

Szachy

Szymon Makulec Jan Moskal

10 lutego 2024

Cel i efekty pracy

Opis

Projekt zakładał stworzenie gry w szachy dla dwóch graczy. Kod oparty jest na bibliotece wxWidgets, a interfejs użytkownika wykorzystuje siatkę wxBitmapButton z obrazkami figur połączonymi z kolorem tła, do reprezentacji szachownicy i figur na niej położonych. Projekt udało nam się doprowadzić do stanu gdzie są przestrzegane zasady gry w szachy takie jak: ruszanie się na zmianę graczy, figury mają tylko takie możliwości ruchy jakby była to normalna gra, mamy dostęp do roszady, pierwszego ruchu piona o 2, promocji piona. Przestrzegane są również zasady związane z szachowaniem tj. nie możemy się ruszyć w taki sposób by po ruchu król był dalej pod biciem.

Tablice obiektów

Użycie obrazków (images[2][13])

```
1 // Black Pieces Images
2 images[0][0] = wxBitmap(wxImage("images/D.jpg"));
3 images[0][1] = wxBitmap(wxImage(_T("images/Pieces/bpB.png")));
4 images[0][2] = wxBitmap(wxImage(_T("images/Pieces/bpD.png")));
5 images[0][3] = wxBitmap(wxImage(_T("images/Pieces/brB.png")));
6 images[0][4] = wxBitmap(wxImage(_T("images/Pieces/brD.png")));
7 images[0][5] = wxBitmap(wxImage(_T("images/Pieces/bnB.png")));
8 images[0][6] = wxBitmap(wxImage(_T("images/Pieces/bnD.png")));
9 images[0][7] = wxBitmap(wxImage(_T("images/Pieces/bbB.png")));
10 images[0][8] = wxBitmap(wxImage(_T("images/Pieces/bbD.png")));
11 images[0][9] = wxBitmap(wxImage(_T("images/Pieces/bqB.png")));
12 images[0][10] = wxBitmap(wxImage(_T("images/Pieces/bqD.png")));
13 images[0][11] = wxBitmap(wxImage(_T("images/Pieces/bkB.png")));
14 images[0][12] = wxBitmap(wxImage(_T("images/Pieces/bkD.png")));
15
16 // White Pieces Images
17 images[1][0] = wxBitmap(wxImage("images/B.jpg"));
18 images[1][1] = wxBitmap(wxImage(_T("images/Pieces/wpB.png")));
19 images[1][2] = wxBitmap(wxImage(_T("images/Pieces/wpD.png")));
20 images[1][3] = wxBitmap(wxImage(_T("images/Pieces/wrB.png")));
21 images[1][4] = wxBitmap(wxImage(_T("images/Pieces/wrD.png")));
22 images[1][5] = wxBitmap(wxImage(_T("images/Pieces/wnB.png")));
23 images[1][6] = wxBitmap(wxImage(_T("images/Pieces/wnD.png")));
24 images[1][7] = wxBitmap(wxImage(_T("images/Pieces/wbB.png")));
25 images[1][8] = wxBitmap(wxImage(_T("images/Pieces/wbD.png")));
26 images[1][9] = wxBitmap(wxImage(_T("images/Pieces/wqB.png")));
27 images[1][10] = wxBitmap(wxImage(_T("images/Pieces/wqD.png")));
28 images[1][11] = wxBitmap(wxImage(_T("images/Pieces/wkB.png")));
29 images[1][12] = wxBitmap(wxImage(_T("images/Pieces/wkD.png")));
```

Do wyświetlania figur na przyciskach wxBitmapButton, używamy zestawu dwudziestu sześciu obrazków, reprezentujących figury szachowe. Dwa puste pola (czarne i białe) oraz sześć różnych typów figur (pion, wieża, skoczek, goniec,

hetman, król) w dwóch kolorach (biały i czarny) na dwóch tłach (jasnym, ciemnym). `images`, jest to tablica, w której przechowywane są obrazy figur szachowych używane do wyświetlania na przyciskach `wxBitmapButton` na szachownicy. Umożliwia łatwe zarządzanie obrazkami figur szachowych w grze.

Użycie przycisków `wxBitmapButton` (`board[8][8]`)

```
1 board[0][0] = BitmapButton1;
2 board[0][1] = new wxBitmapButton(this, wxNewId(), images[0][6],
   wxDefaultPosition, wxDefaultSize, wxBU_AUTODRAW,
   wxDefaultValidator);
3 board[0][2] = new wxBitmapButton(this, wxNewId(), images[0][7],
   wxDefaultPosition, wxDefaultSize, wxBU_AUTODRAW,
   wxDefaultValidator);
4 board[0][3] = new wxBitmapButton(this, wxNewId(), images[0][10],
   wxDefaultPosition, wxDefaultSize, wxBU_AUTODRAW,
   wxDefaultValidator);
5 board[0][4] = new wxBitmapButton(this, wxNewId(), images[0][11],
   wxDefaultPosition, wxDefaultSize, wxBU_AUTODRAW,
   wxDefaultValidator);
6 board[0][5] = new wxBitmapButton(this, wxNewId(), images[0][8],
   wxDefaultPosition, wxDefaultSize, wxBU_AUTODRAW,
   wxDefaultValidator);
7 board[0][6] = new wxBitmapButton(this, wxNewId(), images[0][5],
   wxDefaultPosition, wxDefaultSize, wxBU_AUTODRAW,
   wxDefaultValidator);
8 board[0][7] = new wxBitmapButton(this, wxNewId(), images[0][4],
   wxDefaultPosition, wxDefaultSize, wxBU_AUTODRAW,
   wxDefaultValidator);
9
10 for(int j = 0; j < 8; j++){
11     if(j%2 == 0){
12         board[1][j] = new wxBitmapButton(this, wxNewId(), images[0][2] ,
13             wxDefaultPosition, wxDefaultSize, wxBU_AUTODRAW,
14             wxDefaultValidator);
15     }else{
16         board[1][j] = new wxBitmapButton(this, wxNewId(), images[0][1] ,
17             wxDefaultPosition, wxDefaultSize, wxBU_AUTODRAW,
18             wxDefaultValidator);
19     }
20 }
21
22 for(int i = 2; i < 6; i++){
23     for(int j = 0; j < 8; j++){
24         if((i+j)%2 == 0){
25             board[i][j] = new wxBitmapButton(this, wxNewId(), images
26                 [1][0], wxDefaultPosition, wxDefaultSize, wxBU_AUTODRAW,
27                 wxDefaultValidator);
28         }else{
29             board[i][j] = new wxBitmapButton(this, wxNewId(), images
30                 [0][0], wxDefaultPosition, wxDefaultSize, wxBU_AUTODRAW,
31                 wxDefaultValidator);
32         }
33     }
34 }
```

```

28
29 for(int j = 0; j < 8; j++){
30     if((j)%2 != 0){
31         board[6][j] = new wxBitmapButton(this, wxNewId(), images[1][2],
32             wxDefaultPosition, wxDefaultSize, wxBU_AUTODRAW,
33             wxDefaultValidator);
34     }else{
35         board[6][j] = new wxBitmapButton(this, wxNewId(), images[1][1],
36             wxDefaultPosition, wxDefaultSize, wxBU_AUTODRAW,
37             wxDefaultValidator);
38     }
39 }
40
41 board[7][0] = new wxBitmapButton(this, wxNewId(), images[1][4],
42     wxDefaultPosition, wxDefaultSize, wxBU_AUTODRAW,
43     wxDefaultValidator);
44 board[7][1] = new wxBitmapButton(this, wxNewId(), images[1][5],
45     wxDefaultPosition, wxDefaultSize, wxBU_AUTODRAW,
46     wxDefaultValidator);
47 board[7][2] = new wxBitmapButton(this, wxNewId(), images[1][8],
48     wxDefaultPosition, wxDefaultSize, wxBU_AUTODRAW,
49     wxDefaultValidator);
50 board[7][3] = new wxBitmapButton(this, wxNewId(), images[1][9],
51     wxDefaultPosition, wxDefaultSize, wxBU_AUTODRAW,
52     wxDefaultValidator);
53 board[7][4] = new wxBitmapButton(this, wxNewId(), images[1][12],
54     wxDefaultPosition, wxDefaultSize, wxBU_AUTODRAW,
55     wxDefaultValidator);
56 board[7][5] = new wxBitmapButton(this, wxNewId(), images[1][7],
57     wxDefaultPosition, wxDefaultSize, wxBU_AUTODRAW,
58     wxDefaultValidator);
59 board[7][6] = new wxBitmapButton(this, wxNewId(), images[1][6],
60     wxDefaultPosition, wxDefaultSize, wxBU_AUTODRAW,
61     wxDefaultValidator);
62 board[7][7] = new wxBitmapButton(this, wxNewId(), images[1][3],
63     wxDefaultPosition, wxDefaultSize, wxBU_AUTODRAW,
64     wxDefaultValidator);
65
66 for(int i = 0; i < 8; i++){
67     for(int j = 0; j < 8; j++){
68         if(i + j == 0){
69             j++;
70         }
71         GridSizer1->Add(board[i][j], 1, wxALIGN_CENTER_HORIZONTAL |
72             wxALIGN_CENTER_VERTICAL, 5);
73         Connect(board[i][j]->GetId(), wxEVT_COMMAND_BUTTON_CLICKED, (
74             wxObjectEventFunction)&Chess2DDialog::OnBitmapButton1Click)
75             ;
76     }
77 }

```

Każdy element board[i][j] (8x8) reprezentuje jedno pole z wyświetloną figurą danego koloru i tłem na szachownicy, zawiera przycisk do interakcji z użytkow-

nikiem. Każdy przycisk jest obsługiwany przez tą samą funkcję `OnBitmapButton1Click`.

Użycie figur (`pieces[2][7]`)

```
1 pieces[0][0] = new Pawn(false);
2 pieces[0][1] = new Rook(false);
3 pieces[0][2] = new Knight(false);
4 pieces[0][3] = new Bishop(false);
5 pieces[0][4] = new Queen(false);
6 pieces[0][5] = new King(false);
7 pieces[0][6] = new Rook(false);
8 pieces[1][0] = new Pawn(true);
9 pieces[1][1] = new Rook(true);
10 pieces[1][2] = new Knight(true);
11 pieces[1][3] = new Bishop(true);
12 pieces[1][4] = new Queen(true);
13 pieces[1][5] = new King(true);
14 pieces[1][6] = new Rook(true);
```

`pieces[i][j]` jest to dwuwymiarowa tablica wskaźników do obiektów klasy `Piece`. Jest to tablica, w której przechowywane są obiekty reprezentujące różne typy figur szachowych w dwóch kolorach: białym i czarnym. Każdy obiekt klasy `Piece` przechowuje informacje o tym czy figura już się ruszała. Dlatego tworzymy dwie wieże, żeby każdej z osobna można było sprawdzać czy się ruszała. W przypadku pionów uznaliśmy tworze oddzielnych Pawnów za zbyt duże, ponieważ każdy może poruszać się tylko w 1 kierunku.

Przechowywanie przycisków i figur w jednym obiekcie (`squares[8][8]`)

```
1 for(int i = 0; i < 8; i++){
2     for(int j = 0; j < 8; j++){
3         if((i+j) % 2 == 0){
4             squares[i][j] = new Square(i, j , true, board[i][j]);
5         }else{
6             squares[i][j] = new Square(i, j , false, board[i][j]);
7         }
8     }
9 }
10
11 for(int i = 0; i < 8; i++){
12     squares[1][i] -> setPiece(pieces[0][0]);
13     squares[6][i] -> setPiece(pieces[1][0]);
14 }
15
16 // Assigning other Pieces to Squares
17 squares[0][0] -> setPiece(pieces[0][1]);
18 squares[0][1] -> setPiece(pieces[0][2]);
19 squares[0][2] -> setPiece(pieces[0][3]);
20 squares[0][3] -> setPiece(pieces[0][4]);
21 squares[0][4] -> setPiece(pieces[0][5]);
22 squares[0][5] -> setPiece(pieces[0][3]);
23 squares[0][6] -> setPiece(pieces[0][2]);
```

```

24 squares[0][7] -> setPiece(pieces[0][6]);
25 squares[7][0] -> setPiece(pieces[1][1]);
26 squares[7][1] -> setPiece(pieces[1][2]);
27 squares[7][2] -> setPiece(pieces[1][3]);
28 squares[7][3] -> setPiece(pieces[1][4]);
29 squares[7][4] -> setPiece(pieces[1][5]);
30 squares[7][5] -> setPiece(pieces[1][3]);
31 squares[7][6] -> setPiece(pieces[1][2]);
32 squares[7][7] -> setPiece(pieces[1][6]);

```

`squares[i][j]` to dwuwymiarowa tablica wskaźników do obiektów typu `Square`, reprezentująca pola szachownicy. Każdy element tej tablicy to obiekt `Square`, który przechowuje informacje o polu (przycisk, figura oraz wiersz, kolumna i kolor pola).

Struktura klas

Projekt składa się z kilku klas, które są ze sobą powiązane w celu zorganizowania logiki gry w szachy. Poniżej przedstawiona jest struktura klas, ich relacje oraz krótkie opisy.

Chess2DDialog: Główna klasa projektu, dziedzicząca po `wxDialog`, reprezentująca okno gry w szachy. Odpowiada za obsługę interfejsu użytkownika i inicjalizację gry.

Board: Klasa reprezentująca szachownicę. Odpowiada za logikę gry, przechowuje obiekty klasy `Square` reprezentujące pola na szachownicy.

Square: Klasa reprezentująca pole na szachownicy. Zawiera informacje o kolorze pola, wierszu i kolumnie pola oraz referencje do przycisku `wxBitmapButton` i do figury `Piece`.

Piece: Klasa abstrakcyjna, po niej dziedziczą konkretne rodzaje figur szachowych. Zawiera podstawowe metody i pola, takie jak: kolor, czy to czy figura się ruszała. Wprowadza metodę wirtualną dotyczącą zwracania typu figury.

Pawn, Rook, Knight, Bishop, Queen, King: Klasy dziedziczące po klasie `Piece`, reprezentujące poszczególne rodzaje figur szachowych. Każda z tych klas implementuje pola z informacjami na temat danej figury.

Relacje między klasami:

Chess2DDialog zawiera obiekt klasy **Board**, który odpowiada za logikę gry i przechowuje informacje o szachownicy.

Klasa **Board** zarządza tablicą obiektów **Square** reprezentujących pola na szachownicy.

Każdy obiekt **Square** zawiera referencję do obiektu `wxBitmapButton` oraz informacje o obecności figury (**Piece**).

Klasy **Pawn**, **Rook**, **Knight**, **Bishop**, **Queen**, **King** dziedziczą po klasie **Piece** i zawierają specyficzne implementacje metod związanych z ich ruchami.

Ta struktura klas pozwala na uporządkowanie kodu i łatwość rozbudowy gry.

Szczegóły konkretnych klas:

Chess2DDialog

```
1 wxBitmapButton* board[8][8];
2 wxBitmap images[2][13];
3
4 int counter = 1;
5 int whiteOrBlack = 1;
6
7 _B = new Board(1);
```

W konstruktorze klasy tworzony jest obiekt klasy `Board`, który będzie reprezentować szachownicę w grze. Tutaj są wczytywane obrazki do `images[2][13]` oraz tworzone przyciski umieszczane w `board[8][8]`, a także liczniki odpowiadające za liczenie kliknięć i pilnowanie czyj mamy ruch. Zawiera również obsługę zdarzeń związanych z interakcją użytkownika (`OnBitmapButtonClick()`). Także w tym miejscu jest kod odpowiedzialny za wyświetlanie szachownicy.

Board

```
1 class Board
2 {
3     public:
4         Board(int i);
5         ~Board();
6         void cleaning();
7         void creatingSquares();
8         void creatingPieces();
9         void assigningPieces();
10        void restart();
11        void setClicked(int _nrBB);
12        void setDestination(int _nrBB);
13        bool isClickedPiece();
14        bool isDestinationPiece();
15        bool arePiecesSameColor();
16        bool isGoodColorMoving();
17        void updateSquares(Square* _clicked, Square* _destination);
18        bool isKingInside();
19        void wasKingMoving();
20        int whereIsKing();
```

```

21 bool castling();
22 bool pawnBlockedByPieceInFront();
23 void pawnMovesButNothingIsInFront();
24 void pawnTakes();
25 void pawnPromotion();
26 void whereICanMove(Square* clicked);
27 bool isInSetOfMoves();
28 bool isSomethingBetween(Square* _squareOne, Square* _squareTwo,
    int _typeInt);
29 bool isBeatable(Square* _square);
30 bool moveSimulation(Square* clicked, Square* destination);
31 bool isMate();
32 Piece* pieces[2][7];
33 Square* squares[8][8];
34 Square* clicked;
35 Square* destination;
36 Square* whiteKing;
37 Square* blackKing;
38 Square* squareBetween;
39 Square* target;
40 set<int> setOfMoves;
41 list<int> listOfThreats;
42 list<int> listOfDefenders;
43 list<int> listOfHope;
44 list<int> listOfInsanity;
45 };

```

Klasa Board jest centralnym punktem logiki gry, zarządzającym szachownicą. Zawiera tablicę obiektów Square oraz tablicę obiektów Piece. Jest w niej kod odpowiedzialny za ustawienie początkowego rozmieszczenia figur na szachownicy. Klasa Board implementuje logikę ruchu figur, sprawdzając, czy dany ruch jest zgodny z zasadami gry w szachy. Odpowiada za obsługę zdarzeń związanych z interakcją użytkownika na poziomie szachownicy, takich jak kliknięcia na pola. Po wykonaniu ruchu odpowiednio aktualizuje widok.

Square

```

1 class Square
2 {
3     public:
4     friend class Board;
5     Square(int _row, int _col, bool _backgroundColor, wxBitmapButton*
        _button, Piece* _piece = nullptr);
6     ~Square();
7     int getRow();
8     int getCol();
9     bool getBackgroundColor();
10    wxBitmapButton* getButton();
11    Piece* getPiece();
12    void setPiece(Piece* _piece);
13
14    protected:
15    int row;

```



```

16     int col;
17     bool backgroundColor;
18     wxBitmapButton* button;
19     Piece* piece;
20 };

```

Klasa Square reprezentuje pojedyncze pole na szachownicy. Przechowuje informację o kolorze pola na szachownicy (jasne/ciemne), wierszu i kolumnie pola. Zawiera referencję do obiektu klasy Piece reprezentującego figurę. Jeśli pole jest puste, wartość ta to nullptr. Przechowuje referencję do przycisku, który jest powiązany z danym polem na szachownicy.

Piece

```

1  class Piece
2  {
3  public:
4      Piece(bool _color, bool _moved);
5      virtual ~Piece();
6      bool getColor();
7      bool getMoved();
8      void setMoved();
9      virtual int getTypeInt()=0;
10
11  protected:
12      bool color;
13      bool moved;
14 };

```

Klasa Piece stanowi abstrakcję dla poszczególnych rodzajów figur (piona, wieży, skoczek, gońca, hetmana, króla). Klasa posiada konstruktor, który inicjuje kolor figury oraz informację o tym, czy figura wykonała ruch. Posiada metodę virtual int getTypeInt()=0: Jest to metoda czysto wirtualna, zwracająca typ figury jako wartość liczbową (przydatna do indeksach naszych tablic).

Pawn, Rook, Knight, Bishop, Queen, King

```

1  class Pawn:public Piece
2  {
3  public:
4      Pawn(bool _color, bool _moved = false);
5      ~Pawn();
6      int getTypeInt();
7
8  private:
9      int TypeInt = 1;
10 };

```

Klasy te dziedziczą po klasie `Piece` i reprezentują konkretne figury w grze. Poniżej przedstawiono kluczowe elementy tych klas: Posiada informację o tym, czy figura wykonała swój pierwszy ruch w grze. To istotne np. dla reguły pierwszego ruchu piona lub dla roszady Posiada informację (`int`) o typie konkretnej figury. Wszystkie te klasy są identyczne.

Metody w klasie `Board`:

Odpowiedzialne za restartowanie gry oraz usuwanie i tworzenie obiektów:

```
1 void cleaning();
2 void creatingSquares();
3 void creatingPieces();
4 void assigningPieces();
5 void restart();
```

- **`cleaning()`**: Usuwa zmienne tworzone dynamicznie (korzysta z niej destruktor i `restart()`).
- **`creatingSquares()`**: Tworzy obiekty reprezentujące pola na szachownicy.
- **`creatingPieces()`**: Tworzy obiekty reprezentujące figury szachowe.
- **`assigningPieces()`**: Przypisuje figury do odpowiednich pól na szachownicy (pozycje startowe).
- **`restart()`**: Przywraca początkowe ustawienie figur na szachownicy.

Odpowiedzialne za obsługę interakcji użytkownika z szachownicą, kontrolujące kliknięcia, aby zapobiec nieoczekiwanym i niechcianym zachowaniom:

```
1 void setClicked(int _nrBB);
2 void setDestination(int _nrBB);
3 bool isClickedPiece();
4 bool isDestinationPiece();
5 bool arePiecesSameColor();
6 bool isGoodColorMoving();
```

- **`setClicked(int _nrBB)`**: ustawia kliknięte pole
- **`setDestination(int _nrBB)`**: ustawia docelowe pole
- **`isClickedPiece()`**: sprawdza czy w klikniętym polu jest jakaś figura (używana po to żeby sprawdzić czy w wybranym polu jest jakaś figura, którą można się ruszyć)
- **`isDestinationPiece()`**: sprawdza czy w docelowym polu jest jakaś figura
- **`arePiecesSameColor()`**: sprawdza czy w klikniętym i docelowym polu są figury tego samego koloru (używana żeby zapobiec ruszaniu się na pola gdzie mamy własne figury)
- **`isGoodColorMoving()`**: sprawdza czy ruch wykonuje kolor, którego jest teraz kolej

Odpowiedzialna za aktualizację szachownicy po ruchu:

```
1 void updateSquares(Square* _clicked, Square* _destination);
```

- **updateSquares(Square* _clicked, Square* _destination)**: ustawia obrazki i figury na polu klikniętym i docelowym

Odpowiedzialne za lokalizację oraz zmianę pozycji króli królów:

```
1 bool isKingInside();
2 void wasKingMoving();
3 int whereIsKing();
```

- **isKingInside()**: sprawdza czy król jest w klikniętym polu
 - **wasKingMoving()**: sprawdza na koniec ruchu czy król się ruszył i aktualizuje jego lokalizację
 - **whereIsKing()**: zwraca pozycję króla strony, która jest teraz na ruchu
- Odpowiedzialna za roszadę**

```
1 bool castling();
```

- **castling()**: sprawdza czy w danym ruchu roszada miała miejsce, jeśli nie miała lub się udała to zwraca false, jeśli nie można było jej zrobić zwraca true
- Odpowiedzialne za ruch i bicie (przez niego) piona oraz jego promocje:**

```
1 bool pawnBlockedByPieceInFront();
2 void pawnMovesButNothingIsInFront();
3 void pawnTakes();
4 void pawnPromotion();
```

- **pawnBlockedByPieceInFront()**: sprawdza czy ruch pion jest blokowany przez figury przed nim
- **pawnMovesButNothingIsInFront()**: wykonuje ruch piona, gdy nic przed nim nie stoi
- **pawnTakes()**: wykonuje bicie po przekątnej gdy są tam jakieś figury przeciwnika
- **pawnPromotion()**: sprawdza czy pion znajduje się w pierwszym rzędzie u przeciwnika wtedy zamienia go w hetmana

Odpowiedzialne za sprawdzanie legalnych ruchów dla każdej figury:

```
1 void whereICanMove();
2 bool isInSetOfMoves();
3 bool isSomethingBetween(Square* _squareOne, Square* _squareTwo, int _typeInt);
```

- **whereICanMove()**: tworzy zbiór legalnych ruchów dla figury wybranej przez gracza
 - **isInSetOfMoves()**: sprawdza czy ruch wybrany przez gracza jest w zbiorze legalnych ruchów dla danej figury
 - **isSomethingBetween(Square* _squareOne, Square* _squareTwo, int _typeInt)**: sprawdza czy między dwoma polami jest jakaś figura (używane do sprawdzenia czy nic nie stoi na drodze na dane pole)
- Odpowiedzialne za szachowanie i symulacje ruchów**

```

1 bool isBeatable(Square* _square);
2 bool moveSimulation();

```

- **isBeatable(Square* _square)**: sprawdza czy wybrana figura jest podbiciem przez jakąś figure (używane głównie do sprawdzania szachowania króla)
- **moveSimulation()**: przeprowadza symulację ruchu (potem cofa wprowadzone zmiany) w sytuacji gdy król jest pod szachem, jeśli ruch gracza sprawia, że król nie jest już pod szachem zwraca true, w przeciwnym wypadku false

Przeprowadzenie przez rozgrywkę

Obsługa przycisków (OnBitmapButton1Click)

```

1 void Chess2DDialog::OnBitmapButton1Click(wxCommandEvent& event){
2     int nrBB = event.GetId() - 100;
3
4     if(counter%2 != 0){
5         _B->setClicked(nrBB);
6         if(!(_B->isClickedPiece()) || !(_B->isGoodColorMoving())){
7             return;
8         }
9         counter++;
10        return;
11    }
12
13    if(counter%2 == 0){
14        _B->setDestination(nrBB);
15        if(_B->isDestinationPiece() && _B->arePiecesSameColor()){
16            counter--;
17            return;
18        }
19        _B->whereICanMove();
20
21        if(!_B->isInSetOfMoves()){
22            counter--;
23            return;
24        }
25        if(_B->isSomethingBetween(_B->clicked, _B->destination, _B->
26            clicked->getPiece()->getTypeInt())){
27            counter--;
28            return;
29        }
30        if(_B->moveSimulation()){
31            wxLogMessage("Tutaj nie wolno");
32            counter--;
33            return;
34        }
35
36        if(_B->castling()){
37            counter--;
38            return;

```

```

39     }
40     _B->updateSquares(_B->clicked, _B->destination);
41
42     _B->pawnPromotion();
43     _B->wasKingMoving();
44
45     whiteOrBlack = (whiteOrBlack + 1)%2;
46
47     if( _B->isMate() == true ){
48         wxLogMessage("Mat");
49     }
50     counter++;
51     return;
52 }
53 }

```

Funkcja OnBitmapButton1Click w każdym kliknięciu zapisuje nrBB (czyli numer bitmapy indywidualny dla każdego przycisku) a następnie sprawdza po kolei warunki z taką zasadą, że jeśli coś jest nie tak to wracamy do kliknięcia 1. W kliknięciu 1, poprzez wskaźnik do clicked zapisuje informacje o klikniętym Squarze. Następnie sprawdza po kolei czy na klikniętym polu znajdowała się figura a potem czy ma ona odpowiedni kolor (czyli czy dobra osoba się rusza). Kolejność jest ważna, ponieważ bez sprawdzenia czy na klikniętym polu jest figura mogłoby dojść do sytuacji, że chcemy użyć metody z klasy Piece na polu, w którym Piece-a nie ma i program zakończyłby działanie. Jeśli wszystko się zgadza zwiększamy counter o 1 i wychodzimy z metody. Przy kliknięciu 2 tak jak w 1 przekazujemy wskaźnik tylko tym razem do destination. Sprawdzamy wpierw czy kliknięte miejsce ma figurę a jeśli ma to sprawdzamy czy jest tego samego koloru. Jeśli jednak nie było z tym problemu to do zbioru poprzez funkcję whereICanMove() zapisujemy możliwe ruchy nie uwzględniając tego, że figura może zostać zablokowana. Następnie sprawdzamy czy destination w ogóle zawiera się w naszym zbiorze ruchów. Jeśli tak to następnie sprawdzamy czy jest coś między polem startowym a docelowym a jeśli i to przejdzie bez problemów to wykonywana jest symulacja ruchu, która ma sprawdzić czy po ruchu nasz król nie będzie pod szachem. Pod koniec sprawdzamy czy może ktoś niechciałby zrobić roszady. Uaktualniamy pozycje, sprawdzamy czy na ostatnich liniach nie stoją piony, które chciałyby dostać awans. Jeśli ruszał się król to poprzez metodę wasKingMoving() sprawdzamy jego kolor i w zależności od tego uaktualniamy blackKing lub whiteKing, dzięki którym możemy na bieżąco sprawdzać gdzie znajdują się króle na planszy. Na koniec naszego ruchu sprawdzamy czy nie zmatowaliśmy przeciwnika.

Wybrane metody z Board.cpp

updateSquares(Square* _clicked, Square* _destination)

```
1 void Board::updateSquares(Square* _clicked, Square* _destination){
2     this->destination->setPiece(this->clicked->getPiece());
3     this->destination->getButton()->SetBitmap(images[this->clicked->
        getPiece()->getColor()][this->clicked->getPiece()->getTypeInt
        ()+1-this->destination->getBackgroundColor()]);
4     this->destination->getPiece()->setMoved();
5     this->clicked->setPiece(nullptr);
6     this->clicked->getButton()->SetBitmap(images[this->clicked->
        getBackgroundColor()][0]);
7 }
```

Metoda działa w taki sposób, że najpierw w miejscu docelowym ustawiamy jaką figurę będzie stać na tym polu, następnie obrazek figury, którą chcemy się ruszyć. Wykorzystujemy tutaj fakt, że w tablicy `images[a][b]` a odpowiada za kolor figury a b za to jaką to figura oraz czy stoi na polu białym czy czarnym gdzie w naszej tablicy obrazki na białym i czarnym polu znajdują się od siebie w odległości 1. Następnie ustawiamy figurze, że się ruszyła a na koniec usuwamy ślady naszej figury z pola startowego.

castling()

```
1 bool Board::castling(){
2     if(this->clicked != this->squares[whereIsKing()/8][whereIsKing()
        %8]){
3         return 0;
4     }
5
6     if(abs((this->destination->getCol() - this->squares[whereIsKing()
        /8][whereIsKing()%8]->getCol())) == 2){
7         if(this->destination->getCol() == 2){
8             if(this->squares[this->destination->getRow()][0]->getPiece()
                != nullptr && this->squares[this->destination->getRow()
                ][0]->getPiece()->getMoved() == false){
9                 if(this->isSomethingBetween(this->clicked, this->squares[
                    this->destination->getRow()][0], 3) == false){
10                    this->squares[this->destination->getRow()][3]->setPiece(
                        pieces[this->clicked->getPiece()->getColor()][1]);
11                    if(!(this->isBeatable(this->clicked)) && !(this->
                        isBeatable(this->squares[this->destination->getRow()
                        ][3]))){
12                        this->squares[this->destination->getRow()][3]->setPiece(
                            (this->squares[this->destination->getRow()][0]->
                            getPiece()));
13                        this->squares[this->destination->getRow()][3]->
                            getButton()->SetBitmap(images[this->squares[this->
                            destination->getRow()][0]->getPiece()->getColor()][
                            this->squares[this->destination->getRow()][0]->
                            getPiece()->getTypeInt()+1-this->squares[this->
                            destination->getRow()][3]->getBackgroundColor()]);
14                        this->squares[this->destination->getRow()][3]->getPiece()
                            ->setMoved();

```

```

15         this->squares[this->destination->getRow()][0]->setPiece
           (nullptr);
16         this->squares[this->destination->getRow()][0]->
           getButton()->SetBitmap(images[this->squares[this->
           destination->getRow()][0]->getBackgroundColor()
           ][0]);
17         return 0;
18     }return 1;
19
20     }return 1;
21
22     }return 1;
23
24 }else if(this->destination->getCol() == 6){
25     if(this->squares[this->destination->getRow()][7]->getPiece()
       != nullptr && this->squares[this->destination->getRow()
       ][7]->getPiece()->getMoved() == false){
26         if(this->isSomethingBetween(this->clicked, this->squares[
           this->destination->getRow()][7], 3) == false){
27             this->squares[this->destination->getRow()][5]->setPiece(
               pieces[this->clicked->getPiece()->getColor()][1]);
28             if(!(this->isBeatable(this->clicked)) && !(this->
               isBeatable(this->squares[this->destination->getRow()
               ][5]))){
29                 this->squares[this->destination->getRow()][5]->setPiece
                   (this->squares[this->destination->getRow()][7]->
                   getPiece());
30                 this->squares[this->destination->getRow()][5]->
                   getButton()->SetBitmap(images[this->squares[this->
                   destination->getRow()][7]->getPiece()->getColor()][
                   this->squares[this->destination->getRow()][7]->
                   getPiece()->getTypeInt()+1-this->squares[this->
                   destination->getRow()][5]->getBackgroundColor()]);
31                 this->squares[this->destination->getRow()][5]->getPiece
                   ()->setMoved();
32                 this->squares[this->destination->getRow()][7]->setPiece
                   (nullptr);
33                 this->squares[this->destination->getRow()][7]->
                   getButton()->SetBitmap(images[this->squares[this->
                   destination->getRow()][7]->getBackgroundColor()
                   ][0]);
34                 return 0;
35             }return 1;
36
37             }return 1;
38
39             }return 1;
40
41         }return 1;
42     }
43     return 0;
44 }

```

Na początek sprawdzamy czy w ogóle próbowaliśmy ruszyć się królem.

Następnie sprawdzamy w warunkach kolejno:

- Czy pole docelowe znajduje się w odległości 2 od króla - Jest to jedyna taka

sytuacja gdy król może poruszyć się o 2 pola

- Czy król próbuje zrobić roszadę długą - Czyli czy kolumna docelowa ma numer 2 (numerując od 0)
- Czy po drugiej stronie w ogóle stoi wieża z którą moglibyśmy zrobić roszadę oraz czy dla wieży to będzie pierwszy ruch
- Czy pomiędzy królem a wieżą jest pusta przestrzeń - z pomocą metody `isSomethingBetween`

Następnie w pole, w którym znajdzie się wieża po roszadzie wpisujemy wieżę by móc dla tego pola wykonać metodę `isBeatable` i sprawdzamy czy pola, na których był lub przez, które będzie przechodził król są szachowane. Jeśli wszystko jest jak powinno to następuje roszada. To samo wykonuje się dla roszady krótkiej tyle, że sprawdzamy inne pola.

`isSomethingBetween(Square* _squareOne, Square* _squareTwo, int _typeInt)`

```
1 bool Board::isSomethingBetween(Square* _squareOne, Square*
  _squareTwo, int _typeInt){
2     listOfDefenders.clear();
3     int _col = _squareOne->getCol();
4     int _row = _squareOne->getRow();
5     int wsp_col = - (_col - _squareTwo->getCol())/abs(_col -
      _squareTwo->getCol());
6     this->squareBetween = squares[this->whereIsKing()/8][this->
      whereIsKing()%8];
7     switch(_typeInt){
8         case 1:
9             if(abs(_row - _squareTwo->getRow()) == 2 && this->clicked->
              getPiece()->getColor() == 1){
10                 if(this->squares[ 5 ][_col]->getPiece() != nullptr){
11                     return true;
12                 }
13             }
14             if(abs(_row - _squareTwo->getRow()) == 2 && this->clicked->
              getPiece()->getColor() == 0){
15                 if(this->squares[ 2 ][_col]->getPiece() != nullptr){
16                     return true;
17                 }
18             }
19             return false;
20             break;
21         case 3:
22             if(_col - _squareTwo->getCol() == 0){
23                 for(int row = _row - (_row - _squareTwo->getRow())/abs(_row -
                    _squareTwo->getRow()); abs(row - _squareTwo->getRow())
                    >0; row = row - (row - _squareTwo->getRow())/abs(row -
                    _squareTwo->getRow())){
24                     this->listOfDefenders.push_back(row * 8 + _col);
25                     if(this->squares[row][_col]->getPiece() != nullptr){
26                         this->squareBetween = this->squares[row][_col];
27                         return true;
28                     }
29                 }
30             }
31             return false;
```



```

31     }
32     if(_row - _squareTwo->getRow() == 0){
33         for(int col = _col - (_col - _squareTwo->getCol())/abs(_col -
            _squareTwo->getCol()); abs(col - _squareTwo->getCol())
            >0; col = col - (col - _squareTwo->getCol())/abs(col -
            _squareTwo->getCol())){
34             this->listOfDefenders.push_back(_row * 8 + col);
35             if(this->squares[_row][col]->getPiece() != nullptr){
36                 this->squareBetween = this->squares[_row][col];
37                 return true;
38             }
39         }
40         return false;
41     }
42     break;
43     case 7:
44         for(int wsp = _row - (_row - _squareTwo->getRow())/abs(_row -
            _squareTwo->getRow()); abs(wsp - _squareTwo->getRow()) >0;
            wsp = wsp - (wsp - _squareTwo->getRow())/abs(wsp -
            _squareTwo->getRow())){
45             this->listOfDefenders.push_back(wsp * 8 + _col + wsp_col);
46             if(this->squares[wsp][_col + wsp_col]->getPiece() != nullptr)
            {
47                 this->squareBetween = this->squares[wsp][_col + wsp_col];
48                 return true;
49             }else{
50                 _col = wsp_col + _col;
51             }
52         }
53         return false;
54         break;
55         case 9:
56         wsp_col = - (_col - _squareTwo->getCol())/abs(_col - _squareTwo
            ->getCol());
57         if(_col - _squareTwo->getCol() == 0){
58             for(int row = _row - (_row - _squareTwo->getRow())/abs(_row -
                _squareTwo->getRow()); abs(row - _squareTwo->getRow())
                >0; row = row - (row - _squareTwo->getRow())/abs(row -
                _squareTwo->getRow())){
59                 this->listOfDefenders.push_back(row * 8 + _col);
60                 if(this->squares[row][_col]->getPiece() != nullptr){
61                     return true;
62                 }
63             }
64             return false;
65         }
66         if(_row - _squareTwo->getRow() == 0){
67             for(int col = _col - (_col - _squareTwo->getCol())/abs(_col -
                _squareTwo->getCol()); abs(col - _squareTwo->getCol())
                >0; col = col - (col - _squareTwo->getCol())/abs(col -
                _squareTwo->getCol())){
68                 this->listOfDefenders.push_back(_row * 8 + col);
69                 if(this->squares[_row][col]->getPiece() != nullptr){
70                     return true;
71                 }
72             }
73             return false;

```

```

74     }
75     for(int wsp = _row - (_row - _squareTwo->getRow())/abs(_row -
        _squareTwo->getRow()); abs(wsp - _squareTwo->getRow()) > 0;
        wsp = wsp - (wsp - _squareTwo->getRow())/abs(wsp -
        _squareTwo->getRow())){
76         this->listOfDefenders.push_back(wsp * 8 + _col + wsp_col);
77         if(this->squares[wsp][_col + wsp_col]->getPiece() != nullptr)
            {
78             return true;
79         }else{
80             _col = wsp_col + _col;
81         }
82     }
83     return false;
84     break;
85
86     default:
87         return false;
88 }
89 return false;
90 }

```

W tej metodzie robimy wiele rzeczy na raz. Była ona robiona z myślą, żeby sprawdzać czy jak chcemy się gdzieś ruszyć to nie stoi nam coś na drodze jednak potem uznaliśmy, że może nam się przydać żeby sprawdzić czy coś nie atakuje króla. Dlatego w momencie gdy funkcja natknie się na jakieś pole, to zapisuje je do squareBetween. Zapisuje ona również do listy listOfDefenders wszystkie pola pomiędzy polem startowym a końcowym. Co do działania funkcji to w argumentach przekazujemy jej pole początkowe oraz końcowe oraz jako kto chcemy się poruszyć (jaką figurą). W zależności od naszego _typeInt wchodzimy do innego case-a w switchu i kolejno dla piona sprawdzamy tylko sytuację gdy może ruszyć się o 2, dla wieży sprawdzamy czy ruch odbywa się na kierunku poziomym czy pionowym a następnie odpowiednio czy góra/prawo czy dół/lewo. Dla gońca sprawdzamy po której przekątnej będziemy sprawdzać na zasadzie +/- 1 na kierunku poziomym i +/- 1 pionowym. Hetman to połączona wieża i goniec. Pozostałe figury nie mają takich ograniczeń.

isBeatable(Square* _square)

```

1  bool Board::isBeatable(Square* _square){
2      this->listOfThreats.clear();
3
4      int _Row = _square->getRow();
5      int _Col = _square->getCol();
6
7      //is Rook checking us
8      for(int i = 0; i < 8; i = i+7){
9          if(_Row != i && isSomethingBetween(squares[_Row][_Col], squares
            [i][_Col], 3)){
10             if(this->squareBetween->getPiece()->getTypeInt() == 3 || this
                ->squareBetween->getPiece()->getTypeInt() == 9){
11                 if(this->squares[_Row][_Col]->getPiece()->getColor() !=
                    this->squareBetween->getPiece()->getColor()){

```

```

12         this->listOfThreats.push_back(this->squareBetween->getRow
13         () * 8 + this->squareBetween->getCol());
14     }
15 }
16 }else{
17     if(_Row != i && this->squares[i][_Col]->getPiece() != nullptr
18         && this->squares[_Row][_Col]->getPiece()->getColor() !=
19         this->squares[i][_Col]->getPiece()->getColor()){
20         if(this->squares[i][_Col]->getPiece()->getTypeInt() == 9 ||
21             this->squares[i][_Col]->getPiece()->getTypeInt() == 3)
22         {
23             this->listOfThreats.push_back(this->squares[i][_Col]->
24             getRow() * 8 + this->squares[i][_Col]->getCol());
25         }
26     }
27 }
28 }
29 if(_Col != i && isSomethingBetween(squares[_Row][_Col], squares
30 [_Row][i], 3)){
31     if(this->squareBetween->getPiece()->getTypeInt() == 3 || this
32     ->squareBetween->getPiece()->getTypeInt() == 9){
33         if(this->squares[_Row][_Col]->getPiece()->getColor() !=
34             this->squareBetween->getPiece()->getColor()){
35             this->listOfThreats.push_back(this->squareBetween->getRow
36             () * 8 + this->squareBetween->getCol());
37         }
38     }
39 }else{
40     if(_Col != i && this->squares[_Row][i]->getPiece() != nullptr
41         && this->squares[_Row][_Col]->getPiece()->getColor() !=
42         this->squares[_Row][i]->getPiece()->getColor()){
43         if(this->squares[_Row][i]->getPiece()->getTypeInt() == 9 ||
44             this->squares[_Row][i]->getPiece()->getTypeInt() == 3)
45         {
46             this->listOfThreats.push_back(this->squares[_Row][i]->
47             getRow() * 8 + this->squares[_Row][i]->getCol());
48         }
49     }
50 }
51 }
52
53 //is Bishop checking us
54 for(int i = 1; i < 8; i++){
55     if((_Row) - i >= 0 && _Col - i >= 0){
56         if(squares[_Row - i][_Col - i]->getPiece() != nullptr){
57             if(this->squares[_Row][_Col]->getPiece()->getColor() != this
58             ->squares[_Row - i][_Col - i]->getPiece()->getColor()){
59                 if(this->squares[_Row - i][_Col - i]->getPiece()->getTypeInt
60                 () == 7 || this->squares[_Row - i][_Col - i]->getPiece()
61                 ->getTypeInt() == 9){
62                     this->listOfThreats.push_back(this->squares[_Row - i][
63                     _Col - i]->getRow() * 8 + this->squares[_Row - i][
64                     _Col - i]->getCol());
65                     continue;
66                 }
67             }
68         }
69     }
70 }

```

```

49         i = i+10;
50     }
51 }
52 }
53
54 for(int i = 1; i < 8; i++){
55     if((_Row)- i >=0 && _Col+i <=7){
56         if(squares[_Row-i][_Col+i]->getPiece() != nullptr){
57             if(this->squares[_Row][_Col]->getPiece()->getColor() != this
58                 ->squares[_Row-i][_Col+i]->getPiece()->getColor()){
59                 if(this->squares[_Row-i][_Col+i]->getPiece()->getTypeInt
60                     () == 7 || this->squares[_Row-i][_Col+i]->getPiece()
61                     ->getTypeInt() == 9){
62                     this->listOfThreats.push_back(this->squares[_Row - i][
63                         _Col + i]->getRow() * 8 + this->squares[_Row - i][
64                         _Col + i]->getCol());
65                     continue;
66                 }
67             }
68         }
69         i = i+10;
70     }
71 }
72 }
73
74 for(int i = 1; i < 8; i++){
75     if((_Row)+ i <=7 && _Col+i <=7){
76         if(squares[_Row+i][_Col+i]->getPiece() != nullptr){
77             if(this->squares[_Row][_Col]->getPiece()->getColor() != this
78                 ->squares[_Row+i][_Col+i]->getPiece()->getColor()){
79                 if(this->squares[_Row+i][_Col+i]->getPiece()->getTypeInt
80                     () == 7 || this->squares[_Row+i][_Col+i]->getPiece()
81                     ->getTypeInt() == 9){
82                 this->listOfThreats.push_back(this->squares[_Row + i][
83                     _Col + i]->getRow() * 8 + this->squares[_Row + i][
84                     _Col + i]->getCol());
85                 continue;
86             }
87         }
88         i = i+10;
89     }
90 }
91
92 for(int i = 1; i < 8; i++){
93     if((_Row)+ i <=7 && _Col-i >=0){
94         if(squares[_Row+i][_Col-i]->getPiece() != nullptr){
95             if(this->squares[_Row][_Col]->getPiece()->getColor() != this
96                 ->squares[_Row+i][_Col-i]->getPiece()->getColor()){
97                 if(this->squares[_Row+i][_Col-i]->getPiece()->getTypeInt
98                     () == 7 || this->squares[_Row+i][_Col-i]->getPiece()
99                     ->getTypeInt() == 9){
100                 this->listOfThreats.push_back(this->squares[_Row + i][
101                     _Col - i]->getRow() * 8 + this->squares[_Row + i][
102                     _Col - i]->getCol());
103                 continue;
104             }
105         }
106     }
107 }

```

```

91         i = i+10;
92     }
93 }
94 }
95
96 //is Pawn checking us
97 for(int i = -1; i < 2; i = i + 2){
98     if(_Row == 0){
99         }else{
100             if(_Col - i >= 0 && _Col - i <= 7 && squares[_Row - 1][_Col -
101                 i]->getPiece() != nullptr){
102                 if(this->squares[_Row][_Col]->getPiece()->getColor() !=
103                     this->squares[_Row - 1][_Col - i]->getPiece()->getColor
104                         ()){
105                     if(this->squares[_Row - 1][_Col - i]->getPiece()->
106                         getTypeInt() == 1 && this->squares[_Row - 1][_Col - i
107                             ]->getPiece()->getColor() == 0){
108                         this->listOfThreats.push_back(this->squares[_Row - 1][
109                             _Col - i]->getRow() * 8 + this->squares[_Row - 1][
110                                 _Col - i]->getCol());
111                     }
112                 }
113             }
114         }
115     }
116 }
117
118 for(int i = -1; i < 2; i = i + 2){
119     if(_Row == 7){
120         }else{
121             if(_Col - i >= 0 && _Col - i <= 7 && squares[_Row + 1][_Col -
122                 i]->getPiece() != nullptr){
123                 if(this->squares[_Row][_Col]->getPiece()->getColor() != this
124                     ->squares[_Row + 1][_Col - i]->getPiece()->getColor()){
125                     if(this->squares[_Row + 1][_Col - i]->getPiece()->
126                         getTypeInt() == 1 && this->squares[_Row + 1][_Col - i
127                             ]->getPiece()->getColor() == 1){
128                         this->listOfThreats.push_back(this->squares[_Row + 1][
129                             _Col - i]->getRow() * 8 + this->squares[_Row + 1][
130                                 _Col - i]->getCol());
131                     }
132                 }
133             }
134         }
135     }
136 }
137
138 // is Knight checking us
139 for(int i = 0; i < 5; i = i+4){
140     for(int j = 0; j < 3; j = j+2){
141         if(_Row + i <= 9 && _Row + i >=2 && _Col + j <= 8 && _Col +
142             j >=1 ){
143             if(squares[_Row + i - 2][_Col + j - 1]->getPiece() !=
144                 nullptr){
145                 if(this->squares[_Row][_Col]->getPiece()->getColor() !=
146                     this->squares[_Row + i - 2][_Col + j - 1]->getPiece()
147                         ->getColor()){
148                     if(this->squares[_Row + i - 2][_Col + j - 1]->getPiece
149                         ()->getTypeInt() == 5){

```

```

130         this->listOfThreats.push_back(this->squares[_Row + i
        - 2][_Col + j - 1]->getRow() * 8 + this->squares[
        _Row + i - 2][_Col + j - 1]->getCol());
131     }
132 }
133 }
134 }
135 if(_Row + j <= 8 && _Row + j >=1 && _Col + i <= 9 && _Col + i
    >= 2){
136     if(squares[_Row + j - 1][_Col + i - 2]->getPiece() !=
        nullptr){
137         if(this->squares[_Row][_Col]->getPiece()->getColor() !=
            this->squares[_Row + j - 1][_Col + i - 2]->getPiece()
            ->getColor()){
138             if(this->squares[_Row + j - 1][_Col + i - 2]->getPiece
                ()->getTypeInt() == 5){
139                 this->listOfThreats.push_back(this->squares[_Row + j
                    - 1][_Col + i - 2]->getRow() * 8 + this->squares[
                    _Row + j - 1][_Col + i - 2]->getCol());
140             }
141         }
142     }
143 }
144 }
145 }
146
147 // is King checking us
148 if(_square->getPiece()->getColor() == 0){
149     if(abs(this->whiteKing->getCol() - _square->getCol())<=1 && abs
        (this->whiteKing->getRow() - _square->getRow())<=1){
150         this->listOfThreats.push_back(this->whiteKing->getRow() * 8 +
            this->whiteKing->getCol());
151     }
152 }else{
153     if(abs(this->blackKing->getCol() - _square->getCol())<=1 && abs
        (this->blackKing->getRow() - _square->getRow())<=1){
154         this->listOfThreats.push_back(this->blackKing->getRow() * 8 +
            this->blackKing->getCol());
155     }
156 }
157 if(!this->listOfThreats.empty()){
158     return true;
159 }
160 return false;
161 }

```

Metoda działa w taki sposób, że dla pola, które chcemy sprawdzić czy jest atakowane wykonuje po kolei oraz w każdym kierunku isSomethingBetween z type intem odpowiednim w zależności od tego czego szukamy np. przed nami w oddali będziemy poszukiwać hetmana lub wieży zaś po skosie będziemy szukać hetmana lub gońca. Każde pole, z którego można zbić wybrane przez nas pole zapisywane jest do listOfThreats. Wiele podobnych rzeczy co w isSomethingBetween więc nie będziemy się nad nimi rozwodzić.

moveSimulation(Square* _clicked, Square* _destination)

```

1  bool Board::moveSimulation(Square* _clicked, Square* _destination){
2      bool x = false;
3      if((this->blackKing == _clicked) + (this->whiteKing == _clicked)
4          == 1){
5          if(_clicked->getPiece()->getColor()){
6              this->whiteKing = _destination;
7          }else{
8              this->blackKing = _destination;
9          }
10         Square* pieceStorage = new Square(-1, -1, 0, board[0][0], nullptr);
11         if(_destination->getPiece() != nullptr){
12             pieceStorage->setPiece(_destination->getPiece());
13         }
14         _destination->setPiece(_clicked->getPiece());
15         _clicked->setPiece(nullptr);
16         if(this->isBeatable(this->squares[whereIsKing()/8][whereIsKing()
17             %8])){
18             x = true;
19         }
20         _clicked->setPiece(_destination->getPiece());
21         if(pieceStorage->getPiece() == nullptr){
22             _destination->setPiece(nullptr);
23         }else{
24             _destination->setPiece(pieceStorage->getPiece());
25         }
26         if((this->blackKing == _destination) + (this->whiteKing ==
27             _destination) == 1){
28             if(_clicked->getPiece()->getColor()){
29                 this->whiteKing = _clicked;
30             }else{
31                 this->blackKing = _clicked;
32             }
33         }
34         delete pieceStorage;
35         return x;
36     }

```

Metoda działa w taki sposób, że najpierw w miejscu docelowym ustawiamy jaką figurę będzie stać na tym polu, następnie obrazek figury, którą chcemy się ruszyć. Wykorzystujemy tutaj fakt, że w tablicy `images[a][b]` `a` odpowiada za kolor figury a `b` za to jaka to figura oraz czy stoi na polu białym czy czarnym gdzie w naszej tablicy obrazki na białym i czarnym polu znajdują się od siebie w odległości 1. Następnie ustawiamy figurze, że się ruszyła a na koniec usuwamy ślady naszej figury z pola startowego.