**Software Design  
Document**

for

Ferret Army Chess

Version 1.0 approved

Prepared by A. Maxwell

Ferret Army

February 2018

# 

# Table of Contents

Table of Contents i

Revisions i

1. Introduction 1

1.1 Purpose 1

1.2 System Overview 1

1.3 Definitions, Acronyms and Abbreviations 4

1.4 Supporting Materials 5

1.5 Document Overview 6

2. Architecture 6

2.1 Overview 6

2.2 User Interface (UI) 7

2.3 Game Engine (GE) 9

2.4 Artificial Intelligence Engine (AIE) 10

3. High-Level Design 10

3.1 Component Overview / Game Engine (GE) 10

3.2 User Interface View / Game Board (GB) 11

3.3 User Interface View / Game Piece (GP) 12

3.4 Overview / AIE Random Generator #1 13

3.5 Overview / AIE Random Generator #2 14

3.6 Overview / AIE Random Generator #3 15

3.7 Overview / Stalemate Resolution (Reinfeld Values) 15

3.8 Overview / Piece Interface 16

3.9 Overview / GameBoard & Tile Classes 18

3.10 Overview / Player Interface 19

3.11 Overview / AIE Class 21

3.12 Generating Move Example / AIE 23

# Revisions

|  |  |  |  |
| --- | --- | --- | --- |
| Version | Primary Author(s) | Description of Version | Date Completed |
| 1.0 | Alexander Maxwell | Completion of *section 1* including sub sections 1.1, 1.2, 1.3, 1.4, and 1.5. | 02/19/2018 |
| 1.1 | Alexander Maxwell | Completion of *section 2* including sub sections 2.1, 2.2, 2.3 and 2.4 | 03/04/2018 |
| 1.2 | Alexander Maxwell | Completion of *section 3* including sub sections  3.1, 3.2, 3.3, 3.4, 3.5, 3.6, 3.7, 3.8, 3.9, 3.10, 3.11, and 3.12 | 3/07/2018 |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

# Introduction

## Purpose

The purpose of this Software Design Document (SDD) is to provide a detailed description of the implementation of the "Ferret Army Chess (FAC)" software. It illustrates the design patterns, components, and units involved with creating an interactive version of chess called FAC. This document was generated according to the Software Requirement Specification (SRS) document *version 2.5* agreed upon with the client. The SRS document can be found included with the final deliverables for this project.

## System Overview

The FAC software is broken into three key components, *game engine* (GE), *user interface* (UI), and *artificial intelligence engine* (AIE). These components interact throughout the course of a chess game. The design of FAC largely revolves around the definitions, design, and implementation of GE, UI, and AIE. The architecture of the GE, UI, and AIE will be discussed later in *section 2.*

In accordance with the SRS the FAC software shall have no network functionality. The software shall run locally on desktop / mobile computers with either Windows or macOS operating systems. Due to the locality of the FAC software there will be no external interaction with other systems or actors beyond the *user*. A detailed description of the requirements can be seen in the SRS and will be discussed as required in later sections of this document. The general functional requirements can be seen, in bulleted form, below:

* FAC software shall provide users with three game modes including computer vs. computer, user vs. computer, user vs. user prior to gameplay.
* System shall implement a user interface (UI) allowing the user(s) to select game mode and settings.
* FAC software shall include functionality for limited artificial intelligence engine (AIE). Using randomization, the FAC software’s AIE shall move game pieces (GP) around the game board (GB).
* In addition to the standard chess moves the FAC software shall allow for three special moves including:
  + En passant – Special pawn capture move.
  + Pawn Promotion – From a player’s pawn to any other available game piece (GP). An available GP is defined as a GP previously captured by an opponent.
  + Castling – Both Queenside and Kingside.

These special moves shall be available to players only when specific conditions are met. The detailed description of each special move and the specific conditions to be met are covered in-depth in section 4.

* FAC Software shall provide functionality for user(s) to be able to move any game piece (GP) according to that GP’s specific game move (GM) attribute. These moves are detailed in-depth in TABLE 1.

TABLE 1.

*Game Pieces* and their associated *Game Moves*

|  |  |  |
| --- | --- | --- |
| **Game Piece** | **Game Move** | **Capture** |
| Pawn | **Forward 1 space**  **Forward 2 spaces** (Starting move only) – Movement cannot cause collision with another piece.  If the option for pawn promotion is chosen, then player can choose the piece the pawn will be promoted to after it reaches the last row of the opposing players side.  **Restrictions:**  Movement cannot extend past the edge of the game. | Left Diagonal 1 space  Right Diagonal 1 space  **Special case:**  En passant Capture – left or right diagonal 1 space  (See specifics in section 4.5) |
| Rook | **Forward 1-7 spaces**  **Backward 1-7 spaces**  **Left 1-7 spaces**  **Right 1-7 spaces**  **Restrictions:**  Movement is unrestricted until another game piece is encountered, or edge of game board is reached.  **Special case:**  Simultaneous movement with King is allowed for castling.  (See specifics detailed in section 4.7) | Same as game move until an opponent’s piece is captured. |
| Bishop | **Diagonal 1-7 spaces** on the game pieces color of origin  **Restrictions:**  Movement is unrestricted until another game piece is encountered, or end of game board is reached. | Same as game move until an opponent’s piece is captured. |
| Queen | **Diagonally 1-7 spaces**  **Vertically 1-7 spaces**  **Horizontally 1-7 spaces**  **Restrictions:**  Movement is unrestricted until another game piece is encountered, or end of game board is reached. | Same as game move until an opponent’s piece is captured. |
| King | **Diagonally 1 space**  **Vertically 1 space**  **Horizontally 1 space**  **Restrictions:**  Cannot move into a position that will place it within 1 space of the opponents King.  Cannot move into a position that will place it in check.  Movement cannot exceed the perimeter of the board.  **Special case:**  Castling will allow movement greater than 1 space along the 1st rank.  (See specifics in section 4.7) | Same as Game Move until an opponent’s piece is captured. Must not be in check when Capture completed. |

* The GB shall include game coordinates, so a user can submit moves using a coordinate on the GB. This will be represented as numbers for the rank (horizontal coordinates) and letters for the file (vertical coordinates).
* Users shall have the option to quit an ongoing game at any time. It is not necessary for both players to agree before an individual player quits a game.
* FAC software shall provider user(s) with the ability to enable a timer to control the flow of the game. The time shall have functionality to support the following operations:
  + Game Time Limit – Set duration for the length of a single game.
    - Winner will be determined by the total points accumulated from the capture of the opposing teams game pieces. See section 4.8 for more on stalemate resolution.
  + Turn Time Limit – Set a duration for the length of a single turn.

## Definitions, Acronyms and Abbreviations

TABLE 2.

Definitions, Acronyms, and Abbreviations

|  |  |
| --- | --- |
| **Term** | **Definition** |
| AIE | Refers to artificial intelligence engine that makes moves for the computer. |
| Bystander | A user who is observing a computer vs. computer game without making a game move. |
| Capture | The act of a player removing another player’s game piece by replacing their opponent’s game piece with the attacking game piece thus Capturing said game piece. |
| Check | Refers to a game move where a player’s King is under attack from another player whether a user or the computer. |
| Checkmate | Refers to a game move where a player’s King has no remaining moves where said game piece is not under attack from another player whether a user or the computer. |
| Computer | The system that represents the artificial intelligence of which a user can compete against. |
| FAC | Refers to Ferret Army Chess the software under development. |
| File | The columns of the chessboard that run vertically and are referred to by letters. |
| GB | Refers to the Game Board which is comprised of an 8 square by 8 square board with alternating colors which total 64 possible squares a game piece may occupy. |
| GE | Refers to the game engine which is collectively the code that runs the game pieces, game moves, and Game Board. |
| GM | Refers to game move which is the act of moving a game piece on the Board. |
| GP | Refers to any of the game piece(s) which may be a Pawn, Rook, Bishop, Knight, Queen, or King |
| GPA | Refers to the game piece array that hold the active state of each piece in a FAC game. |
| Major Piece | Refers to specifically to the queen or rook game pieces. |
| Minor Piece | Refers to specifically to the bishop or knight game pieces. |
| Player | A user who has initiated a game against either another user or the computer. |
| Rank | The rows that go from side to side across the chessboard and are referred to by numbers. |
| RF Value | The numeric Reinfeld Value assigned to each game piece the values are as follows: Pawn (1), Bishop (3), Knight (3), Rook (5), Queen (9), King(1,000,000) |
| SDD | Software Design Document |
| TP | Refers to the Test Plan used to test the functionality of FAC. |
| UI | Refers to interface by which the user interacts with the FAC software. |
| User | A person interacting with the FAC software. |

## Supporting Materials

<Note any references or related materials here.

## Document Overview

This Software Design Document (SDD) is divided into three sections with various subsections. The major sections of this SDD are as follows:

1. Introduction – Structure of the SDD.
2. Architecture – In-depth look at the structure of the FAC software.
3. High Level Design – Overview of each component.

In *section 1. – Introduction*, the purpose, methodology, and reason for this Software Design Document is introduced. In *section 2. – Architecture,* the system design, component design, sub-component design, and class design is discussed in-depth. In *section 3. – High Level Design*, an overview of the structural and functional decomposition of the systems components is discussed. In addition, the general interaction between the system’s components and the *user* is discussed at an abstract level.

# Architecture

## Overview

As previously mentioned in section 1.2 the FAC software is broken down into three key components, the *game engine* (GE), user *interface* (UI), and artificial *intelligence engine* (AIE). Each of these primary components can be further divided into several sub-components. An overview of the FAC software at a component/sub-component level can be seen in figure 1.

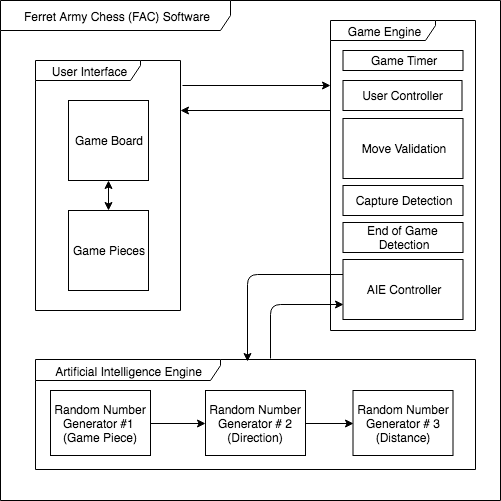


Figure 1. FAC Software Overview

The FAC software was decomposed into the GE, UI, and AIE for several reasons. In order to play a game of chess there are several key components including: *game board* (GB), *game pieces* (GP), *players* (2 *users*), the *computer*, and associated game winning mechanics. Of the key components *users* need only be aware of the GB and GP’s to play FAC. The remaining key components are the “under-the-hood” elements supporting the transaction between the GB, GP’s, and *user*.

Since FAC shall allow a *player* to be either a *human user* or a *computer user* the need arises for the FAC software to support some form of artificial intelligence when a *human user* is playing a game against a *computer user*. In an effort to support modularity it was beneficial to break the *user interface* (UI) elements from the game mechanics including the artificial intelligence engine thus leading to the UI, GE, and AI.

To support future improvements to the artificial intelligence of FAC the AIE was separated from the GE. Since it is highly unlikely that interactions between the UI and GE will change during maintenance the compartmentalization of the AIE leads to greater maintainability. When the AIE ability is improved there will be little refactoring involved since all interaction between the GE and the AIE run through the AIE controller. This means that changes to the AIE will almost be “plug-in-play” since all calls from the GE to the AIE will remain unchanged. This type of interaction leads to an adapter design pattern for the AIE and GE, which can be seen in figure 2. The interactions between the GE and AIE will be discussed in detailed later in this document.

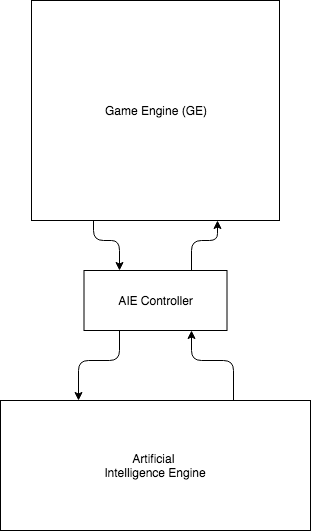


Figure 2. AIE Controller

## User Interface (UI)

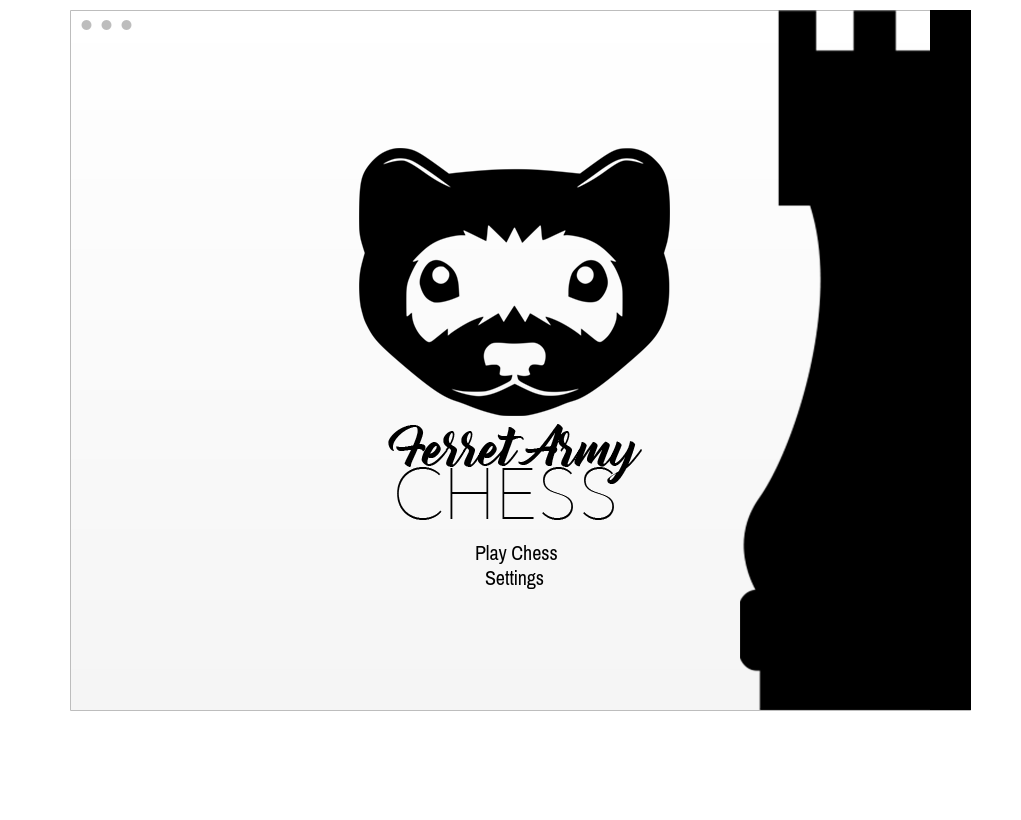


Figure 3. Landing Page

The *user interface* (UI) is broken into several key sub-components. As previously mentioned the two primary components are the *game board* (GB) and the *game pieces* (GP’s). Additionally, there are several other UI elements that encompass the FAC software. These include the:

* *landing page,* which is the first screen a user sees when launching the FAC software, see figure 3.

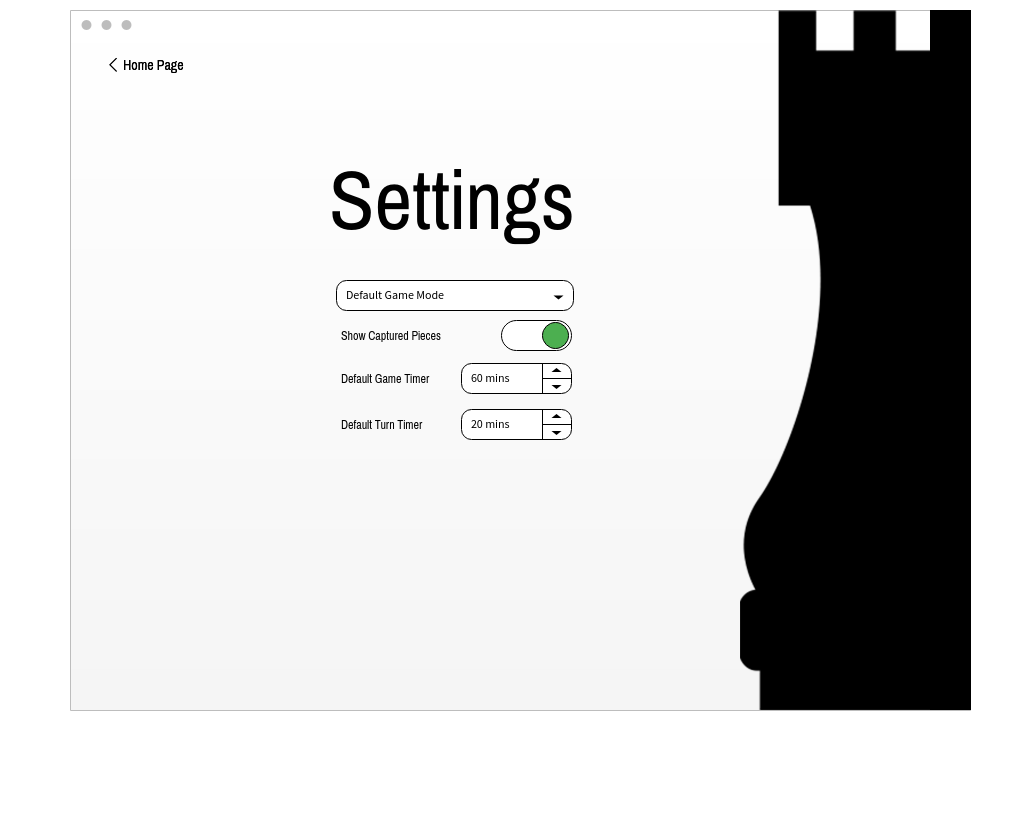


Figure 4. Settings Page

* *settings page* where FAC shall provide the *user* with the ability to change default values of the *game timer*, *turn timer*, enable/disable *captured piece tracking*, and choose a default *game mode*, see figure 4. The settings shall be saved between instances of the FAC software. This will be completed using a .xml, .text, or .json file.
* *game setup page* where FAC shall provide



Figure 5. Game Setup Page

*user(s)* with the ability to provide a *username* and override the default values of the *game timer*, *turn timer*, and *game mode*, see figure 5. The *game page* is also where a player starts a game thus causing the FAC software to initiate the GB, GP’s, *game timer, turn timer,* and *game page* encompasses the GB and all 32 GP’s as well as the *game timer*, *turn timer*,

and *captured pieces* count.

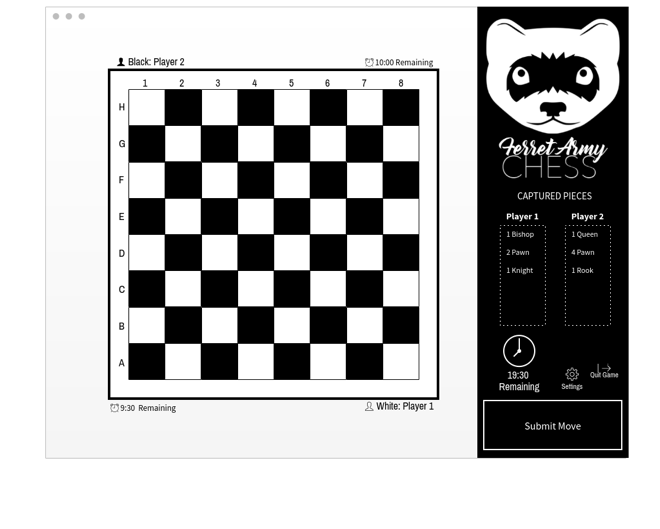


Figure 6. Game Page

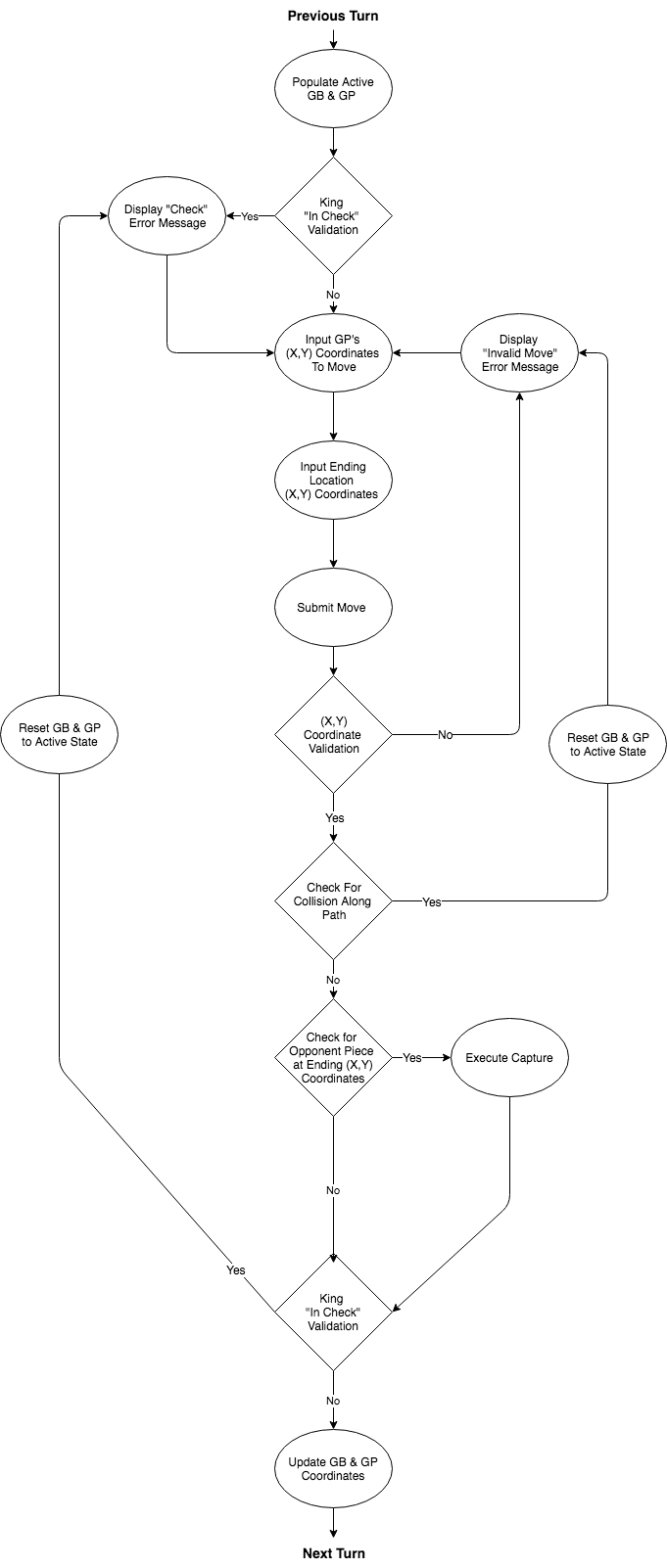
* *game page* is the primary page *users* see while playing FAC, see figure 6. The *game page* shall accurately show the current *game state*, position of the GP’s, allow *players* to submit a move, and notify *players* if a move is *invalid*, leads to *capture*, or leads to *check* or *checkmate*.

It shall also display a message when the game timer “runs outs” triggering a call to the GE to initiate the *stalemate* *resolution*. The *stalemate resolution* feature will be discussed in depth later in this document.

## Game Engine (GE)

The *game engine* (GE) is the most crucial component of the FAC software. It is responsible for handling a majority of the interactions between the UI, *users*, and where applicable the AIE (via the *AIE* *controller*). Additionally, the GE is the component that handles all chess logic associated with a *player* making a move, validating said move, checking for an *end game* state, and finally sending a message to the GB (via the UI) to reflect the change to the *player*. A standard interaction for the GE with a *player* can be seen in figure 7.

Figure 7. Moving Piece Use Case



As previously mentioned a primary function the GE serves beyond interfacing with the user through the UI is sending is controlling the AIE through the *AIE controller* which is the name given to an instance of the AIE during a game of FAC. The *AIE Controller* is responsible for querying the AIE for a potential *computer* related move when a *user* is acting as the role of a *player* in the *user vs. computer* game mode, or when a *user* is acting as the role of a *bystander* in the *computer vs. computer* game mode.

## Artificial Intelligence Engine (AIE)

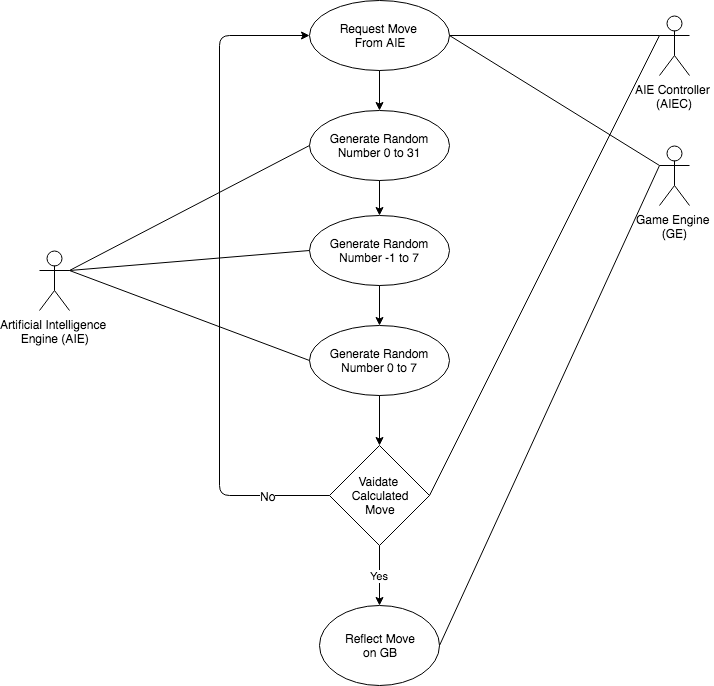


Figure 8. AIE Logic Use Case

The *artificial intelligence engine* (AIE) is arguably the second most crucial component of the FAC software. It provides all logic necessary to generate a move on the behalf of the *computer*. This shall be completed using 3 random number generators. The first random generator shall choose the piece, the second random number generator shall choose the direction the GP will attempt to move, and the third random generator shall choose the distance from the GP’s current position to the desired location.

Although not intelligent the AIE will seemingly appear “smart” as it will choose a piece, direction, and length of move each time it is called. Once the AIE generates a move the GE will validate the move. If the move is valid then the GE will reflect the move on the GB. If the move is not valid then the GE will query the AIE for a new move (this includes a new piece, direction, and distance). This process will repeat until a valid move is reflected on the GB. The process of generating the numbers to calculate the move will be discussed in-depth in section 3; however, an interaction with the AIE, AIE Controller, and GE can be seen in figure 8.

# High-Level Design

## Component Overview / Game Engine (GE)

The game engine is the brain of the FAC software. It acts as the program driver, performs all GP move validation, and handles much of the UI control. Due to the scope and breadth of the tasks the GE is responsible for the GE shall be robust, agile, and able to handle errors gracefully. This means that the GE needs to keep running regardless of errors or degradation in performance by the UI or AIE.

Since the GE is responsible for so many tasks it shall encompass ownership of a majority of the other classes used to implement the FAC software. In a standard game of chess (as with FAC) there are 2 *players*, 32 total *game pieces* (16 pawns, 4 rooks, 4 knights, 4 bishops, 2 queens, and 2 kings), 1 *game-board* with 64 individual *game tiles*. Just from the standard actors in a game of chess the classes: *Piece,* *GameBoard, Tile*, and *Player*. Additionally, since there are several variations of a “piece” the *Piece* shall be created as an interface that is implemented by each GP including, *Rook*, *Knight*, *Bishop*, *Pawn*, *Queen*, and *King*. The details of the *GameBoard*, *Tile*, *Piece* (and sub-piece) classes will be discussed in detail in later sections. An overview of the objects in FAC can be seen in figure 9. The objects shown in figure 9. are not an exhaustive list of the classes that shall be used but merely a guide when creating the GE. The UI will require many of its own classes to interface with the GE as the UI will continually be displaying the GB, 32 or less GP’s,

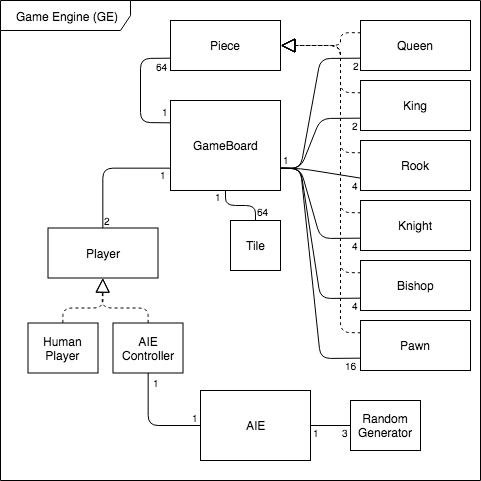


Figure 9. FAC Object Overview

and the playing window.

## User Interface View / Game Board (GB)

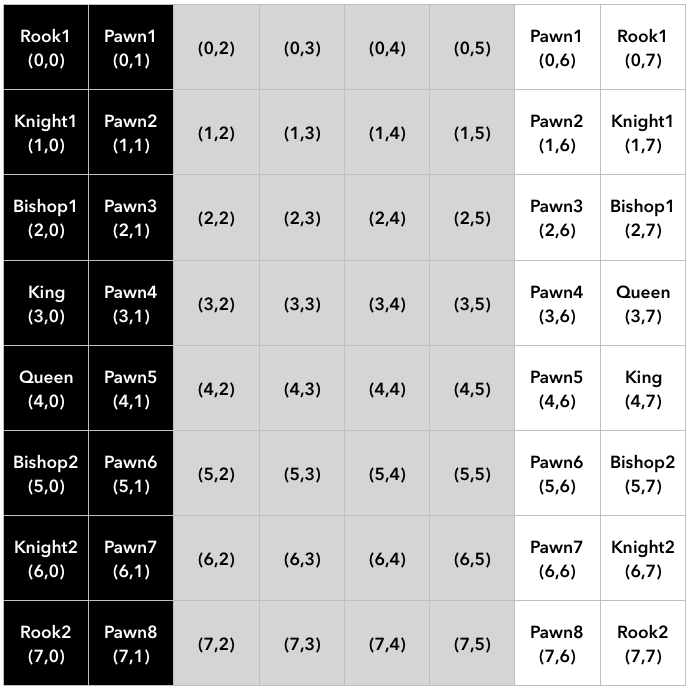


Figure 10. Game Board with Coordinate Listing

Like a traditional game of chess FAC shall implement an 8-by-8 *game board* (GB). The GB shall be colored with alternating colors with the *white player* starting on the bottom of the GB, and the *black player* starting on the top of the GB. To represent the GB a 2-dimensional array shall be used as it naturally facilitates (X,Y) coordinates, with both x-coordinate and y-coordinate being bounded from 0-7.

In an effort to ease the calculation of each GP’s offset the coordinate pattern shown in figure 10. shall be used for the 2-dimensional array.

The reason for aligning the black and white GP’s vertically instead of horizontally is to hold to the principle of “*keep the common case fast*”. Thus, calculating horizontal moves is a matter of subtraction or addition along a single row. Vertical moves are a matter of subtraction or addition along a single column. Diagonal moves require calculating offsets from the GP’s current position. The offsets for *vertical*, *horizontal*, and *diagonal* moves can be seen in figure 11. It is important to note that the offsets shown in figure 11. only apply if the coordinate labeling in figure 10. is implemented.

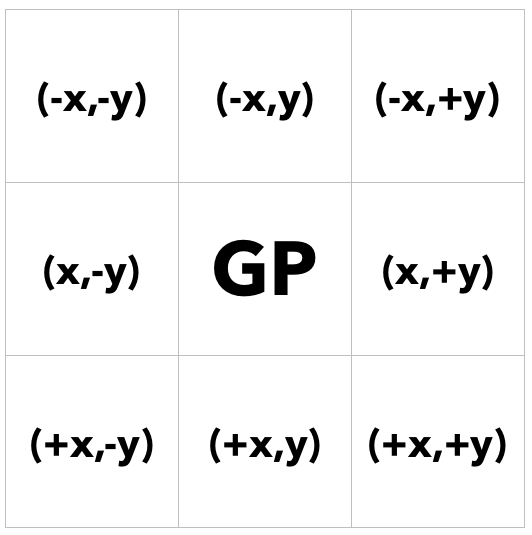
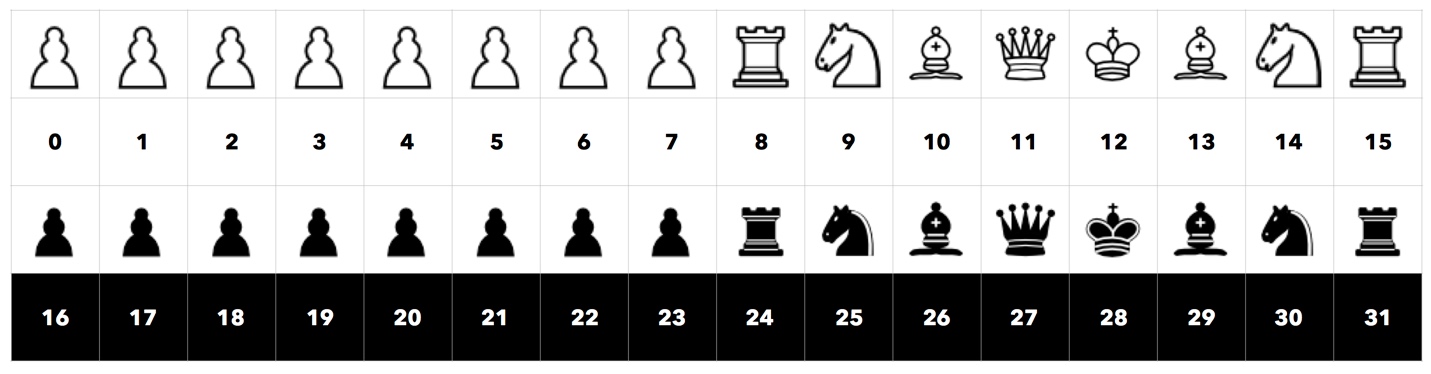


Figure 11. Offset Calculations

## User Interface View / Game Piece (GP)

In a game of FAC there are 32 individual *game pieces* (GP). With both playable colors (white and black) having 8-pawns, 2-rooks, 2-knights, 2-bishops, 1-queen, and 1-king for a total of 16 GP’s per *player.* GP’s may have one of two states, *active* or *non-active*. Since each space on the GB will have a valid corresponding (X,Y) coordinate to signify a capture an invalid coordinate of (-1,-1) will be assigned. The GE owns the list of *active* and *non-active* (captured) GP’s for both the *white player* and *black player.* This shall be implemented using a 1-dimensional array. The alignment of pieces, by index, in the array including color collation can be seen in figure 12. with *white player’s* pawns stored in indices 0-7 and *white player’s* value pieces stored in indices 8-15, *black player’s* pawns in 16-23 and *black player’s* value pieces stored in indices 24-31.

Figure 12. Game Piece Array



This 1-dimensional array will also be called upon heavily when the AIE randomly selects a GP to move. The logic of the AIE and its use of the *game piece array* (GPA)will be discussed in-depth in the next section.

## Overview / AIE Random Generator #1

As previously mentioned the AIE is made up of 3 random number generators. Random generator #1 will generate a number between 0-31 corresponding to the 1-dimensional array outlined in section 3.2. When it’s the *white player’s* random generator #1 shall be seeded with a number between 0-15 as shown in figure 13. When it’s the *black player’s* turn random generator #1 shall be seeded with a number between 16-31 as shown in figure 14.

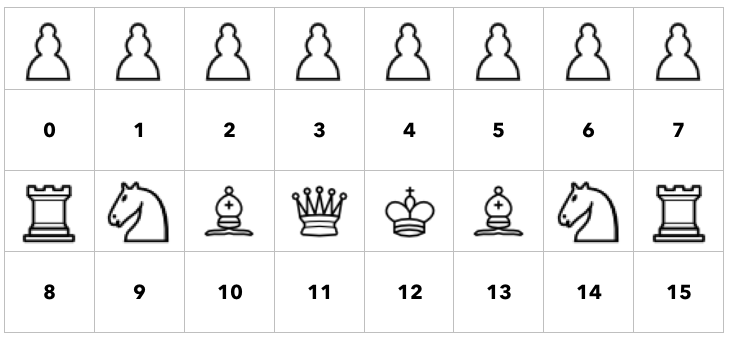
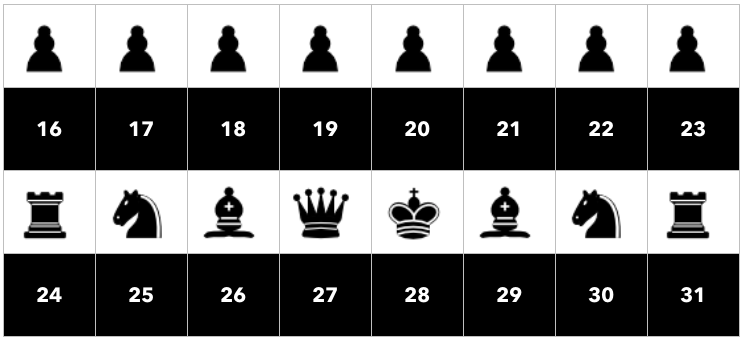


Figure 13. Game Piece Array (White Only)

After a random number is generated 0-15 or 16-31 the piece at that location shall have its active coordinates checked to determine if they are (-1,-1) or another value. As described if the active coordinates for the selected GP in the *game piece array* are (-1,-1) then that piece has already been captured during a previous turn. In this case a new random number is generated until an active GP is found. If it is white’s turn then a number between 0-15 will be generated. If it is black’s turn then a number between 16-31 will be generated. In order to signify a captured piece the (X,Y) coordinate (-1, -1) will be used. Every time a number 0-15 or 16-31 is generated the AIE will check the selected GP’s current coordinate to determine if that piece is captured or active.

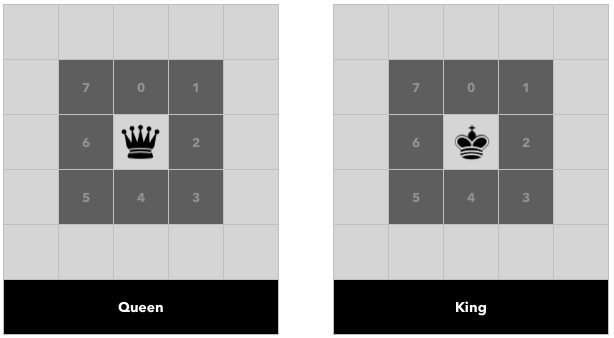
Figure 14. Game Piece Array (Black Only)



## Overview / AIE Random Generator #2

The second random number generator shall generate a number from 0-7 representing the direction the GP will move from its current location. In order to generate an (X,Y) coordinate for the chosen direction a numeric system shall be assigned to each unique GP’s available direction of movement. Since a queen and king have the most available directions of movement: *forward*, *backward,* *left*, *right*, *diagonally-left-upper, diagonally-right-upper, diagonally-left-lower, diagonally-right-lower* the numbers 0-7 will be used to correspond to the available directions starting vertically moving counter-clockwise. The numbering system for the queen and king can be seen in figure 15.

Figure 15. Queen & King Numbering System



For the remaining pieces they each have specific directions of movement. The bishops move along a diagonal path, so in the 0-7 numbering scale they can only move along an odd numbered path.



Figure 16. Bishop & Rook Numbering System

The rooks move along a horizontal or vertical path, so in the 0-7 numbering scale they can only move along an even numbered path. The numbering system for the bishop and rook can be seen in figure 16. There are a couple exceptions to the numbering scale. These include the knight and the pawn. The knight can move in exactly 7 directions; however, unlike the other pieces the knight will only make use of random number generator #1 and random number generator #2. For the pawn there are two different movement scenarios, pawns 1st move or pawns 2nd move. If the pawn is able to make its 1st move then it can move forward two spaces, otherwise it can only move forward one space. This means the pawn has to have two different numbering scales. The numbering scales for the knight and pawns these can be seen in figure 17.

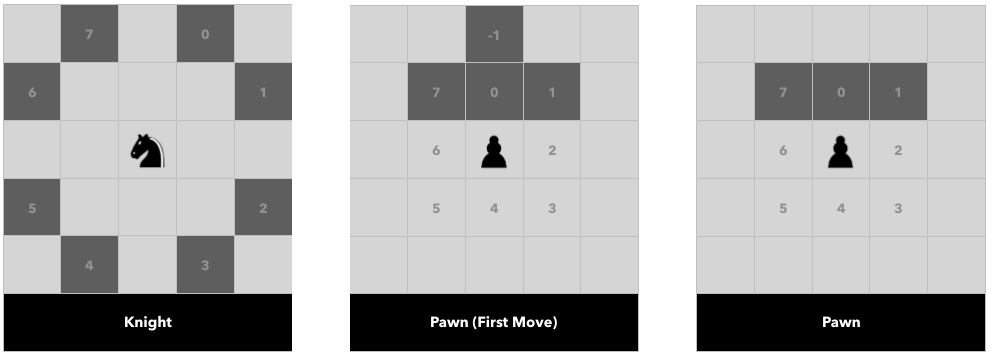


Figure 17. Knight & Pawn Numbering System

## Overview / AIE Random Generator #3

The third random number generator shall generate a number from -1 to 7 representing the distance a GP will move from its current position. For the various GP’s this will be different. The queen, rook, and bishop can move a maximum of 7-spaces along their supported paths. For knights they can move a maximum of 4-spaces along their supported paths; however, as mentioned their move logic is calculated using a 0-7 scale. For pawns the distance is limited to 1-space forward or 2-spaces (if it’s their first move) as outlined in section 3.4.

## Overview / Stalemate Resolution (Reinfeld Values)

A common occurrence in a game of chess (as in a game of FAC) is something called a stalemate where both active players cannot put the other player in checkmate for an extend period of time. This will happen quite often in FAC when the game timer runs out a *stalemate resolution* will be executed using Reinfeld Values (RF Values). Each *player* in a game of chess will be assigned a total RF Value of 39 points signifying they have all 16 of their pieces active. The math for how an RF Value of 39 points was reached and the RFValues assigned to each GP can be seen in figure 18. When a *stalemate* is reach the GE shall calculate each player’s RF Value and the *player* with the highest RF Value shall be determined the winner. This logically makes sense as the more important GP’s have a higher RF Value than less important GP’s. Meaning a *player* with a higher RF Value has lost important GP’s less often than their opponent. The only problem that could arise from this method of *stalemate* *resolution* is it lends to players keeping high RF Value pieces simply to win when a *stalemate* occurs.

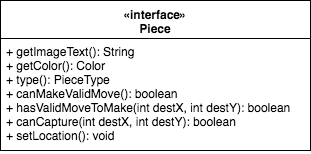


Figure 18. RF Value Point System

## Overview / Piece Interface

In section 3.2 the UI view of the GP was outlined.

Figure 19. Piece Interface Diagram



In this section a detailed class view of *Piece* will be outlined including its methods. In a standard game of chess all pieces share a few similar traits including: color, type (move validation, current (X,Y) coordinates, and current game. The *Piece* interface diagram can be seen in figure 19. The method descriptions for *Piece* can be seen below:

|  |  |
| --- | --- |
| **Piece.getImageText()** | |
| Purpose | Returns current piece’s color and type for UI |
| Prototype | public String getImageText() |
| Inputs | None |
| Outputs | The (String) representing the pieces name and type |

|  |  |
| --- | --- |
| **Piece.getColor()** | |
| Purpose | Returns the color of the current piece |
| Prototype | public Color getColor() |
| Inputs | None |
| Outputs | A Color object representing the color of the current piece |

|  |  |
| --- | --- |
| **Piece.type()** | |
| Purpose | Returns the type of this piece |
| Prototype | public PieceType type() |
| Inputs | None |
| Outputs | The type of the current piece PAWN, ROOK, KNIGHT, BISHOP, QUEEN, or KING. |

|  |  |
| --- | --- |
| **Piece.canMakeValidMove()** | |
| Purpose | Determine if piece has any valid moves to make |
| Prototype | public boolean canMakeValidMove(int x, int y) |
| Inputs | The (X,Y) coordinate of move destination |
| Outputs | TRUE if a valid move can be made. FALSE if no valid moves can be made. |

|  |  |
| --- | --- |
| **Piece.hasValidMoveToMake()** | |
| Purpose | Determine if moving piece puts king in check and execute move accordingly. |
| Prototype | public boolean hasValidMoveToMake() |
| Inputs | None |
| Outputs | TRUE if a valid move has been made. FALSE if no valid moves have been made. |

|  |  |
| --- | --- |
| **Piece.canCapture()** | |
| Purpose | Determine if piece can capture opponent piece after it moves to  destination (X,Y) coordinates |
| Prototype | public boolean canCapture(int destX, int destY) |
| Inputs | The (X,Y) coordinate of move destination |
| Outputs | TRUE if a valid move has been made. FALSE if no valid moves have been made. |

|  |  |
| --- | --- |
| **Piece.setLocation()** | |
| Purpose | Set the current location (X,Y) coordinates of piece on GB. |
| Prototype | public void setLocation(int destX, int destY) |
| Inputs | The (X,Y) coordinate of current location on the GB |
| Outputs | None |

## Overview / GameBoard & Tile Classes

Outlined first in section 3.1 and section 3.2 the *GameBoard* class is responsible for owning the *Tile* class instances that make up a modern chess board, generating the GB and keep tracking of the current state of the GB during a game. The *Gameboard* and *Tile* class diagram with relationships can be seen in figure 20. The method descriptions for *GameBoard* and *Tile* can be seen below:

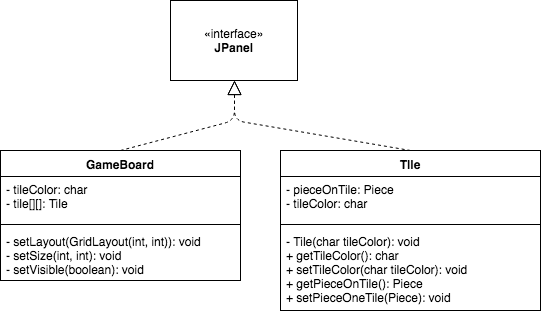


Figure 20. GameBoard & Tile Class Diagram

|  |  |
| --- | --- |
| **GameBoard.setLayout()** | |
| Purpose | Implementation of the JFrame class method in the JPanel library of the same name. Is used to set the LayoutManger for UI elements. |
| Prototype | public void setLayout(GridLayout GridLayout(int,int)) |
| Inputs | LayoutManger object from the JFrame class |
| Outputs | UI element framework correlating to a grid of 8 by 8 *Tile*s |

|  |  |
| --- | --- |
| **GameBoard.setSize()** | |
| Purpose | Implementation of the Dimension class method in the JPanel library of the same name. It is used to set the height and width of a JFrame |
| Prototype | public void setSize(int width,int height) |
| Inputs | Two (int) representing the width and height of the JFrame framework |
| Outputs | A JFrame framework of specified width and height |

|  |  |
| --- | --- |
| **GameBoard.setVisible()** | |
| Purpose | Implementation of the Component class method in the JPanel library of the same name. |
| Prototype | public void setVisible(boolean b) |
| Inputs | A single (boolean) representing whether a component should be visible withing a JFrame framework or not. |
| Outputs | Specific display criteria based on true (show) or false (hide) |

|  |  |
| --- | --- |
| **Tile.getTileColor()** | |
| Purpose | Get the current color of a tile object |
| Prototype | public char getTileColor() |
| Inputs | None |
| Outputs | Returns the contents of the private member variable (char) tileColor |

|  |  |
| --- | --- |
| **Tile.setTileColor()** | |
| Purpose | Set the current color of a tile object |
| Prototype | public void setTileColor(char tileColor) |
| Inputs | A (char) representing the color, which can be either “b” for black or “w” for white. |
| Outputs | None |

|  |  |
| --- | --- |
| **Tile.getPieceOnTile()** | |
| Purpose | Gives the Piece object currently resting on a Tile object |
| Prototype | public Piece getPieceOnTile() |
| Inputs | None |
| Outputs | Returns a copy of the current Piece object on the Tile object |

|  |  |
| --- | --- |
| **Tile.setPieceOnTile()** | |
| Purpose | Sets the Piece object to rest on a Tile object |
| Prototype | public void setPieceOnTile(Piece pieceReceived) |
| Inputs | A Piece object passed in and the current Piece object set on Tile object is set to the passed in Piece object. |
| Outputs | None |

## Overview / Player Interface

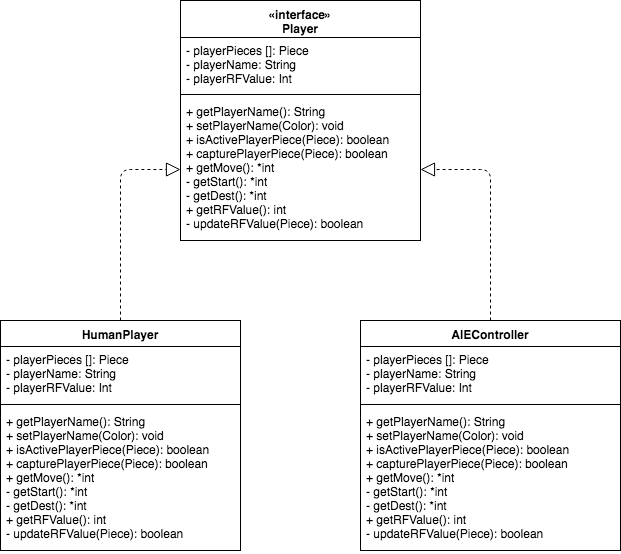


Figure 21. Player, HumanPlayer, AIEController Class Diagram

Since FAC is designed to support *human* vs. *human*, *human* vs. *computer*, and *computer* vs. *computer* the need for fluidness when components interact with a *player* is crucial. Since we are dealing with *human players* and *computer players* an interface for the *player* class shall be used. With the *player class* being an interface, it allows the creation of a *human class* and the *AIE Controller.* This means regardless of the type of player the GE can make the same set of calls when setting up a game, getting *player moves*, executing captures, and performing move validation. The class diagram for the *Player* interface and extended classes *HumanPlayer* and *AIEController* can be seen in figure 21. The method descriptions for the *Player* interface can be seen on the next page:

|  |  |
| --- | --- |
| **Player.getPLayerName()** | |
| Purpose | Get the contents of the private playerName variable. |
| Prototype | public String getPlayerName() |
| Inputs | None |
| Outputs | The (String) name assigned to this instance of the Player class. |

|  |  |
| --- | --- |
| **Player.setPLayerName()** | |
| Purpose | Set the contents of the private playerName variable to “Player 1” if color is “white” or “Player 2” if the color is “black”. |
| Prototype | public void setPlayerName(Color color) |
| Inputs | The name (String) to set a Player’s name to. |
| Outputs | None |

|  |  |
| --- | --- |
| **Player.isActivePiece()** | |
| Purpose | Checks the static 1-dimensional array storing all 32 pieces to see if the specified piece is active or captured. |
| Prototype | public boolean isActivePlayerPiece(Piece piece) |
| Inputs | The (Piece) to be checked. |
| Outputs | Return FALSE if the player’s piece is captured. Returns TRUE if the player’s piece is active still. |

|  |  |
| --- | --- |
| **Player.capturePiece()** | |
| Purpose | Locates the specified piece in the static 1-dimensional array storing all 32 pieces and changes its active coordinates to (-1, -1) marking it captured. |
| Prototype | public boolean capturePlayerPiece(Piece piece) |
| Inputs | The (Piece) to be captured. |
| Outputs | Return FALSE if the player’s piece was not captured. Returns TRUE if the player’s piece was captured. |

|  |  |
| --- | --- |
| **Player.getMove()** | |
| Purpose | Calls upon two private functions of the *Player* interface getStart() and getDest(). |
| Prototype | public \*int getMove() |
| Inputs | None |
| Outputs | Return a pointer to a 1-dimensional array containing the starting (X,Y) coordinates [0],[1] and destination (X,Y) coordinates [2], [3] for the move. |

|  |  |
| --- | --- |
| **Player.getStart()** | |
| Purpose | Private function to prompt a player for starting (X,Y) coordinates of the piece they want to move. |
| Prototype | private \*int getStart() |
| Inputs | None |
| Outputs | A (int) pointer to an array of size 2 representing the (X,Y) coordinates of the piece to move. |

|  |  |
| --- | --- |
| **Player.getDest()** | |
| Purpose | Private function to prompt a player for destination (X,Y) coordinates of the piece they want to move. |
| Prototype | private \*int getStart() |
| Inputs | None |
| Outputs | A (int) pointer to an array of size 2 representing the (X,Y) coordinates of the destination for the piece to move. |

|  |  |
| --- | --- |
| **Player.getRFValue()** | |
| Purpose | Get the contents of the private playerRFValue variable. |
| Prototype | private int getRFValue() |
| Inputs | None |
| Outputs | Return the current (Int) RF Value of the player. |

|  |  |
| --- | --- |
| **Player.updateRF()** | |
| Purpose | Private function to prompt a player for destination (X,Y) coordinates of the piece they want to move. |
| Prototype | private boolean updateRFValue(Piece piece.type()) |
| Inputs | The (Piece) type of the current piece PAWN, ROOK, KNIGHT, BISHOP, QUEEN, or KING being removed. |
| Outputs | Return FALSE if the RF Value was NOT updated. Return TRUE if the RF Value was successfully updated. |

It is important to note that the implementations for .getMove(), getStart(int, int), and getDest(int,int) for the *AIEController* class are significantly different than that of the *HumanPlayer* class. Since the AIE will be using three random number generators to calculate the selection of the piece, direction to move, and distance to move the AIE is not generating standard (X,Y) coordinates, so it shall be the job of the *AIEController* class to accomplish this. This process will be discussed in-depth in the next section.

## Overview / AIE Class

In sections 3.4, 3.5, and 3.6 the general concept and purpose of the AIE’s three random number generators were discussed. In this section the class overview and pseudo code for the AIE will be discussed in detail. As mentioned previously the AIE interacts with the GE via the *AIEController* class which implements the *Player* interface. The AIE acts as a compartmentalized class comprised primarily of the three number generators and associated methods to access the calculated values. The class diagram for the *AIE* class and its relationship to the *AIEController* class can be seen in figure 22. The method descriptions for the *AIE* class can be seen on the below:

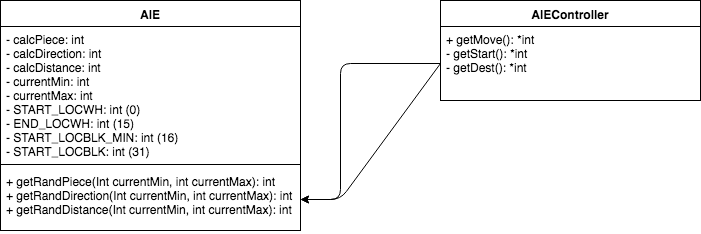


Figure 22. AIE Class Diagram & Relationship to AIEController Class

|  |  |
| --- | --- |
| **AIE.getRandPiece()** | |
| Purpose | Randomly generate a number between 0-31. 0-15 if current player is white. 16-31 if current player is black. |
| Prototype | private int getRandPiece(int currentMin, int currentMax) |
| Inputs | One (int) representing the minimum bound, and another (int) representing the maximum bound for the random generator. For white (min: 0, Max:15) and for black (min:16, max:31). |
| Outputs | Returns an (int) representing a number in the 1-dimensional array of player pieces discussed in section 3.4. |

|  |  |
| --- | --- |
| **AIE.getRandDirection()** | |
| Purpose | Randomly generate a number between min-max representing a direction for selected piece to move. |
| Prototype | private int getRandDirection(int currentMin, int currentMax) |
| Inputs | One (int) representing the minimum bound, and another (int) representing the maximum bound for the random generator. This varies based piece to piece. For exact bounds refer to section 3.5. |
| Outputs | Returns an (int) representing a direction (usually 0-7). |

|  |  |
| --- | --- |
| **AIE.getRandDistance()** | |
| Purpose | Randomly generate a number between min-max representing a distance for selected piece to move along a specific direction. |
| Prototype | private int getRandDistance(int currentMin, int currentMax) |
| Inputs | One (int) representing the minimum bound, and another (int) representing the maximum bound for the random generator. This is usually 0-7 since a chess board has a maximum of 8 spaces vertically or diagonally. |
| Outputs | Returns an (int) representing a distance (usually 0-7). |

The relationship between the AIE and AIE Controller comes via the *AIE* class methods .getRandPiece(), .getRandDirection(), and .getRandDistance(). These functions are called from within the *AIEController* class methods .getStart() and .getDest(). The pseudo code for .getStart() and .getDest() is shown on the next page:

**private \*int getStart(){**

1. *Determine if it’s WHITE’s turn or BLACK’s turn:*
   1. *If it’s WHITE’s turn then currentMin equals “0” and currentMax equals “15”*
   2. *If it’s BLACKS’s turn then currentMin equals “16” and currentMax equals “31”*
2. *make call to AIE.getRandPiece()*
3. *find piece in playerPieces array based on random integer found in 2.*
4. *check if piece in playerPieces array is active:*
   1. *If piece is active then proceed to 5.*
   2. *If piece is NOT active then go back to 2.*
5. *Return the (int) pointer to the array holding the starting coordinates.*

***}***

**private \*int getDest(){**

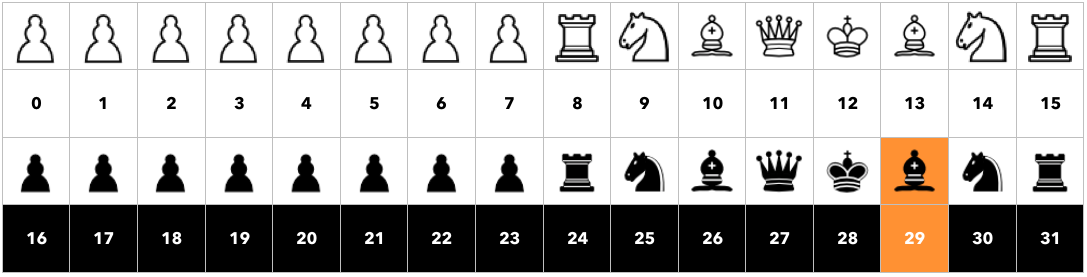
1. *Determine if the current piece (found in .getStart()) is a pawn in starting position or another piece:*
   1. *If the current piece is a pawn in starting position, then currentMin equals “-1” and currentMax equals “7”*
   2. *If the current piece is NOT a pawn in starting position, then currentMin equals “0” and currentMax equals “7”*
2. *make call to AIE.getRandDirection()*
3. *set currentMin equal to “0” and currentMax equal to “7”*
4. *make call to AIE.getRandDistance()*
5. *Using the offsets calculations (shown in figure 11.) that correlate to the direction (0-7) determine the destination coordinates using the distance integer found in .getRandDistnace subtracted/added to the offsets of the starting position coordinates of the piece found by .getStart().*
6. *Return the (int) pointer to the array holding the destination coordinates.*

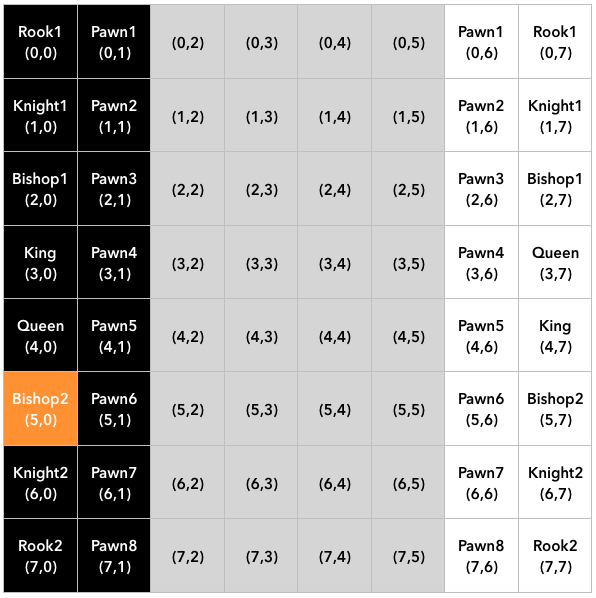
**}**

After the starting coordinates and destination coordinates are calculated they are returned to the GE (via an (int) pointer to a 4-integer array) the GE shall attempt to execute the move, which shall be validated by the selected piece.

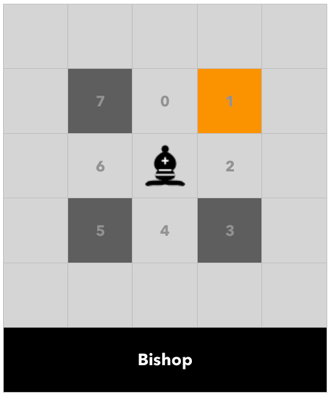
## Generating Move Example / AIE

Since much of the complexity of FAC comes from the functionality of the AIE this section aims to give a practical example of all the components in action using previously mentioned definitions, diagrams, and method functionality. The example of an interaction with the AIE, AIE Controller, and GE in plain English is shown on the next couple of pages:

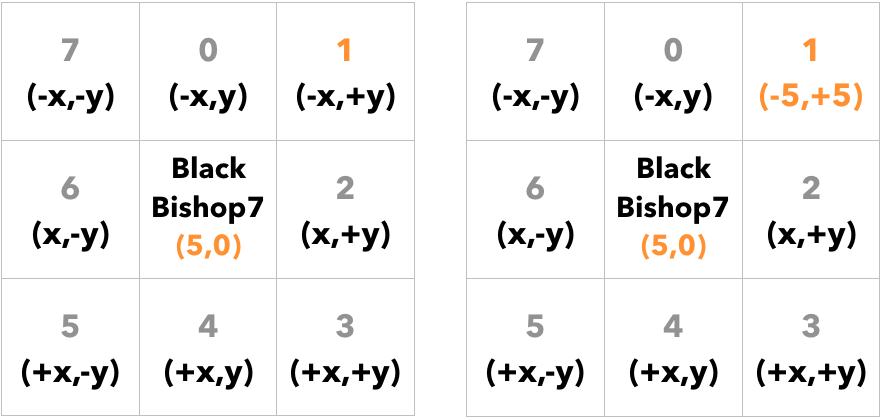
1. Assume it’s **black’s** turn. Black is a *computer player*.
2. GE makes a call to *AIEController.getMove()*, which in turn calls *AIEController.getStart()* and *AIEController.getDest().*
3. *AIEController.getStart()* follows the pseudo code outlined in section 3.11 and after a call to *AIE.getRandomPiece()* the number **29** is generated.
4. *AIEController.getStart()* looks up the index of the piece (29) in playerPieces[] array. An updated version of figure 12. with the selected piece highlighted is shown below:
5. The number **29** in the playerPieces[] array correlates to **Black Bishop2** whose current starting (X,Y) coordinates are **(5,0).** An updated version of the “*Gameboard with Coordinate Listing”* (originally shown in figure 10.) is shown below with the **starting coordinates** for **Black Bishop2** highlighted:



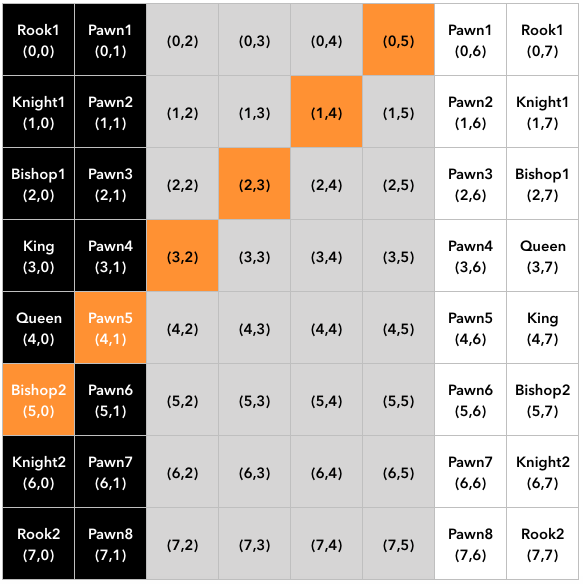
1. Earlier in 2. a call was made to *AIEController.getMove()* which in turn called *AIEController.getDest().*
2. *AIEController.getDest()* follows the pseudo code in section 3.11 and after calls to *AIE.getRandomDirection()* and *AIE.getRandomDistance()* the random numbers **1** and **5** are generated. This means on the direction numbering scale of 0-7 the piece will move in direction **1.** An updated versionof the “*Bishop Numbering System”* (originally shown in figure 16.) with the calculated **direction** highlighted is shown below:



This means on the distance numbering scale of 0-7 the piece will move a distance of **5**. An updated version of “*Offset Calculations*” (originally shown in figure 11.) is shown below with the **direction** and the updated **distance offsets** calculated:



1. If you take the starting (X,Y) coordinates of (5,0) and add the offsets found along direction **1** (-5,+5) you get the destination coordinates of (0, 5). An updated version of the “*Gameboard with Coordinate Listing”* (originally shown in figure 10.) is shown below with the **starting coordinates**, path, and **destination coordinates** for **Black Bishop2** highlighted:



1. The calculations done in step 8. are completed inside the *AIEController.getDest()* method which returns an (int) pointer to an array holding the destination coordinates.
2. The starting coordinates and destination coordinates are combined into a single array and the (int) pointer to that array is returned to the GE for move validation based on the selected piece. If validation fails, the process is repeated until a viable piece and move are found. After validation is successful then the GE updates the GB accordingly to reflect the new location of the GP that was moved.