Technical solution

```python
#Mutable Chess

#imports

import os #for loading rule sets from files
import pickle #for saving rule sets to files
import pygame #for playing and creating rule sets
import math #for calculating squaresize
import operator #used for repeated move calculation
import random #for Fishcer Random+
from copy import copy, deepcopy #for check and checkmate functions

import ctypes
ctypes.windll.user32.SetProcessDPIAware() #for getting the size of the user's monitor


```

Some computers have a screen size that extends beyond what is actually visible. In order to ensure that the board does not extend beyond the screen, we need to use this function to set the window. This ensures that we only gather height and width from the area on the screen where the DPI is ‘aware’ or where the mouse can move to.

```python
#colours

GREY = (200,200,200)
BLACK = (20,20,20)
GREEN = (50, 150, 50)
EMPTY = (0,0,0)

```

Colours of the squares used to build the board

The state of the board in my code is defined as a 2-dimensional array where each sub-array represents a row of the board. The number of items in each array will remain permanent throughout each game but the value of each item will be representative of what is on that square. The number 1 will indicate that the square is empty, so an empty board, with height and width 4, would be defined thusly:

```
Board = [ [1,1,1,1],
          [1,1,1,1],
          [1,1,1,1],
          [1,1,1,1], ]
```

Throughout my code I use two forms of notation to reference the location of pieces within this board.

The first is a 2-dimensional position vector where the first number represents the index of the sub-array representing the row that the piece is on and the second number represents the index of item in that array.

So, in general terms a position vector can be any element from the set:

$$\{ (a, b) \mid 0 < a < \text{board height}, 0 < b < \text{board width} \}$$

And a piece could be referenced by said position vector using the following syntax:

Piece = Board[a][b]

The second is a string made up of two characters. The first, a letter, refers to the column or rank of the square and the second, the number, refers to the row or file. This is standard algebraic chess notation.

I opted to use two methods instead of just one because the different formats are useful in different situations. Vector notation is more useful for mathematical operations, for example calculating moves. Algebraic notation is what is standardly used in chess, so it is easier for humans to understand. This is useful for the record game feature which prints every move that’s made and is also helpful to anyone working on the code during the debugging process.

As a result, I have defined some data structures and functions to support the manipulation of positions throughout my code.

```
#positions

#generate alphabet
a = ord('a')
alph = [chr(i) for i in range(a,a+26)]

#will contain letters as keys with their corresponding index positions in the alphabet as values
chess_map_from_alpha_to_index = {}

#will contain numbers as keys with their corresponding letter with at that index value in the alphabet as values
chess_map_from_index_to_alpha = {}

def algebra_to_vector(pos):
        column = pos[0]
        row = pos[1:]
        row = int(row) - 1
        column = chess_map_from_alpha_to_index[column]
        coords = [column, row]
        return coords

def vector_to_algebra(coords):
        i = chess_map_from_index_to_alpha[coords[0]]
        j = str(coords[1] + 1)
        pos = i+j
        return pos

```

First, I generate a list containing the whole alphabet. I have chosen to dynamically generate the list rather than specify every value because it takes up less space in the code and can easily be altered in the future to contain more than 26 values, allowing for boards with heights and widths greater than 26.

Secondly, I initialise two dictionaries which will be used to store the relationship between letters and their index in the alphabet. They will be populated when the user chooses a height and width. Again, this gives them the potential to extend beyond 26 in future.

Thirdly, I create two functions. These allow me to easily convert vector notation to algebra notation and vice versa.

The program uses two principle classes. The Gameboard class and the Piece class.

```
class GameBoard:

#Gameboard initialisation
    def __init__(self, agents, board, width, height, win_condition):
        self.agents = agents
        self.board = board
        self.w = width
        self.h = height
        self.wincon = win_condition
        self.create_notation()

    def create_notation(self):
        for i in range(0, self.w):
            #uses width because number of columns is equal to width and rows dont use letters
            chess_map_from_alpha_to_index[alph[i]] = i
            chess_map_from_index_to_alpha[i] = alph[i]

```

The Gameboard Class initialises with 4 variables: The initial setup of the board in the array format previously mentioned; the height and width of the board so that this board can be interpreted within the class and the win condition this board should be played with. The win condition is a pre-defined function.

After this we need to populate the chess map dictionaries from earlier. This is done by the create notation function.

```
#getter and setter for the board
    def getboard(self):
        return self.board

    def setboard(self, newboard):
        self.board = newboard

```

Used in board creation function later and potentially useful in the future. For example, if support for a board that changes throughout the game is added.

```python
#draw board
    def draw_board(self, squaresize, w, h):

        width = w * squaresize
        height = h * squaresize

        size = (width, height)
        screen = pygame.display.set_mode(size)

        i = 1

        colours = [GREY, BLACK]

        #An offset is needed to ensure that the squares of the board alternate in colour
        if h%2 == 0:
            offset = True
        else:
            offset = False

        #drawing squares
        for c in range(w):
            if offset:
                i = i%2 + 1
            for r in range(h):
                pygame.draw.rect(screen, colours[i-1], (c*squaresize, r*squaresize, squaresize, squaresize))
                i = i%2 + 1

        #drawing pieces onto squares
        for c in range(w):
            for r in range(h):
                if type(self.board[r][c]) == str:
                    image = pygame.image.load('images\{}.png'.format(pieces[self.board[r][c]][0]))
                    image = pygame.transform.scale(image, (round(0.9*squaresize), round(0.9*squaresize)))
                    screen.blit(image, (c*squaresize+round(squaresize*0.05), r*squaresize+round(squaresize*0.05)))

```

(See draw board flowchart)

Pieces is a dictionary defined later on. This line of code gets the string of filename of the piece being referenced and loads the image from that file in.

It is key that everything in the function is dependent on the dynamically changing square size. This means that no matter the size of the board all objects on it appear in the same ratio at the same location.

The next function plays the game.

```
    #play pre-amble
        def play(self):

                self.create_notation()

                pygame.init()

                end = False
                #game stops when this becomes true

                turn = 1
                #starts on turn 1

                touched = False
                #indicates whether a piece has been clicked on

                show_moves = False
                #indicates whether or not to display moves on the screen

                check = False
                #indicates whether or not a piece is in check

                if self.wincon == checkmate or self.wincon == KotH:
                        check_check = True
                else:
                        check_check = False

                #this variable is used as a parameter for the move generation function
                #it determines whether or not to factor in the laws of check when calculating moves


    #dynamic squaresize
                squaresize = 0

                mon_width, mon_height = pygame.display.Info().current_w, pygame.display.Info().current_h

                while True:
                    if squaresize * self.h < mon_height-100 and squaresize * self.w < mon_width-50:
                        squaresize+=1
                    else:
                        break

```

The square size is calculated dynamically for each board. The square size starts at 0 and the height and width of the user's monitor are found. Then the program checks if the square size multiplied by the height and width of the board exceeds the size of the monitor in either direction. If not, the square size is incremented, and this is tested again. If it does, the current square size is used. In added in an arbitrary buffer of 50 for the width and a semi-arbitrary buffer of 100 for the height so that the actual height or width isn't exceeded. The height buffer is slightly bigger so that the task bar isn’t covered, hence it being semi-arbitrary.

```
#record game
            ans = input('Record game?:\n')

            if ans.lower() == 'yes':
                log = True
            else:
                log = False
# play draw board
            while True:

                self.draw_board(squaresize, self.w, self.h)


```

The game loop officially starts. At the start of each loop the board is redrawn.

```
#win condition
                if not end: #once the game has ended, stops checking for win
                        if self.wincon == KotH:
                                end = self.wincon(self.board, self.w, self.h, turn, check, self.hills)
                                #the king of the hill win condition requires the hill squares to be passed as well as the normal variables
                        else:
                                end = self.wincon(self.board, self.w, self.h, turn, check)


```

The program checks if either player has won first.

```
#quit events
                for event in pygame.event.get():
                    if end == True:
                        if event.type == pygame.MOUSEBUTTONDOWN or event.type == pygame.KEYDOWN:
                            pygame.display.quit()
                            return

                    if event.type == pygame.QUIT:
                        pygame.display.quit()
                        return

                    if event.type == pygame.KEYDOWN:
                        if event.key == pygame.K_ESCAPE:
                            pygame.display.quit()
                            return

                    pygame.display.update()
```

```

```

There are three ways to exit the game loop: pressing the close window button or the escape key at any point in the game, or, once the game has ended, if the mouse is clicked or any button is pressed the window closes. This is so that the code doesn't abruptly close once a game has finished because reviewing the final board state in retrospect is an important part of strategy games.

```
      #show moves
                          if show_moves:
                          #if the last event was a piece being clicked, this statement displays all its legal moves
                              for i in legal_moves:
                                  coords = algebra_to_vector(i)
                                  screen.blit(image, (coords[0]*squaresize+round(squaresize/4), coords[1]*squaresize+round(squaresize/4)))
                                  #moves are represented by a red dot
                                  pygame.display.update()

```

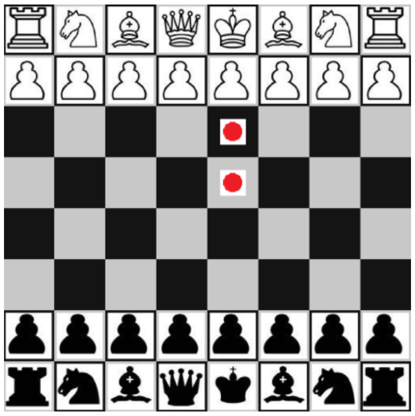
If there are moves to show, they are shown.

```
      #mouse clicks
                          if event.type == pygame.MOUSEBUTTONDOWN:
                              x, y = event.pos
                              column = int(math.floor(x/squaresize))
                              row = int(math.floor(y/squaresize))
                              #works out column and row based on x and y co-ordinate of mouse press
                              coords = column, row
                              pos = vector_to_algebra(coords)
                              #pos is the square thats been clicked on
                              choice = self.board[row][column]
                              #choice is the object on the square thats been clicked on

```

When the mouse clicks on the board there are two possible scenarios.

1: A piece has already been selected.

If this is the case the board will have had that piece's legal moves displayed. For example



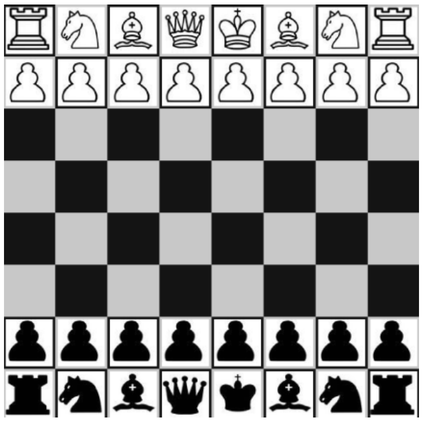
```
      #if a piece has been selected
                              if type(touched) == str:
                              #if a piece was touched last event...
                                  if pos in legal_moves:
                                  #and if a legal move for that piece was touched this event..
                                      #square that the piece was on is made empty
                                      self.board[last_row][last_column] = 1
                                      #contents of target square are recorded for the game log
                                      target = self.board[row][column]
                                      #target square is populated with chosen piece
                                      self.board[row][column] = touched
                                      legal_moves = []
                                      #a move is made


      #updates game info
                                      #promotes any unpormoted piece
                                      self.board = promotion_check(self.board, self.h, self.agents)
                                      #checks if a piece is in check
                                      check = checkifcheck(self.board, turn, self.w, self.h)
                                      #moves onto next turn
                                      turn = next_turn(turn)
                                      pygame.display.update()

                                      #if the user requested to record the game, logs the move that has just occurred
                                      if log:
                                          if target == 1:
                                              print(touched, pos)
                                          else:
                                              print(touched, 'x', pos)

```

```
              #lets go of the piece that was touched last event
              touched = False

          else:
              legal_moves = []
              #lets go of the piece that was touched last event
              touched = False
              pygame.display.update()

```

In this scenario the algorithm checks if one of the displayed legal moves has been clicked, and if it has then it makes that move. Then information about the game is updated.

If the square clicked isn't a legal move, then the piece is dropped, and the moves disappear.

This changes the state of the board to the second scenario possible when the mouse is clicked. In this case the board will look, for example, like this:



```
#if a piece hasn't been selected
      else:
          #if an empty square or wall is clicked, nothing happens
          if choice == 1 or choice == 4:
                  pass
          #checks if its that players turn
          elif choice.isupper() ^ turn-1:
                  #calculates that piece's moves
                  legal_moves = (pieces[choice])[1].getMoves(pos, self.board, turn, check_check, self.w, self.h)
                  if legal_moves:
                      touched = choice
                      #saves previous position for use in next loop
```

```
                                        last_column = column
                                        last_row = row
                                        #loads the image of the red dot for use in next loop
                                        image = pygame.image.load('images\\red_dot.png')
                                        image = pygame.transform.scale(image, (round(0.5*squaresize), round(0.5*squaresize)))
                                        width = self.w * squaresize
                                        height = self.h * squaresize
                                        size = (width, height)
                                        screen = pygame.display.set_mode(size)

                                        show_moves = True

                            #if an enemy piece is clicked
                            else:
                                print('enemy piece\n')


```

If an empty square or a wall is clicked, then nothing happens.

Then the 'elif' condition verifies that a player is clicking on their own piece. This works because white pieces are stored as an uppercase, one letter string and black pieces as a lowercase, one letter string. White moves on turn 1 and black on turn 2. Therefore, if a white piece is clicked on turn 1, the statement reads (True XOR 0) and if a black piece is clicked on turn 2, it reads (False XOR 1). If an enemy piece is clicked this is printed to the console to avoid turn confusion.

Next the Custom Game Board class is defined, a child class of the game board class.

```
#custom board class
class CustomGameBoard(GameBoard): #Class used to create user customised game boards

    #custom game board initialisation
    def __init__(self, agents, fairies, board, width, height, win_condition):
        #fairy pieces specific to this ruleset
        self.fairies = fairies
        self.hills = []
        #parent initialisation function
        super().__init__(agents, board, width, height, win_condition)
        #board deesign
        if board != defaultBoard:
            self.creation()

```

In the initialisation function, first it defines the fairy pieces specific to that ruleset and creates an empty array of hill squares which may be populated later. Second it runs the initialisation of its parent class. Thirdly, if the user did not select to use the default board it runs its board creation function to allow the player to design their own board.

```
    #getter for faries
        def getfairies(self):
            return self.fairies

    #getter and setter for hills
        def gethills(self):
            return self.hills

        def sethills(self, newhills):
            self.hills = newhills

    #custom draw board function
        #function to draw board with pygame graphics
        def draw_board(self, squaresize, w, h): #different squaresize for design and play
                                                #width and heigh must be specified becuase different values are used for design and play

            width = w * squaresize
            height = h * squaresize

            size = (width, height)
            screen = pygame.display.set_mode(size)

            i = 1

            colours = [GREY, BLACK]

            #An offset is needed to ensure that the squares of the board alternate in colour
            if h%2 == 0:
                offset = True
            else:
                offset = False

            for c in range(w):
                if offset:
                    i = i%2 + 1
                for r in range(h):
                    pygame.draw.rect(screen, colours[i-1], (c*squaresize, r*squaresize, squaresize, squaresize))
                    i = i%2 + 1

            for i in self.hills:
                pygame.draw.rect(screen, GREEN, (i[1]*squaresize+round(squaresize*0.1), i[0]*squaresize+round(squaresize*0.1), 0.8*squaresize, 0.8*squaresize))

```

```python
        for c in range(w):
            for r in range(h):
                if type(self.board[r][c]) == str:
                    image = pygame.image.load('images\{}.png'.format(pieces[self.board[r][c]][0]))
                    image = pygame.transform.scale(image, (round(0.9*squaresize), round(0.9*squaresize)))
                    screen.blit(image, (c*squaresize+round(squaresize*0.05), r*squaresize+round(squaresize*0.05)))
                if self.board[r][c] == 2:
                    pygame.draw.rect(screen, EMPTY, (c*squaresize, r*squaresize, squaresize, squaresize))
                if self.board[r][c] == 3:
                    image = pygame.image.load('images\\tick.png')
                    image = pygame.transform.scale(image, (round(0.9*squaresize), round(0.9*squaresize)))
                    screen.blit(image, (c*squaresize+round(squaresize*0.05), r*squaresize+round(squaresize*0.05)))
                if self.board[r][c] == 4:
                    image = pygame.image.load('images\wall.png')
                    image = pygame.transform.scale(image, (round(0.9*squaresize), round(0.9*squaresize)))
                    screen.blit(image, (c*squaresize+round(squaresize*0.05), r*squaresize+round(squaresize*0.05)))
                if self.board[r][c] == 5:
                    image = pygame.image.load('images\hill.png')
                    image = pygame.transform.scale(image, (round(0.9*squaresize), round(0.9*squaresize)))
                    screen.blit(image, (c*squaresize+round(squaresize*0.05), r*squaresize+round(squaresize*0.05)))
                    #this represents a hill marker which tells the creation function where hills should be placed

```

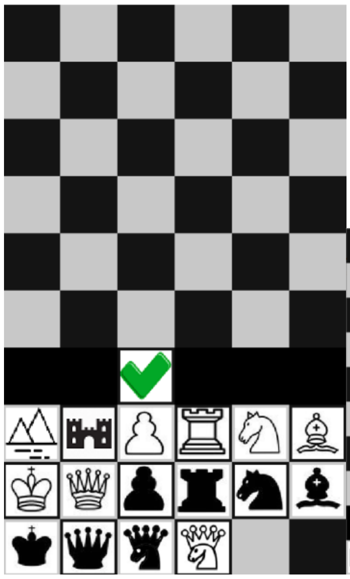
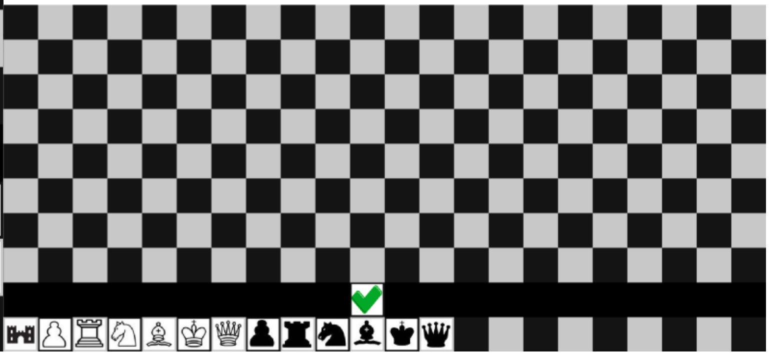
The custom game board class has a different draw board method. After drawing the empty board, it iterates though the array of hill squares and draws a green rectangle where each one should be. It also has cases for 3 more types of objects that can be drawn onto the board apart from pieces. This includes walls, which are inanimate objects that obstruct movement and give shape to the board; a hill-setting image, which the player uses to choose the location of hills in the creation function and the tick button, which is used to save a board state in the creation function.

```python
#function in which the user designs board
    def creation(self):

#creation board construction

        pygame.init()

        #function to work out extra height
        extra_h = extra_row_count(self.w, 1, self.agents)

        squaresize = 0

        mon_width, mon_height = pygame.display.Info().current_w, pygame.display.Info().current_h


        while True:
            if squaresize * (self.h+extra_h+1) < mon_height-100 and squaresize * self.w < mon_width-50:
                squaresize+=1
            else:
                break

```

The creation function interprets the board differently from the play function. I originally intended to have two windows, one containing the board being designed and one containing the toolset being used to design it. However, this would require multiprocessing and I decided it wasn’t worth the time and complexity because it was purely aesthetic, and a similar look could be achieved in a simpler way.

What the program does instead is: display the board being designed, add on a separating layer containing the tick button underneath it and then add on another board segment containing one of each board object available to the user underneath that.

Here are some examples what it might look like:

I wanted to maintain the rectangular shape of the window, so instead of just adding on a list of pieces I use a function to work out what extra height is needed to add on an auxiliary section with the same width as the user specified board that can fit all the creative tools needed.

Then the square size is calculated in the same way as before but factoring in the new auxiliary board and the separating layer in between.

```
#Creation function pre-amble

        piece = False
        #whether a piece is held

        while True:


                playable = False
                #is the gamemode playable yet

                white = False
                #are there white pieces

                black = False
                #are there black pieces

                white_royal = False
                #are there royal white pieces

                black_royal = False
                #are there royal black pieces

                hill = False
                #are there any hill squares

```

A variable indicating which piece is held is initialised as False outside of the loop. Inside the loop all the variables containing information about the board are set to False.

```
#checks playability
                for y in range(self.h):
                    for x in range(self.w):
                        if type(self.board[y][x]) == str and self.board[y][x].isupper():
                            white = True
                            if pieces[self.board[y][x]][1].getStatus() == 'royal':
                                white_royal = True
                #checks board for white pieces

                for y in range(self.h):
                    for x in range(self.w):
                        if type(self.board[y][x]) == str and self.board[y][x].islower():
                            black = True
                            if pieces[self.board[y][x]][1].getStatus() == 'royal':
                                    black_royal = True
                #checks board for black pieces

                for y in range(self.h):
                    for x in range(self.w):
                        if self.board[y][x] == 5:
                                    hill = True
                #checks if the user has set any hills

                if self.wincon == extinction:
                    if white and black:
                        playable = True


                if self.wincon == checkmate or self.wincon == regicide:
                    if white_royal and black_royal:
                                playable = True

                if self.wincon == KotH:
                    if white_royal and black_royal and hill:
                                playable = True
                #checks if the game is playable, different for each win condition

                for y in range(self.h):
                    for x in range(self.w):
                        if type(self.board[y][x]) == str and pieces[self.board[y][x]][1].getStatus() == 'royal':
                                if checkifcheck(self.board[:self.h], 1, self.w, self.h) or checkifcheck(self.board[:self.h], 2, self.w, self.h):
                                        playable = False
                #if a piece starts in check the board state is not playable

```

Then at the start of every loop the program checks if the win condition is playable with the current board state.

```
    #event loop
                self.draw_board(squaresize, self.w, self.h+extra_h+1)


                for event in pygame.event.get():
                    if event.type == pygame.QUIT:
                        pygame.display.quit()
                        self.board = []
                        #if the user quits, the board isn't defined
                        return

                    pygame.display.update()

                if event.type == pygame.MOUSEBUTTONDOWN:
                    x, y = event.pos
                    column = int(math.floor(x/squaresize))
                    row = int(math.floor(y/squaresize))
                    coords = column, row
                    pos = vector_to_algebra(coords)

                    choice = board[row][column]

                    if choice in self.agents:
                    #if a board object has been chosen
                        if row > self.h:
                        #if the chosen object is below the separation line (in the creative tools)
                            piece = choice
                            #turns the held piece to that object
                        else:
                        #if its above
                            self.board[row][column] = 1
                            #removes the object

                    elif choice == 1:
                    #if an empty square is chosen
                        if piece and row < self.h:
                        #if a piece has been selected and the square chosen is on the board
                            self.board[row][column] = piece
                            #places piece there

                    elif choice == 2:
                        pass
                    #if an empty square is clicked, does nothing

                    elif choice == 3:
                    #if the tick button was pressed
                        if playable:
                            for y in range(self.h):
                                for x in range(self.w):
                                    if self.board[y][x] == 5:
                                        self.board[y][x] = 1
                                        self.hills.append((y, x))
                                        #iterates through board and adds any hills found to the hills array
                            self.board = self.board[:self.h]
                            #sets the game board to be just the board component of the creation board.
                            pygame.display.quit()
                            return

                        else:
                                print('not playable')


                Pygame.display.update

```

(see creation loop flowchart)

Next the program draws the board and waits for an event, either a quit or a mouse click. If an object on the board is chosen it works out which side of the separation it is on. If it's in the creative tools it selects that object. If it's an object on the board it removes it.

If an empty square on the board is chosen and a piece was selected last event, that piece is placed there.

If the tick button is pressed and the game is playable, the board is saved and ready to be played.

```python
#default board

defaultBoard = [['R', 'N', 'B', 'Q', 'K', 'B', 'N', 'R'],
              ['P', 'P', 'P', 'P', 'P', 'P', 'P', 'P'],
              [  1,   1,   1,   1,   1,   1,   1,   1],
              [  1,   1,   1,   1,   1,   1,   1,   1],
              [  1,   1,   1,   1,   1,   1,   1,   1],
              [  1,   1,   1,   1,   1,   1,   1,   1],
              ['p', 'p', 'p', 'p', 'p', 'p', 'p', 'p'],
              ['r', 'n', 'b', 'q', 'k', 'b', 'n', 'r']]

```

Pre-made board designed for conventional chess.

Next is the piece class.

```python
#piece class initialisation
class Piece:

    def __init__(self, letter, vectors_ride = None, vectors_leap = None):
        self.piece = letter
        #unique identifier
        self.vec_ride = vectors_ride
        self.vec_leap = vectors_leap

    def getStatus(self):
        return 'common'

```

The piece class is instantiated with a unique letter to identify it and movement vectors for riding and leaping. (see requirement 5)

A base piece’s getStatus method will always return common, meaning the laws of check don’t apply to it.

```python
#get moves

    def getMoves(self, pos, board, turn, check_check, w, h):
        coords = algebra_to_vector(pos)
        #piece's location
        column, row = coords
        #will continue to represent the piece's location
        moves = []

```

This method is used to calculate a piece’s moves. It makes use of two functions called leap move and ride move which are defined later.

```
#leaps
        if self.vec_leap:

            for v in self.vec_leap:

                operators = [operator.add, operator.sub]

                for f in operators:
                        for g in operators:
                                moves += leap_move(f, g, v[0], v[1], board, self.piece, column, row)
                                moves += leap_move(f, g, v[1], v[0], board, self.piece, column, row)

#rides

        if self.vec_ride:
            for v in self.vec_ride:

                operators = [operator.add, operator.sub]

                for f in operators:
                        for g in operators:
                                moves += ride_move(f, g, v[0], v[1], board, self.piece, column, row)
                                moves += ride_move(f, g, v[1], v[0], board, self.piece, column, row)

```

Every chess piece can use its movement vector in 8 directions. The method achieves this by applying both components of the vector with addition and subtraction. This means there are 2 ways to apply both so there are 4 different possible combinations. Then the vectors can be swapped doubling the possible directions hence 8 directions. This is done for leaps which apply the vector once and then for rides which apply it infinitely.

```
#Filter all negative values

        temp = [i for i in moves if i[0] >=0 and i[1] >=0]

        allPossibleMoves = []

        for i in temp:
                allPossibleMoves.append(vector_to_algebra(i))

```

In the move getting process, any vector that equates to a move beyond the edge of the board will raise an index error and trigger the except clause. However, this is only for the moves that extend beyond the right and the bottom of the board because their magnitude is greater than the length of the array they are used as indexes for. When a negative value is reached, it wraps around to the back of the list adding a move that shouldn't be possible. Therefore, this line of code is necessary to filter those values.

```
#check for check
        if check_check:
        #if the laws of check are being considered
            royal_moves = []

            for i in allPossibleMoves:
                newpos = algebra_to_vector(i)
                if not royal_attacked(ChessBoard.getboard(), self.piece, turn, newpos, coords, w, h):
                #filters all moves that result in a royal piece being attacked
                    royal_moves.append(i)

            allPossibleMoves = royal_moves

        return allPossibleMoves

```

When the laws of check apply it is necessary for each piece to factor them into their moves. This is because of discovered checks which would occur when a friendly piece, blocking an enemy piece from causing check, moves out of the way.

```
#Pawn Class
class Pawn(Piece):

        def getMoves(self, pos, board, turn, check_check, w, h):
                coords = algebra_to_vector(pos)
                j, i = coords
                column, row = coords
                moves = []

                #white pawn
                if self.piece.isupper():
                    try:
                        temp = board[i+1][column]
                        if temp == 1:   #checks if the square is empty
                            moves.append((column,i+1))
                    except:
                        pass

                    if row == 1:
                        temp = board[i+1][column]
                        temp2 = board[i+2][column]#checks if both squares are empty
                        if temp == 1 and temp2 == 1:
                            moves.append((column,i+2))

                    try:
```

```
            temp = board[i+1][j+1] #diagonal capture
            if temp.lower() == temp:
                moves.append((j+1,i+1))
        except:
            pass

        try:
            temp = board[i+1][j-1]
            if temp.lower() == temp:
                moves.append((j-1,i+1))
        except:

            pass

    #black pawn
    if self.piece.islower(): #same as above but in opposite direction
        if i > 0:
            try:
                temp = board[i-1][column]
                if temp == 1:
                    moves.append((column,i-1))
            except:
                pass

        if row == (h-2):
            temp = board[i-2][column]
            temp2 = board[i-1][column]
            if temp == 1 and temp2 == 1:
                moves.append((column,i-2))

        try:
            temp = board[i-1][j+1]
            if temp.upper() == temp:
                moves.append((j+1,i-1))
        except:
            pass

        try:
            temp = board[i-1][j-1]
            if temp.upper() == temp:
                moves.append((j-1,i-1))
        except:
            pass

    #Filter all negative values

    temp = [i for i in moves if i[0] >=0 and i[1] >=0]


    allPossibleMoves = []

    for i in temp:
```

```
                    allPossibleMoves.append(vector_to_algebra(i))

                #check for check
                if check_check:
                #if the laws of check are being considered
                    royal_moves = []

                    for i in allPossibleMoves:
                        newpos = algebra_to_vector(i)
                        if not royal_attacked(ChessBoard.getboard(), self.piece, turn, newpos, coords, w, h):
                        #filters all moves that result in a royal piece being attacked
                            royal_moves.append(i)

                    allPossibleMoves = royal_moves

                return allPossibleMoves

```

The movement of pawns in chess is unique. They can only move in one direction and this is different depending on which colour they are, they get to move two squares on their first turn and, unlike all other pieces, the squares which they can capture on are different from the squares they attack. Since they can't be generalised, they inherit everything from the Piece class apart from their getMoves method which specifies pawn moves.

```
#royal piece
class Royal(Piece):
        def getStatus(self):
                return 'royal'

```

A royal piece is the same as a piece but it's getStatus returns royal instead of common.

```
#standard pieces
#(self, letter, vectors_ride = None, vectors_leap = None)
white_pawn = Pawn('P')
white_rook = Piece('R', [[0,1]])
white_knight = Piece('N', None, [[1,2]])
white_bishop = Piece('B', [[1,1]])
white_king = Royal('K', None, [[0,1],[1,1]])
white_queen = Piece('Q', [[0,1],[1,1]])
black_pawn = Pawn('p')
black_rook = Piece('r', [[0,1]])
black_knight = Piece('n', None, [[1,2]])
black_bishop = Piece('b', [[1,1]])
black_king = Royal('k', None, [[0,1],[1,1]])
black_queen = Piece('q', [[0,1],[1,1]])

#fairy pieces
# fairy pieces are initialise as none and then set values later if the user decides to use them
```

```
    white_fairy1 = None
    white_fairy2 = None
    white_fairy3 = None
    black_fairy1 = None
    black_fairy2 = None
    black_fairy3 = None

    fairies = [[white_fairy1, black_fairy1], [white_fairy2, black_fairy2], [white_fairy3, black_fairy3]]
    #put into list for easy access

    letters = ['x', 'y', 'z']
    #unique identifiers

    #pieces
    pieces = {
              'P':['white_pawn',white_pawn],
              'R':['white_rook',white_rook],
              'N':['white_knight',white_knight],
              'B':['white_bishop',white_bishop],
              'K':['white_king',white_king],
              'Q':['white_queen',white_queen],
              'X':['white_fairy1',white_fairy1],
              'Y':['white_fairy2',white_fairy2],
              'Z':['white_fairy3',white_fairy3],
              'p':['black_pawn',black_pawn],
              'r':['black_rook',black_rook],
              'b':['black_bishop',black_bishop],
              'n':['black_knight',black_knight],
              'k':['black_king',black_king],
              'q':['black_queen',black_queen],
              'x':['black_fairy1',black_fairy1],
              'y':['black_fairy2',black_fairy2],
              'z':['black_fairy3',black_fairy3]
              }

    #dictionary containing pieces and their image file

```

Pieces are stored in a dictionary. The unique letter is used as the key to identify the piece and the object is stored along with the string used for getting the piece's image file.

Then some functions which are used throughout the code are defined.

```
    #next turn
    def next_turn(turn):
        turn = turn%2+1
        return turn

```

The turn starts as 1 and then the remainder is found with modular division by two and then 1 is added to that. This means it alternates between 1 and 2. Modular arithmetic is a good method for calculating turns because, where n is the number that the turn is divided by, the turns form a cyclic group of order n under the operation of modular addition. This means that this number can be changed to any positive integer and the system still cycles through the turns in the correct order. This would make it significantly easier to edit the program to support more than two players and even possibly a user defined number of players in future.

```
    #leap moves
    def leap_move(f, g, m, n, board, piece, column, row):

        moves = []
        try:
            temp = board[f(row, m)][g(column, n)]
            #tests if a square is on the board
            if temp == 1 or (temp.isupper() ^ piece.isupper()):
            #if square is empty or contains enemy piece
                moves = [[g(column, n), f(row, m)]]
        except:
            pass
        return moves

```

The function is passed: The operations being used, both components of the movement vector, the current board, the identifier of the piece (so that it knows what side it is on) and the location on the board. It applies the vector with the operations and if the square is empty or contains an enemy piece it adds that square to possible moves

```
    #ride moves
    def ride_move(f, g, m, n, board, piece, column, row):
        moves = []
        i = 1
        while True:
            try:
                temp = board[f(row,(m*i))][g(column,(n*i))]
                #tests if a square is on the board
            except:
                break
                #if not, stops calculating moves in that direction
            if temp == 1:
                moves+=[[g(column,(n*i)), f(row,(m*i))]]
                #if a square is empty adds it to possible moves
                i+=1
            elif temp == 4:
                break
                #if a wall is hit, stops calculating moves in that direction
            elif temp.isupper() ^ piece.isupper():
                moves+=[[g(column,(n*i)), f(row,(m*i))]]
                #if an enemy piece is hit, adds its locations to possible moves then stops calculating moves in that direction
                break
            else:
                break
                #this case only occurs when a friendly piece is hit, in which case it stops calculating moves in that direction
        return moves

```

A counter is initialised at 1. Both components of this vector are multiplied by the counter and added to the piece's location. The square arrived at is added to the list of possible moves. If the square contains a friendly piece or wall, or isn't on the board, the move is not added, and the function stops calculating moves in that direction. If the square contains an enemy piece, the function stops but the move is still added (this move will capture that piece). If nothing stops it the process is repeated.

```
#extra row count
def extra_row_count(width, count, agents):
    #count starts at 1

  if int(width)*int(count) >= len(agents):
  #multiplies the width of the board by the count of extra rows to see if it can contain all the creation tools
    return count
    #base case
    #if it fits all the pieces, the count is returned

  else:
    return extra_row_count(width, count+1, agents)
    #general case
    #if not, it increases the count and performs the function again

```

This function checks if 1 extra row is enough to contain all the creation tools, if not it increases by 1 and checks again until the extra row count is enough.

```
#promotion
def promotion_check(chessBoard, h, agents_):

  agents = []

  for i in agents_:
      if type(i) == str and i.lower() != 'p':
          agents.append(i)
    #selects only the non-pawn pieces to be promoted



  for i in range(len(chessBoard[0])):
    if chessBoard[0][i] == 'p':
        #black pawn
            for p in agents:
                if p.islower():
                    print(p)
            while True:
                choice = input('Select a piece to promote to...\n')
                if choice in agents and choice.islower():
                    break
                else:
                    print('Invalid answer')
            chessBoard[0][i] = choice

    for i in range(len(chessBoard[h-1])):
        if chessBoard[h-1][i] == 'P':
        #white pawn
            for p in agents:
                if p.isupper():
                    print(p)
            while True:
                choice = input('Select a piece to promote to...\n')
                if choice in agents and choice.isupper():
                    break
                else:
                    print('Invalid answer')
            chessBoard[h-1][i] = choice

    return chessBoard

```

Checks if any pawns are on the last rank. If there is one, provides a text-based promotion menu with all the pieces in play for the player to choose from.

```
#function to check for check
def checkifcheck(board, turn, w, h):

    chessBoard = deepcopy(board)
    #creates deepcopy of the board

    attacked_spaces = []

    moves = []

    for x in range(h):
        for y in range(w):
            if chessBoard[x][y] == 5:
                chessBoard[x][y] = 1
    #replaces all hills markers with empty squares

    for x in range(h):
        for y in range(w):
                if (type(chessBoard[x][y]) == str)  and (((chessBoard[x][y]).islower()) == turn-1):
                #finds every piece belonging to the attacking player
                moves.append(pieces[chessBoard[x][y]][1].getMoves((chess_map_from_index_to_alpha[y]+str(x+1)), chessBoard, turn-1, False, w, h))

    for z in moves:
        if z:
            for q in z:
                if q not in attacked_spaces:
                    attacked_spaces.append(q)
                    #filters all squares that are attacked twice

    for i in attacked_spaces:
        y = algebra_to_vector(i)
        if chessBoard[y[1]][y[0]] != 1:
            if pieces[chessBoard[y[1]][y[0]]][1].getStatus() == 'royal':
               return True
        #if one of these squares contains a royal piece, then there is check

    return False

```

This function takes the board, the board's dimensions and the current turn. First it creates a deep copy of the board so that any changes made aren't kept. Then it turns every hill square into an empty square so that they don't obstruct moves. It then iterates through the board finding every piece belonging to the attacking player and collects their moves.

(piece.islower() == turn –1) will return True for white on turn 1 and True for black on turn 2.

When these moves are calculated check is not factored in because it doesn't matter for the attacking player (even if a piece can't technically move somewhere because it would reveal a discovered check it can still threaten that square). This distinction is made by passing False as the 'check_check' parameter. Then all the squares that have already been attacked are filtered and then the final list is searched for royal pieces. If such a piece exists on an attacked square, the function returns True, if not, False.

```
    #function to see if a royal piece is being attacked after a move is made
    def royal_attacked(chessBoard, piece, turn, newpos, origpos, w, h):
        #is passed all the infor about the board as well as the info about the move being tested for legality

        moves = []
        attacked_spaces = []

        tempBoard = deepcopy(chessBoard)
        #creates a deep copy of the board

        tempBoard[origpos[1]][origpos[0]] = 1
        tempBoard[newpos[1]][newpos[0]] = piece
        #makes the move about to be made

        return checkifcheck(tempBoard, next_turn(turn), w, h)
        #sees if the move has resulted in a check

```

The royal attacked function is how the program works out if a player’s move will result in their own royal piece being in check. This function is what is used to filter any moves obtained by the getMoves methods which cause a discovered check. It is passed all the information about the board as well as the move being tested. A deep copy of the board is made and the program advances as if the move had been made, (makes move and moves onto next turn). Then the program applies the check for check function on this new board and returns the result. The getMoves method tests every move with the royal_attacked function and anything that returns True is filtered.

Next the win conditions are defined.

```
#win conditions

#extinction
def extinction(board, w, h, turn, check = None, hills = None):
    white_exists = False

    black_exists = False

    #assumes no pieces exist


    for y in range(h):
        for x in range(w):
            if type(board[y][x]) == str and board[y][x].isupper():
                white_exists = True
            if type(board[y][x]) == str and board[y][x].islower():
                black_exists = True
    #checks each side for a piece

    if not white_exists:
        pygame.display.update()
        print('==========\n\nBlack wins\n\n===========\n')
        return True

    if not black_exists:
        pygame.display.update()
        print('==========\n\nWhite wins\n\n===========\n')
        return True
    #if either side has no pieces, game ends

    return False
    #otherwise game continues

```

The extinction function iterates through the board checking for a piece of each colour. If either side has no pieces or is ‘extinct’ the game ends.

```
#checkmate
def checkmate(board, w, h, turn, check, hills = None):









    check_check = True
    allowed_moves = []

    #calculate every move of current player
    for x in range(h):
        for y in range(w):
            if type(board[x][y]) == str  and (board[x][y]).isupper() ^ turn-1:
                move = (pieces[board[x][y]][1].getMoves((chess_map_from_index_to_alpha[y]+str(x+1)), board, turn, True, w, h))

                if move:
                    allowed_moves.append(move)


    #if there are no moves, the game ends
    if not allowed_moves:
        if check == True:
        #if a royal is in check then it is a checkmate
            if turn == 1:
                pygame.display.update()
                print('==========\n\nBlack wins\n\n===========\n')
                return True

            elif turn == 2:
                pygame.display.update()
                print('==========\n\nWhite wins\n\n===========\n')
                return True
        else:
        #if no piece is in check then it is a stalemate
                  pygame.display.update()
                  print('==========\n\nStalemate\n\n===========\n')
                  return True

    return False
    #if the player can still make a move, the game continues

```

The checkmate function checks if a player can make any moves, if not the game is over. This is a checkmate if the player is in check and a stalemate if they are not.

```
#king of the hill
def KotH(board, w, h, turn, check, hills):
    if checkmate(board, w, h, turn, check):
        return True

    if hills:
        for i in hills:
                if type(board[i[0]][i[1]]) == str and pieces[board[i[0]][i[1]]][1].getStatus() == 'royal' and board[i[0]][i[1]].isupper():
                    pygame.display.update()
                    print('==========\n\nWhite wins\n\n===========\n')
                    return True

                if type(board[i[0]][i[1]]) == str and pieces[board[i[0]][i[1]]][1].getStatus() == 'royal' and board[i[0]][i[1]].islower():
                    pygame.display.update()
                    print('==========\n\nBlack wins\n\n===========\n')
                    return True
                #if a royal is on a hill, game ends
    return False

```

The king of the hill function first checks if a piece has been mated. If not, it searches for any hill squares that contain royal pieces and if such a square exists, the owner of that piece wins.

```
#regicide
def regicide(board, w, h, turn, check = None, hills = None):

    white_royals = False

    black_royals = False

    for y in range(h):
        for x in range(w):
            if type(board[y][x]) == str and board[y][x].isupper():
                if pieces[board[y][x]][1].getStatus() == 'royal':
                    white_royals = True


    for y in range(h):
        for x in range(w):
```

```
                        if type(board[y][x]) == str and board[y][x].islower():
                                if pieces[board[y][x]][1].getStatus() == 'royal':
                                        black_royals = True
        #searches for royal pieces

        if not white_royals:
                pygame.display.update()
                print('==========\n\nBlack wins\n\n===========\n')
                return True

        if not black_royals:
                pygame.display.update()
                print('==========\n\nWhite wins\n\n===========\n')
                return True
        #if a side lacks a royal piece, game ends

        return False

```

This function works in the same way as extinction, but it only considers royal pieces.

```
#prompt
if __name__ == '__main__':

    mainloop = True

    while mainloop:

        try:
            choice = input('1:play default\n2:play Fischer Random+\n3:create ruleset\n4:delete ruleset\n5:play custom\n')
            #user is propmpted
            if choice.lower() == 'quit':
                mainloop = False
            #quit option
            elif choice.lower() == 'help':
            #help menu
                print('\n\nNaviagting the program:\nWhen given items listed numerically, type in the number of the item you wish to select\nWhen asked a binary question type \'yes\' to confirm or enter enything else to deny\n')
                print('Pieces:')
                print('Piece movement is represented by 2-dimensional vectors. These can be interpreted in two ways:\n1:As a leaping vector, in which case the piece can move once by that vector in all directions\n2:As a riding vector, in which case the piece can move infinitely by that vector until it is blocked.\n(so a knight is a [1,2] leaper and a bishop is a [1,1] rider')
                print('A piece\'s status can either be common or royal. The properties of royal pieces are dependent on the game\'s win condition\n')
                print('Win conditions:\n1:Checkmate: the standard rules of Chess, games can result in checkmate on any royal piece for either side or stalemate\n2:Regicide: the laws of check do not apply, if an enemy royal is captured, the game is won\n3:King of the Hill: Same as the Checkmate but with hill squares which give victory to a player that moves their royal piece onto the hill.\n4:Extinction: The laws of check do not apply, all enemy pieces must be captured to win.')
                print('Custom rulesets are not playable unless there are sufficient pieces for the win condition to be achieved.\n')
            else:
                choice = int(choice)

        except:
            choice = 0
            #if choice is invalid, sets it to 0 for error handling clause

```

Then the main loop begins. The user is prompted to choose from the menu and can also ask for help o

```
#default
        if choice == 1:
            agents = [4,'P','R','N','B','K','Q','p','r','n','b','k','q']
            ChessBoard = GameBoard(agents, deepcopy(defaultBoard), 8, 8, checkmate)
            #default rules
            ChessBoard.play()

```

If the choice is 1, it creates a default game and plays it.

```
#Fischer random+
        elif choice == 2:
            agents = [4,'P','R','N','B','K','Q','p','r','n','b','k','q']

            while True:
                try:
                    w = int(input('Width of the board(4-26):\n'))
                    if w < 27 and w > 3:
                        break
                    else:
                        print('Not within bounds')
                except:
                    print('invalid input')


            while True:
                try:
                    h = int(input('Height of the board(4-26):\n'))
                    if h < 27 and h > 3:
                        break
                    else:
                        print('Not within bounds')
                except:
                    print('invalid input')

            #gets height and width of the board

            random_pieces = []

            for i in (agents):
                if type(i) is str and i == i.lower() and i.lower() != 'k' and i.lower() != 'p':
                    random_pieces.append(i)
            #adds all the white pieces to a list

            board = [[1] * w for i in range(h)]
            #creates a board of the dimensions specified by the user

            white_pawns = []
            for i in range(w):
                 white_pawns.append('P')
            board[1] = white_pawns

            black_pawns = []
            for i in range(w):
                 black_pawns.append('p')
            board[h-2] = black_pawns
            #fills the second and penultimate rows with pawns

            for i in range(w):
               x = random.choice(random_pieces)
               board[0][i] = x.upper()
               board[h-1][i] = x
               #chooses a random piece and adds a white one to the white side and black on to the blakc side

            kingspot = random.randint(0,i)

            board[0][kingspot] = 'K'

            board[h-1][kingspot] = 'k'
            #randomly places the kings

            ChessBoard = GameBoard(agents, board, w, h, checkmate)
            ChessBoard.play()
            #defines the game and starts playing

```

If the choice is 2, the user is asked to enter their desired height and width. The all the white pieces from the list of agents are added to a random piece list. This is because the sides are mirrored so the white set up can be randomly generated and then just reflected to get the black set up. The king is not added to this list but is randomly added afterwards to ensure there is exactly one on each side.

```
    #custom ruleset
```

```python
            elif choice == 3:

                agents = [4,'P','R','N','B','K','Q','p','r','n','b','k','q']

                loop = 0

                while loop < 3:
                    ans = input('Would you like to make a new piece?({}/3):\n'.format(loop+1))
                    #the user is asked to make a custom piece up to 3 times

                    if ans.lower() == 'yes':

                        fairy = letters[loop]

                        vec_ride = []
                        vec_leap = []

                        while True:
                            ans = input('Add rides?\n')
                            if ans.lower() != 'yes':
                                break

                            #can't use while not m/n because m/n might be set to 0

                            m = False
                            while not type(m) is int:
                                try:
                                    m = int(input('First dimension?\n'))

                                except:
                                    print('answer must be an integer')


                            n = False
                            while not type(n) is int:
                                try:
                                    n = int(input('Second dimension?\n'))

                                except:
                                    print('answer must be an integer')


                            vec_ride.append([m,n])


                        while True:
                            ans = input('Add leaps?\n')
                            if ans.lower() != 'yes':
                                break

                            m = False
                            while not type(m) is int:
                                try:
                                    m = int(input('First dimension?\n'))
                                except:
                                    print('answer must be an integer')


                            n = False
                            while not type(n) is int:
                                try:
                                    n = int(input('Second dimension?\n'))
                                except:
                                    print('answer must be an integer')

                            vec_leap.append([m,n])
                        #rides and leaps are inputed by the user


                        ans = input('Is the piece royal?:\n')
                        if ans.lower() == 'yes':
                            define = Royal
                        else:
                            define = Piece

                        if not vec_ride and not vec_leap and define == Piece:
                            print('no piece created.')

                        else:
                            fairies[loop][0] = define(fairy.upper(), vec_ride, vec_leap)
                            #defines piece using user input

                            pieces[fairy.upper()].pop(-1)
                            #gets rid of previous piece in that piece's slot

                            pieces[fairy.upper()].append(fairies[loop][0])
                            #marks the piece as a fairy

                            #creates an identical black version
                            fairies[loop][1] = define(fairy, vec_ride, vec_leap)

                            pieces[fairy].pop(-1)

```

```
            pieces[fairy].append(fairies[loop][1])

            agents += [fairy, fairy.upper()]

            loop += 1
        else:
          loop = 3

      win_condition = False

```

The user is asked if they want to design a custom piece. If they say yes, they can choose to add as many rides and leaps as they wish and specify the vectors for these. Then they are asked if they want their piece to be defined as a standard piece or a royal. The pieces are added to the piece dictionary to be stored within the code temporarily and to a list of fairies to be stored within the board object permanently.

```
#win condition choice
      while not win_condition:
        try:
          ans = int(input('Select a win condition\n1:Checkmate\n2:Regicide\n3:King of the Hill\n4:Extinction\n'))
        except:
          ans = 0



        if ans == 1:
            win_condition = checkmate
        elif ans == 2:
            win_condition = regicide
        elif ans == 3:
            agents = [5] + agents
            win_condition = KotH
        elif ans == 4:
            win_condition = extinction

        else:
          print('Invalid answer')

      ChessBoard = False

```

Now the user must select a win condition. This decides which function is passed to the custom gameboard class. After that they choose whether to use a default board or a custom board.

```
while not ChessBoard:
        try:
          ans = int(input('1:Default Board\n2:Custom Board\n'))
        except:
          ans = 0

        if ans == 1:
```

```python
                        board = deepcopy(defaultBoard)

                        TempBoard = CustomGameBoard(agents, fairies, board, 8, 8, win_condition)
                        if win_condition == KotH:
                            TempBoard.sethills([[3, 3], [3, 4], [4, 3], [4, 4]])
                            #if default board is selected with king of the hill, central four squares become hills
                        ChessBoard = TempBoard

                elif ans == 2:


                        while True:
                            try:
                                w = int(input('Width of the board(2-26):\n'))
                                if w < 27 and w > 1:
                                    break
                                else:
                                    print('Not within bounds')
                            except:
                                print('invalid input')

                        while True:
                            try:
                                h = int(input('Height of the board(2-26):\n'))
                                if h < 27 and h > 1:
                                    break
                                else:
                                    print('Not within bounds')
                            except:
                                print('invalid input')

                        board = [[1] * w for i in range(h)]
                        #board generated with user's height and width

                        extra_h = extra_row_count(w, 1, agents)


                        auxiliary_1 = [[2] * w]
                        #represents separation row
                        auxiliary_1[0][round(w/2)] = 3
                        #tick button is placed in the middle of the separation row
                        auxiliary_2 = [[1] * w for i in range(extra_h)]
                        #space for the board design tools is added
                        auxiliary = auxiliary_1+auxiliary_2
                        #the two are combined

                        counter = 0
                        for i in range(len(agents)):
                            if i % w == 0:
                                counter += 1
                            auxiliary[counter][i%w] = agents[i]
                        #populated with pieces

                        for i in auxiliary:
                            board.append(i)
                        #adds auxiliaries to board

                        TempBoard = CustomGameBoard(agents, fairies, board, w, h, win_condition)

                        ChessBoard = TempBoard





                else:
                    print('Invalid answer')

```
If a custom board is selected, like in Fischer random+, it asks the user for height and width. However, this time it prepares to use the board creation function. A board is generated with the user's dimensions, then the extra height needed for the creation tools is calculated. The creation board is constructed with the help of two auxiliaries. The first is the separating layer between the board and the creative tools, and the tick is placed in the middle of this. The next auxiliary contains space for the creative tools. It is then populated with all the pieces in the list of agents, including any custom pieces the user defined. Then it is initialised as a custom gameboard, triggering the board design function. If the user presses the close window button, the user is returned to the main menu.

```
#saving
            if ChessBoard.getboard():
            #if a board has been created
                    presets = []

                    for entry in os.listdir('.\Presets'):
                        presets.append(entry.lower())
                    #creates list of all existing rule sets

                    while True:
                        name = input('Choose a name for your ruleset:\n')
                        if name.lower() in presets:
                            print('Name already taken\n')
                        else:
                            break
                    #verifies that name is unique

                    if name:
                        with open('.\Presets\{}'.format(name), 'wb') as file:
                            pickle.dump(ChessBoard, file)
                    #saves the board object to a file

```

First the program loads in all the previous rulesets from a file. Then the user is asked to name their ruleset and the list of previous ones is used to verify that it has a unique name. Once a unique name is found the ruleset is saved to a file.

```
    #delete
            elif choice == 4:
                presets = []

                for entry in os.listdir('.\Presets'):
                    presets.append(entry)

                if presets:
                #verifies that rulesets exist
                    print('Select a ruleset from the following to delete...\n')

                    print('0: cancel')
                    for i in range(len(presets)):
                        print('{}:'.format(i+1), presets[i])

                    while True:
                        try:
                            choice = int(input(''))
                            if choice >= 0 and choice <= len(presets):
                                break
                            else:
                                print('No such ruleset exists')
                        except:
                            print('Invalid input')

                    if choice == 0:
                        pass
                    else:
                        name = presets[choice-1]

                        os.remove('.\Presets\{}'.format(name))
                        #deletes selected ruleset
                        print('Ruleset deleted\n')
                else:
                    print('no rulesets currently exist\n')

    #play custom
            elif choice == 5:

                    presets = []

                    for entry in os.listdir('.\Presets'):
                        presets.append(entry)
                    #presets are loaded from a file

                    if presets:
                    #verifies that they exist
                        print('Select a ruleset from the following...\n')

                        print('0: cancel')
                        for i in range(len(presets)):
                            print('{}:'.format(i+1), presets[i])

                        while True:
                            try:
                                choice = int(input(''))
                                if choice >= 0 and choice <= len(presets):
                                    break
                                else:
                                    print('No such ruleset exists')
                            except:
                                print('Invalid input')

                        if choice == 0:
                            pass
                        else:

                            with open('.\Presets\{}'.format(presets[choice-1]), 'rb') as file:
                                ChessBoard = pickle.load(file)

                            #loads selected ruleset

                            custom_pieces = ChessBoard.getfairies()


                            for i in range(len(custom_pieces)):
                                pieces[letters[i].upper()].pop(-1)
                                pieces[letters[i].upper()].append(custom_pieces[i][0])
                                pieces[letters[i]].pop(-1)
                                pieces[letters[i]].append(custom_pieces[i][1])
                            #fills dictionary with custom pieces from loaded ruleset

                            ChessBoard.create_notation()
                            ChessBoard.play()

                    else:
                        print('No custom rulesets currently exist.\nChoose 2 to play Fischer random+ chess\nChoose 3 to create your own\n')

```

```python

                elif choice == 0:
                    print('Invalid answer\n')
```

A similar process occurs if delete or play are selected. When play is selected, the program replaces the custom pieces that are currently saved with the ones from the file that has been just loaded.