Reversi w .NET MAUI

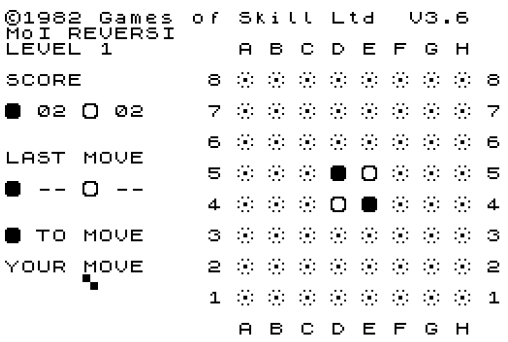
Tworzymy grę planszową dla systemów Windows i Android

W dwóch wcześniejszych artykułach (Programista 2/2022 i 3/2022) oswajałem nową technologię .NET MAUI, która umożliwia tworzenie aplikacji wieloplatformowych z graficznym interfejsem użytkownika. Do tej pory testowaliśmy MAUI na przykładzie bardzo prostej aplikacji, umożliwiającej wybór trzech składowych RGB koloru za pomocą suwaków. Dziś sprawdzimy, czy możliwe jest stworzenie czegoś bardziej złożonego.

Jako przykładu użyję klasycznej gry planszowej dla dwóch osób o nazwie „Reversi". Podobnie jak dwa poprzednie artykuły, ten również będzie miał formę tutorialu. Trudnością będzie m.in. dynamicznie tworzony interfejs (plansza do gry), okna komunikatów czy dostosowywanie interfejsu do rozmiarów okna. Sprawdzimy też, czy możliwe jest przygotowanie testów jednostkowych. Rozwijając ten projekt, nie będziemy korzystać z architektury MVVM, wyposażymy go jednak w porządny model, w którym umieszczony będzie silnik gry pilnujący przestrzegania zasad gry.

Za bazę tekstu wziąłem rozdział z książki *Visual Studio 2017. Tworzenie aplikacji Windows w języku C#* i starałem się odtworzyć opisaną tam aplikację Windows zbudowaną z użyciem kontrolek WPF w .NET MAUI.

Zasad gry „Reversi" (znanej również pod nazwą „Othello") można nauczyć się w kilka minut, ale dostarcza ona rozrywki przez znacznie dłuższy czas. Pierwszy raz zetknąłem się z nią na ZX Spectrum w latach 80. ub.w. (Rysunek 1) i do tej wersji mam szczególny sentyment. Do tego stopnia, że czasem wracam do niej nawet, korzystając z emulatorów. Teraz spróbujemy przygotować wersję tej gry w .NET MAUI. Zaczniemy od wersji dla dwóch osób, a potem przygotujemy prostego bota, który w razie potrzeby zastąpi żywego gracza.



Rysunek 1. Reversi na ZX Spectrum

Gra „Reversi" jest rozgrywana na planszy o wymiarach 8×8, na której w chwili rozpoczęcia gry zajęte są cztery środkowe pola, tworząc małą szachownicę (por. Rysunek 1). Zasady gry są następujące:

1. Gracze zajmują na przemian pola planszy, kładąc na nich swoje kamienie, przejmując wszystkie pola przeciwnika znajdujące się między nowo zajętym polem a innymi polami gracza wykonującego ruch. Między nimi nie może być pól pustych – wszystkie muszą być zajęte przez kamienie przeciwnika.
2. Celem gry jest zdobycie większej liczby pól niż przeciwnik. To decyduje o zwycięstwie.
3. Gracz może zająć jedynie takie pole, które pozwoli mu przejąć przynajmniej jedno pole przeciwnika. Jeżeli takiego pola nie ma, musi oddać ruch.
4. Gra kończy się, gdy wszystkie pola są zajęte lub gdy żaden z graczy nie może wykonać ruchu.

MODEL – SILNIK GRY

Jak wspomniałem, nie będziemy korzystać z architektury MVVM, ale oczywiście oddzielimy model z silnikiem gry od widoku (kodu XAML i klasy strony). Serce projektu stanowi klasa modelu `ReversiModel`, która będzie implementować reguły gry i przechowywać bieżący stan planszy. Jej klasą potomną będzie `ReversiModelAI`, która rozszerzy model o metody wskazujące najlepszy ruch. Tym jednak zajmiemy się w dalszej części artykułu.

Podobnie jak w dwóch poprzednich artykułach, nadal korzystamy z *Visual Studio Community 2022 Preview*, ale warto zaznaczyć że od początku sierpnia 2022 .NET MAUI jest też dostępna w „zwykłej" wersji Visual Studio 2022. Zacznijmy od utworzenia nowego projektu zgodnego z szablonem *Aplikacja platformy .NET MAUI* o nazwie *Reversi*. Do projektu dodajmy od razu dodatkowy plik klasy o nazwie *ReversiModel.cs*. Na razie umieścimy go w głównym folderze tego samego projektu.

Zastanówmy się nad tym, co jest niezbędne do implementacji gry „Reversi". Jakie pola musi mieć klasa `ReversiModel`? Potrzebujemy przede wszystkim „logicznej" reprezentacji planszy z możliwością ustawiania na niej kamieni (zajmowania pól). Rozmiar planszy będzie ustalany w argumentach konstruktora, co pozwoli na jej ewentualne zwiększenie ponad standardowe 8×8, jeżeli gra okaże się dla nas zbyt krótka, lub zmniejszenie np. do 4×4, żeby łatwiejsze było jej testowanie, a w szczególności testowanie zakończenia rozgrywki. Każde pole na planszy może mieć trzy stany: może być puste lub zajęte przez kamień jednego lub drugiego gracza. Poza tym potrzebujemy zmiennej przechowującej numer gracza, który ma wykonać następny ruch. W obu przypadkach narzuca się utworzenie typu wyliczeniowego. Jednak w praktyce takie rozwiązanie nie okazało się wygodne ze względu na częste konwersje między stanem pola a liczbami całkowitymi będącymi indeksami tablic wykorzystywanych w programie.

DNA

Boisz się,

że Twoja ścieżka rozwoju technicznego **dobiega końca?**

Otóż... **nie!**

DNA Droga Nowoczesnego Architekta to nowa jakość w budowaniu programistycznej kariery. Tony materiałów kierowane do architektów, seniorów i midów. Prosto od najwyższej klasy specjalistów z ogromnym doświadczeniem w projektach oraz w edukacji.

Prawie **15 000** korzysta z naszego Mailingu Nowoczesnych Architektów!

Dołącz do nas za darmo i rozwijaj techniczną karierę na **droga.dev**!

Dwuwymiarową tablicę liczb całkowitych reprezentujących pola planszy zadeklarujemy jako prywatne pole klasy `ReversiModel`. Aby umożliwić dostęp do planszy „z zewnątrz", zdefiniujemy publiczną metodę `GetSpaceStatus`, pozwalającą na sprawdzenie wartości wybranego pola. Zdefiniujemy również publiczną właściwość `NextPlayerNumber`, z której można będzie odczytać numer gracza, ale jego zmiana będzie możliwa tylko z samej klasy modelu.

Antycypując rozwój klasy, mogę zdradzić, że będziemy potrzebowali także metody `PutStone`, która będzie zmieniała stan wskazanego przez gracza pola (oczywiście o ile jest ono puste). Metoda ta będzie dbała również o zgodne z zasadami gry przejęcie pól przeciwnika. Przypomnę, że aby możliwe było położenie kamienia na polu, musi ono prowadzić do przejęcia przynajmniej jednego pola przeciwnika. Potrzebować będziemy również metod sprawdzających, czy gracz może wykonać ruch oraz czy został osiągnięty stan planszy, w którym gra jest zakończona.

> Zwróćmy uwagę, że metody `GetSpaceStatus` i `PutStone` są *de facto* akcesorem i mutatorem do tablicy przechowującej stan planszy. Wobec tego możliwe byłoby zdefiniowanie indeksera przyjmującego współrzędne pola, który w części `get` wywoływałby pierwszą, a w części `set` drugą z tych metod. Nie zdecydowałem się jednak na to, bo wydaje mi się, że wywołania metod prowadzą do bardziej czytelnego kodu.

Stan planszy

Skupmy się najpierw na realizacji pierwszej części powyższego planu, definiując tablicę reprezentującą planszę oraz metodę pozwalającą na odczyt stanu poszczególnych pól tej planszy. W Listingu 1 pokazano zawartość pliku *ReversiModel.cs* z polem będącym deklaracją dwuwymiarowej tablicy liczb całkowitych typu `int` opisującą planszę wraz z ustawionymi na niej kamieniami. Proszę zwrócić uwagę na nową formę użycia słowa kluczowego `namespace`, która ustala przestrzeń nazw dla całego pliku. Użyłem jej, pomimo iż szablon generujący plik klas zdefiniował przestrzeń nazw w starym stylu, tj. z klasą w jej nawiasach klamrowych. W listingu widoczna jest również prosta struktura `Coordinates`, którą wykorzystamy do wygodniejszego identyfikowania pól na planszy. Widoczna jest także zapowiedziana metoda `GetSpaceStatus`, a także wykorzystywana w niej statyczna metoda sprawdzająca poprawność podanej pozycji pola oraz własność wskazująca gracza wykonującego następny ruch. W klasie `ReversiModel` zdefiniowana jest poza tym prosta metoda statyczna `opponentsNumber` wyznaczająca numer gracza-oponenta do gracza wykonującego najbliższy ruch, a więc zwracająca wartość 1 dla argumentu równego 2 i 2 dla 1.

Listing 1. Klasa ReversiModel

```
namespace Reversi;

public struct Coordinates
{
    public int Horizontal, Vertical;

    public Coordinates(int horizontal, int vertical)
    {
        this.Horizontal = horizontal;
        this.Vertical = vertical;
    }
}

public class ReversiModel
{
    public int Nx { get; private set; }

    public int Ny { get; private set; }

    private int[,] board;

    public int NextPlayerNumber
        { get; private set; } = 1;

    private static int opponentsNumber(int playerNumber)
    {
        return (playerNumber == 1) ? 2 : 1;
    }

    private bool areCoordinatesCorrect(Coordinates space)
    {
        return space.Horizontal >= 0 &&
               space.Horizontal < Nx &&
               space.Vertical >= 0 &&
               space.Vertical < Ny;
    }

    public int GetSpaceStatus(Coordinates space)
    {
        if (!areCoordinatesCorrect(space))
            throw new Exception(
                "Nieprawidłowe współrzędne pola");
        return board[space.Horizontal, space.Vertical];

    }
}
```

Własności `Nx` i `Ny` przechowują rozmiary planszy, które będą przekazywane przez argumenty konstruktora klasy `ReversiModel` (jeszcze go nie zdefiniowaliśmy). Zgodnie z zapowiedzią, aby uniknąć nierozmyślnej modyfikacji tablicy `board`, została ona zdefiniowana jako prywatna, a odczyt jej elementów możliwy jest tylko dzięki publicznej metodzie `GetSpaceStatus`. Interfejs klasy nie umożliwia na razie zmian wartości pól planszy – te będą się odbywać tylko zgodnie z zasadami gry, za co odpowiedzialna będzie metoda `PutStone`, którą za chwilę zdefiniujemy Wcześniej jednak zajmiemy się odpowiednim przygotowaniem planszy do gry, a więc położeniem czterech kamieni w jej centrum.

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 Rysunkiem 1 i wyznaczeniu gracza wykonującego pierwszy ruch. Za inicjację planszy odpowiedzialna jest metoda `clearBoard`. Argumenty konstruktora pozwalają na wskazanie numeru gracza rozpoczynającego oraz wielkości planszy. Konstruktor i metodę pomocniczą pokazano w Listingu 2.

Listing 2. Początkowe ustawienie kamieni odpowiada temu z rysunku 1

```
private void clearBoard()
{
    for (int i = 0; i < Nx; i++)
        for (int j = 0; j < Ny; j++)
            board[i, j] = 0;

    int halfWidth = Nx / 2;
    int halfHeight = Ny / 2;
    board[halfWidth - 1, halfHeight - 1] = 1;
    board[halfWidth, halfHeight] = 1;
    board[halfWidth - 1, halfHeight] = 2;
    board[halfWidth, halfHeight - 1] = 2;
}

public ReversiModel(int startingPlayerNumber,
                    int boardWidth = 8,
                    int boardHeight = 8)
```