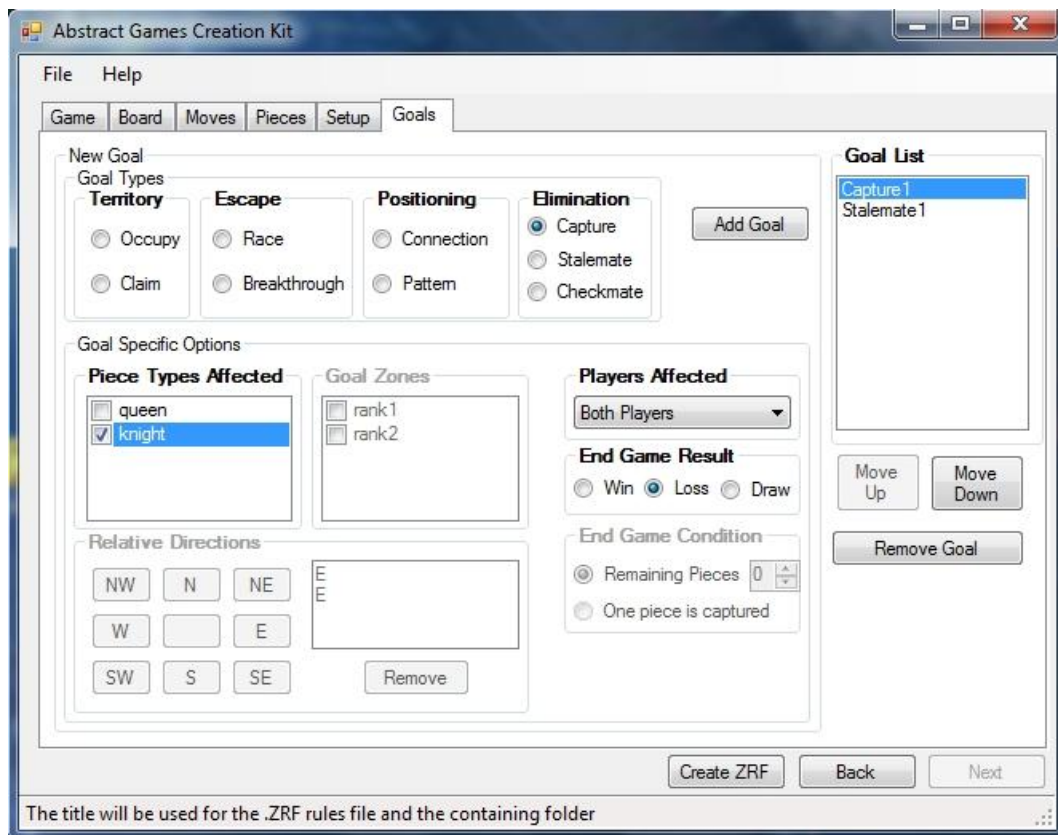


Figure B.6: Goals Tab Prototype Screenshot

## Appendix C

### Game Creation Example: Maze Game

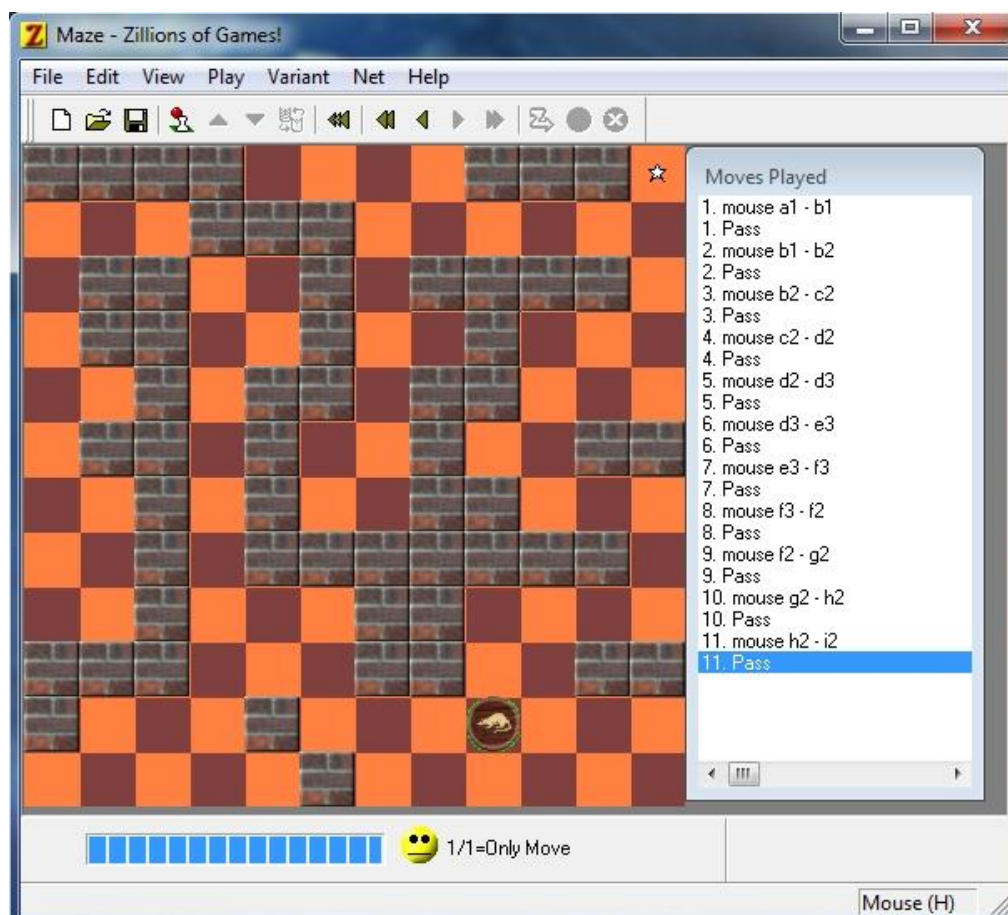


Figure C.1: Screenshot of the Maze Game running on Zillions

## Appendix D

### Examples of generated ZRF code

```
(define dropempty ((verify empty?) add))
(game
  (title "Tictactoe")
  (players cross circle)
  (turn-order cross circle)
  (board
    (image "c:\zillions\mytictactoe\3x3board.bmp")
    (grid
      (start-rectangle 0 0 133 133) ; top-left position
      (dimensions ; 3x3
        ("a/b/c" (133 0)) ; columns
        ("3/2/1" (0 133))) ; rows
      (directions (nw -1 -1) (n 0 -1) (ne 1 -1) (e 1 0) (se 1 1)
        (s 0 1) (sw -1 1) (w -1 0))))
  (piece
    (name stone)
    (image cross "c:\zillions\mytictactoe\stone_cross.bmp"
      circle "c:\zillions\mytictactoe\stone_circle.bmp")
    (drops
      (dropempty)))
  (board-setup
    (cross
      (stone off 5))
    (circle
      (stone off 5)))
  (win-condition(cross circle) (relative-config stone nw stone nw stone))
  (win-condition(cross circle) (relative-config stone n stone n stone))
  (win-condition(cross circle) (relative-config stone e stone e stone))
  (win-condition(cross circle) (relative-config stone ne stone ne stone))
)
```

Figure D.1: Generated ZRF code: Tic-Tac-Toe

```

(define Pawn-add
  (if (not-in-zone? promotion)
      add
      else
        (add Rook Bishop Queen knight)))
(define slide_orth ($1 (while empty? add $1) (verify not-friend?) add))
(define slide_diag ($1 (while empty? add $1) (verify not-friend?) add))
(define pawn_slide ($1 (verify empty?) add))
(define pawn_capture ($1 (verify enemy?) add))
(define pawn_first ((verify (in-zone? promotion)) (verify (in-zone? promotion)) $1 (verify empty?) $1 (verify empty?) add))
(define king_slide ($1 (verify not-friend?) add))
(define knight_leap ($1 $2 $2 (verify not-friend?)add)($1 $1 $2 (verify not-friend?)add))
(define Pawn-pawn_slide ($1 (verify empty?) (Pawn-add)))
(define Pawn-pawn_capture ($1 (verify enemy?) (Pawn-add)))
(define Pawn-pawn_first ((verify (in-zone? promotion)) (verify (in-zone? promotion)) $1 (verify empty?) $1 (verify empty?) (Pawn-add)))

(game
  (title "Chess")
  (players White Black)
  (turn-order White Black)
  (move-priorities priority-0)
  (board
    (image "c:\zillions\Chess\8x8board.bmp")
    (grid
      (start-rectangle 0 0 50 50) ; top-left position
      (dimensions ; 8x8
        ("a/b/c/d/e/f/g/h" (50 0)) ; columns
        ("8/7/6/5/4/3/2/1" (0 50))) ; rows
      (directions (nw -1 -1) (n 0 -1) (ne 1 -1) (e 1 0) (se 1 1)
        (s 0 1) (sw -1 1) (w -1 0)))
      (symmetry Black (n s) (s n) (e w) (w e) (nw se) (se nw) (ne sw)
        (sw ne))
      (zone
        (name promotion)
        (players White)
        (positions a8 b8 c8 d8 e8 f8 g8 h8))
      (zone
        (name promotion)
        (players Black)
        (positions a1 b1 c1 d1 e1 f1 g1 h1))
      (zone
        (name rank2)
        (players Black)
        (positions a7 b7 c7 d7 e7 f7 g7 h7)))

```

```

(zone
  (name rank2)
  (players White)
  (positions a2 b2 c2 d2 e2 f2 g2 h2)))
(piece
  (name Pawn)
  (image White "c:\zillions\Chess\Pawn_White.bmp"
    Black "c:\zillions\Chess\Pawn_Black.bmp")
  (moves
    (move-type priority-0)
    (Pawn-pawn_slide n)
    (Pawn-pawn_capture nw)
    (Pawn-pawn_capture ne)
    (Pawn-pawn_first n)))
(piece
  (name King)
  (image White "c:\zillions\Chess\King_White.bmp"
    Black "c:\zillions\Chess\King_Black.bmp")
  (moves
    (move-type priority-0)
    (king_slide nw)
    (king_slide n)
    (king_slide ne)
    (king_slide w)
    (king_slide e)
    (king_slide sw)
    (king_slide s)
    (king_slide se)))
(piece
  (name Rook)
  (image White "c:\zillions\Chess\Rook_White.bmp"
    Black "c:\zillions\Chess\Rook_Black.bmp")
  (moves
    (move-type priority-0)
    (slide_orth n)
    (slide_orth w)
    (slide_orth e)
    (slide_orth s)))

```

```

(piece
  (name Bishop)
  (image White "c:\zillions\Chess\Bishop_White.bmp"
    Black "c:\zillions\Chess\Bishop_Black.bmp")
  (moves
    (move-type priority-0)
    (slide_diag nw)
    (slide_diag ne)
    (slide_diag sw)
    (slide_diag se)))

(piece
  (name Queen)
  (image White "c:\zillions\Chess\Queen_White.bmp"
    Black "c:\zillions\Chess\Queen_Black.bmp")
  (moves
    (move-type priority-0)
    (slide_diag nw)
    (slide_diag ne)
    (slide_diag sw)
    (slide_diag se)
    (slide_orth n)
    (slide_orth w)
    (slide_orth e)
    (slide_orth s)))

(piece
  (name knight)
  (image White "c:\zillions\Chess\knight_White.bmp"
    Black "c:\zillions\Chess\knight_Black.bmp")
  (moves
    (move-type priority-0)
    (knight_leap n e)
    (knight_leap s e)
    (knight_leap s w)
    (knight_leap n w)))

(board-setup
  (White
    (Pawn a2 b2 c2 d2 e2 f2 g2 h2)
    (King d1)
    (Rook a1 h1)
    (Bishop c1 f1)
    (Queen e1)
    (knight b1 g1))

```



```
        (Black
          (Pawn a7 b7 c7 d7 e7 f7 g7 h7)
          (King d8)
          (Rook a8 h8)
          (Bishop c8 f8)
          (Queen e8)
          (knight b8 g8)))
    (loss-condition(White Black) (checkmated King))
)
```

**Figure D.2: Generated ZRF code: Chess**