AUTOMATYKA • 2005 • Tom 9 • Zeszyt 3

Paweł Skrzyński*, Michał Turek*

Generowanie kodu oraz wykonywalnych aplikacji z modelu w UML 2.0

1. Wprowadzenie

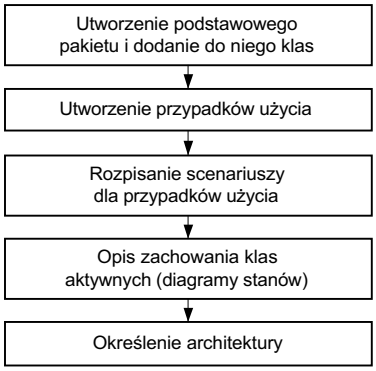
Na projekt w UML (*Unified Modelling Language*) składa się zbiór diagramów, które opisują system z różnych perspektyw. Na różnych diagramach ta sama informacja przedstawiona może być w różny sposób, uwypuklając róże aspekty działania systemu. Cechą charakterystyczną modelu zapisanego w UML jest to, iż jest on semantycznie bogatszy od jakiegokolwiek kodu w języku programowania. Pomimo to przez długi czas narzędzia CASE dla UML pozostawały daleko w tyle – nie było możliwości wygenerowania kodu, który zawierałby coś więcej niż tylko szkielet klas [1]. Mapowanie takie jest robione na podstawie diagramu klas i jest bardzo proste. Sytuacja zmieniła się z chwilą pojawienia się narzędzia Telelogic Tau G2. Jest to bardzo dojrzałe narzędzie posiadające jako jedno z pierwszych na rynku możliwość generowania wykonywalnych aplikacji na podstawie modelu zapisanego w UML. Ponadto moduł symulacyjny pozwala na sprawdzenie poprawności systemu lub jego części. W artykule krótko omówiony zostanie sposób rozwijania aplikacji w Tau G2 z naciskiem na elementy, które mają bezpośredni wpływ na „wykonywalność".

1.1. Rozwijanie modelu w Tau G2

Tworząc projekt systemu w UML z wykorzystaniem narzędzia Tau G2 (rys. 1), należy rozpocząć od utworzenia pakietu – pakiety powalają na elegancką organizację projektu i do nich mogą być następnie dodawane diagramy. Pierwszym diagramem, jaki może zostać utworzony, może być diagram przypadków użycia, na którym prezentowana jest funkcjonalność systemu z punktu widzenia użytkownika. Wiedząc już, co system ma robić, należy się zastanowić, w jaki sposób ma to realizować. Trzeba zatem zacząć rozpisywać scenariusze poszczególnych przypadków użycia, posługując się diagramami interakcji oraz czynności. Równolegle można rozpocząć pracę nad opracowaniem diagramu klas, gdyż w systemie obiektowym to właśnie współpracujące obiekty (czyli egzemplarze klas) realizują funkcjonalność systemu. Następnie modeluje się zachowanie poszczególnych klas aktywnych systemu, posługując się diagramami stanów. Z punktu widzenia generowania kodu

* Katedra Automatyki, Akademia Górniczo-Hutnicza, Kraków

oraz plików wykonywalnych aplikacji, jest to jeden z najważniejszych rodzajów diagramów. Na końcu określana jest architektura systemu, czyli sposób, w jaki poszczególne obiekty mają być instancjonalizowane – do tego celu wykorzystywany jest diagram architektury czy też struktury [2, 7].



Rys. 1. Metodyka tworzenia modelu w Tau G2

1.2. Modelowania struktury i zachowania systemu

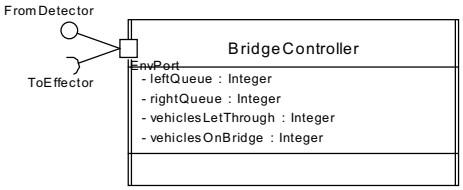
Jednym z głównych wymagań, jakie stawiano wersji 2.0 języka UML, była możliwość zarządzania złożonością w dużych i skomplikowanych projektach. Do modelu składającego się do tej pory z płaskiego zbioru, luźno ze sobą powiązanych diagramów, wprowadzono hierarchię. Odpowiednie zaprojektowanie struktury systemu oraz dekompozycja ułatwia podział zadań przy rozwijaniu systemu oraz zarządzanie jego integralnością. Wiele nowoczesnych języków programowania zaadoptowało interfejsy jako sposób opisu usług, jakie są udostępniane przez pojedynczą klasę.

W UML 2.0 wprowadza się rozróżnienie pomiędzy interfejsami [3] na:

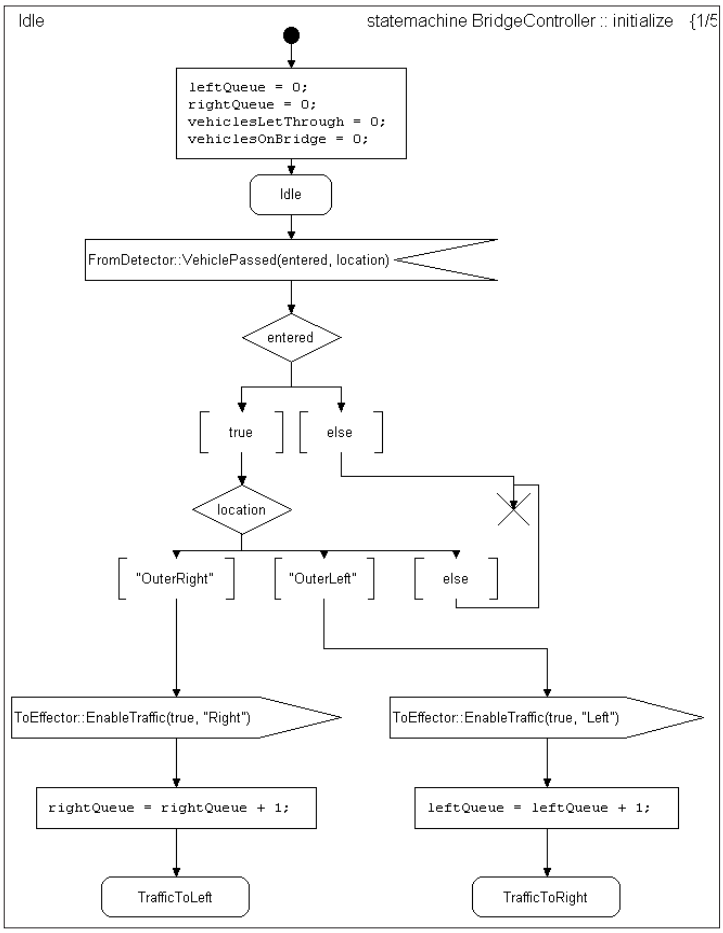
– interfejsy dostarczane (*provided interfaces*) będące usługami, które klasa implementuje;
– interfejsy wymagane (*required interfaces*) będące usługami, które inne klasy muszą dostarczyć w celu zapewnienia odpowiedniego funkcjonowania klasy w środowisku.

Takie rozróżnienie zapewnia możliwość rozwijania pojedynczej klasy jako autonomicznej jednostki nie potrzebującej nic wiedzieć o szczegółach klas, które dostarczają jej wymaganych interfejsów. Każda klasa może być wykorzystywana na różne sposoby i różne klasy mogą wymagać od niej różnych usług. Realizowane jest to oczywiście poprzez implementowanie różnych interfejsów, lecz potrzebny jest mechanizm grupowania tych interfejsów tak, by w jednej grupie znajdowały się interfejsy dostarczane określonemu odbiorcy. Każda taka grupa dostarcza innej perspektywy widoku klasy. W UML 2.0 realizowane jest to poprzez porty. Dodatkowo porty pełnią inną istotną rolę – łączą implementację klasy ze środowiskiem, w którym działa. Komunikacja z klasą możliwa jest zatem poprzez porty – wszelkie komunikaty do i z klasy są przez nie wysyłane w zależności od tego, czy z portem są skojarzone interfejsy wymagane, czy realizowane. Port, z którym są skojarzone oba rodzaje interfejsów, nosi nazwę dwukierunkowego.

Z interfejsami z kolei wiązane są sygnały, które są wymieniane pomiędzy obiektami w działającym systemie. Zatem komunikacja pomiędzy klasami, ściślej – egzemplarzami klas (rys. 2), odbywa się poprzez porty, z którymi są z kolei związane interfejsy określające, jakich zachowań klasa dostarcza oraz jakich wymaga od innych w celu poprawnego działania.



Rys. 2. Przykładowa klasa wraz z interfejsami



Rys. 3. Przykładowy diagram stanów kontrolera mostu

System nie składa się z samej struktury. Równie ważna jest jego dynamika, czyli zachowanie. W UML do modelowania zachowania pojedynczych klas wykorzystuje się diagramy stanów – zatem z każda klasą aktywną związana jest maszyna stanowa, która definiuje jej zachowanie, na które składa się sposób obsługi sygnałów odbieranych przez egzemplarz klasy od innych klas oraz sygnały, jakie ta instancja wysyła do środowiska lub innych klas.

Przykładowy diagram stanów jest przedstawiony na rysunku 3.

2. Możliwości translacji w procesie inżynierii do przodu oraz inżynierii wstecz

Najciekawsza gałąź procesów przetwarzania kodu z zastosowaniem UML sprowadza się do translacji z UML do kodu języka obiektowo zorientowanego. Proces takiej transformacji nazywany jest inżynierią do przodu. W wyniku takiej transformacji, w zależności od wyboru docelowego języka programowania, może dojść do straty pewnej ilości informacji, ponieważ modele zapisane w UML są bogatsze semantycznie od każdego współczesnego języka obiektowego. Podstawowym problemem i powodem utraty informacji przy konwersji są różnego rodzaju braki i rozbieżności w strukturze oraz składni zarówno po jednej, jak i drugiej stronie. Wiele konstrukcji języka obiektowego nie może być w sposób jednoznaczny mapowana na UML (z oczywistych względów jest to przykładowo treść operacji / metod/). Podobnie konstrukcje zapisane w UML (np. mechanizm przekazywania sygnałów) są trudne do implementacji w kodzie języka programowania. Z reguły narzędzia typu CASE kończą takie procesy na mapowaniu struktury statycznej klas, co z oczywistych względów nie jest zadowalające. W zakres takich czynności wchodzi wówczas przenoszenie informacji o klasach oraz atrybutach i operacjach składowych klas (tylko deklaracje). W ramach informacji o klasach umieszczane są czasem jeszcze relacje pomiędzy klasami. Model dynamiczny (implementacja sekwencji, aktywności, przejść stanów) rzadko jest brane pod uwagę.

Obecne narzędzia CASE, jeśli posiadają funkcjonalność w dziedzinie przetwarzania kodu aktywnego, to przeważanie zawężają ją do mapowania treści operacji do diagramów tekstowych (w modelu mamy diagram, który zawiera pobrany z pierwotnego pliku kod i jest on skojarzony np. z przypadkiem użycia czy operacją klasy) [5]. W procesie inżynierii wprzód czy wstecz treść jest wtedy jedynie kopiowana pomiędzy zawartością diagramów tekstowych a ciałami metod w plikach kodu.

Jak wspomniano, narzędzie CASE firmy Telelogic pozwala na generowanie kody również na podstawie maszyn stanowych. Ponieważ definiują one zachowanie klas, to możliwe jest wygenerowanie działającej aplikacji. Proces generowanie kodu oraz łączenia go z kodem tworzonym w środowiskach SDK takich jak MS Visual C++ zostanie przedstawiony w dalszej części artykułu.

2.1. Artefakty i mapowanie UML na kod

W procesie konwersji modelu UML do kodu wykonywalnego konieczne jest przeprowadzenie fragmentacji modelu. Fragmentacja ta powinna być przede wszystkim przej-

rzysta dla użytkownika, jasno prezentując, które fragmenty modelu są odpowiednikami jakich modułów kodu źródłowego. Dobrą techniką jest zastosowanie tzw. artefaktów. Artefakt jest z definicji elementem modelu lub wzmaganiem definiowanym wobec modelu. Definicje tę można rozszerzyć o możliwość wprowadzenia konfiguracji artefaktu, zawierającej informację o jego odpowiedniku w postaci kodu [6]. Informacja ta wyspecyfikuje pliki docelowe, pakiet lub klasę docelową, ustawienia samej konwersji (szczegóły notacji, rodzaje treści, jakie należy z danego elementu modelu wyeksportować do plików).

Artefakty w procesie generowania kodu są wykorzystywane do tworzenia tzw. „stref poszukiwań", obejmujących wszystkie wykryte w modelu zależności pomiędzy obiektami. Przykładowo, gdy w modelowanej maszynie stanów wykryto wysyłanie sygnału z obiektu typu A do obiektu B, artefakt obiektu B poszerzy strefę poszukiwań o obiekt B lub dodatkową klasę sygnału, służącą do przekazania w jej instancjach konkretnej informacji z A do B (odpowiadającej danemu sygnałowi). Konsekwencją dla wygenerowanego kodu klasy B będzie przykładowo dołożenie dyrektyw włączających nagłówki innych klas, dodanie wskaźnika na obiekt klasy A będący w interakcji, wygenerowanie metody obsługującej zdarzenie związane z odebraniem komunikatu o konkretnym sygnale (i posiadającej dostęp wyspecjalizowanego obiektu sygnału).

2.2. Dynamika tworzonego systemu – maszyny stanów i instancje klas

Znaczenie maszyn stanów prezentowanych w UML na „diagramach przejść stanów" dla procesu generowania kodu na pewno nie jest kwestionowane. To te diagramy umożliwiają jednoznaczne i precyzyjne definiowanie czynności podejmowanych w metodzie danej klasy. Z uwagi na notację proceduralną z zaznaczeniem ogniw decyzyjnych, punktów wejścia i wyjścia czy nawrotów istnieje możliwość bezpośredniego przeniesienia poszczególnych symboli diagramu na kod. Dodatkowo diagramy te umożliwiają prezentację komunikacji pomiędzy metodami (operacjami) klas, realizowaną za pośrednictwem wysyłanych i odbieranych sygnałów. Wysłany sygnał może być wymagany przez inną operację i w innej klasie. Mechanizm taki jest banalny do zakodowania, jeśli każdej modelowanej klasie odpowiada jeden fizyczny obiekt, posiadający własną aktywność i będący jedynym adresatem wysyłanych przez operacje jego klasy sygnałów. Często obiektów jest jednak więcej i tutaj powstaje problem, związany z pytaniem, jak ująć w kodzie wynikowym komunikację na poziomie obiektów, bazując jedynie na komunikacji pomiędzy klasami. Dlatego także kluczową rolę dla generowania takiego kodu obok „diagramów przejść stanów" stanowią „diagramy obiektów". Są one konstruowane w sposób zbliżony do diagramów klas, jednak przedstawiają także wzajemne relacje pomiędzy fizycznymi obiektami (instancjami klas), również tworzonymi dynamicznie. Możliwe jest więc modelowanie komunikacji pomiędzy instancjami klas z wytypowaniem konkretnych portów klas i konkretnych interfejsów, jako właściwe do pośredniczenia na drodze przesyłanego sygnału. Podobnie – możliwe jest modelowanie obiektów tablicowanych, z uwzględnieniem komunikacji pomiędzy nimi. Wszystkie te informacje są bezpośrednim źródłem danych do procesu generowania kodu aktywnego klas, umożliwiającego automatyczną komunikację między instancjami klas w wynikowym programie [6].

2.3. Komunikacja ze środowiskiem

Komunikację ze środowiskiem zewnętrznym prezentowanego systemu model UML pozostawia z reguły interfejsom środowiskowym, koncentrującym zdefiniowane wcześniej zbiory sygnałów. W niniejszym podrozdziale przedstawiono propozycje praktycznego wykorzystania interfejsów środowiskowych UML w realizacji komunikacji pomiędzy modułem wygenerowanym z diagramów a modułem zewnętrznym, mogącym być przykładowo graficznym interfejsem użytkownika, choćby w aplikacji okienkowej.

Nie ulega wątpliwości, że fakt wysłania przez instancję klasy określonego sygnału i skierowania go do również określonego adresata, będzie powiązany w wywołaniem metody obiektu adresata. Metoda ta przejmie ewentualne parametry, będące parametrami wysyłanego sygnału. W przypadku obiektów-adresatów, stanowiących integralną część modelu, problem nie przysparza żadnych trudności. Zbiór metod obiektu docelowego oraz ich parametrów jest jawnie zdefiniowany. Jednak przy rozważaniu sygnałów środowiskowych takiego zbioru nie ma. Ciężko też odgórnie narzucić w modelu strukturę obiektów otaczających projektowany system.

Struktura ta jest ściśle zależna między innymi od:

- rodzaju docelowego środowiska programistycznego, dla którego będzie produkowany kod;
- rodzaju interfejsu zewnętrznego (interfejs zdarzeniowy użytkownika, kolejki komunikatów przesyłane pomiędzy bieżącym modułem i modułem zdalnym itp.).

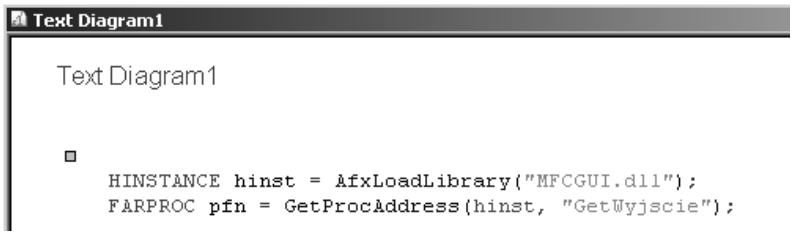
Wydaje się, że problem komunikacji rozwiązać można całościowo. Prostym rozwiązaniem jest uzależnienie modelu od struktury środowiska poprzez import bibliotek z metodami wywoływanymi zewnętrznie, modelowanie obiektów reprezentujących elementy środowiska i posiadających importowane metody. Obiekty takie nie będą posiadały wygenerowanych odpowiedników w kodzie, gdyż zakłada się, że te definicje zostaną dostarczone z zewnątrz. Z uwagi na powyższe nie będzie możliwości przeprowadzenia natychmiastowej kompilacji w taki sposób wygenerowanego kodu (braki w definicjach).

Inną metodą jest wprowadzenie biblioteki klas pośredniczących, stanowiącej model komunikacji i oparcie rozwiązania na sygnałach modelowanych w UML [4]. Rozwiązanie to jest bardziej skomplikowane, ale posiada dużą zaletę – istnieje możliwość odseparowania projektu związanego ze środowiskiem (np. graficzny interfejs użytkownika) i projektu modelowanego w UML (np. struktura aplikacji). Obydwa projekty można (już na etapie wygenerowanego kodu) prowadzić niezależnie – do fazy kompilacji kodu włącznie. Do kompilacji wymagana jest jedynie znajomość klas pośredniczących, identycznych dla obydwu projektów. Sama komunikacja następuje w wyniku rejestrowania konkretnych instancji w odpowiednich obiektach klas pośredniczących.

Dalszym krokiem może być rozwinięcie poprzedniego rozwiązania o zastosowanie eksportów bibliotek łączonych dynamicznie. Klasy pośredniczące nadal realizują komunikację, jednak ich funkcje są redukowane wyłącznie do kolejkowania wiadomości i rozsyłania ich do właściwych adresatów (obiektów) modelowych w UML.

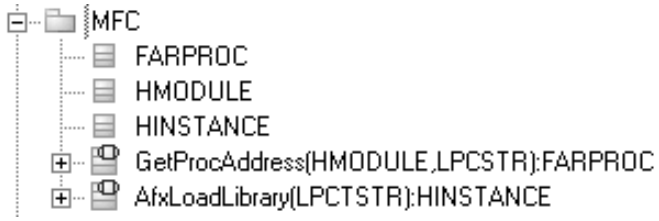
Od strony UML model pobiera bezpośrednio eksporty biblioteczne, nawiązuje komunikację poprzez wywoływanie metod eksportowanych lub metod obiektów przez nie wskazywanych.

Pobieranie eksportów bibliotecznych będzie wymagało użycia konkretnych funkcji bibliotecznych, dostarczonych z zewnątrz przez zasoby środowiska, na które tworzony jest kod. W tej technice mamy więc ponowne uzależnienie kodu od bibliotek zewnętrznych. Dzięki „diagramom tekstowym" UML (również mogącymi stanowić implementację maszyny stanowej) jesteśmy w stanie zapisać w modelu kod dedykowany dla konkretnego już środowiska programistycznego i korzystający z jego funkcji bibliotecznych.



Rys. 4. Użycie w modelu UML zasobów bibliotecznych z zewnątrz

W przypadku MS Visual C++ i zasobów MFC eksport biblioteki można pobrać (rys. 4), korzystając z uprzednio dodanego do modelu pakietu, zawierającego metody i atrybuty specyficzne dla MFC (rys. 5) i użyte, jak pokazano na rysunku 4.



Rys. 5. Zewnętrzne zasoby w drzewie modelu

Eksport zdefiniowany w projekcie biblioteki (w tym przykładzie – biblioteki DLL) musi jedynie zwracać wskaźnik na obiekt, posiadający metody odbierające wysyłane sygnały, jak pokazano na rysunku 6.

```
CMFCGUIDlg theDialog;
// Eksport DLL, klasa CMFCGUIDlg posiada metody interfejsu
extern "C" class Wyjscie* PASCAL EXPORT GetWyjscie()
{
    AFX_MANAGE_STATE(AfxGetStaticModuleState());
    return &theDialog;
}
```

Rys. 6. Przykład definicji eksportu bibliotecznego, umożliwiającego nawiązanie komunikacji

Naturalnie nadal do rozwiązania pozostaje problem kompatybilności interfejsu wysyłającego sygnały „z modelu" a interfejsu odbierającego je po stronie środowiska (np. biblioteka z interfejsem okienkowym użytkownika).

Ten problem najprościej rozwiązać poprzez dołączenie do generatora funkcjonalności umożliwiającej wytworzenie kodu interfejsu i wstawienie kodu do biblioteki. Kod taki będzie oczywiście automatycznie tworzony na podstawie sygnałów UML zdefiniowanych w interfejsie wychodzącym do środowiska.

Przykładowo:

Rys. 7. Przykład interfejsów środowiskowych w UML

W modelu na rysunku 7 mamy interfejs „Wyjscie", skupiający sygnały Wy1, Wy2 i Wy3. Każdy sygnał może być reprezentowany jako oddzielna klasa w wygenerowanym kodzie (rys. 8). Instancja takiej klasy będzie przekazywana w momencie wysyłania czy odbierania sygnału.

```
UML_INTERFACE Wyjscie : virtual
public SignalCommunication::EventReceiver { //klasa pośrednicząca
    public: //<WYGNEROWANE>
    class Wy1_Natural : virtual public SignalCommunication::Event {
        Wy1_Natural(SignalCommunication::Natural d);
        SignalCommunication::Natural d; //parametr przekazany
    };
    class Wy2 : virtual public SignalCommunication::Event {
        Wy2();
    };
    class Wy3_Charstring : virtual public SignalCommunication::Event {
        Wy3_Charstring(SignalCommunication::Charstring s);
        SignalCommunication::Charstring s; //parametr przekazany
    };
    //</WYGNEROWANE>
};
```

Rys. 8. Wygenerowany kod klas, odpowiadających sygnałom interfejsów środowiskowych

Interfejs dziedziczy z klas pośredniczących, zarządzających kolejkami komunikatów i wywołującymi metody odbierające komunikaty przez klienta, które można przeciążać. Przykładowym odpowiednikiem przedstawionego wyżej interfejsu „Wyjscie" może być kod przedstawiony na rysunku 9.

```
bool CMFCGUIDlg::receive(SignalCommunication::Event* e) {

    if (Wyjscie::Wy2* pSignal
        = SignalCommunication::cast<Wyjscie::Wy2*>(e)) {
        // AKCJA
    }
}
```

Rys. 9. Kod pisany przez programistę w celu przechwycenia sygnału wysyłanego do środowiska z modelu

Z wygenerowanego kodu, zawierającego metody i klasy może dziedziczyć dowolna klasa „implementująca" interfejs (rys. 10). Wskaźnik na tę klasę jest eksportowany przez funkcję biblioteki, co czyni metody interfejsu dostępnymi na zewnątrz.

```
class CMFCGUIDlg : public CDialog, public Wyjscie
{
```

Rys. 10. Deklaracja klasy – odbiorcy sygnałów w środowisku. Przez dziedziczenie z klasy „Wyjscie" uzyskujemy alternatywę dla „implementacji" interfejsu „Wyjscie"

Komunikacja w przeciwnym kierunku będzie łatwiejsza do zrealizowania – wystarczy wywołać metodę klasy pośredniczącej i podać jej instancję klasy z interfejsu (tu będzie interfejs skierowany w stronę przeciwną, w przykładzie nazwany „Wejscie", którego kod również może zostać wygenerowany na podstawie modelu, rys. 11)

```
void CMFCGUIDlg::OnSignalNatural()
{
    SignalCommunication
    ::sendTo(new Wejscie::We1_Natural(m_Edit)
        , m_pWejscie);
}
```

Rys. 11. Przykład kodu wysyłającego sygnały ze środowiska do klas modelu

Efekt końcowy – to biblioteka i zależny od niej wygenerowany kod, który posiadając własny *entry point*, realizuje następujące czynności:

- importuje funkcję biblioteczną;
- pobiera za pomocą zaimportowanej funkcji wskaźnik na eksportowany interfejs biblioteki;
- tworzy za pomocą interfejsu biblioteki instancję klasy zdefiniowanej w bibliotece; utworzona instancja buduje (przykładowo) graficzny interfejs użytkownika;
- utrzymuje dwustronną komunikację poprzez wymianę obiektów, będących odpowiednikami sygnałów w modelu UML.

Zaprezentowane rozwiązanie umożliwia automatyczne wygenerowanie modułu, gwarantującego wierne odzwierciedlenie sygnałów środowiskowych modelu swoim w protokole komunikacyjnym. Technikę taką można zastosować w przypadku każdego generatora kodu, który wyprowadza sygnały środowiskowe modelu jako klasy. Zaletą jest tutaj brak konieczności ingerencji programisty w wygenerowany kod, gdyż granica pomiędzy kodem generowanym a kodem tworzonym ręcznie przez programistę w ramach środowiska dla modelu jest jasno określona.

3. Podsumowanie

Proces generowania kodu z modelu UML jest zdaniem autorów bardzo ważnym procesem mającym istotny wpływ na jakość oprogramowania. Dodatkowo integracja modelu w UML z kodem tworzonym w zewnętrznych środowiskach programistycznych pozwala na łatwiejsze zarządzanie projektem. W artykule został przedstawiony sposób integracji modelu zapisanego w UML 2.0 z kodem interfejsu użytkownika, który jest tworzony w MS Visual C++. Podejście takie pozwala na wytworzenie w pełni funkcjonalnej i działającej aplikacji. Dzięki takim mechanizmom dostarczanym przez narzędzie Tau G2 możliwe jest utrzymanie pełnej zgodności pomiędzy projektem a implementacją. Dodatkowo przejście od fazy projektowania (tworzenia modelu UML) do implementacji (tworzenia kodu np. w C++) jest realizowane w sposób płynny. Płynność ta ma dodatkowa zaletę – kod jest elegancko udokumentowany przez model UML, na podstawie którego został wytworzony. Jest to istotne zwłaszcza w pracy nad dużymi projektami informatycznymi, w którym bierze udział duża liczba programistów i projektantów.

Literatura

[1] Bjorkander M., Kobryn C.: *Architecting Systems with UML 2.0.* IEEE Software, lipiec 2003, 57–61

[2] Bjorkander M.: *Real-Time Systems in UML (and SDL).* Embedded System Engineering, październik/listopad 2000

[3] Booch G., Jacobson I., Rumbaugh J.: *The Unified Modeling Language User Guide (The Addison-Wesley Object Technology Series).* Addison-Wesley, 1998

[4] Booch G., Jacobson I., Rumbaugh J.: *The Unified Modeling Language Reference Manual*. Addison Wesley Longman Inc., 1999

[5] Douglass B.P.: *Real-Time UML. Developing Efficient Objects for Embedded Systems*. Addison-Wesley, 1998

[6] Klimek R., Skrzyński P., Turek M.: *UML i Telelogic Tau Generation2.* Materiały dydaktyczne, 2005

[7] OMG RFP: *Unified Modeling Language: Superstructure version 2.0. 3rd revised submission.* 2003