Poznań, 22.01.2010

Adam MATYSIAK
Michał URBANIAK
Tomasz ŻYŹNIEWSKI
Gr. IV

DOKUMENTACJA PROJEKTU „*CHANGING MONEY*”

Informatyka, sem. V

I. Opis zadania

Dana jest pewna liczba monet o różnych nominałach. Niech c_i oznacza liczbę dostępnych monet o nominale v_i, dla $i = 1, \ldots, n$. Wyznaczyć minimalną liczbę monet niezbędną do wypłacenia kwoty PRICE.

Opracować algorytm rozwiązujący powyższy problem w języku Prolog, wykorzystując bibliotekę więzów na ograniczonych dziedzinach (CLP(FD)).

II. Założenia realizacyjne

1. **Założenia dodatkowe**

   Nominały monet są odpowiednikami polskiego systemu monetarnego.
2. **Metody, strategie oraz algorytmy wykorzystane do rozwiązania zadania**
   a. Informacje o dostępnych monetach reprezentuje tablica $[c_1, \ldots, c_n]$ dla i = 1,…,n, gdzie pozycja w tablicy determinuje nominał, którego dotyczy przechowywana wartość.
   b. Rozwiązanie problemu stanowi ciąg liczb $a_1, \ldots, a_n$, gdzie $0 \le a_i \le c_i$ i = 1,…,n oraz spełnione jest równanie $c_1*v_1 + c_2*v_2 + \ldots + c_n*v_n = PRICE$.
   c. Wszystkie wartości monet zostały podane w tej samej jednostce (w groszach), by uniknąć zbędnych przekształceń.
3. **Języki programowania, narzędzia informatyczne i środowiska używane do implementacji systemu**
   a. Język Prolog i środowisko SWI-Prolog
   b. Język C# i Microsoft Visual Studio 2008
   c. Biblioteka SbsSW.SwiPlCs napisana w języku C#

III. Opis implementacji:

- I wersja projektu (oparta na mechanizmach prologa) - 26.10.2009

```
change(PRICE,
[  ZL5,   ZL2,   ZL1,   G50,   G20,   G10,   G05,   G02,   G01],
[C_ZL5, C_ZL2, C_ZL1, C_G50, C_G20, C_G10, C_G05, C_G02, C_G01]) :-
divider(PRICE, 500, X1), member(ZL5, X1), ZL5 =< C_ZL5,      /* pięciozłotówki */
divider(PRICE, 200, X2), member(ZL2, X2), ZL2 =< C_ZL2,      /* dwuzłotówki */
divider(PRICE, 100, X3), member(ZL1, X3), ZL1 =< C_ZL1,      /* złotkówki */
divider(PRICE, 050, X4), member(G50, X4), G50 =< C_G50,      /* pięćdziesięciogroszówki */
divider(PRICE, 020, X5), member(G20, X5), G20 =< C_G20,      /* dwudziestogroszówki */
divider(PRICE, 010, X6), member(G10, X6), G10 =< C_G10,      /* dziesięciogroszówki */
divider(PRICE, 050, X7), member(G05, X7), G05 =< C_G05,      /* pięciogroszówki */
divider(PRICE, 020, X8), member(G02, X8), G02 =< C_G02,      /* dwugroszówki */
SUM is ZL5 * 500 + ZL2 * 200 + ZL1 * 100 + G50 * 50 + G20 * 20 + G10 * 10 + G05 * 5 + G02 * 2,
SUM =< PRICE,
G01 is PRICE - SUM, G01 =< C_G01.

divider(PRICE, VALUE, X2) :- divide(PRICE, VALUE, X1), reverse(X1, X2).

divide(PRICE, VALUE, [0]) :- VALUE > PRICE.
divide(PRICE, VALUE, [0, 1]) :- VALUE =:= PRICE.
divide(PRICE, VALUE, X) :- VALUE < PRICE, integers(PRICE, VALUE, X1), append([0], X1, X).
```

```
integers(PRICE, VALUE, [1|X]) :- integers(PRICE, VALUE, 2, X).
integers(PRICE, VALUE, I, [I|X]) :- I * VALUE =< PRICE, I1 is I + 1, integers(PRICE, VALUE, I1, X).
integers(PRICE, VALUE, I, []) :- I * VALUE > PRICE.
```

Rozwiązanie to opierało się na mechanizmach prologa. Przykładowe wywołanie:

```
?- change(51, [ZL5, ZL2, ZL1, G50, G20, G10, G05, G02, G01], [1, 2, 3, 2, 3, 4, 2, 6, 5]).
```

Najlepsze rozwiązanie zostało zwrócone przez program:

```
ZL5 = 0,
ZL2 = 0,
ZL1 = 0,
G50 = 1,
G20 = 0,
G10 = 0,
G05 = 0,
G02 = 0,
G01 = 1
```

Czyli by rozmienić 51 groszy należy wykorzystać 1 pięćdziesięciogroszówkę i 1 jednogroszówkę.

Mechanizm więzów został zastąpiony predykatem *divider*, który generuje możliwe przedziały wartości dla każdej monety oddzielnie.

Powyższy program rozwiązuje problem całkowicie, zwraca poprawne wyniki, ale nie korzystał z biblioteki więzów, która była jednym z wymagań dot. projektu, dlatego też przygotowano drugą wersję projektu.

- II wersja projektu (na więzach) - 09.11.2009

Po zapoznaniu się z biblioteką więzów, na IV laboratoria przygotowaliśmy nową wersję programu, która tym razem wykorzystuje programowanie logiczne z więzami na skończonych dziedzinach. Oto kod:

```
:- use_module(library(clpfd)).

change(PRICE,
       [  ZL5,   ZL2,   ZL1,   G50,   G20,   G10,   G05,   G02,   G01],
       [C_ZL5, C_ZL2, C_ZL1, C_G50, C_G20, C_G10, C_G05, C_G02, C_G01]) :-
           Vars = [ZL5, ZL2, ZL1, G50, G20, G10, G05, G02, G01],
           ZL5 in 0..C_ZL5,
           ZL2 in 0..C_ZL2,
           ZL1 in 0..C_ZL1,
           G50 in 0..C_G50,
           G20 in 0..C_G20,
           G10 in 0..C_G10,
           G05 in 0..C_G05,
           G02 in 0..C_G02,
           G01 in 0..C_G01,
           ZL5 * 500 + ZL2 * 200 + ZL1 * 100 + G50 * 50 + G20 * 20 + G10 * 10 + G05 * 5 + G02 * 2 +
G01 #= PRICE,
           labeling([min(ZL5 + ZL2 + ZL1 + G50 + G20 + G10 + G05 + G02 + G01)], Vars).
```

Wywołanie programu jest identyczne do tego z I wersji programu i wartości zwrócone również. Program zajmuje mniej linijek, nie wykorzystuje dodatkowych predykatów, jest bardziej zrozumiały i (dla większych kwot) szybszy.

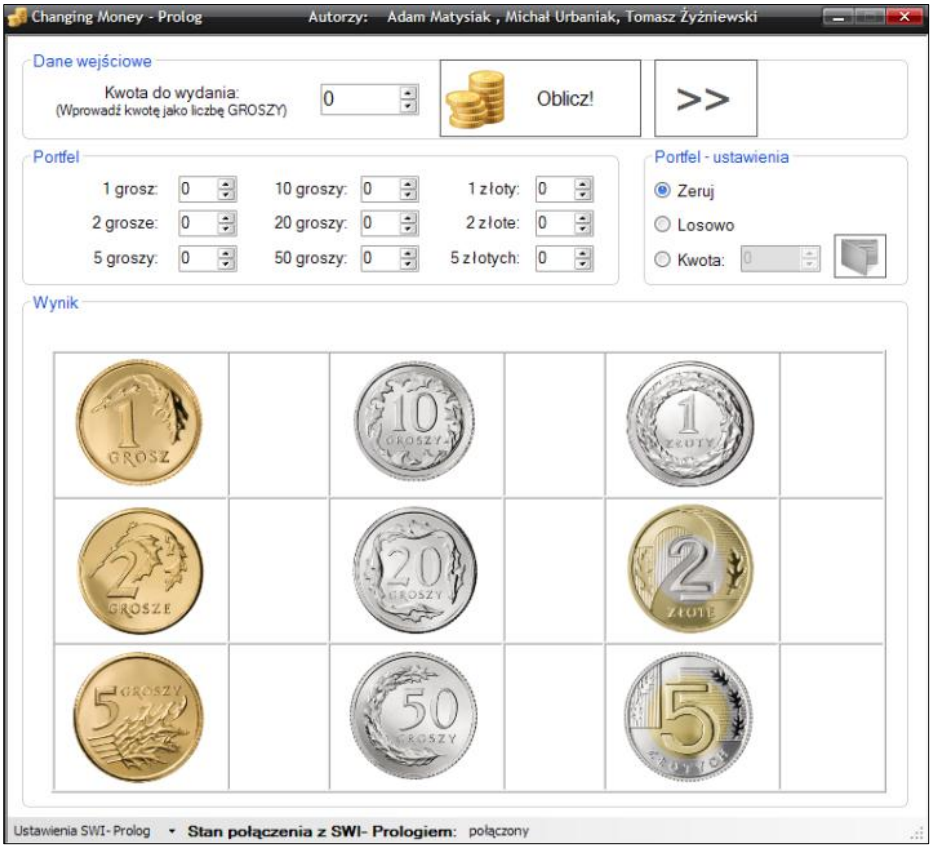
Opis parametrów predykatu *change*:

- PRICE – kwota do rozmienienia podana w groszach,
- [ZL5, ZL2, ZL1, G50, G20, G10, G05, G02, G01] – lista reprezentująca liczbę poszczególnych monet potrzebnych do rozmienienia kwoty PRICE.
- [C_ZL5, C_ZL2, C_ZL1, C_G50, C_G20, C_G10, C_G05, C_G02, C_G01] - lista będąca wirtualnym „portfelem”, gdzie każda pozycja oznacza liczbę monet danego nominału dostępną w puli.

Predykat *labeling* połączony z *min* służy do generowania wyników w kolejności od najbardziej pożądanej, czyli tej, gdzie wykorzystano jak najmniej monet.

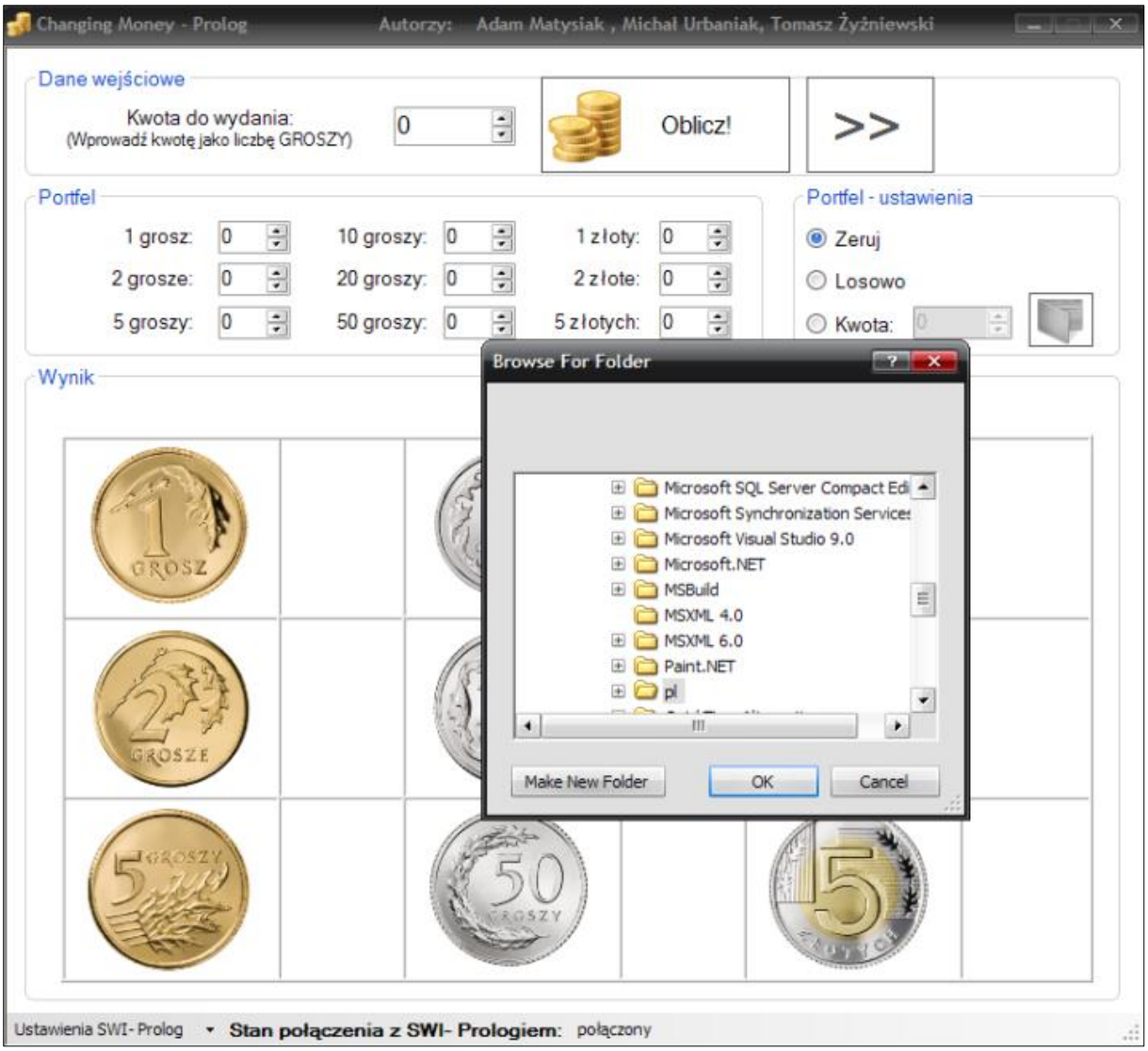
- III wersja projektu (graficzny interfejs użytkownika / C#) - 07.12.2009

W III wersji naszego projektu postanowiliśmy dodać graficzny interfejs użytkownika stworzony w środowisku .NET z wykorzystaniem języka C#. Projekt zyskuje w ten sposób na funkcjonalności oraz przejrzystości.



Rysunek 1: Główne okno programu

Pierwszym zadaniem użytkownika jest podanie ścieżki dostępu do folderu z pakietem SWI-Prolog. W tym celu należy kliknąć **Ustawienia SWI-Prolog** -> **Zlokalizuj SWI-Prolog** na pasku stanu aplikacji**.** Teraz wybieramy odpowiedni katalog na dysku. Stan połączenia zmieni się na : **połączony**.



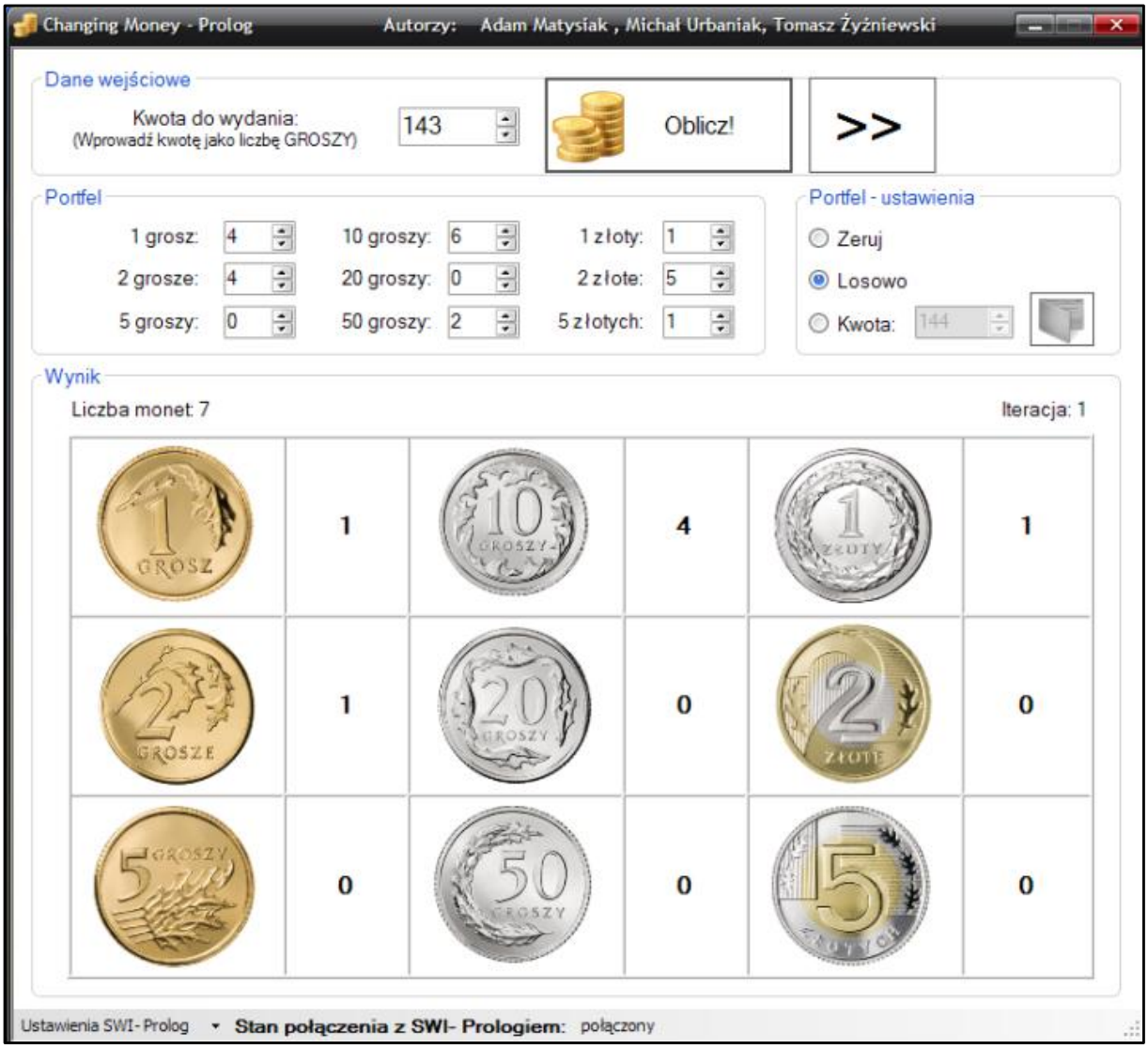
Rysunek 2: Wybór lokalizacji SWI-Prolog

Kwotę (w groszach) do wydania wpisujemy w odpowienio opisane pole u góry, a zawartość portfela możemy określać na trzy sposoby:

a) Użytkownik sam ustala, jakie monety ma w portfelu,
b) Zawartość portfela jest losowa,
c) W portfelu znajduję sie określona przez użytkownika kwota.

By uzyskać wynik działania programu, czyli liczbę konkretnych monet jakie należy wydać, należy nacisnąć przycisk „**Oblicz!**”. Pokaże on najlepszy rezultat, czyli zawierający najmniejszą liczbę monet.

Kolejne najlepsze dopasowania liczby monet do wydania określonej kwoty, uzyskamy przez naciśnięcie przycisku „**>>**” (obok przycisku Oblicz). W przypadku braku możliwości wydania podanej kwoty, program wygeneruje odpowiedni komunikat.

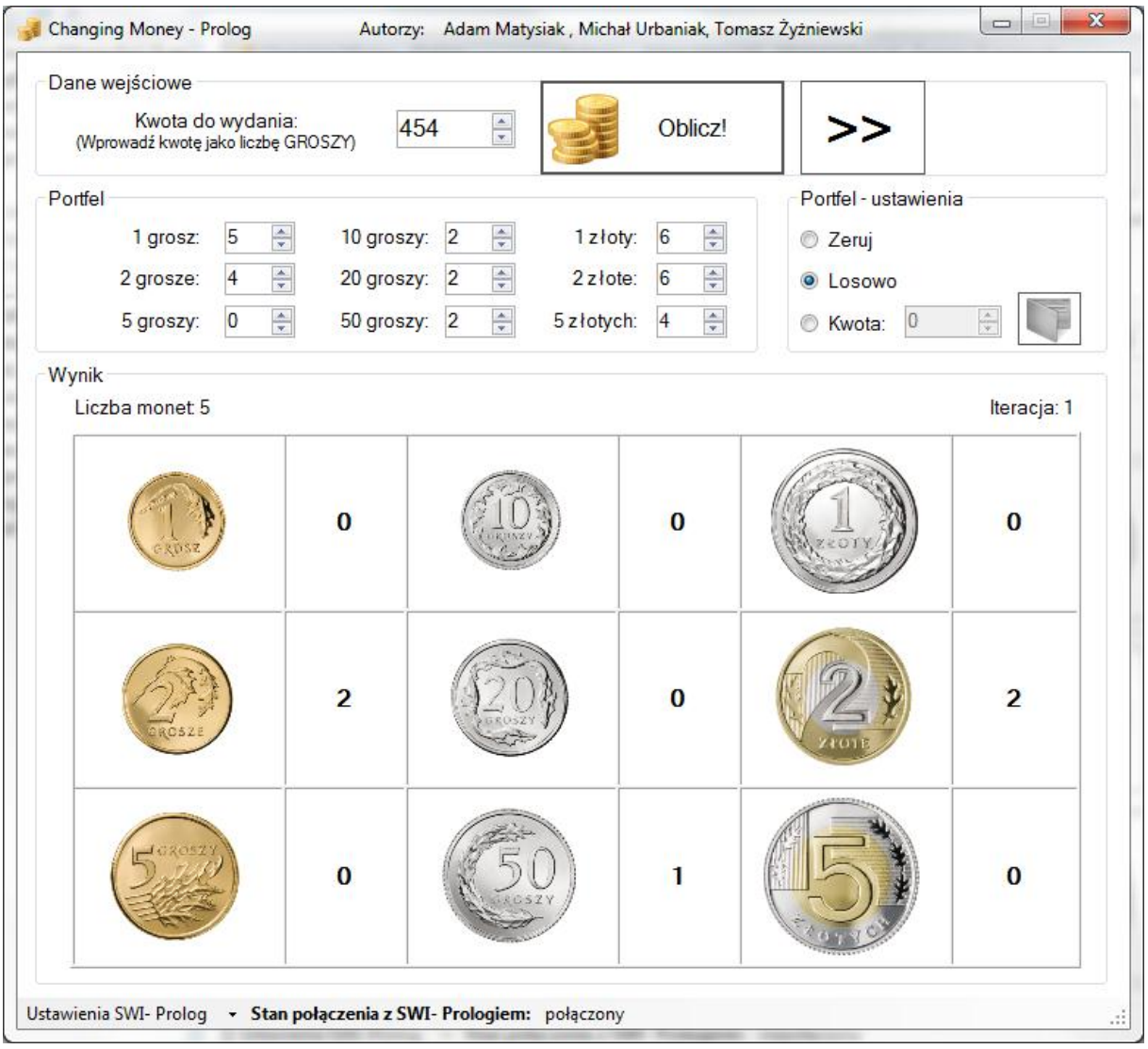


Rysunek 3: Przykładowe zapytanie o najmniejszą liczbę monet potrzebną do wydania podanej kwoty

- IV wersja projektu (graficzny interfejs użytkownika / C#) - 04.01.2010

W ostatecznej wersji naszego projektu zmieniliśmy dwie rzeczy:

- Przycisk „**>>**” pokazuje następne dopasowania tylko wtedy, gdy jest ono tak samo dobre, czyli gdy wykorzystuje taką samą liczbę monet, jak pierwszy zwrócony wynik. W przeciwnym razie, program informuje użytkownika o braku następnych możliwości.
- Zmieniono rozmiary obrazków monet – teraz ich wielkość jest proporcjonalna względem siebie do ich rzeczywistych odpowiedników.



Rysunek 4: Przykładowe zapytanie o najmniejszą liczbę monet potrzebną do wydania podanej kwoty

IV. Testy.

Test dla 51 groszy:

```
1 ?- change(51, [ZL5, ZL2, ZL1, G50, G20, G10, G05, G02, G01], [1, 2, 3, 2, 3, 4, 2, 6, 5]).
ZL5 = 0,
ZL2 = 0,
ZL1 = 0,
G50 = 1,
G20 = 0,
G10 = 0,
G05 = 0,
G02 = 0,
G01 = 1 ; % żądanie kolejnego najlepszego dopasowania monet
---------------------------------------------------------------------------
ZL5 = 0,
ZL2 = 0,
ZL1 = 0,
G50 = 0,
G20 = 2,
G10 = 1,
G05 = 0,
G02 = 0,
G01 = 1 ; % żądanie kolejnego najlepszego dopasowania monet
---------------------------------------------------------------------------
ZL5 = 0,
ZL2 = 0,
ZL1 = 0,
G50 = 0,
G20 = 1,
G10 = 3,
G05 = 0,
G02 = 0,
G01 = 1 … % kolejne możliwe rozwiązania
```

Test dla 777 groszy:

```
1 ?- change(777, [ZL5, ZL2, ZL1, G50, G20, G10, G05, G02, G01], [1, 2, 3, 2, 3, 4, 2, 6, 5]).
ZL5 = 1,
ZL2 = 1,
ZL1 = 0,
G50 = 1,
G20 = 1,
G10 = 0,
G05 = 1,
G02 = 1,
G01 = 0 ;

---------------------------------------------------------------------------

ZL5 = 1,
ZL2 = 0,
ZL1 = 2,
G50 = 1,
G20 = 1,
G10 = 0,
G05 = 1,
G02 = 1,
G01 = 0 ;

---------------------------------------------------------------------------

ZL5 = 1,
ZL2 = 1,
ZL1 = 0,
G50 = 1,
G20 = 0,
G10 = 2,
G05 = 1,
G02 = 1,
G01 = 0 .
```

Test dla 1428 groszy (pamiętajmy, że mamy ograniczony portfel – w tym przypadku możemy wydać kwotę nie większą niż 1427 groszy):

```
4 ?- change(1428, [ZL5, ZL2, ZL1, G50, G20, G10, G05, G02, G01], [1, 2, 3, 2, 3, 4, 2, 6, 5]).
false.  % nieznaleziono żadnego rozwiązania.
```

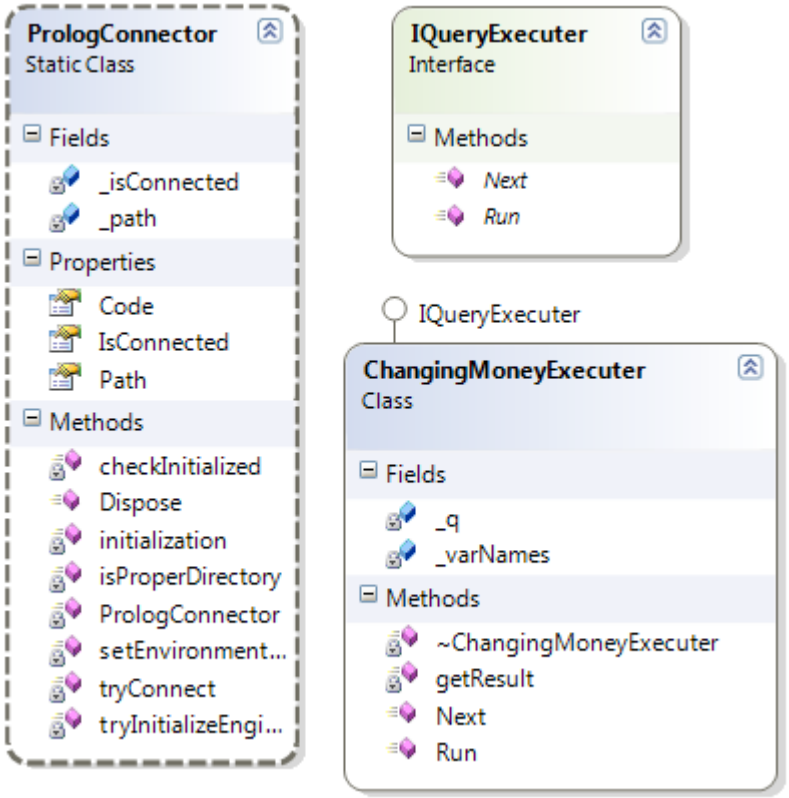
V. Kod źródłowy

a) Program prologowy

```
:- use_module(library(clpfd)).

change(PRICE,
       [  ZL5,   ZL2,   ZL1,   G50,   G20,   G10,   G05,   G02,   G01],
       [C_ZL5, C_ZL2, C_ZL1, C_G50, C_G20, C_G10, C_G05, C_G02, C_G01]) :-
           Vars = [ZL5, ZL2, ZL1, G50, G20, G10, G05, G02, G01],
           ZL5 in 0..C_ZL5,
           ZL2 in 0..C_ZL2,
           ZL1 in 0..C_ZL1,
           G50 in 0..C_G50,
           G20 in 0..C_G20,
           G10 in 0..C_G10,
           G05 in 0..C_G05,
           G02 in 0..C_G02,
           G01 in 0..C_G01,
           ZL5 * 500 + ZL2 * 200 + ZL1 * 100 + G50 * 50 + G20 * 20 + G10 * 10 + G05 * 5 + G02 *
2 + G01 #= PRICE,
           labeling([min(ZL5 + ZL2 + ZL1 + G50 + G20 + G10 + G05 + G02 + G01)], Vars).
```

b) Program interfejsu



Klasa **PrologConnector** odpowiada za połączenie aplikacji z silnikiem SWI-Prolog.

Interfejs **IQueryExecuter** określa zbiór metod, które należy implementować w klasie, by aplikacja mogła ją wykorzystać do wykonywania zapytań.

Klasa **ChangingMoney-Executer** wywołuje zapytanie *change* i pobiera wyniki.

PrologConnector – połączenie się z prologiem

Aplikacja wykorzystuje bibliotekę SbsSW.SwiPlCs (dostępną na stronie http://www.lesta.de/prolog/swiplcs/Generated/). Za jej pomocą nasz program łączy się z silnikiem SWI-Prolog. Dlatego też przy pierwszym uruchomieniu naszego programu, należy wskazać na lokalizację programu SWI-Prolog.

Dalszą obsługą połączenia z SWI-Prolog zajmuje się statyczna klasa PrologConnector.

Zawiera ona tylko 3 publiczne pola:

- **Code** – zawiera kod napisany w języku prolog, który ma być ładowany przy połączeniu się programu z SWI-Prologiem.
- **IsConnected** – zwraca true, jeśli jesteśmy połączeni z silnikiem SWI-Prolog.
- **Path** – zawiera ścieżkę do katalogu SWI-Prolog. Gdy się zmieni tą wartość, klasa spróbuje połączyć się automatycznie z silnikiem SWI-Prolog pod nową ścieżką.

Dzięki takiemu interfejsowi publicznemu – cała logika łączenia i zarządzania połączeniem jest ukryta przed programistą. Jedyne co jest ważne – ścieżka do SWI-Prolog, stan połączenia i kod, jaki należy mieć wczytany – są dostępne dla użytkownika klasy.

Oto najciekawszy fragment (PrologConnector.cs):

```
// próba połączenia się z silnikiem SWI-PROLOG
private static void tryConnect()
{
    if (!isProperDirectory()) return;

    setEnvironmentVariables();

    tryInitializeEngine();

    checkInitialized();
}

// sprawdza czy podany katalog zawiera plik boot32.prc
// potrzebny do działania bibloteki SwiPlCs
private static bool isProperDirectory()
{
    if (File.Exists(string.Format("{0}\\{1}", Path, "boot32.prc")))
        return true;

    ErrorMessager.NotInstalledInThisDirectory();
    return false;
}

// próba uruchomienia silnika SWI-Prolog
// i obsługa błędów
private static void tryInitializeEngine()
{
    try
    {
        initialization();
    }
    catch (PlException e)
    {
        ErrorMessager.FileNotFound(e);
    }
```

```
            catch (FileNotFoundException e)
            {
                ErrorMessager.NotInstalledInThisDirectory();
            }
            catch (AccessViolationException e)
            {
                try
                {
                    initialization();
                }
                catch (Exception ex)
                {
                    ErrorMessager.InitializationError(ex);
                }
            }
        }

        // uruchamianie silnika SWI-Prolog
        // przesłanie do prologa zapisanego kodu w polu Code
        private static void initialization()
        {
            PlEngine.Initialize(new string[] { "-q" });

            if (Code.Count == 0)
            {
                ErrorMessager.NoCodeReaded();
                return;
            }

            foreach(string c in Code)
                PlQuery.PlCall(c);
        }

        // sprawdzenie czy jest nawiązane połączenie
        private static bool checkInitialized()
        {
            if (PlEngine.IsInitialized)
                _isConnected = true;
            else
            {
                _isConnected = false;
                Dispose();
            }

            return IsConnected;
        }

        // ustawienie zmiennych środowiskowych
        // potrzebnych bibliotece SwiPlCs
        private static void setEnvironmentVariables()
        {
            Environment.SetEnvironmentVariable("SWI_HOME_DIR", Path);
            Environment.SetEnvironmentVariable("PATH",
String.Format(@"{0}\bin", Path));
            Environment.SetEnvironmentVariable("LIB",
String.Format(@"{0}\lib", Path));
        }

        // przerwanie połączenia z silnikiem SWI-Prolog
        public static void Dispose()
        {
            PlEngine.PlCleanup();
        }
```

Dzięki takiemu wykorzystaniu statycznej klasy – mamy przez cały czas działania programu managera połączenia z silnikiem SWI-Prolog.

Kilka kwestii jest wartych uwagi:

- Należy sprawdzać przy każdej zmianie ścieżki do katalogu z SWI-Prolog, czy katalog ten zawiera plik boot32.prc (dla wersji 32-bitowej), gdyż jest wymagany przez bibliotekę SwiPlCs. W przypadku wybrania niepoprawnego katalogu, zostanie wygenerowany odpowiedni komunikat.
- Do zestawienia połączenia z prologiem, potrzebne są także trzy zmienne środowiskowe: SWI_HOME_DIR, PATH i LIB. Dokumentacja biblioteki SwiPlCs wspomina tylko o tej pierwszej, przez co na systemach Windows Vista i 7 niemożliwe jest połączenie się z silnikiem SWI-Prolog. Potrzebne są ustawione wszystkie 3 ścieżki.
- Po połączeniu się z SWI-Prologiem, należy przesłać kod napisany w prologu, z którego będziemy korzystać. Przykładowy kod wygląda następująco:

```
string consult = String.Format("consult('{0}\\ChangingMoney.pl')",
Environment.CurrentDirectory);
consult = consult.Replace("\\", "\\\\");

PrologConnector.Code.Add(consult);
```

Skomplikowanie powyższego kodu jest spowodowane błędem w bibliotece SwiPlCs. Pierwsze dwie linie do zmiennej *consult* wstawiają ścieżkę do pliku z programem prologowym znajdującym się w tym samym katalogu, co wywoływany program (np. consult('D:\\cm\\ChangingMoney.pl') – dwa slash'e, gdyż w C# slash wykorzystuje się do znaków specjalnych).

Następnie należy zdublować wszystkie slashe, gdyż biblioteka SwiPlCs modyfikuje przesłany tekst i ciąg „\\" zamienia na „\", co silnik SWI-Prolog traktuje jako znak specjalny. Dlatego podwojenie slash'y rozwiązuje ten problem, gdyż „\\\\" zostanie przekształcone na „\\" co jest oczekiwanym przez nas ciągiem znaków.

Dopiero ścieżkę w takiej postaci można przesłać do silnika SWI-Prolog.

ChangingMoneyExecuter - wywołanie zapytania – uzyskanie rezultatów

Najprostszy kod wysyłający zapytanie do silnika SWI-Prolog wygląda następująco:

```
            // utworzenie obiektu wywołującego zapytanie do SWI-Prolog
            IQueryExecuter _executer = new ChangingMoneyExecuter();

            // pobranie kwoty jaką należy rozmienić
            int kwota = (int)this.kwota.Value;
            // tekst zawierający nazwy zmiennych
            const string variables = "ZL5, ZL2, ZL1, G50, G20, G10, G05,
G02, G01";
            // lista zawierająca liczbę monet każdego rodzaju w portfelu
            // taka sama postać jak wyżej np. „1, 2, 3, 5, 7, 9, 4, 5, 6"
            string portfelList = listaMoneyWPortfelu();

            // wywołanie zapytania z trzema parametrami
            // uzyskanie wyników w postaci słownika np.
            // ['ZL5'] => 2,
            // ['ZL2'] => 0,...
            Dictionary<string, string> result = _executer.Run(kwota,
variables, portfelList);
```

Powyższy kod ułatwia wywoływanie zapytań, lecz cała logika wykonywania zapytania i pobierania wyniku znajduje się w klasie ChangingMoneyExecuter:

```
// klasa wywołująca predykat change
// interfejs IQueryExecuter nakazuje implementację dwóch metod:
// Run(params object[]) i Next()
public class ChangingMoneyExecuter : IQueryExecuter
{
    // obiekt zapytania prologowego - zawiera wyniki
    private PlQuery _q;
    // tablica nazw zmiennych
    private string[] _varNames;

    // wywołanie zapytania z danymi parametrami po raz pierwszy
    public Dictionary<string, string> Run(params object[] vars)
    {
        // jeżeli istnieje w pamięci aktywne zapytanie, usunięcie go
        if(_q != null) _q.Dispose();

        // utworzenie zapytania z otrzymanych parametrów
        // {0} - kwoty podanej w groszach
        // {1} - listy etykiet zmiennych
        // {2} - listy liczby monet o podanych nominałach w portfelu
        string query = string.Format("change({0}, [{1}], [{2}]).", vars);

        // wykonanie powyższego zapytania
        _q = new PlQuery(query);

        // zapamiętanie nazw zmiennych
        _varNames = vars[1].ToString().Replace(" ", "").Split(',');

        // zwrócenie otrzymanego wyniku
        return getResult();
    }

    // wywołanie następnych wyników (odpowiednik ";" w SWI-Prolog)
    public Dictionary<string, string> Next()
    {
        return getResult();
    }

    // funkcja zwracająca wyniki w słowniku z obiektu zapytania
    private Dictionary<string, string> getResult()
    {
        Dictionary<string, string> result = new Dictionary<string, string>();

        // dla każdego zbioru rozwiązań...
        foreach (PlQueryVariables qv in _q.SolutionVariables)
        {
            // dla każdej zmiennej...
            foreach (string v in _varNames)
                // zapisz wynik do słownika
                result.Add(v, qv[v].ToString());

            // po zebraniu wyników rozwiązania, przerwij wykonywanie
            // interesuje nas jeden zbiór rozwiązań naraz
            break;
        }
        // zwróć wyniki
        return result;
    }
```

Powyższy kod w całości odpowiada za wywoływanie i pobieranie kolejnych wyników zapytania.

VI. Zastosowanie programu

Przystosowaliśmy projekt do polskich realiów, zamieniając amerykańskie monety (50 centów, 25 centów, 10 centów, 5 centów , 1 cent) na polskie monety (5zł, 2zł, 1zł, 50gr, 20 gr, 10gr, 5gr, 2gr, 1gr). Dzięki temu program ten mógłby być wykorzystany z powodzeniem w każdego rodzaju automatach z napojami czy do gier. Trzeba by jedynie dodać mechanizm wyliczania ile reszty należy zwrócić klientowi.

Po małej przeróbce – zamianie monet na banknoty (200zł, 100zł, 50zł, 20zł, 10zł) program można by zastosować w bankomatach, które wymagają sprawdzania dostępności banknotów. Program zwracałby zawsze najmniejszą liczbę dostępnych banknotów.

Projekt sprawdziłby się również w kasach fiskalnych.

Na podstawie powyższych przykładów widać mnogość zastosowań projektu.

VII. Bibliografia

- http://www.mozart-oz.org/documentation/fdt/node26.html#section.scripts.change
- http://www.swi-prolog.org/pldoc/doc_for?object=section%282%2c%20%27A.7%27%2c%20swi%28%27%2fdoc%2fManual%2fclpfd.html%27%29%29
- http://en.wikipedia.org/wiki/Constraint_programming
- http://en.wikibooks.org/wiki/Prolog/Constraint_Logic_Programming