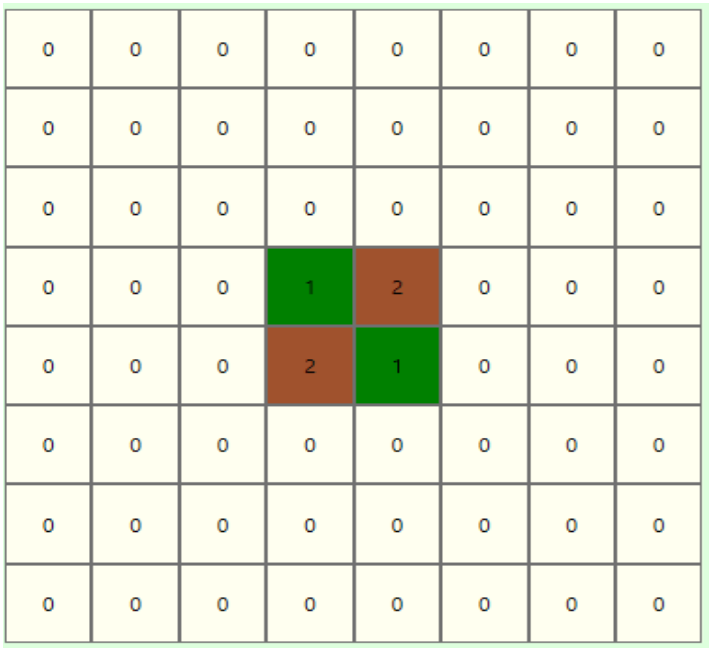
## Gra Reversi (*Othello*). Model i widok

Gra *Reversi* jest rozgrywana na planszy o wymiarach 8×8, na której w chwili rozpoczęcia gry zajęte są cztery środkowe pola, tworzące małą szachownicę. Zasady gry są następujące:

1. Gracze zajmują na przemian pola planszy, przejmując przy tym wszystkie pola przeciwnika znajdujące się między nowo zajętym polem a innymi polami gracza wykonującego ruch. Między nimi nie może być pól pustych.
2. Celem gry jest zdobycie większej liczby pól niż przeciwnik.
3. Gracz może zająć jedynie takie pole, które pozwoli mu przejąć przynajmniej jedno pole przeciwnika. Jeżeli takiego pola nie ma, musi oddać ruch.

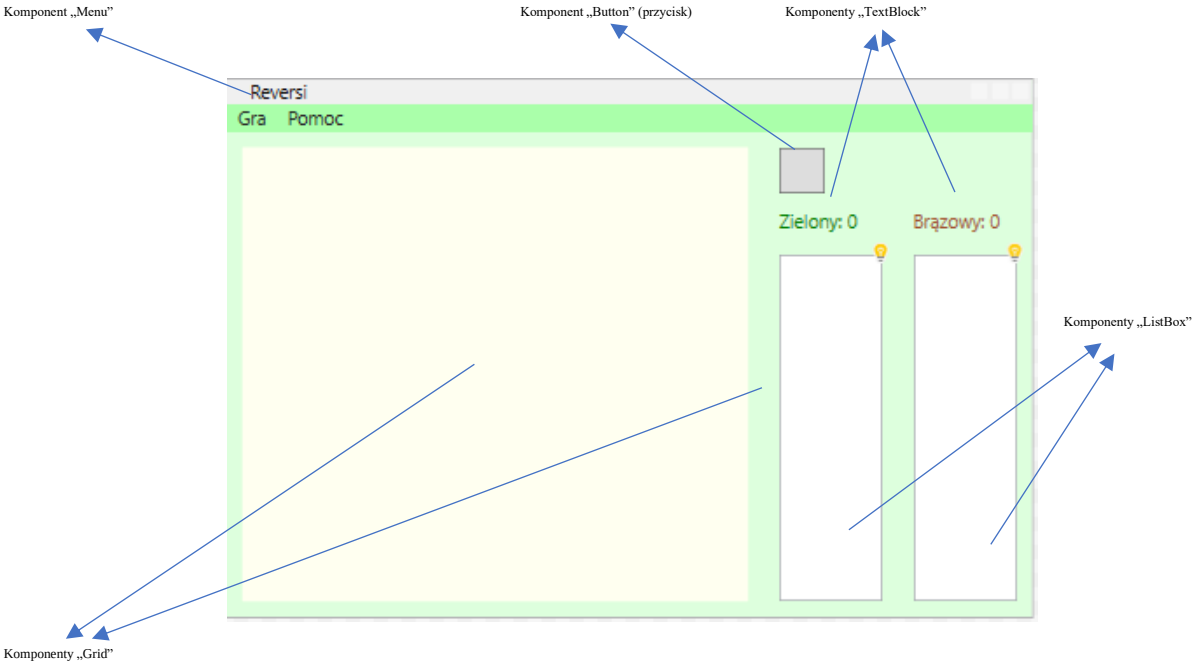



Gra kończy się, gdy wszystkie pola są zajęte lub gdy żaden z graczy nie może wykonać ruchu. O zwycięstwie decyduje liczba zajętych pól.

## Widok - Graficzna prezentacja planszy

Zacznijmy od utworzenia nowego projektu *WPF App* o nazwie *Reversi*.

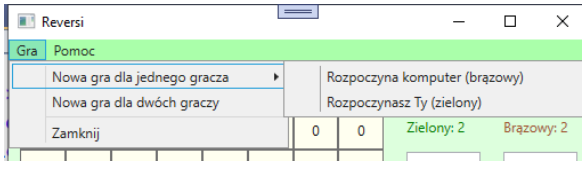
Pierwszym krokiem będzie zaprojektowanie widoku, który umożliwia wyświetlenie zawartości planszy w oknie aplikacji. W oknie tym oprócz planszy powinno być także miejsce na prezentację liczb kamieni na planszy należących do poszczególnych graczy oraz na listę ich dotychczasowych ruchów. Całość powinna oczywiście dostosowywać się do aktualnego rozmiaru okna.

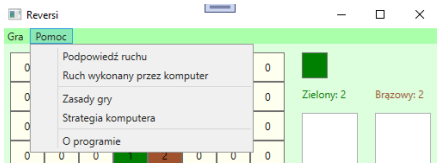



Siatkę (Grid) planszy oraz Menu umieścimy w pojemniku typu DockPanel

```
<DockPanel>
        <Menu>
        …
        </Menu>
        <Grid>
        …
         </Grid>
</DockPanel>
```

Komponent Menu z dwustopniową strukturą elementów MenuItem. Ich atrybuty Header określają etykietę widoczną w menu, a atrybuty Click wskazują na metody zdarzeniowe:



```
<Menu DockPanel.Dock="Top" Background="#AAFFAA">

          <MenuItem Header="Gra">
              <MenuItem Header="Nowa gra dla jednego gracza">
                  <MenuItem Header="Rozpoczyna komputer (brązowy)"
Click="MenuItem_NowaGraDla1Gracza_RozpoczynaKomputer_Click" />
                  <MenuItem Header="Rozpoczynasz Ty (zielony)"
Click="MenuItem_NowaGraDla1Gracza_Click" />
              </MenuItem>
              <MenuItem Header="Nowa gra dla dwóch graczy"
Click="MenuItem_NowaGraDla2Graczy_Click" />
                <Separator />
                <MenuItem Header="Zamknij" Click="MenuItem_Zamknij_Click" />
          </MenuItem>
          <MenuItem Header="Pomoc">
              <MenuItem Header="Podpowiedź ruchu"
Click="MenuItem_PodpowiedźRuchu_Click" />
              <MenuItem Header="Ruch wykonany przez komputer"
Click="MenuItem_RuchWykonanyPrzezKomputer_Click" />
              <Separator />
              <MenuItem Header="Zasady gry" Click="MenuItem_ZasadyGry_Click" />
              <MenuItem Header="Strategia komputera"
Click="MenuItem_StrategiaKomputera_Click" />
              <Separator />
              <MenuItem Header="O programie" Click="MenuItem_Informacje_Click" />
            </MenuItem>
        </Menu>
```

Grid (siatka) – składa się z dwóch kolumn. Pierwsza kolumna jest dwa razy szersza niż druga:

```
<Grid>
     <Grid.ColumnDefinitions>
         <ColumnDefinition Width="2*" />
         <ColumnDefinition Width="1*" />
     </Grid.ColumnDefinitions>
```

W pierwszej kolumnie umieszczamy nową siatkę (Grid), która będzie planszą do gry.

Druga kolumna składa się z dwóch „podkolumn” (o takiej samej szerokości)

```
<Grid Grid.Column="1" Margin="0,0,0,0">
                    <Grid.ColumnDefinitions>
                        <ColumnDefinition Width="*" />
                        <ColumnDefinition Width="*" />
                    </Grid.ColumnDefinitions>
```

i trzech wierszy (o wysokościach odpowiednio: 50, 20, „reszta”).

```
 <Grid.RowDefinitions>
     <RowDefinition Height="50" />
     <RowDefinition Height="20" />
     <RowDefinition Height="*" />
 </Grid.RowDefinitions>
```

W pierwszym wierszu znajduje się Button (szerokość = wysokość = 30). Wyrównywanie wewnątrz układu strony właściwości: HorizontalAlignment i VerticalAlignment.

```
<Button x:Name="przyciskKolorGracza" Margin="10,10,0,0" Grid.Row="0" Width="30"
                Height="30" VerticalAlignment="Top" HorizontalAlignment="Left" />
```

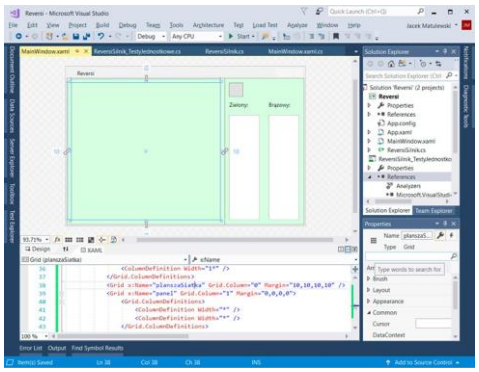
W drugim wierszu znajdują się komponenty TextBlock (w pierwszej i drugiej kolumnie). Komponent TextBlock - zapewnia dobrą wydajność pod kątem wyświetlania maksymalnie kilku wierszy tekstu. Przypisujemy graczom kolory: gracz pierwszy będzie „zielony”, a drugi — „brązowy”.

```
<TextBlock Grid.Row="1" Grid.Column="0" Margin="10,0,0,0" Foreground="Green"
        Text="Zielony: "><Run x:Name="liczbaPólZielony" Text="0"/></TextBlock>
<TextBlock Grid.Row="1" Grid.Column="1" Margin="10,0,0,0" Foreground="Sienna"
        Text="Brązowy: "><Run x:Name="liczbaPólBrązowy" Text="0"/></TextBlock>
```

W trzecim wierszu znajdują się komponenty Listbox (w pierwszej i drugiej kolumnie)

```
<ListBox x:Name="listaRuchówZielony" Grid.Column="0" Grid.Row="2"
                Margin="10,10,10,10" />
<ListBox x:Name="listaRuchówBrązowy" Grid.Column="1" Grid.Row="2"
                 Margin="10,10,10,10" />
```

Projekt interfejsu



Metody zdarzeniowe w kodzie C#.

```
#region Metody zdarzeniowe menu głównego
        //Gra, Nowa gra dla jednego gracza, Rozpoczyna komputer (brązowy)
        private void MenuItem_NowaGraDla1Gracza_RozpoczynaKomputer_Click(object
sender,RoutedEventArgs e)
        {
        }
        //Gra, Nowa gra dla jednego gracza, Rozpoczynasz Ty (zielony)
        private void MenuItem_NowaGraDla1Gracza_Click(object sender, RoutedEventArgs e)
        {
        }
        //Gra, Nowa gra dla dwóch graczy
        private void MenuItem_NowaGraDla2Graczy_Click(object sender, RoutedEventArgs e)
        {
        }
        //Gra, Zamknij
        private void MenuItem_Zamknij_Click(object sender, RoutedEventArgs e)
        {
            Close();
        }
        //Pomoc, Podpowiedź ruchu
        private void MenuItem_PodpowiedźRuchu_Click(object sender, RoutedEventArgs e)
        {
        }
        //Pomoc, Ruch wykonany przez komputer
        private void MenuItem_RuchWykonanyPrzezKomputer_Click(object sender, RoutedEventArgs e)
        {
        }
        //Pomoc, Zasady gry
        private void MenuItem_ZasadyGry_Click(object sender, RoutedEventArgs e)
        {
        }
        //Pomoc, Strategia komputera
        private void MenuItem_StrategiaKomputera_Click(object sender, RoutedEventArgs e)
        {
        }
        //Pomoc, O programie
        private void MenuItem_Informacje_Click(object sender, RoutedEventArgs e)
        {
        }
        #endregion
```

Najważniejszy element interfejsu - plansza, to jedynie siatka. Przyciski, których użyjemy jako pól planszy, będą dynamicznie generowane z kodu C# klasy okna w konstruktorze klasy MainWindow. Na razie ustalmy, że plansza ma wymiary 8x8 (później będziemy definiować rozmiar planszy w modelu).

```
public partial class MainWindow : Window
    {
        private Button[,] plansza;
        public MainWindow()
        {
            InitializeComponent();
            //podział siatki na wiersze i kolumny
            for (int i = 0; i < 8; i++)
                planszaSiatka.ColumnDefinitions.Add(new ColumnDefinition());
            for (int j = 0; j < 8; j++)
                planszaSiatka.RowDefinitions.Add(new RowDefinition());

            //tworzenie przycisków
            plansza = new Button[8, 8];
            for (int i = 0; i < 8; i++)
                for (int j = 0; j < 8; j++)
                {
                    Button przycisk = new Button();
                    przycisk.Margin = new Thickness(0);
                    planszaSiatka.Children.Add(przycisk);
                    Grid.SetColumn(przycisk, i);
                    Grid.SetRow(przycisk, j);
                    plansza[i, j] = przycisk;
                }
        }
    }
```

Sercem projektu będzie klasa ReversiSilnik, która będzie implementować reguły gry i przechowywać bieżący stan planszy. Jej klasą potomną będzie ReversiSilnikAI, która dodatkowo będzie posiadać metody wskazujące najlepszy ruch. To pozwoli nauczyć grać w *Reversi* komputer.

## Model — silnik gry

Do projektu dodajmy dodatkowy plik klasy o nazwie *ReversiSilnik.cs*.

Zacznijmy od pól klasy *ReversiSilnik*. Wymiary planszy:

```
public int SzerokośćPlanszy { get; private set; }
public int WysokośćPlanszy { get; private set; }
```

Rozmiar planszy może być ustalany w argumentach konstruktora, co pozwoli na jej ewentualne zwiększenie ponad standardowe 8×8.

Zdefiniujmy właściwość NumerGraczaWykonującegoNastępnyRuch, z której można będzie odczytać numer gracza, który ma wykonać następny ruch.

```
public int NumerGraczaWykonującegoNastępnyRuch { get; private set; } = 1;
```

Potrzebujemy logicznej reprezentacji planszy z możliwością ustawiania na niej kamieni (zajmowania pól). Każde pole na planszy może mieć trzy stany: może być puste lub zajęte przez kamień jednego lub drugiego gracza.

```
private int[,] plansza;
```

Aby umożliwić dostęp do planszy „z zewnątrz”, zdefiniujmy publiczną metodę PobierzStanPola, pozwalającą na sprawdzenie wartości wybranego pola oraz metodę sprawdzającą poprawność podanej pozycji pola:

```
private bool czyWspółrzędnePolaPrawidłowe(int poziomo, int pionowo)
{
    return poziomo >= 0 && poziomo < SzerokośćPlanszy &&
           pionowo >= 0 && pionowo < WysokośćPlanszy;
}
public int PobierzStanPola(int poziomo, int pionowo)
{
    if (!czyWspółrzędnePolaPrawidłowe(poziomo, pionowo))
        throw new Exception("Nieprawidłowe współrzędne pola");
    return plansza[poziomo, pionowo];
}
```

Zdefiniujmy metodę numerPrzeciwnika wyznaczającą numer gracza - inny niż ten podany w argumencie (w praktyce 1 dla 2 i 2 dla 1).

```
private static int numerPrzeciwnika(int numerGracza)
{
    if (numerGracza == 1) return 2;
    else return 1;
}
```

## Konstruktor klasy

Zadaniem konstruktora jest inicjacja stanu obiektu, co oznacza zainicjowanie jego pól. W naszym przypadku polega ono na utworzeniu planszy (tablicy liczb całkowitych), ustawieniu na niej kamieni zgodnie z zasadami gry i wyznaczeniu gracza wykonującego pierwszy ruch. Za inicjację planszy odpowiedzialna jest metoda czyśćPlanszę. Argumenty konstruktora pozwalają na wskazanie numeru gracza rozpoczynającego oraz wielkości planszy.

```
private void czyśćPlanszę()
{
   for (int i = 0; i < SzerokośćPlanszy; i++)
      for (int j = 0; j < WysokośćPlanszy; j++)
          plansza[i, j] = 0;

   int srodekSzer = SzerokośćPlanszy / 2;
   int srodekWys = WysokośćPlanszy / 2;
   plansza[srodekSzer - 1, srodekWys - 1] = plansza[srodekSzer, srodekWys] = 1;
   plansza[srodekSzer - 1, srodekWys] = plansza[srodekSzer, srodekWys - 1] = 2;
}

public ReversiSilnik(int numerGraczaRozpoczynającego,
                     int szerokośćPlanszy = 8, int wysokośćPlanszy = 8)
{
   if (numerGraczaRozpoczynającego < 1 || numerGraczaRozpoczynającego > 2)
      throw new Exception("Nieprawidłowy numer gracza rozpoczynającego grę");

   SzerokośćPlanszy = szerokośćPlanszy;
   WysokośćPlanszy = wysokośćPlanszy;
   plansza = new int[SzerokośćPlanszy, WysokośćPlanszy];
   czyśćPlanszę();
   NumerGraczaWykonującegoNastępnyRuch = numerGraczaRozpoczynającego;
}
```

## Implementacja zasad gry

Teraz zajmiemy się najważniejszym elementem klasy ReversiSilnik, a mianowicie metodą PołóżKamień pozwalającą na położenie nowego kamienia i zgodne z regułami gry przejęcie pól przeciwnika. Oznacza to implementację przedstawionych wcześniej reguł gry.

A mówią one, że kamień może być ustawiony tylko wówczas, gdy w wyniku jego dodania zostanie przejęte przynajmniej jedno pole należące do przeciwnika. Oznacza to, że w przynajmniej jednym kierunku od wybranego pola musi znajdować się inny kamień gracza wykonującego ruch, ustawiony tak, że pomiędzy nim a wybranym polem wszystkie pola zajmują kamienie przeciwnika. Dodajmy do klasy ReversiSilnik metodę zmieńBieżącegoGracza oraz PołóżKamień.

Sprawdzamy, czy wokół wskazanego pola (w ośmiu kierunkach) znajdują się kamienie bieżącego gracza odseparowane jedynie kamieniami przeciwnika

```
private void zmieńBieżącegoGracza()
{
   NumerGraczaWykonującegoNastępnyRuch =
                numerPrzeciwnika(NumerGraczaWykonującegoNastępnyRuch);
}

protected int PołóżKamień(int poziomo, int pionowo, bool tylkoTest)
{
   //czy współrzędne są prawidłowe
   if (!czyWspółrzędnePolaPrawidłowe(poziomo, pionowo))
       throw new Exception("Nieprawidłowe współrzędne pola");

   //czy pole nie jest już zajęte
   if (plansza[poziomo, pionowo] != 0) return -1;

   int ilePólPrzejętych = 0;

   //pętla po 8 kierunkach
   for (int kierunekPoziomo = -1; kierunekPoziomo <= 1; kierunekPoziomo++)
       for (int kierunekPionowo = -1; kierunekPionowo <= 1; kierunekPionowo++)
       {
          //wymuszenie pominięcia przypadku, gdy obie zmienne są równe 0
          if (kierunekPoziomo == 0 && kierunekPionowo == 0) continue;

          //szukanie kamieni gracza w jednym z 8 kierunków
          int i = poziomo;
          int j = pionowo;
          bool znalezionyKamieńPrzeciwnika = false;
          bool znalezionyKamieńGraczaWykonującegoRuch = false;
          bool znalezionePustePole = false;
          bool osiągniętaKrawędźPlanszy = false;
          do
          {
              i += kierunekPoziomo;
              j += kierunekPionowo;
              if (!czyWspółrzędnePolaPrawidłowe(i, j))
                  osiągniętaKrawędźPlanszy = true;
              if (!osiągniętaKrawędźPlanszy)
              {
```

```
if (plansza[i, j] == NumerGraczaWykonujacegoNastepnyRuch)
                                znalezionyKamieńGraczaWykonujacegoRuch = true;
                if (plansza[i, j] == 0) znalezionePustePole = true;
                if (plansza[i, j] ==
                    numerPrzeciwnika(NumerGraczaWykonujacegoNastepnyRuch))
                    znalezionyKamieńPrzeciwnika = true;
            }
        }
        while (!(osiągniętaKrawędźPlanszy ||
                 znalezionyKamieńGraczaWykonujacegoRuch || nalezionePustePole));

        //sprawdzenie warunku poprawności ruchu
        bool położenieKamieniaJestMożliwe = znalezionyKamieńPrzeciwnika &&
             znalezionyKamieńGraczaWykonujacegoRuch && !znalezionePustePole;

        //"odwrócenie" kamieni w przypadku spełnionego warunku
        if (położenieKamieniaJestMożliwe)
        {
            int maks_indeks =
                 Math.Max(Math.Abs(i - poziomo), Math.Abs(j - pionowo));

            if (!tylkoTest)
            {
                for (int indeks = 0; indeks < maks_indeks; indeks++)
                plansza[poziomo + indeks * kierunekPoziomo,
                 pionowo + indeks * kierunekPionowo] =
                                NumerGraczaWykonujacegoNastepnyRuch;
            }

            ilePólPrzejętych += maks_indeks - 1;
        }
    } //koniec pętli po kierunkach

    //zmiana gracza, jeżeli ruch został wykonany
    if (ilePólPrzejętych > 0 && !tylkoTest)
        zmieńBieżącegoGracza();

  //zmienna ilePólPrzejętych nie uwzględnia dostawionego kamienia
  return ilePólPrzejętych;

}
```

W pierwszej kolejności metoda PołóżKamień sprawdza, czy podane w jej argumentach współrzędne wskazują pole należące do planszy, a następnie czy to pole jest wolne. Jeżeli tak, to przechodzi do sprawdzenia, czy po prawej stronie w poziomie od wybranego pola znajduje się kamień gracza wykonującego ruch, ale jednocześnie czy nie ma tam pustego pola i jest tam przynajmniej jeden kamień przeciwnika. Tak samo sprawdza pozostałe siedem kierunków. Za zmiany kierunków odpowiedzialna jest podwójna pętla for przyjmująca za indeksy zmienne kierunekPoziomo i kierunekPionowo, które mogą mieć wartości -1, 0 i 1. Należy jednak pamiętać o pominięciu przypadku, w którym oba indeksy są jednocześnie równe 0. Wewnątrz wykonywana jest pętla do...while, która kończy się, jeżeli w danym kierunku znaleziony został kamień gracza lub osiągnięta została krawędź planszy, lub znalezione zostało puste pole. Następnie sprawdzany jest warunek zgodności z regułami gry pozwalającymi na położenie kamienia, a więc czy w danym kierunku jest pole gracza kładącego kamień, a wszystkie wcześniej napotkane pola należą do jego przeciwnika. Jeżeli ten warunek jest spełniony,

kamień jest umieszczany na wybranym polu i wszystkie kamienie między dostawionym a tymi znalezionymi w różnych kierunkach są zamieniane na kamienie gracza wykonującego ruch. Liczba wszystkich przejętych kamieni jest sumowana i zwracana przez wartość funkcji. Przejmowanie kamieni przeciwnika w metodzie PołóżKamień jest realizowane, gdy jej drugi argument tylkoTest ma wartość false. Ustawienie go na true umożliwia sprawdzenie, ile kamieni zostanie przejętych, bez rzeczywistego wykonania ruchu.

Zdefiniujmy prostą metodę przeciążoną. O ile podstawowa wersja zwraca liczbę przejętych kamieni, o tyle jej uproszczona wersja zwraca tylko wartość logiczną informującą o tym, czy ruch jest możliwy. Poza tym ta metoda będzie publiczna, podczas gdy wcześniejsza jest chroniona.

```
public bool PołóżKamień(int poziomo, int pionowo)
{
    return PołóżKamień(poziomo, pionowo, false) > 0;

}
```

Jeżeli nowy kamień został położony na planszy, to metoda PołóżKamień automatycznie zmienia numer identyfikujący gracza wykonującego kolejny ruch. Wywołuje w tym celu metodę pomocniczą zmieńBieżącegoGracza.

## Obliczanie liczb pól zajętych przez graczy

Potrzebne są jeszcze metody obliczające liczby pól, które są zajęte przez kamienie obu graczy. To pozwoli ocenić, który z graczy ma przewagę. Dodajmy do klasy ReversiSilnik prywatną metodę obliczLiczbyPól. Wynik jej działania będzie zapisywany w trójelementowej tablicy liczbyPól, której pola będą udostępniane przez trzy właściwości tylko do odczytu.

```
private int[] liczbyPól = new int[3]; //puste, gracz 1, gracz 2
public int LiczbaPustychPól { get { return liczbyPól[0]; } }
public int LiczbaPólGracz1 { get { return liczbyPól[1]; } }
public int LiczbaPólGracz2 { get { return liczbyPól[2]; } }

private void obliczLiczbyPól()
{
    for (int i = 0; i < liczbyPól.Length; ++i) liczbyPól[i] = 0;

    for (int i = 0; i < SzerokośćPlanszy; ++i)
        for (int j = 0; j < WysokośćPlanszy; ++j)
            liczbyPól[plansza[i, j]]++;

}
```

Wywołanie metody obliczLiczbyPól należy umieścić na końcu konstruktora klasy ReversiSilnik oraz w metodzie PołóżKamień po spełnieniu warunku sprawdzającego poprawność ruchu (tuż po zmianie numeru gracza). W ten sposób silnik sam będzie dbał o aktualizację tablicy liczbyPól i odczytanie powyższych właściwości w każdej chwili będzie dawało poprawne rezultaty.

W kodzie widoku przypisujemy graczom kolory gracz pierwszy będzie „zielony”, a drugi — „brązowy”.

```
private SolidColorBrush[] kolory = {Brushes.Ivory,Brushes.Green,Brushes.Sienna};
string[] nazwyGraczy = { "", "zielony", "brązowy" };
```

Generujemy planszę

```
private ReversiSilnik silnik = new ReversiSilnik(1);
```

Uzgadniamy zawartość planszy. Zmieniamy także wcześniej przypisane na sztywno szerokość planszy i wysokość planszy(były ustalone na 8). Teraz możemy im przypisać wartości silnik.SzerokośćPlanszy, silnik.WysokośćPlanszy

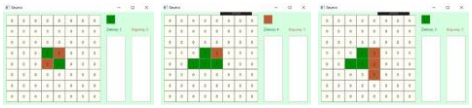
```
private bool planszaZainicjowana
{
 get
 {
 return plansza[silnik.SzerokośćPlanszy - 1, silnik.WysokośćPlanszy - 1] != null;
 }
}
private void uzgodnijZawartośćPlanszy()
{
 if (!planszaZainicjowana) return;

 for (int i = 0; i < silnik.SzerokośćPlanszy; i++)
     for (int j = 0; j < silnik.WysokośćPlanszy; j++)
     {
        plansza[i, j].Background = kolory[silnik.PobierzStanPola(i, j)];
        plansza[i, j].Content = silnik.PobierzStanPola(i, j).ToString();
     }

  przyciskKolorGracza.Background =
                kolory[silnik.NumerGraczaWykonujacegoNastepnyRuch];
  liczbaPólZielony.Text = silnik.LiczbaPólGracz1.ToString();
  liczbaPólBrazowy.Text = silnik.LiczbaPólGracz2.ToString();
 }
```

W konstruktorze klasy MainWindow (na końcu) wywołujemy metodę uzgodnijZawartośćPlanszy();

Całość kodu

```
public partial class MainWindow : Window
{
private ReversiSilnik silnik = new ReversiSilnik(1);

private SolidColorBrush[] kolory = {Brushes.Ivory,Brushes.Green,Brushes.Sienna};
string[] nazwyGraczy = { "", "zielony", "brązowy" };
private Button[,] plansza;

private bool planszaZainicjowana
{
 get
 {
 return plansza[silnik.SzerokośćPlanszy - 1, silnik.WysokośćPlanszy - 1] != null;
 }
}
private void uzgodnijZawartośćPlanszy()
{
 if (!planszaZainicjowana) return;

 for (int i = 0; i < silnik.SzerokośćPlanszy; i++)
     for (int j = 0; j < silnik.WysokośćPlanszy; j++)
     {
        plansza[i, j].Background = kolory[silnik.PobierzStanPola(i, j)];
        plansza[i, j].Content = silnik.PobierzStanPola(i, j).ToString();
     }

  przyciskKolorGracza.Background =
                kolory[silnik.NumerGraczaWykonującegoNastępnyRuch];
  liczbaPólZielony.Text = silnik.LiczbaPólGracz1.ToString();
  liczbaPólBrazowy.Text = silnik.LiczbaPólGracz2.ToString();
 }

public MainWindow()
{
  InitializeComponent();

   //podział siatki na wiersze i kolumny
   for (int i = 0; i < silnik.SzerokośćPlanszy; i++)
        planszaSiatka.ColumnDefinitions.Add(new ColumnDefinition());
   for (int j = 0; j < silnik.WysokośćPlanszy; j++)
        planszaSiatka.RowDefinitions.Add(new RowDefinition());

   //tworzenie przycisków
   plansza = new Button[silnik.SzerokośćPlanszy, silnik.WysokośćPlanszy];
   for (int i = 0; i < silnik.SzerokośćPlanszy; i++)
       for (int j = 0; j < silnik.WysokośćPlanszy; j++)
       {
           Button przycisk = new Button();
           przycisk.Margin = new Thickness(0);
           planszaSiatka.Children.Add(przycisk);
           Grid.SetColumn(przycisk, i);
           Grid.SetRow(przycisk, j);
           plansza[i, j] = przycisk;
       }
   uzgodnijZawartośćPlanszy();

            //testy
            //silnik.PołóżKamień(2, 4);
            //silnik.PołóżKamień(4, 5);
            //uzgodnijZawartośćPlanszy();
 }
}
```

Po uruchomieniu aplikacji powinniśmy zobaczyć osiem wierszy zer z szachownicą utworzoną z dwóch jedynek i dwóch dwójek pośrodku (lewa część rysunku 6.3). Natomiast jeżeli dodamy do konstruktora okna polecenie silnik.PołóżKamień(2, 4); kładące kamień na polu D5, plansza powinna wyglądać jak na środkowym rysunku. Ponieważ metoda ReversiSilnik.PołóżKamień automatycznie zmienia gracza, możemy dodać kolejne jej wywołania, np. silnik.PołóżKamień(4, 5) - prawa część rysunku .



## Interakcja z użytkownikiem

Kluczową metodą klasy ReversiSilnik jest PołóżKamień. Będzie ona wywoływana przez kliknięcie poszczególnych pól planszy. Do rysowania pól użyliśmy przycisków, które są wyposażone w zdarzenie Click. Musimy wobec tego zdefiniować metodę zdarzeniową zgodną z sygnaturą zdarzenia Click i związać ją z każdym przyciskiem dodawanym do planszy w konstruktorze klasy MainWindow.

W tej metodzie potrzebne będą współrzędne kliknięty pola. Przesłana jednak będzie do niej tylko referencja do przycisku. Wykorzystamy własność Tag typu object, którą posiadają wszystkie kontrolki WPF. Tag to miejsce, w którym można przechować dowolne dane. My zapiszemy tam obiekt, w którym przechowamy współrzędne pola na planszy. Obiekt ten należy tam umieścić podczas tworzenia przycisku w konstruktorze klasy MainWindow.

Metoda kliknięciePlanszy odczytuje zapisane we własności Tag przycisku współrzędne pola, próbuje położyć na tym polu kamień bieżącego gracza i jeżeli to się udało, aktualizuje widok planszy.

```
private struct WspółrzędnePola
{
    public int Poziomo, Pionowo;
}

void kliknięciePolaPlanszy(object sender, RoutedEventArgs e)
{
     Button kliknietyPrzycisk = sender as Button;
     WspółrzędnePola współrzędne = (WspółrzędnePola)kliknietyPrzycisk.Tag;
     int kliknietePoziomo = współrzędne.Poziomo;
     int kliknietePionowo = współrzędne.Pionowo;

     //wykonanie ruchu
     int zapamietanyNumerGracza = silnik.NumerGraczaWykonujacegoNastepnyRuch;
     if (silnik.PołóżKamień(kliknietePoziomo, kliknietePionowo))
         uzgodnijZawartośćPlanszy();

}
public MainWindow()
{
   InitializeComponent();

    ...
   plansza = new Button[silnik.SzerokośćPlanszy, silnik.WysokośćPlanszy];
   for (int i = 0; i < silnik.SzerokośćPlanszy; i++)
       for (int j = 0; j < silnik.WysokośćPlanszy; j++)
       {
           Button przycisk = new Button();
           przycisk.Margin = new Thickness(0);
           planszaSiatka.Children.Add(przycisk);
           Grid.SetColumn(przycisk, i);
           Grid.SetRow(przycisk, j);
           przycisk.Tag = new WspółrzędnePola { Poziomo = i, Pionowo = j };
           przycisk.Click += new RoutedEventHandler(kliknięciePolaPlanszy);
           plansza[i, j] = przycisk;
        }
   uzgodnijZawartośćPlanszy();
}
```

## Historia ruchów

Wyświetlenie historii ruchów obu graczy. Użyjemy do tego oznaczeń pól przejętych z szachów. Pole o współrzędnych (0, 0) oznaczymy jako A1, pole (7,7) - jako H8. Tłumaczenie współrzędnych na symbol szachowy to zadanie metody symbolPola. Użyjemy jej w metodzie kliknięciePolaPlanszy.

Uwaga. Gdy pól jest zbyt wiele w poziomie i nie wystarcza liter w alfabecie lub zbyt wiele w pionie i nie wystarcza cyfr, to zamiast symbolu znanego z szachów wyświetlane będą zwykłe współrzędne pola podane w nawiasach.

```
private static string symbolPola(int poziomo, int pionowo)
{
   if (poziomo > 25 || pionowo > 8) return "(" +poziomo.ToString() + ","
      +pionowo.ToString() + ")";
   return "" + "ABCDEFGHIJKLMNOPQRSTUVWXYZ"[poziomo] + "123456789"[pionowo];
}

void kliknieciePolaPlanszy(object sender, RoutedEventArgs e)
{
    Button kliknietyPrzycisk = sender as Button;
    WspółrzędnePola współrzędne = (WspółrzędnePola)kliknietyPrzycisk.Tag;
    int kliknietePoziomo = współrzędne.Poziomo;
    int kliknietePionowo = współrzędne.Pionowo;

    //wykonanie ruchu
   int zapamietanyNumerGracza = silnik.NumerGraczaWykonujacegoNastepnyRuch;
   if (silnik.PołóżKamień(kliknietePoziomo, kliknietePionowo))
   {
      uzgodnijZawartośćPlanszy();
      //lista ruchów
      switch (zapamietanyNumerGracza)
      {
          case 1:
          listaRuchówZielony.Items.Add(symbolPola(kliknietePoziomo,
                                                  kliknietePionowo));
          break;
          case 2:
          listaRuchówBrazowy.Items.Add(symbolPola(kliknietePoziomo,
                                                  kliknietePionowo));
          break;
      }
      listaRuchówZielony.SelectedIndex = listaRuchówZielony.Items.Count - 1;
      listaRuchówBrazowy.SelectedIndex = listaRuchówBrazowy.Items.Count - 1;
   }
}
```

### Wykrywanie szczególnych sytuacji w grze

Aplikację możemy już używać do gry dla dwóch osób. Klasa ReversiSilnik nie sprawdza jeszcze warunków zakończenia gry. Nie wykrywa także sytuacji, w której gracz nie ma możliwości wykonania ruchu i jest zmuszony do jego oddania.

Najpierw zajmijmy się wykrywaniem sytuacji, w której gracz nie może wykonać ruchu, ponieważ żadne pole nie spełnia warunków stawianych przez reguły gry. Do realizacji tego zadania będziemy potrzebowali metody, która pozwoli na sprawdzenie, czy na wskazanym polu można położyć kamień. Właśnie tak działa trójargumentowa wersja metody PołóżKamień, jeżeli jej argument tylkoTest jest równy true. Korzystając z niej, możemy z łatwością sprawdzić, czy na planszy jest pole, na którym bieżący gracz może położyć kamień. Wystarczy wywołać tę metodę dla wszystkich pustych pól i sprawdzić, czy którekolwiek z nich zwraca wartość większą od zera.

```
private bool czyBieżącyGraczMożeWykonaćRuch()
{
    int liczbaPoprawnychPól = 0;
    for (int i = 0; i < SzerokośćPlanszy; ++i)
        for (int j = 0; j < WysokośćPlanszy; ++j)
            if (plansza[i, j] == 0 && PołóżKamień(i, j, true) > 0)
                liczbaPoprawnychPól++;

    return liczbaPoprawnychPól > 0;
}
```

Jeżeli gracz nie ma żadnej możliwości wykonania ruchu, zgodnie z regułami gry zmuszony jest do jego oddania. Pozwoli na to prosta metoda Pasuj, która nie tylko zmieni numer gracza wykonującego następny ruch, ale także przypilnuje, aby oddanie ruchu nie odbywało się niezgodnie z regułami, tj. było możliwe tylko w sytuacji, gdy wykonanie ruchu nie jest możliwe.

```
public void Pasuj()
{
  if (czyBieżącyGraczMożeWykonaćRuch())
     throw new Exception("Gracz nie może oddać ruchu, jeżeli wykonanie ruchu jest
                          możliwe");

  zmieńBieżącegoGracza();
}
```

Możliwość sprawdzenia, czy bieżący gracz może wykonać ruch, pozwoli nam również przygotować metodę, która sprawdzi, czy gra się zakończyła, i jeśli tak, to w jaki sposób (zapełnienie wszystkich pól czy niemożność wykonania ruchu przez obu graczy). Na jej potrzeby w klasie ReversiSilnik zdefiniujemy typ wyliczeniowy SytuacjaNaPlanszy zawierający wszystkie możliwe stany gry

```
public enum SytuacjaNaPlanszy
{
   RuchJestMożliwy,
   BieżącyGraczNieMożeWykonaćRuchu,
   ObajGraczeNieMogąWykonaćRuchu,
   WszystkiePolaPlanszySąZajęte
}
```

Zdefiniujemy publiczną metodę klasy ReversiSilnik sprawdzającą, czy gra może toczyć się dalej, czy też należy już wyłonić zwycięzcę. Do sprawdzenia, czy na wszystkich polach leżą kamienie, użyjemy właściwości ReversiSilnik.LiczbaPustychPól, której wartość jest aktualizowana po każdym ruchu. Jeżeli na planszy są puste pola, sprawdzamy, czy bieżący gracz może wykonać ruch. Jeżeli nie, to sprawdzamy, czy przeciwnik może wykonać ruch. Jeżeli obaj gracze nie mogą położyć kamienia, gra się kończy.

```
public SytuacjaNaPlanszy ZbadajSytuacjęNaPlanszy()
{
  if (LiczbaPustychPól == 0)
     return SytuacjaNaPlanszy.WszystkiePolaPlanszySąZajęte;

  //badanie możliwości ruchu bieżącego gracza
  bool czyMożliwyRuch = czyBieżącyGraczMożeWykonaćRuch();
  if (czyMożliwyRuch) return SytuacjaNaPlanszy.RuchJestMożliwy;
  else
  {
     //badanie możliwości ruchu przeciwnika
     zmieńBieżącegoGracza();
     bool czyMożliwyRuchOponenta = czyBieżącyGraczMożeWykonaćRuch();
     zmieńBieżącegoGracza();
     if (czyMożliwyRuchOponenta)
         return SytuacjaNaPlanszy.BieżącyGraczNieMożeWykonaćRuchu;
     else return SytuacjaNaPlanszy.ObajGraczeNieMogąWykonaćRuchu;
   }
}
```

Dzięki tej metodzie możemy sprawdzić po każdym ruchu, czy gra jest skończona. Jeżeli tak, należy wyłonić zwycięzcę (wystarczy porównać liczbę kamieni obu graczy). Odpowiedzialna za to będzie własność NumerGraczaMającegoPrzewagę, którą należy dodać do silnika gry. W przypadku remisu własność będzie zwracała 0.

```
public int NumerGraczaMajacegoPrzewage
{
    get
    {
        if (LiczbaPólGracz1 == LiczbaPólGracz2) return 0;
        else
            if (LiczbaPólGracz1 > LiczbaPólGracz2) return 1;
            else return 2;
    }

}
```

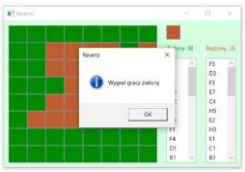
Po zakończeniu gry i ustaleniu zwycięzcy powinniśmy wyświetlić graczom stosowny komunikat informujący o wyniku – metoda kliknięciePolaPlanszy. Do klasy MainWindow dodamy także metodę, która będzie resetować planszę.

```
void kliknieciePolaPlanszy(object sender, RoutedEventArgs e)
{
...
  //sytuacje specjalne
  ReversiSilnik.SytuacjaNaPlanszy sytuacjaNaPlanszy =
                silnik.ZbadajSytuacjeNaPlanszy();
  bool koniecGry = false;
  switch (sytuacjaNaPlanszy)
  {
        case ReversiSilnik.SytuacjaNaPlanszy.BieżącyGraczNieMożeWykonaćRuchu:
        MessageBox.Show("Gracz "
+nazwyGraczy[silnik.NumerGraczaWykonujacegoNastepnyRuch] + " zmuszony jest do
oddania ruchu");
        silnik.Pasuj();
        uzgodnijZawartośćPlanszy();
        break;
        case ReversiSilnik.SytuacjaNaPlanszy.ObajGraczeNieMogaWykonaćRuchu:
        MessageBox.Show("Obaj gracze nie mogą wykonać ruchu");
        koniecGry = true;
        break;
        case ReversiSilnik.SytuacjaNaPlanszy.WszystkiePolaPlanszySaZajete:
        koniecGry = true;
        break;
  }
  //koniec gry - informacja o wyniku
  if (koniecGry)
  {
     int numerZwycięzcy = silnik.NumerGraczaMajacegoPrzewage;
     if (numerZwycięzcy != 0) MessageBox.Show("Wygrał gracz "
+nazwyGraczy[numerZwycięzcy], Title, MessageBoxButton.OK,
MessageBoxImage.Information);
    else MessageBox.Show("Remis", Title, MessageBoxButton.OK,
MessageBoxImage.Information);

     if (MessageBox.Show("Czy rozpocząć grę od nowa ?", "Reversi",
MessageBoxButton.YesNo, MessageBoxImage.Question, MessageBoxResult.Yes) ==
MessageBoxResult.Yes)
                    {
                        przygotowaniePlanszyDoNowejGry(1,
silnik.SzerokośćPlanszy, silnik.WysokośćPlanszy);
                    }
                    else
                    {
```

```
                    planszaSiatka.IsEnabled = false;
                    przyciskKolorGracza.IsEnabled = false;
                }

                }

    }

    }
```

```
private void przygotowaniePlanszyDoNowejGry(int numerGraczaRozpoczynającego,
                        int szerokośćPlanszy = 8, int wysokośćPlanszy = 8)
{
    silnik = new ReversiSilnik(numerGraczaRozpoczynającego,
                               szerokośćPlanszy, wysokośćPlanszy);
    listaRuchówZielony.Items.Clear();
    listaRuchówBrazowy.Items.Clear();
    uzgodnijZawartośćPlanszy();
    planszaSiatka.IsEnabled = true;
    przyciskKolorGracza.IsEnabled = true;
}
```

Po zakończeniu gry i ogłoszeniu zwycięzcy lub remisu



użytkownikowi zadawane jest pytanie, czy chce zacząć nową grę. Jeżeli wybierze odpowiedź „tak”, aplikacja wróci do sytuacji wyjściowej. Jeżeli wybierze „nie” - plansza zostanie dezaktywowana i nie będzie możliwości, żeby coś z nią zrobić. Pozostanie tylko zamknięcie aplikacji.