STUDIA INFORMATICA 2011
Volume 32 Number 2B (97)

Łukasz WYCIŚLIK
Politechnika Śląska, Instytut Informatyki

ZARZĄDZANIE INSTALACJĄ I AKTUALIZACJĄ ZALEŻNYCH SYSTEMÓW APLIKACYJNYCH W ŚRODOWISKACH KORPORACYJNYCH

Streszczenie. Artykuł przedstawia problemy związane z funkcjonowaniem systemów aplikacyjnych w środowiskach korporacyjnych. Czynienie zadość współczesnym trendom architektury wytwarzania systemów informatycznych doprowadza do sytuacji, w której w środowiskach wykonawczych istnieje potrzeba rozmieszczania i utrzymywania wielu modułów wykonawczych, które do realizacji stawianych wymogów funkcjonalnych muszą ze sobą kooperować. Ponieważ każdy z tych modułów może być objęty innym cyklem wytwórczym, zaś całość systemu podlega ciągłemu rozwojowi funkcjonalnemu, pojawia się niebezpieczeństwo awarii systemu, wynikającej z błędnego rozmieszczania, konfigurowania i aktualizowania poszczególnych modułów. Autor w artykule wyjaśnia, dlaczego na wyżej wspomniane problemy narażone są zwłaszcza współcześnie tworzone systemy informatyczne. W dalszej części artykułu przedstawiony został ogólny model zależności między modułami wchodzącymi w skład złożonych systemów korporacyjnych. Przykładowa implementacja tego modelu, pozwalająca automatyzować proces rozmieszczania i aktualizowania modułów wykonawczych, z zachowaniem zdefiniowanych wcześniej zależności międzymodułowych, została zrealizowana za pomocą systemu NSIS (ang. *Nullsoft Scriptable Install System*).

Słowa kluczowe: systemy korporacyjne, wersjonowanie systemów, instalowanie aplikacji, uaktualnianie aplikacji, NSIS, Oracle

DEPLOYMENT AND UPGRADE MANAGEMENT OF DEPENDENT SYSTEMS IN CORPORATE ENVIRONMENTS

Summary. The article presents problems of deployment and upgrade processes in complex, corporate computing systems. Building computer systems according to contemporary trends in a software architecture leads to a situation where there is a need to deploy and upgrade multiple modules in an execution environment that to deliver their functional requirements must cooperate with each other. Since each of these modules

can be covered by different lifecycle and the whole system is a subject of continuous functional development, there is the risk of system failure resulting from the incorrect deployment, configuration and upgrading individual modules. The author of the article explains why the above-mentioned problems are especially vulnerable to systems created today. Later in the paper a general model of the relationship between the modules which form part of complex enterprise systems is presented. Exemplary implementation of this model, allowing to automate the process of deploying and upgrading modules with preservation of intermodular dependencies has been proposed with the help of NSIS system.

Keywords: corporate systems, system versioning, application deployment, application upgrade, NSIS, Oracle

1. Wstęp

Ciągły postęp technologiczny, skutkujący coraz większą miniaturyzacją sprzętu komputerowego oraz szybkością jego działania, powoduje gwałtowne spadki jednostkowych cen mocy procesorów oraz pojemności pamięci. Wszystko to sprzyja sytuacji, w której technologia informatyczna jest zaprzęgana do wyręczania człowieka w coraz to nowych obszarach. Pierwotnie komputery wykorzystywane były jako niezależne jednostki, które służyły do rozwiązywania problemów nielicznej grupy osób, do której dany komputer przynależał. W sytuacji gdy pojawiała się potrzeba objęcia zasięgiem działania większej populacji, budowano szybszy komputer (ang. *mainframe*) i udostępniano go za pomocą technologii terminalowych. Sytuacja zmieniła się wraz z rozwojem technologii sieciowych oraz rozwojem obiektowego podejścia do projektowania oprogramowania. Pojawiły się wtedy pomysły na dekompozycję złożonych problemów na mniejsze, przy zastosowaniu koncepcji hermetyzacji, luźnego wiązania czy wstrzykiwania zależności. Podejście takie skutkuje podziałem złożonych systemów na specjalizowane, autonomiczne i „reużywalne" jednostki wykonawcze (ang. *executables*) [2], które dla realizacji złożonych zadań współpracują, komunikując się ze sobą w ściśle zdefiniowany sposób. Jednak podobne pomysły na dekompozycję złożonych problemów spowodowały dalszy podział poszczególnych jednostek wykonawczych na warstwy specjalizujące się w realizacji ogólnie powtarzających się zagadnień, takich jak: interakcja z użytkownikiem, realizacja logiki biznesowej i trwałego przechowywania danych.

Obydwa wspomniane podejścia w oczywisty sposób ułatwiają analizowanie, projektowanie, wytwarzanie i testowanie systemów informatycznych, a przez możliwość umieszczania poszczególnych jednostek wykonawczych na różnych węzłach infrastruktury, pozwalają również na zwiększanie skalowalności. Powoduje to, że tańsze staje się zarówno wytwarzanie oprogramowania, jak i niezbędna infrastruktura, na której zostanie ono posadowione. Podejście to skutkuje jednak komplikacją w rozmieszczaniu (instalowaniu) oraz uaktualnianiu po-

szczególnych modułów wykonawczych oraz struktur baz danych i ich zawartości, co często może prowadzić do awarii systemu.

2. Przedstawienie problemu i koncepcja jego rozwiązania

2.1. Geneza problemu

Problem zgodności (inaczej kompatybilności) interfejsów modułów programowych pojawił się wraz z koncepcją wielokrotnego użycia, a dokładniej użycia w wielu zastosowaniach tychże modułów. Moduły takie, zwane potocznie bibliotekami, najczęściej implementują rozwiązania pewnych, często powtarzających się problemów z danej dziedziny, np. obliczenie wartości funkcji skrótu, konwersja ciągu znaków między różnymi systemami kodowania itp. Dopóki podczas procesu wytwarzania oprogramowania wykorzystywano techniki statycznego łączenia kodu (biblioteka kompilowana była wraz z całością oprogramowania), odpowiedzialność za dobór odpowiednich wersji bibliotek leżała po stronie dostawcy oprogramowania, co pozwalało na unikanie awarii wynikających z niezgodności wersji. Sytuacja zaczęła się radykalnie pogarszać z chwilą wprowadzenia w życie techniki dynamicznego łączenia bibliotek (np. dll – ang. dynamic-link library). Przykładowo, w środowisku Windows istniał dedykowany folder na współdzielone biblioteki. Aplikacja korzystająca z danej biblioteki nie miała gwarancji co do jej poprawnego funkcjonowania, gdyż bibliotekę tę mogła podmienić instalowana później inna aplikacja. Podobny problem dotykał również inne systemy operacyjne. Producenci środowisk wspomagających wytwarzanie oprogramowania zaczęli sobie z nim radzić przez dołączanie niezależnych kopii bibliotek do każdej aplikacji, która z nich korzystała. Dzięki takiemu zabiegowi każda aplikacja może korzystać dokładnie z tej wersji biblioteki, której oczekuje.

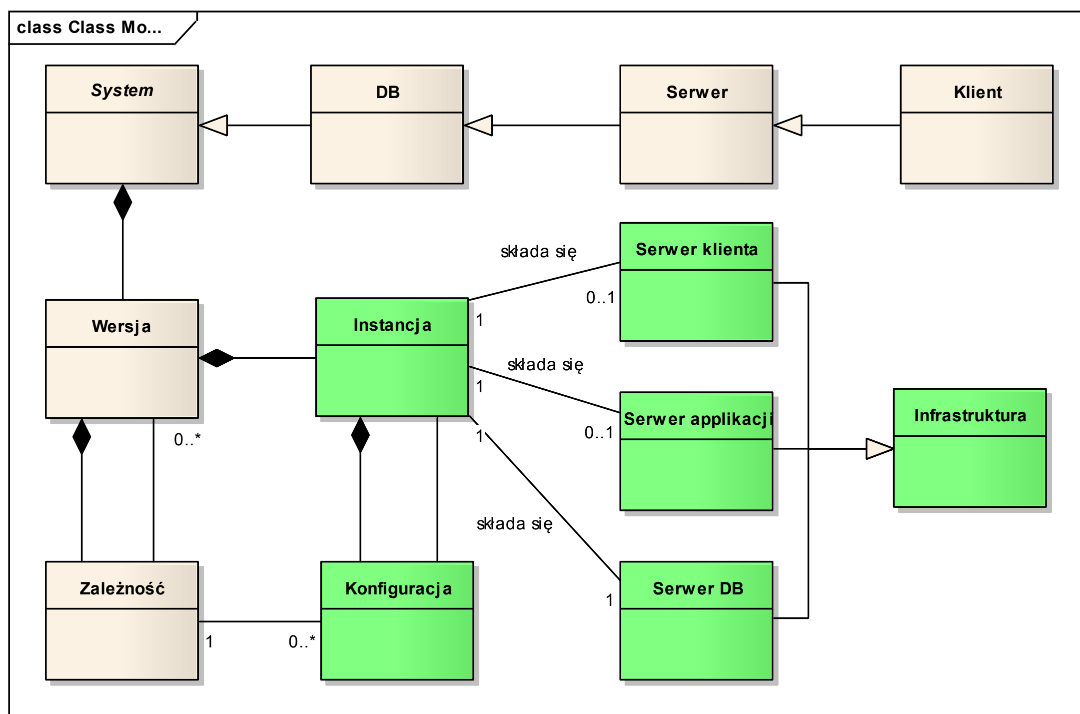
Niestety, w sytuacji gdy funkcjonalność udostępniana przez daną bibliotekę nie jest bezstanowa, czyli zależy ona od historii działania samej biblioteki, możliwość wykorzystania niezależnych jej kopii jest ograniczona. Przykładem może być funkcjonalność autoryzowania użytkowników, kiedy pożądana jest sytuacja, w której dwie aplikacje współdzielą ich listę.

Sytuacja komplikuje się jeszcze bardziej w chwili, gdy rozważania zaczynają dotyczyć nie systemu operacyjnego funkcjonującego na pojedynczym komputerze, a usług rozmieszonych w rozproszonym środowisku heterogenicznych węzłów.

2.2. Model pojęciowy i koncepcja rozwiązania

Skuteczne przeciwdziałanie zagrożeniom wyszczególnionym w poprzednim podrozdziale może być realizowane tylko przy zachowaniu odpowiednich rygorów w procesach wytwórczym i wdrożeniowym, których realizacja powinna uwzględniać zależności istniejące między systemami w poszczególnych wersjach. Spełnienie zależności zdefiniowanych na etapie wytwarzania oprogramowania przez system zainstalowany w danym środowisku produkcyjnym zapewni, że wersje współpracujących ze sobą systemów będą ze sobą kompatybilne (system podrzędny będzie realizował dokładnie taką funkcjonalność, jakiej oczekuje od niego system nadrzędny).

Propozycja opisu tych zależności przedstawiona została za pomocą poniższego diagramu klas.



Rys. 1. Diagram klas
Fig. 1. Class diagram

W literaturze spotkać można dość swobodne i wieloznaczne użycie terminów system, aplikacja, moduł, biblioteka itp., dlatego na potrzeby niniejszych rozważań należy uściślić niektóre z nich. System to zbiór arbitralnie wybranych funkcjonalności, które w danej wersji udostępniają użytkownikowi lub innym systemom swoje usługi. Jest on najmniejszą jednostką podlegającą samodzielnej instalacji i uaktualnianiu. Przesłanką do wyodrębnienia zbioru funkcjonalności i nazwania ich systemem może być fakt bycia stanowym (np. posiadania warstwy bazodanowej) oraz potrzeba jego „reużywalności”. System jest klasą abstrakcyjną, gdyż jako taki może istnieć w trzech specjalizacjach – usług implementowanych tylko w war-

stwie serwera bazy danych (np. udostępnianych jako implementacje procedur składowanych), usług implementowanych dodatkowo po stronie serwera aplikacji (np. usług udostępnianych za pomocą mechanizmów usług sieciowych (ang. *web services*)) oraz usług implementowanych dodatkowo po stronie klienta (np.aplikacji RIA (ang. *rich internet application*), która pobierana jest i wykonywana na komputerze klienta i realizuje swoją funkcjonalność przez wysyłanie żądań do warstwy serwera aplikacji).

System może udostępniać wiele usług, których zarówno interfejsy, jak i implementacje mogą podlegać niezależnemu wersjonowaniu, jednak z punktu widzenia niniejszych rozważań znaczenie ma tylko fakt, że danej wersji systemu przypisane są konkretne wersje usług, zaś system zależny, korzystający z danej wersji systemu, oczekuje implementacji usług w tych właśnie wersjach, co zagwarantuje ich poprawną współpracę (kompatybilność). Zależność ową wyraża klasa o takiej właśnie nazwie. Dana wersja systemu wymagać może innych systemów w którejś z akceptowanych wersji.

Instancją wersji systemu jest jego wystąpienie (zainstalowanie na którymś z serwerów bazodanowych, aplikacyjnych czy webowych) w infrastrukturze korporacyjnej. Nic nie stoi na przeszkodzie, aby w jednej infrastrukturze istniało wiele wystąpień tego samego systemu w jednej lub wielu wersjach.

Klasa konfiguracja reprezentuje powiązanie instancji danej wersji systemu z instancją innych wersji systemów – jest ona niejako opisem sposobu zapewnienia wymogów zdefiniowanych przez klasę zależność, a dodatkowo wskazuje na instancje wersji, od których dana instalacja systemu zależy.

Implementacja umożliwiająca zdefiniowanie wyżej omówionego modelu oraz jego systematyczne uaktualnianie w trakcie całego cyklu wytwórczego i wdrożeniowego poszczególnych systemów, pozwoli na kontrolę zgodności wersji poszczególnych systemów. Należy zwrócić uwagę, że klasy w kolorze białym[1] stanowią definicję statycznego obrazu zależności pomiędzy wersjami systemów i powinny być definiowane w trakcie cyklu wytwórczego, zaś klasy w kolorze zielonym oznaczają wystąpienie w konkretnej infrastrukturze i powinny być definiowane w procesie wdrożeniowym.

3. Implementacja

Celem implementacji mechanizmów zapobiegających opisanym wyżej problemom jest przede wszystkim automatyzacja procesu kontroli zależności wersji, ale również ułatwienie procesu skomplikowanej instalacji (ang. deployment) oprogramowania oraz późniejszej kon-

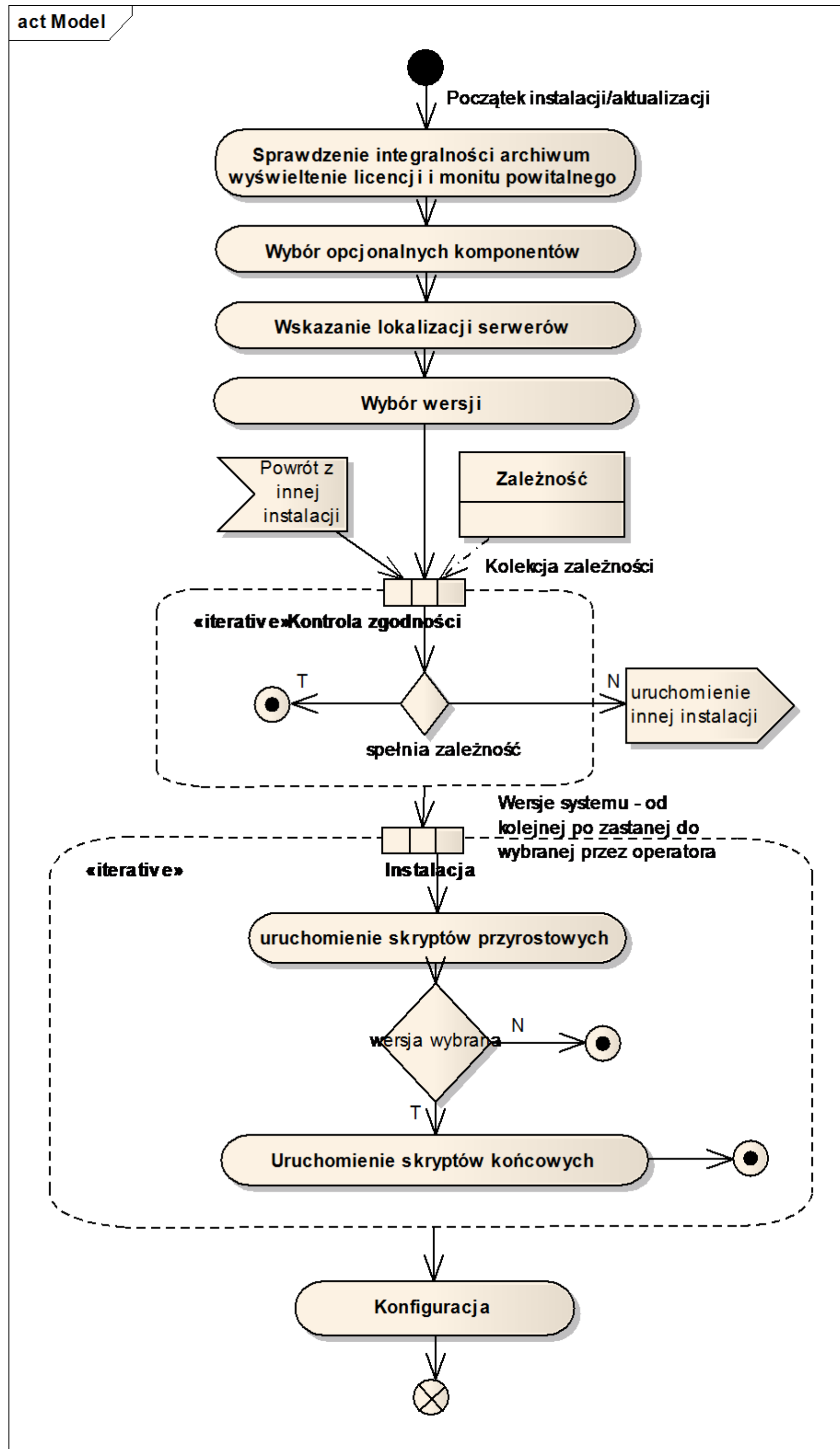
[1] Na portalu zeszytów Studia Informatica: http://znsi.aei.polsl.pl/, artykuł dostępny jest z rysunkami w kolorze.

figuracji, w zakresie niezbędnym do jego poprawnego uruchomienia. Przyjęto dodatkowe założenie, że cały proces instalacji/aktualizacji powinien być realizowany ze stacji klienckiej z systemem Windows, mającej jedynie niezbędny dostęp do serwerów przez sieciowe protokoły SMB oraz TCP/IP. Komunikacja z serwerem bazy danych (dla niniejszej implementacji wybrano serwer Oracle) realizowana jest za pomocą mechanizmów dostarczanych przez oprogramowanie „instant client" [1].

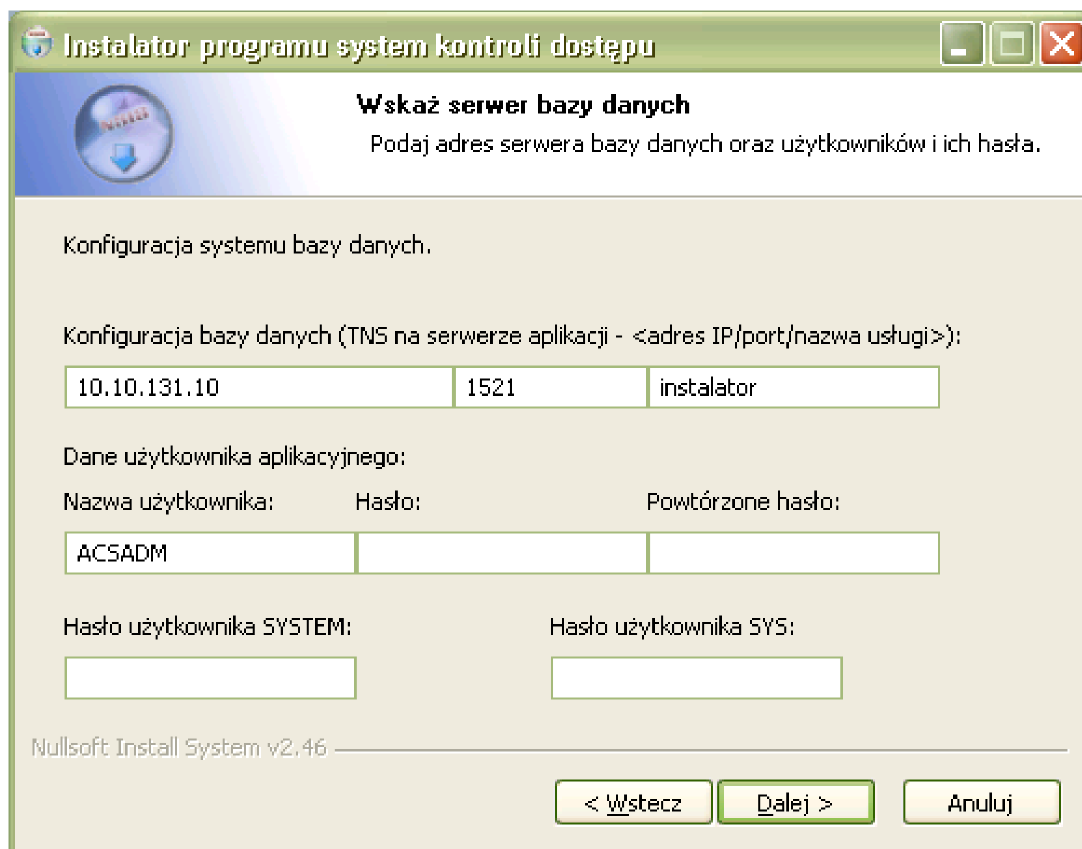
Jako silnik modułu instalacyjnego oraz graficzne środowisko użytkownika zastosowano narzędzie NSIS [3], które ma tę zaletę, że do swojego funkcjonowania nie wymaga żadnych dodatkowych bibliotek po stronie systemu operacyjnego. Dodatkowo narzędzie to posiada wbudowane mechanizmy kompresji i kontroli integralności plików instalacyjnych oraz dość rozbudowane możliwości, jeśli chodzi o tworzenie interfejsu graficznego. Zgodnie z ogólną koncepcją, każdy system posiada oddzielny, wykonywalny plik instalacyjny, po uruchomieniu którego można go zainstalować lub zaktualizować z danej wersji do wersji wybranej przez operatora. W razie zagrożenia niespełnienia zależności między wersją systemu wybraną przez operatora a wersją systemu zależnego, zostanie automatycznie uruchomiony moduł instalacyjny systemu zależnego celem jego aktualizacji do wersji, która pozwoli spełnić zdefiniowaną zależność. Jeśli nie będzie takiej możliwości, to proces instalacji systemu zostanie przerwany z odpowiednim komunikatem. Proces instalacji danej wersji systemu odzwierciedla diagram czynności (rys. 2).

Każda wersja systemu składa się z plików będących skryptami wykonywalnymi, „binariami" dla poszczególnych serwerów, skryptami baz danych itp. Należy zwrócić uwagę, że pewne czynności podczas aktualizacji systemu wymagają zachowania ciągłości operacji – np. skrypty modyfikujące dane lub strukturę bazy danych w odróżnieniu np. od kompilacji procedur składowanych, która może być zrealizowana tylko w najnowszej wersji. Dlatego podczas przygotowywania wersji systemu należy pogrupować pliki według miejsca ich przyszłego rozmieszczenia oraz podzielić na takie, które realizują operacje całościowo oraz przyrostowo. Dzięki temu, w przypadku aktualizowania wersji danego systemu moduł instalacyjny będzie w stanie rozróżnić pliki i skrypty, które ma wykonać tylko dla wybranej przez operatora wersji od tych, które ma wykonać kolejno dla wszystkich wersji, począwszy od wersji kolejnej od zastanej. Definicja struktury możliwych powiązań pomiędzy poszczególnymi wersjami systemów (białe klasy z diagramu klas) zawarta jest w plikach konfiguracyjnych samego modułu instalacyjnego, natomiast stan faktyczny (zielone klasy z diagramu klas) odczytywany jest na bieżąco z infrastruktury systemowej. Oczywiście w przypadkach bardzo restrykcyjnych definicji, np. gdy dany system dopuszcza współdziałanie z drugim systemem tylko w jednej wersji oraz drugi system z pierwszym systemem również w jednej wersji, wówczas przeprowadzenie pojedynczej instalacji pozostawi całą infrastrukturę w stanie niespójnym – jednak są to sytuacje wyjątkowe i operator będzie o nich poinformowany.

Całość interfejsu graficznego zrealizowana została zgodnie z konwencją proponowaną przez twórców narzędzia, czyli w postaci podobnych do siebie stron kreatora (ang. wizard), po których nawiguje się przyciskami *Wstecz* oraz *Dalej*. Przykładową stronę umieszczono na rys. 3.



Rys. 2. Diagram czynności procesu instalacji
Fig. 2. Activity diagram of deployment



Rys. 3. Przykładowe okno dialogowe
Fig. 3. An exemplary dialog box

Narzędzie NSIS zawiera „kompilator", który poza kompilacją programów tworzonych w prostym języku stworzonym specjalnie na jego potrzeby (w celach poglądowych fragment kodu przedstawiono poniżej), pozwala wszystkie pliki instalacyjne skompresować do jednego, wykonywalnego archiwum.

```
${NSD_GetText} $DialogDBServersTNSIP $R0
WriteRegStr HKLM "Software\general" "TNSIP" $R0
${NSD_GetText} $DialogDBServersTNSPORT $R1
WriteRegStr HKLM "Software\general" "TNSPORT" $R1
${NSD_GetText} $DialogDBServersTNSNAME $R2
WriteRegStr HKLM "Software\general" "TNSNAME" $R2
StrCpy ${TNS} "//$R0:$R1/$R2"

${NSD_GetText} $DialogDBServersAPPPASS $APPPASS
${NSD_GetText} $DialogDBServersAPPPASS2 $TMPAPPPASS

 ${If} $APPPASS != $TMPAPPPASS
   MessageBox MB_OK|MB_ICONSTOP "Podano niezgodne hasła dla użytkownika aplika-
cyjnego!"
   Abort
 ${EndIf}
```

4. Podsumowanie

W artykule przedstawiono zagrożenia, na które narażone są aplikacje funkcjonujące w środowiskach korporacyjnych, których różne obszary podlegają niezależnym cyklom wytwarzania oprogramowania. Opracowano model zależności, którego zdefiniowanie i utrzymywanie dla każdego z wytwarzanych systemów pozwoli na kontrolę możliwości ich współpracy z pozostałymi częściami całości oprogramowania. Przykładowa implementacja zaproponowanego modelu zawiera kompletne rozwiązanie dla współcześnie tworzonych systemów, w tym w architekturze trójwarstwowej. Zastosowane narzędzie NSIS umożliwiło w łatwy sposób zapewnienie kontroli spójności danych zawartych w plikach instalacyjnych oraz stworzenie przyjaznego i intuicyjnego interfejsu graficznego.

BIBLIOGRAFIA

1. Oracle Database Client Installation Guide, November 2005.
2. Fowler M.: Inversion of Control Containers and the Dependency Injection pattern. 23.01.2004.
3. NSIS, nsis.sourceforge.net.

Recenzent: Dr hab. inż. Mirosław Zaborowski

Wpłynęło do Redakcji 31 stycznia 2011 r.

Abstract

The article presents the risks faced by applications running in enterprise environments, where different parts of software are covered by independent software lifecycles. Definition of the presented model for each of the systems will allow the possibility of their cooperation with other parts of software. Exemplary implementation of the proposed model provides a complete solution for modern systems also these developed in the three-tier architecture. Thanks to NSIS tool it was possible to ensure consistency checks of the data contained in the installation files and to provide user friendly and intuitive graphical interface.

Adres

Łukasz Wyciślik: Politechnika Śląska, Instytut Informatyki, ul. Akademicka 16,
44-100 Gliwice, Polska, lwycislik@polsl.pl.