

Projekt indywidualny

Wykonanie biblioteki Managed C# dla pakietu Aria

Kamil Nowak
Informatyka, sem. VI

Spis treści

1.Wprowadzenie.....	2
2.Własna aplikacja do generwania mapowań.....	2
3.Swig	3
4.Swig- użycie.....	3
5.Architektura biblioteki.....	5
6.Użycie biblioteki.....	5
7.Przykład użycia.....	6
8.Tworzenie własnej biblioteki.....	7
9.Problemy.....	7

1. Wprowadzenie

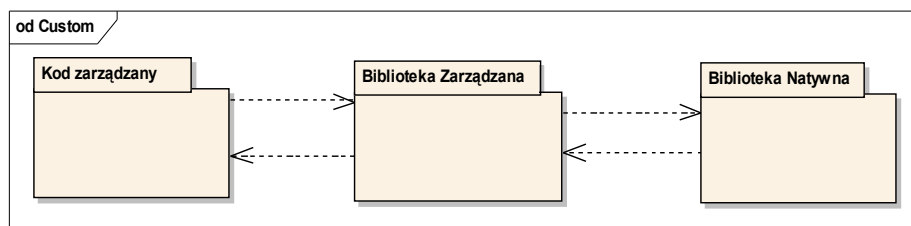
Celem projektu było stworzenie biblioteki umożliwiającej wygodne tworzenie aplikacji używającej pakietu ARIA w środowisku Microsoft Visual Studio .NET 2003/2005.

Pakiet ARIA jest to zestaw klas w języku C++, stanowiących interfejs programistyczny do sterowania oraz zarządzania robotami stworzonymi przez firmę ActivMedia Robotics. W swojej formie klasy te najwygodniej jest używać do tworzenia programów kompilowanych bezpośrednio do kodu maszynowego – aplikacji win32/linux.

W ostatnich latach coraz bardziej rozpowszechnia się idea pisania programów tłumaczonych do kodu pośredniego (zarządzanego) i uruchamianego na maszynie wirtualnej np CLR. Jednak stosowanie zwykłych, niezarządzanych bibliotek w programach pod tę maszynę jest dość uciążliwe. Microsoft udostępnił kilka rozwiązań, które pozwalają mieszać te dwa rodzaje kodu, ale używanie ich bezpośrednio w kodzie głównym nie jest zbyt eleganckie i czytelne.

Aby ułatwić tworzenie zarządzanych programów, gdzie dysponujemy jedynie bibliotekami niezarządzanymi, najłatwiej stworzyć zestaw klas proxy opakowujących klasy natywne. Pozwoli to na wygodne pisanie kodu ze wszystkimi udogodnieniami oferowanymi przez środowisko programistyczne.

W podejściu tym każda klasa proxy ma swój odpowiednik w postaci klasy niezarządzanej, a każde wywołanie metody proxy powoduje wywołanie metody natywnej.



Podczas tworzenia biblioteki pośredniej dla pakietu Aria podjęto dwie koncepcje, z której do końca zrealizowana została tylko jedna. Poniżej znajdują się krótki opis pierwszego rozwiązania oraz dokładniejszy rozwiązania drugiego.

2. Własna aplikacja do generowania mapowań

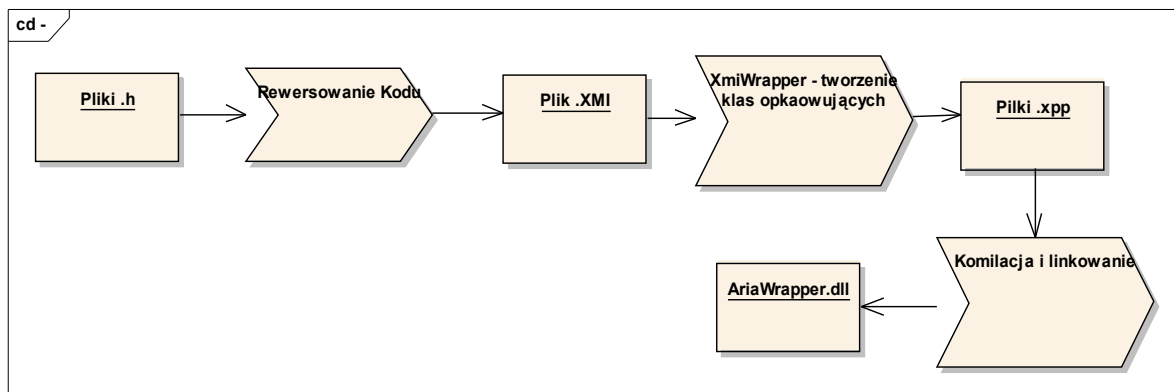
Pierwszą drogą jaka została obrana przy realizacji projektu to napisanie nowej aplikacji generującej opakowujący kod.

Program XmiWrapper miał na wejście pobierać plik w formacie XMI, który to zawierał XML-owe definicje klas oraz opisy zależności występujących między nimi. Na jego podstawie generowane miały być klasy C++/CLI opakowujące wywołania natywne. Po zakończeniu działania wynikowe pliki z kodem miały być kompilowane i linkowane jako biblioteka .NET.

O ile napisanie programu generującego nie stanowiło większych przeszkód, to problem pojawił się przy znalezieniu dobrego narzędzia do rewersowania kodu C++ i eksportu do XMI. Przetestowanych zostało kilka programów, radzących sobie lepiej lub gorzej, lecz przy bardziej zaawansowanych konstrukcjach takich jak np. szablony klas tworzone definicje nie były poprawne. Aplikacjami użyte do generacji to m.in:

- Enterprise Architect
- Microsoft Visio
- Visual Paradigm

Schemat postępowania:



W rzeczywistości część aplikacji XmiWrapper powstała, ale prace nad jej rozwojem zostały zaniechane z powodu natknięcia się na już gotowe narzędzie, które w dużym stopniu uprościło wykonywanie projektu.

3. Swig

Swig jest to narzędzie, które pierwotnie stworzone zostało do generowania interfejsu do programów C/C++ dla języków skryptowych. Z czasem narzędzie się rozrosło i obsługuje również takie języki jak Java i C#.

Użycie swig-a może polegać jedynie na uruchomieniu programu z odpowiednimi parametrami, wówczas na wyjściu otrzymamy wygenerowane klasy w rządany języku. Jednak w przypadkach, kiedy chcemy wyspecyfikować otrzymany rezultat możemy stworzyć plik *.i, w którym zawrzemy odpowiednie dyrektywy.

Parsując kod przy użyciu Swig-a nie musimy nic wiedzieć o metodzie wywoływania metod natywnych w języku docelowym. Narzędzie wykona wszystko za nas.

W przypadku C# metodą wywoływania funkcji niezarządzanych jest P/Invoke. Odbyna się to za pomocą klauzuli `[DllImport("nazwa_biblioteki.dll", EntryPoint="nazwa_metody")]`, która umieszczana jest przed deklaracją extern danej metody.

Pakiet Aria zawierał już plik wrapper.i dla języków Java oraz Python.

4. Swig- użycie

Aby wygenerować klasy mapujące zdefiniowany został plik wrapper.i dla języka C#:

```
//Definicja nazwy modułu
%module AriaWrapper

//Pliki nagłówkowe, które mają być dołączone w pliku klasy pośredniczącej
%{
#include "Aria.h"
#include "ArSoundPlayer.h"
#include "ArSpeech.h"
#include "ArSoundsQueue.h"
#include "ArExport.h"
%}

//Specyfikacja użytego szablonu - std::vector
#include "std_vector.i"
%template(ArPoseWithTimeVector) std::vector<ArPoseWithTime>;
%template(ArSensorReadings) std::vector<ArSensorReading>;

//Zmiana nazwy dla metod i parametrów, które mogą powodować problemy dla
//SWIG-a, bądź też są słowami kluczowymi w C#
%rename (setLock) lock;
%rename (equals) operator=;
%rename (compare) operator();
```

```
%rename (retVar) ret;

//Fragment odpowiadający za używanie w C# tablicy string[] jako argumentów
//przekazywanych do funkcji zamiast int *, char **

//Definicja klasy zamieniającej string na char *
#pragma(csharp) imclasscode=%{
    public class StringArrayMarshal : IDisposable {
        //przekazywana do C++ tablica char**
        public readonly IntPtr[] _ar;
        public StringArrayMarshal(string[] ar) {
            _ar = new IntPtr[ar.Length];
            for (int cx = 0; cx < _ar.Length; cx++) {
                _ar[cx] =
System.Runtime.InteropServices.Marshal.StringToHGlobalAnsi(ar[cx]);
            }
        }
        public virtual void Dispose() {
            for (int cx = 0; cx < _ar.Length; cx++) {
                System.Runtime.InteropServices.Marshal.FreeHGlobal(_ar[cx]);
            }
            GC.SuppressFinalize(this);
        }
    }
}%

//odpowiednie podstawienie typów podczas parsowania
%typemap(ctype) (int *argc, char **argv) "int size, void *"
%typemap(imtype, out="IntPtr") (int *argc, char **argv) "int size,IntPtr[]"
%typemap(cstype) (int *argc, char **argv) %{string[]%}
%typemap(in) (int *argc, char **argv) %{ $1 = (int *)&size; $2 =
($2_ltype)$input; %}
//Przekazanie do C++ tablicy _ar
%typemap(csin) (int *argc, char **argv) "$csinput.Length,new
$modulePINVOKE.StringArrayMarshal($csinput)._ar"

//Lista plików nagłówkowych, które mają być sparsowane i w których znajdują się
//deklaracje do opakowania
#include "ArBasePacket.h"
#include "ArPTZ.h"
#include "ArThread.h"
#include "ArAsyncTask.h"
#include "ArRangeDevice.h"
#include "ArRangeDeviceThreaded.h"
#include "ArResolver.h"
#include "ArAction.h"
.....
```

Następnie uruchomiony został Swig z następującymi parametrami:

```
swig.exe -c++ -o "aria_wrap.cpp" -I"include" -dllimport "Aria.dll" -DAREXPORT -outdir
"././AriaWrapper" -csharp plik_wejściowy
```

-c++ - język który parsujemy

-o "aria_wrap.cpp" - nazwa pliku wyjściowego C++ znajduje się w nim definicja klasy, której metody są bezpośredni wywoływane z zarządzanego kodu

-I"include" – folder z plikami nagłówkowymi

-dllimport "Aria.dll" nazwa biblioteki dll, która ma być użyta w klauzuli DllImport

-DAREXPORT – definiowanie jako pustą stałą AREXPORT; nie jest potrzebna podczas tworzenia opakowania, używana podczas tworzenia biblioteki dynamicznej

-outdir "././AriaWrapper" – folder wyjściowy dla opakowanych klas

-csharp – język docelowy

Po wygenerowaniu odpowiednich klas są one kopilowane i linkowane do dwóch bibliotek

- Aria.dll – biblioteka natywna, gdzie znajdują się skompilowane klasy pakietu Aria oraz wygenerowana klasa SwigValueWrapper
- AriaWrapper.dll – biblioteka zarządzana, w której znajdują się opakowane klasy C#

5. Architektura biblioteki

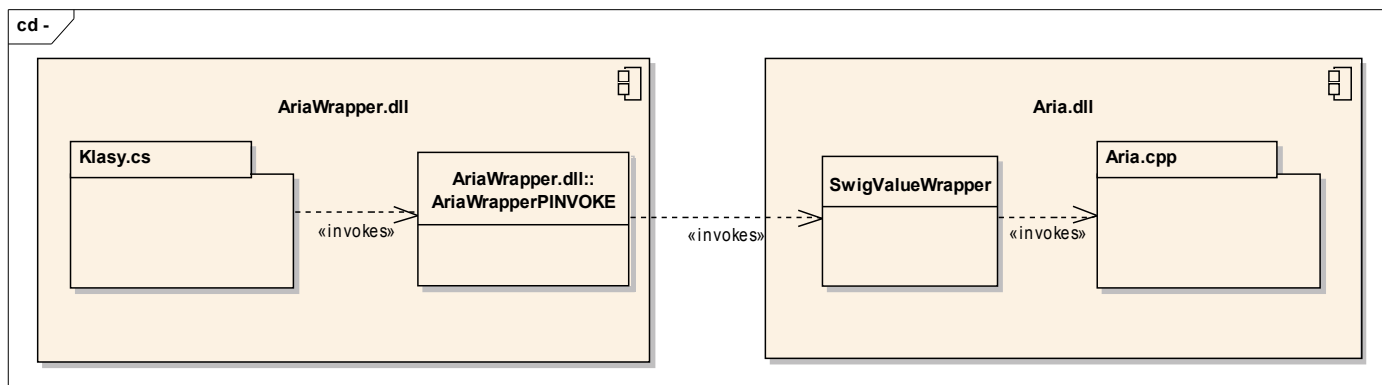
W skład bibliotek wchodzi następujące pliki:

AriaWrapper.dll

- NazwaKlasy.cs – pliki opakowujące odpowiadające klasy C++; z ich poziomu wywoływane są odpowiednie metody klasy AriaWrapperPINVOKE.
- AriaWrapperPINVOKE.cs – klasa, której metody to rozszerzenia odpowiednich metod klasy SwigValueWrapper

Aria.dll

- aria_wrapper.cpp – znajdują się tu klasa SwigValueWrapper, w której metodach dokonuje się konwersja typów oraz wywołanie funkcji odpowiednich klas



6. Użycie biblioteki

Aby móc używać powyższej biblioteki we własnych aplikacjach należy:

- dodać referencję AriaWrapper.dll do wybranego projektu
- umieścić plik Aria.dll w katalogu gdzie umieszczona jest wersja Debug/Release programu tak, aby aplikacja mogła go odnaleźć

Po wykonaniu tych czynności możemy pisać programy w pakiecie Aria tak, jak robilibyśmy to w języku C++, korzystając z podpowiadania składni w środowisku Visual Studio .NET.

7. Przykład użycia

Dwa proste przykłady użycia biblioteki zostały zamieszczone w Solution 'AriaWrapper'. Jeden z nich to aplikacja konsolowa, drugi okienkowa. Do ich uruchomienia dobrze jest mieć zainstalowany symulator robota – MobileSym. Możemy wówczas obserwować zachowanie urządzenia.

W powyższym sprawozdaniu przedstawię aplikację konsolową. Jest ona podobna do tej opisanej jako przykład dla biblioteki proxy Java.

Jest to program, który łączy się z robotem, przesuwa go na pewną odległość, wyświetla współrzędne, rozłącza się i kończy działanie.

Kod przedstawiam poniżej:

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

namespace AriaCsharpExample
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Starting Csharp test");
            //Inicjalizacja Aria
            Aria.init(Aria.SigHandleMethod.SIGHANDLE_THREAD, true);
            //Dodanie nowego robota
            ArRobot robot = new ArRobot("robot1", true, true, true);
            //Tworzenie konetkora dla robota
            ArSimpleConnector conn = new ArSimpleConnector(args);
            //Łączenie się z urządzeniem
            if (!conn.connectRobot(robot))
            {
                Console.WriteLine("Could not connect robot.");
                return;
            }
            //rozpoczyna pracę z robotem w nowm wątku
            robot.runAsync(true);
            //zakłada zamek
            robot.setLock();
            //przesuwa na pewną odległość
            robot.move(1000);
            //zdjemuje zamek
            robot.unlock();
            //wstrzymuje działanie
            ArUtil.sleep(5000);
            robot.setLock();
            //pobiera i wypisuje koordynaty
            Console.WriteLine("Robot Coords (" + robot.getX() + ", " +
robot.getY() + ", " + robot.getTh() + ")");
            robot.unlock();
            robot.setLock();
            //Zatrzymuje wszystkie akcje robota, wychodzi z wątku
            robot.stopRunning(true);
            robot.unlock();
            robot.setLock();
            //rozłącza się
            robot.disconnect();
            robot.unlock();
            //zamyka pakiet Aria
            Aria.shutdown();
        }
    }
}
```

Testując program na symulatorze otrzymaliśmy wyniki zgodne z rzeczywistością – robot zachowywał się poprawnie.

Można stwierdzić, że obecnie tworzenie aplikacji w C# wygląda bradzo podobnie do pisania ich w C++. W celu bliższego zapoznania się ze sposobem tworzenia tych programów należy zapoznać się z biblioteką Aria.

Jedynymi różnicami są zmienione nazwy metod:

- lock() -> setLock()
- operator(...) -> compare(...)
- operator==(...) -> equals(...)

8. Tworzenie własnej biblioteki

Ta sekcja opisuje co zrobić gdy ukaże się nowa wersja pakietu Aria, chcemy przerobić istniejące klasy C++ bądź też dopisać własne.

Po wykonaniu powyższych czynności należy jeszcze raz zbudować bibliotekę. Najlepiej jest użyć już gotowych projektów zawartych w Solution AriaWrapper.

Czynności do wykonania:

1. Upewniamy się, że posiadamy narzędzie Swig(<http://www.swig.org>) i że jego folder główny dodany jest do systemowego PATH
2. Umieścić folderach include/ i src/ projektu Aria pliki nagłówkowe oraz te z kodem.
3. Uruchomić budowanie projektu Aria
4. Po zakończeniu należy w projekcie AriaWrapper za pomocą funkcji Add -> Existing Item dodać nowe pliki klas, które zostały dodane się w nowej wersji bądź też, które napisaliśmy samemu.
5. Budujemy projekt AriaWrapper
6. Jeśli opcje konfiguracji ustawione były na release, w folderze AriaWrapper/lib znajdują się gotowe do użycia biblioteki .dll.

9. Problemy

Ponieważ jest to pierwsza wersja biblioteki przy jej mogą w niej istnieć pewne nieprzewidziane sytuacje.

Jedną z nich jest przedwczesne usunięcie przez Garbage Collectora z pamięci obiektów utworzonych na stacku. Będzie to prowadziło do błędnych zachowań.

```
1) ArRobot robot = new ArRobot("robot1", true, true, true);
2) robot.addAction(new ArActionGotoStraight("act"), 1);
3) System.GC.Collect();
4) ArAction action = robot.findAction("act");
```

W tym przypadku action będzie wskazywać na spodziewany obiekt – po opuszczeniu linii 2) żadna zmienna nie jest dowiązana do obiektu stworzonego przez new ArActionGotoStraight("act"). Dowiązanie przechowywane jest w kodzie niezarządzanym. Tak więc Garbage Collector zwolni zalegającą wg. niego pamięć.

Rozwiązaniem może być przechowywanie referencji do takich obiektów w dodatkowej zmiennej / kolekcji:

```
1) ArRobot robot = new ArRobot("robot1", true, true, true);
2) ArAction goStraightBind = new ArActionGotoStraight("act");
3) robot.addAction(goStraightBind, 1);
4) System.GC.Collect();
5) ArAction action = robot.findAction("act");
```

W tym przypadku zmienna action będzie wskazywała na odpowiedni obiekt.

W przyszłej wersji biblioteki C# dopisana będzie opcja samoczynnie dowiązująca obiekty, które mogą być przedwcześnie posprzątane.