Jacek Matulewski, e-mail: *jacek@fizyka.umk.pl*
Strona wykładu: *http://www.fizyka.umk.pl/~jacek/dydaktyka/programowanie/*

# Przykład: Maszyna Turinga 2.0

Symulator maszyny Turinga to najbardziej rozbudowany projekt przedstawiony na tym wykładzie. To cenny przykład, bo pozwoli zdefiniować nam więcej niż jedną klasę w jednym systemie. Oczywiście, podobnie jak w opisanym przed chwilą projekcie równań kwadratowych, także i w tym przypadku zdefiniujemy klasę opakowującą cały kod symulatora, ale tym razem zdefiniujemy również dodatkowe klasy, które opisują poszczególne elementy symulatora: czwórka, czyli linia programu maszyny Turinga i obiekty opisujące jej stan i program.

Ponieważ zmian względem oryginalnego projektu będzie sporo, nie będę opisywał transformacji, a zbuduję projekt od nowa. Jednak kod poszczególnych metod, będzie podobny do tego z rozdziału 10 i nie będę ich drugi raz szczegółowo omawiał. Stwórzmy zatem nowy projekt aplikacji konsolowej dla platformy .NET Core. Ja nazwałem go *MaszynaTuringa2*. Następnie w tym projekcie dodajmy dodatkowy plik klasy, który nazwijmy *Turing.cs*. W tym celu z menu *Projekt* wybieramy polecenie *Dodaj klasę...*. Pojawi się wówczas okno dialogowe o nazwie *Dodaj nowy element*, w którym zaznaczona jest już pozycja *Klasa*. W polu edycyjnym u dołu tego okna należy wpisać nazwę *Turing.cs* i kliknąć przycisk *Dodaj*.

## Dwójka

Zacznijmy od przygotowania kodu obsługującego wczytywanie i parsowanie programu dla maszyny Turinga. Pamiętamy, że składa się z „czwórek", w których pierwsze dwa znaki to bieżący stan maszyny, a więc wartość rejestru i wartość na taśmie wskazywana przez głowicę, a kolejne dwa znaki zawierają nowy stan maszyny, czyli nowy stan głowicy oraz wartość, jaką należy zapisać na taśmie (napisując poprzednią wartość), ewentualnie polecenia przesunięcia głowicy w lewo lub w prawo. Stany rejestru zapisywane są małymi literami, a wartości na taśmie – dużymi. Polecenia przesunięcia głowicy kodowane są literami *L* i *R*. Więcej informacji na temat maszyny Turinga znajduje się w rozdziale 9. Pamiętamy z niego, że wygodne jest przechowywanie programu dla maszyny Turinga (zbioru czwórek) w postaci słownika, w którym bieżący stan maszyny jest kluczem pozwalającym na identyfikację tej linii kodu, która powinna być wykonana. Dlatego zanim zdefiniujemy klasę opisującą czwórkę, zdefiniujemy wpierw klasę opisującą dwójkę: wartość rejestru i wartość na taśmie (ewentualnie polecenie przesunięcia głowicy). Pokazuje ją listing 14.9. Z tych dwójek łatwo będzie zbudować czwórkę.

Listing 14.9. Struktura dwójki

```
using System;


namespace MaszynaTuringa2
{
    struct Dwójka : IComparable<Dwójka>
    {
        public char StanGłowicy;
        public char WartośćLubPolecenieNaTaśmie;


        public int CompareTo(Dwójka other)
        {
            int wartosc = StanGłowicy.CompareTo(other.StanGłowicy);
```

```
            if (wartosc != 0) return wartosc;
            else return WartośćLubPolecenieNaTaśmie.CompareTo(
                                                    other.WartośćLubPolecenieNaTaśmie);
        }
    }

    class Turing
    {
    }
}
```

W typie `Dwójka` znajdują się dwa publiczne pola przechowujące pojedyncze znaki (pola typu `char`): `StanGłowicy` oraz `WartośćLubPolecenieNaTaśmie`. Ale dlaczego nie własności? Można byłoby ich przecież użyć do dopilnowania, żeby do pierwszego pola przypisywane były małe litery, a do drugiego – duże. Z takiej kontroli oczywiście nie zrezygnujemy, ale przeprowadzimy ją na poziomie czwórki. Klasa `Dwójka` ma być tylko prostym nośnikiem danych. Z tego powodu jest strukturą, a nie klasą. Będzie często kopiowana, zatem użycie klasy niepotrzebnie obciążyłoby odśmiecacz platformy .NET.

Typ `Dwójka` implementuje interfejs `IComparable<Dwójka>`, aby kolekcje z tego typu elementami mogły być sortowane. A tak będzie, gdy użyjemy słownika `SortedList<Dwójka,Dwójka>`. Implementacja tego interfejsu oznacza, że w strukturze `Dwójka` musi być zdefiniowana metoda `CompareTo`, która zwraca liczbę całkowitą równą zeru jeżeli porównywalne elementy są równe oraz mniejszą i większą od zera, jeżeli jeden z elementów jest większy od drugiego. Porównywane są przede wszystkim stany głowicy, a dopiero w drugiej kolejności wartości na taśmie (por. klasę `Para` z rozdziału 15.).

# Czwórka

Typ `Czwórka` ma opisywać pojedynczą linię programu dla maszyny Turinga. Już ustaliliśmy, że będzie składał się z dwóch dwójek – opisujących bieżący i nowy stanu. To również przede wszystkim typ, który będzie przechowywał dane – dlatego i on będzie strukturą (listing 14.10). W tej strukturze umieścimy wprawdzie także metody odpowiedzialne za parsowanie pojedynczej linii kodu programu wczytanego z pliku i tworzących obiekty typu `Czwórka` (tzw. pseudokonstruktory), ale będą to metody statyczne, a więc nie mające wpływu na stan obiektu.

Listing 14.10. Czwórka, czyli para dwójek

```
struct Czwórka
{
    public Dwójka BieżącyStan, NowyStan;

    private static bool czyŁańcuchCzwórkiPoprawny(string linia)
    {
        Func<char, bool> isLowerLetter = (char c) => c >= 'a' && c <= 'z';
        Func<char, bool> isUpperLetter = (char c) => c >= 'A' && c <= 'Z';
        return isLowerLetter(linia[0]) && isUpperLetter(linia[1]) &&
isUpperLetter(linia[2]) && isLowerLetter(linia[3]);
    }

    public static Czwórka Parsuj(string linia)
    {
        if (string.IsNullOrWhiteSpace(linia)) throw new Exception("Pusta linia kodu");
        if (!czyŁańcuchCzwórkiPoprawny(linia))
            throw new Exception("Niepoprawna linia kodu (" + linia + ")");
        Czwórka czwórka = new Czwórka()
```

```
            {
                BieżącyStan = new Dwójka()
                {
                    StanGłowicy = linia[0],
                    WartośćLubPolecenieNaTaśmie = linia[1]
                },
                NowyStan = new Dwójka()
                {
                    StanGłowicy = linia[3],
                    WartośćLubPolecenieNaTaśmie = linia[2]
                }
            };
            return czwórka;
        }
    }
```

Czemu w metodzie `czyŁańcuchCzwórkiPoprawny` zdefiniowałem własne wyrażenia Lambda sprawdzające, czy litera jest mała lub duża, zamiast użyć gotowych metod `char.IsLower` i `char.IsUpper`? Przede wszystkim dlatego, żeby ćwiczyć użycie wyrażeń Lambda w praktycznych sytuacjach, a dodatkowo, dlatego żeby ograniczyć znaki do małych i dużych liter alfabetu łacińskiego.

Metoda `Parse` to fragment metody `parsujProgram` opisanej w rozdziale 9.; ten jej fragment, który z linii programu tworzy instancję typu `Czwórka`. Metoda ta sprawdza, czy łańcuch czwórki jest prawidłowy, a następnie tworzy czwórkę przekazując do niej odpowiednie litery z linii kodu.

# Program

Kolejny poziom będzie stanowiła klasa reprezentująca program maszyny Turinga widoczna na listingu 14.11. Będzie przechowywać zapowiadany słownik z poleceniami (pole `czwórki`) oraz wyposażona będzie w metodę parsującą podaną tablicę linii programu oraz w metodę `znajdźPolecenie` wyszukującą w tym słowniku polecenie pasujące do podanego bieżącego stanu maszyny (metoda ta przyjmować będzie i zwracać obiekty typu `Dwójka`). Metoda `parsuj` wykorzystuje wcześniej zdefiniowaną metodę `Czwórki.Parsuj` do parsowania poszczególnych linii kodu. W klasie znajduje się też konstruktor, który przyjmuje tablicę łańcuchów z kodem programu i który używa metody `parsuj`, aby zainicjować zwracaną przez nią wartością pole `czwórki`. Ponieważ `ProgramMaszynyTuringa` to klasa, zdefiniowanie własnego konstruktora powoduje, że kompilator nie definiuje już konstruktora domyślnego (bezargumentowego). Użycie słownika `SortedList` wymaga dodania do sekcji poleceń `using` deklarującej użycie przestrzeni nazw, polecenia `using System.Collections.Generic;`. Metoda `ZnajdźPolecenie` może nie znaleźć odpowiedniej czwórki w słowniku. Dlatego zwraca typ `Nullable<Dwójka>` (pamiętajmy, że typ `Dwójka` to struktura, a więc nie przyjmuje wartości `null`).

Listing 14.11. Klasa przechowująca program dla maszyny Turinga

```
class ProgramMaszynyTuringa
{
    private SortedList<Dwójka, Dwójka> czwórki;

    static SortedList<Dwójka, Dwójka> parsuj(string[] kodProgramu)
    {
        SortedList<Dwójka, Dwójka> czwórki = new SortedList<Dwójka, Dwójka>();
        foreach (string linia in kodProgramu)
        {
            try
            {
```

```
                Czwórka czwórka = Czwórka.Parsuj(linia);
                if (czwórki.ContainsKey(czwórka.BieżącyStan)) throw new
Exception("Program nie może zawierać dwóch poleceń o takim stanie głowicy i wartości na
taśmie");
                czwórki.Add(czwórka.BieżącyStan, czwórka.NowyStan);
            }
            catch (Exception exc)
            {
                Console.Error.WriteLine(exc.Message);
            }
        }
        return czwórki;
    }

    public ProgramMaszynyTuringa(string[] kodProgramu)
    {
        czwórki = parsuj(kodProgramu);
    }

    public Dwójka? ZnajdźPolecenie(Dwójka bieżącyStan)
    {
        if (czwórki.ContainsKey(bieżącyStan)) return czwórki[bieżącyStan];
        else return null;
    }
}
```

# Stan

Jak pamiętamy z rozdziału 9. z plików odczytujemy nie tylko kod programu, ale również stan maszyny tj. całą zawartość taśmy oraz pozycję głowicy i stan rejestru[1]. W nowej implementacji za obsługę tego stanu odpowiedzialna będzie klasa `StanMaszynyTuringa` (listing 14.12), która będzie zawierać tablicę znaków `Taśma`, pole `StanGłowicy` przechowujące znak z rejestru i pole typu `int` o nazwie `PołożenieGłowicy`. Zwróćmy uwagę, że z tych wielkości można także zbudować dwójkę opisującą bieżący stan (stan głowicy i bieżącą wartość na taśmie). Zdefiniujmy wobec tego własność `BieżącyStan` typu `Dwójka`, który będzie taką dwójkę zwracał (listing 14.12). Dodamy również konstruktor, który pozwala na inicjację obiektu, przyjmujący jako argumenty trzy elementy przechowywane w polach.

Listing 14.12. Klasa opisująca stan maszyny Turinga

```
class StanMaszynyTuringa
{
    public char[] Taśma;
    public char StanGłowicy;
    public int PołożenieGłowicy;

    public Dwójka BieżącyStan
    {
        get => new Dwójka()
```

[1] Tu pojawia się niejednoznaczność, bo stanem nazywałem też wcześniej dwójkę zawierającą wartość rejestru (stanu głowicy) i bieżącą wartość na taśmie wskazywaną przez pozycję głowicy.

```
        {
            StanGłowicy = this.StanGłowicy,
            WartośćLubPolecenieNaTaśmie = Taśma[this.PołożenieGłowicy]
        };
    }

    public StanMaszynyTuringa(char[] taśma, char stanGłowicy, int położenieGłowicy)
    {
        Taśma = taśma;
        StanGłowicy = stanGłowicy;
        PołożenieGłowicy = położenieGłowicy;
    }
}
```

Wyposażmy teraz klasę `StanMaszynyTuringa` w statyczną metodę pomocniczą, która pobiera łańcuch opisujący stan maszyny, a zwraca instancję obiektu `StanMaszynyTuringa` (listing 14.13). Jest to w zasadzie kopia metody `analizujOpisStanuMaszyny` z rozdziału 9. Z takich metod można zrobić tzw. pseudokonstruktor, czyli statyczną metodę, która pozwala na utworzenie instancji obiektu. Przykład takiej metody o nazwie `Twórz` też jest widoczny na listingu 14.13. Na tym samym listingu widoczny jest również zwykły konstruktor, który jako argument przyjmuje łańcuch opisujący stan maszyny. Dublowanie konstruktora i pseudokonstruktora nie ma oczywiście sensu – chodzi jedynie o prezentacji obu możliwości.

Listing 14.13. Zmiana łańcucha na obiekt opisujący stan maszyny

```
class StanMaszynyTuringa
{
    ...

    private static StanMaszynyTuringa analizuj(string łańcuchOpisującyStanMaszyny)
    {
        char stanGłowicy = ' ';
        int położenieGłowicy = -1;
        for (int i = 0; i < łańcuchOpisującyStanMaszyny.Length; ++i)
        {
            char c = łańcuchOpisującyStanMaszyny[i];
            if (c == 'L' || c == 'R')
                throw new Exception("Taśma nie może zawierać wartości L lub R");
            if (char.IsLower(c))
            {
                if (położenieGłowicy != -1) throw new Exception("Znaleziono więcej niż
jeden znak oznaczający głowicę");
                stanGłowicy = c;
                położenieGłowicy = i;
                łańcuchOpisującyStanMaszyny =
                    łańcuchOpisującyStanMaszyny.Remove(położenieGłowicy, 1);
            }
            else
            {
            if (c < 'A' || c > 'Z')
                throw new Exception($"Niepoprawna wartość na taśmie ({c}) na pozycji
{i}");
```

```
            }
        }
        if (położenieGłowicy < 0)
            throw new Exception("Nie znaleziono znaku oznaczającego głowicę");
        return new StanMaszynyTuringa(
            łańcuchOpisującyStanMaszyny.ToCharArray(),
            stanGłowicy,
            położenieGłowicy);
    }

    public StanMaszynyTuringa Twórz(string łańcuchOpisującyStanMaszyny)
    {
        return analizuj(łańcuchOpisującyStanMaszyny);
    }

    public StanMaszynyTuringa(string łańcuchOpisującyStanMaszyny)
    {
        StanMaszynyTuringa stan =
            StanMaszynyTuringa.analizuj(łańcuchOpisującyStanMaszyny);
        Taśma = stan.Taśma;
        StanGłowicy = stan.StanGłowicy;
        PołożenieGłowicy = stan.PołożenieGłowicy;
    }
}
```

Wyposażmy klasę `StanMaszynyTuringa` także w metodę umożliwiającą konwersję w drugą stronę tj. do łańcucha opisującego stan maszyny, w którym do zawartości taśmy dodawana jest literka oznaczająca głowicę. Na listingu 14.14 widoczna jest własność, której sekcje `get` buduje odpowiedni łańcuch. W sekcji set użyliśmy natomiast metody analizuj. Dzięki temu dysponujemy wygodnym sposobem odczytu i zapisu stanu maszyny Turinga w postaci łańcucha. Korzystając z własności możemy również zdefiniować metodę `ToString`, która pozwoli na automatyczną konwersję do łańcucha.

Listing 14.14. Własność pozwalająca zarówno na zmianę stanu z użyciem łańcuchu w odpowiednim formacie, jak i odczytanie takiego łańcucha dla bieżącego stanu maszyny

```
class StanMaszynyTuringa
{
    ...

    public string ŁańcuchOpisującyStanMaszyny
    {
        get
        {
            string s = new string(Taśma);
            s = s.Insert(PołożenieGłowicy, StanGłowicy.ToString());
            return s;
        }
        set
        {
            StanMaszynyTuringa stan = StanMaszynyTuringa.analizuj(value);
```

```
            Taśma = stan.Taśma;
            StanGłowicy = stan.StanGłowicy;
            PołożenieGłowicy = stan.PołożenieGłowicy;
        }
    }

    public override string ToString()
    {
        return ŁańcuchOpisującyStanMaszyny;
    }
}
```

Zwróćmy jednak uwagę, że kod w sekcji `set` własności `ŁańcuchOpisującyStanMaszyny` jest niemal taki sam, jak kod w zdefiniowanym przed chwilą jednoargumentowym konstruktorze. Aby uniknąć powielania kodu, zmieńmy konstruktor, wykorzystując własność. Pokazuje to listing 14.15.

Listing 14.15. Uproszczenie konstruktora dzięki użyciu własności ŁańcuchOpisującyStanMaszyny

```
public StanMaszynyTuringa(string łańcuchOpisującyStanMaszyny)
{
    ŁańcuchOpisującyStanMaszyny = łańcuchOpisującyStanMaszyny;
}
```

# Maszyna

Cały kod jest pewnie dłuższy od tego, który stworzyliśmy w rozdziale 9. Ale tworząc klasę `MaszynaTuringa` (listing 14.16) zarządzającą całym symulatorem maszyny Turinga docenimy, że warto było się wysilić i zdefiniować zaprezentowane wyżej klasy. Docenimy to również przygotowując testy jednostkowe (zob. zadania na końcu rozdziału).

Jedynymi polami tej klasy będzie `stan` i `program` typu `StanMaszynyTuringa` i `ProgramMaszynyTuringa`. Oba będą prywatne. Publiczny będzie natomiast konstruktor, który zainicjuje oba pola oraz metoda `WykonajProgram` (listing 14.16). Kod tej metody jest niemal identyczny, jak metody `wykonajProgram` z rozdziału 9. – poleceń szuka jednak w obiekcie `program`, korzystając z jego metody `ProgramMaszynyTuringa.ZnajdźPolecenie`. Do tej metody przesyła dwójkę odczytaną z własności `StanMaszynyTuringa.BieżącyStan`, a odbiera dwójkę opisującą nowy stan lub polecenie przesunięcia głowicy.

Listing 14.16. Klasa zarządzająca wszystkimi klasami zdefiniowanymi na potrzeby symulatora maszyny Turinga

```
public class MaszynaTuringa
{
    private StanMaszynyTuringa stan; //aktualny stan
    private ProgramMaszynyTuringa program; //program maszyny Turinga

    public MaszynaTuringa(string łańcuchOpisującyStanMaszyny, string[] kodProgramu)
    {
        stan = new StanMaszynyTuringa(łańcuchOpisującyStanMaszyny);
        program = new ProgramMaszynyTuringa(kodProgramu);
    }

    public string[] WykonajProgram()
    {
        List<string> historia = new List<string>();
        historia.Add(stan.ŁańcuchOpisującyStanMaszyny);
```

```
            Dwójka? polecenie;
            while ((polecenie = program.ZnajdźPolecenie(stan.BieżącyStan)) != null)
            {
                stan.StanGłowicy = polecenie.Value.StanGłowicy;
                switch (polecenie.Value.WartośćLubPolecenieNaTaśmie)
                {
                    case 'L':
                        stan.PołożenieGłowicy--;
                        break;
                    case 'R':
                        stan.PołożenieGłowicy++;
                        break;
                    default:
                        stan.Taśma[stan.PołożenieGłowicy] =
                            polecenie.Value.WartośćLubPolecenieNaTaśmie;
                        break;
                }
                historia.Add(stan.ŁańcuchOpisującyStanMaszyny);
            }
            return historia.ToArray();
        }
    }
```

Zwróćmy uwagę, że klasa `MaszynaTuringa` jest jedyną klasą publiczną w pliku *Turing.cs*. Ponieważ ten plik umieszczony jest w tym samym projekcie, co klasa `Program`, ma ona dostęp także do pozostałych klas, nawet jeżeli nie są one publiczne. Jednak, jeżeli plik *Turing.cs* przenieślibyśmy do biblioteki DLL, dostępna spoza niej będzie tylko klasa `MaszynaTuringa`.

Zmieńmy mimo to nazwę przestrzeni nazw w pliku *Turing.cs* z `MaszynaTuringa2` na `Turing`. Możemy wówczas sprawdzić, że rzeczywiście nie zmienia się dostępność nowych struktur i klas z klasy `Program` (por. zadanie utworzenia biblioteki DLL).

```
using System;
using System.Collections.Generic;

namespace Turing
{
    struct Dwójka : IComparable<Dwójka>
    {
        ...
```

# Rozruch

Sposób, w jaki przygotowywaliśmy program – długo pisząc kod i dopiero na końcu próbując go uruchomić, jest możliwy w książce, ale nie w prawdziwym życiu. Postępując w taki sposób, narażamy się na wielogodzinne odkopywanie się z błędów. Znacznie lepiej by to wyglądało, gdybyśmy każdy z definiowanych typów obłożyli testami jednostkowymi (zob. zadania). Wówczas szanse na to, żeby symulator zadziałał z marszu byłyby większe.

Tak, czy inaczej przechodzimy wreszcie do pierwszego uruchomienia nowej wersji maszyny. To oznacza, że jeszcze raz napiszemy metodę `Main` w klasie `Program` (plik *Program.cs*). Spora jej część będzie taka sama, jak w rozdziale 9., dlatego proponuję skopiować tamten kod i nieco go zmodyfikować. Nie zmienią się polecenia wczytujące z plików stan maszyny i jej program. Zmieni się natomiast ta zasadnicza część umieszczona w sekcji

`try`, która odpowiada za utworzenie i uruchomienie maszyny. Nie musi być w niej już poleceń parsujących stan i kod maszyny – wszystkim zajmują się metody ukryte w poszczególnych klasach i wywoływane w konstruktorze klasy `MaszynaTuringa`. Dzięki użyciu definiowania typów, użytkownik korzysta jedynie z konstruktora tej, który przyjmuje łańcuchy odczytane z plików i metody bezargumentowej `WykonajProgram`, która zwraca „output" programu (oznaczone na listingu 14.17). Korzystanie z przygotowanego przez nas kodu jest wobec tego znacznie łatwiejsze, a tym samym bardziej odporne na błędy korzystającego z naszej klasy programisty.

Listing 14.17. Przykład użycia

```
using System;
using System.IO;

namespace MaszynaTuringa2
{
    class Program
    {
        static void Main(string[] args)
        {
            if (args.Length < 2)
            {
                Console.Error.WriteLine(
                    "Do uruchomienia program wymaga dwóch argumentów");
                return;
            }

            string ścieżkaPlikuProgramu = args[0];
            if (!File.Exists(ścieżkaPlikuProgramu))
            {
                Console.Error.WriteLine(
                    "Brak pliku z kodem programu " + ścieżkaPlikuProgramu);
                return;
            }
            else Console.WriteLine("Plik z kodem programu: " + ścieżkaPlikuProgramu);
            string[] kodProgramu = File.ReadAllLines(ścieżkaPlikuProgramu);

            string ścieżkaPlikuTaśmy = args[1];
            if (!File.Exists(ścieżkaPlikuTaśmy))
            {
                Console.Error.WriteLine(
                    "Brak pliku ze stanem maszyny " + ścieżkaPlikuTaśmy);
                return;
            }
            else Console.WriteLine("Plik ze stanem maszyny: " + ścieżkaPlikuTaśmy);
            string łańcuchOpisującyStanMaszyny = File.ReadAllText(ścieżkaPlikuTaśmy);

            Console.WriteLine(
                "Początkowy stan maszyny: " + łańcuchOpisującyStanMaszyny);
            Console.WriteLine("Program:");
```

```
            foreach (string linia in kodProgramu) Console.WriteLine(linia);

            try
            {
                Turing.MaszynaTuringa maszyna =
                    new Turing.MaszynaTuringa(łańcuchOpisującyStanMaszyny, kodProgramu);
                string[] historia = maszyna.WykonajProgram();

                Console.WriteLine("\nEwolucja stanu maszyny:");
                foreach (string linia in historia)
                    Console.WriteLine(linia);
            }
            catch (Exception exc)
            {
                ConsoleColor color = Console.ForegroundColor;
                Console.ForegroundColor = ConsoleColor.Red;
                Console.Error.WriteLine("Błąd: " + exc.Message);
                Console.ForegroundColor = color;
            }
        }
    }
}
```

Aby uruchomić program, musimy do niego skopiować dwa pliki tekstowe *program.txt* i *taśma.txt*, które przygotowaliśmy w rozdziale 9. Można je przenieść myszką np. z *Eksploratora Windows* i upuścić na podokno *Eksplorator rozwiązań* w środowisku *Visual Studio*. Ważne, żeby umieścić je w projekcie. Po dodaniu ich do projektu, należy jeszcze zmienić ich własność *Kopiuj do katalogu wyjściowego* (okno *Właściwości*) na *Zawsze kopiuj* lub *Kopiuj, jeżeli nowszy*. Następnie wejdźmy do menu *Projekt, Właściwości MaszynaTuringa2...* i na zakładce *Debuguj* w polu *Argumenty aplikacji* wpiszmy „program.txt taśma.txt”. Po tych przygotowaniach możemy wreszcie uruchomić program naciskając klawisz *F5*.

## Wezwanie do refaktoryzacji

Nie należy bać się optymalizacji kodu (do czego daje odwagę obłożenie go testami jednostkowymi). Zastanówmy się na przykład, czy rzeczywiście w kodzie symulatora potrzebna jest struktura `Czwórka`? Pomimo, że ten rzeczownik był często używany, gdy opisywaliśmy „teorię” maszyny Turinga, to w kodzie czwórka rozpada się na dwie dwójki i to struktura `Dwójka` pojawia się w nim znacznie częściej. W szczególności program nie jest kolekcją czwórek, a słownikiem par dwójek.

# Zadania

1. Umieść klasy symulatora maszyny Turinga w bibliotece o nazwie *Turing*. Sprawdź, czy rzeczywiście dostępna z zewnątrz będzie tylko klasa `MaszynaTuringa`.
2. Przeprowadź refaktoryzację klas symulatora maszyny Turinga, w ramach której usuniesz klasę `Czwórka`. Po każdym etapie, po którym można skompilować rozwiązanie, badaj wyniki kompletu testów jednostkowych.