STUDIA INFORMATICA 2012
Volume 33 Number 2B (106)

Łukasz WYCIŚLIK
Politechnika Śląska, Instytut Informatyki

KONTROLA SPÓJNOŚCI WDROŻENIOWEJ ZALEŻNYCH SYSTEMÓW APLIKACYJNYCH W ŚRODOWISKACH KORPORACYJNYCH

Streszczenie. Artykuł przedstawia problemy związane z funkcjonowaniem systemów aplikacyjnych w środowiskach korporacyjnych. Najczęstszym modelem wdrożenia i eksploatacji systemów korporacyjnych jest ich wykonanie i dostarczenie przez wyspecjalizowane podmioty zajmujące się produkcją, integracją i wdrażaniem oprogramowania. Systemy te są następnie wdrażane w infrastrukturze zamawiającego i eksploatowane pod jego nadzorem. Wobec wymogów ciągłej aktualizacji oprogramowania i strojenia całego systemu, istnieje poważne niebezpieczeństwo utraty integralności poszczególnych składowych wdrożenia. Artykuł przedstawia potencjalne źródła problemów, koncepcję przeciwdziałania im oraz przykładową implementację dla potencjalnej składowej systemu, jaką może być system zarządzania bazą danych Oracle.

Słowa kluczowe: systemy korporacyjne, utrzymanie systemu, eksploatacja systemu, kontrola spójności systemu, Oracle

DEPLOYMENT INTEGRITY CONTROL OF DEPENDENT SYSTEMS IN ENTEPRISE ENVIRONMENTS

Summary. The article presents problems related to enterprise systems maintenance. The common model of deploying and using these systems is when they are build by specialized entities while after deployment they are operated and tuned by customers. Due to the need of continuous tailoring and upgrading of the system there is the risk of integrity loss of some deployment artifacts. The article presents possible sources of problems, the concept of how to prevent them and the exemplary implementation based on the Oracle DBMS.

Keywords: corporate systems, system maintenance, integrity management, Oracle

1. Wstęp

Współcześnie projektowane i wdrażane systemy informatyczne klasy korporacyjnej (ang. *enterprise*) są zwykle złożonymi strukturami, których każdy element pełni ściśle zdefiniowaną rolę, czyniąc zadość stawianym przed nim wymogom zarówno funkcjonalnym (spełnianie reguł biznesowych obowiązujących w danej dziedzinie przedmiotowej), jak i niefunkcjonalnym (niezawodność, bezpieczeństwo, wydajność itd.). Umiejętny podział złożonego systemu na mniejsze komponenty o ściśle zdefiniowanej odpowiedzialności czyni go łatwiejszym w zarządzaniu, co zostało zauważone już w latach siedemdziesiątych przez sformułowanie zasad luźnego powiązania (ang. *loosely coupling*) czy powtórnego użycia (ang. *reusing*). Jednak dopiero rozwój technologii komputerowych oraz upowszechnienie Internetu spowodowały ich zaszczepienie na gruncie systemów informatycznych. Systemy takie są przez to łatwiejsze w zarządzaniu rozumianym jako ich wytwarzanie, testowanie, wdrażanie i utrzymywanie. Ponieważ jednak raz wdrożony system ciągle podlega rozwojowi, koniecznością staje się objęciem, poszczególnych jego składowych mechanizmem kontroli zarządzania wersjami, tak aby przykładowy komponent, korzystający z funkcjonalności innego komponentu usługowego, miał pewność, że funkcjonalność ta będzie realizowana zgodnie z obowiązującymi wymogami. Przyczyny potrzeb wdrażania mechanizmów kontroli wersji, ogólna ich koncepcja dla złożonych systemów oraz przykładowa implementacja zostały omówione w [1].

Mechanizmy kontroli wersji wdraża się zazwyczaj przez implementację dedykowanych interfejsów, za pomocą których poszczególne składowe systemu mogą się wzajemnie „odpytywać" o wersję dostarczanej funkcjonalności. Sposób realizacji tych mechanizmów jest najczęściej deklaratywny, co oznacza, że w trakcie cyklu wytwórczego oprogramowania podejmuje się decyzję o realizacji zmian danej grupy funkcjonalności w ramach wersji o konkretnym identyfikatorze, który to następnie jest „zaszywany" (ang. *hardcoded*) w implementacji interfejsu kontroli wersji.

Brak bezpośredniego powiązania pomiędzy identyfikatorem wersji a artefaktami wdrożeniowymi (pliki wykonywalne, pliki interpretowane, pliki konfiguracyjne, baza danych) uniemożliwia w środowisku wdrożeniowym dokonania weryfikacji, czy są one zgodne z tymi, które opuściły środowisko wytwórcze, a co za tym idzie, czy faktycznie spełniają założone wymagania, co często może prowadzić do awarii systemu.

O ile w dziedzinie zarządzania cyklem wytwórczym oprogramowania istnieje wiele znanych metodyk – traktujących również o fazie wdrażania i utrzymania systemów informatycznych – to ich zasięg kończy się bardzo często na przedstawieniu zbiorów przepisów odpowiadających na pytania kiedy i jak postępować. Jeśli są one wspomagane przez jakieś narzędzia, to zazwyczaj narzędzia te ograniczają się do wspierania technologii, na których wyrośli

autorzy danych metodyk. Artykuł przedstawia koncepcję i przykładową implementację narzędzia służącego do kontroli spójności artefaktów wdrożeniowych, którego architektura jest otwarta i rozszerzalna.

2. Przedstawienie problemu i koncepcja jego rozwiązania

2.1. Geneza problemu

Problem spójności wdrożonego systemu, a co za tym idzie realizowania funkcjonalności z dokładnością do deklarowanej wersji, pojawił się już z chwilą pierwszych zastosowań komercyjnych technologii komputerowych. W sytuacji kiedy inny podmiot wytwarza oprogramowanie, a inny je użytkuje, zawsze zachodzi wątpliwość, kto ponosi odpowiedzialność za ewentualną niezgodności artefaktów wdrożeniowych z tymi, które opuściły środowisko wytwórcze. Ponieważ jednak jest to zazwyczaj sytuacja niedopuszczalna, więc dzięki implementacji dodatkowych mechanizmów programowych, mogłaby ona być wykrywana przez samo oprogramowanie.

W przypadku prostych systemów aplikacyjnych wiele nowoczesnych platform wytwórczych dostarcza już gotowe mechanizmy umożliwiające kontrolę spójności artefaktów wdrożeniowych. Przykładem może być tutaj platforma JAVA, która umożliwia pakowanie wszystkich składowych plików aplikacji do jednego archiwum, które stanowi samodzielną jednostkę wykonawczą, zaś spójność samego archiwum jest weryfikowana przy każdym uruchomieniu aplikacji. Mechanizm umożliwia również zastosowanie podpisu elektronicznego do sprawdzenia, czy zawartość archiwum jest dokładnie taka sama, jak podpisana przez wytwórcę oprogramowania [2].

W przypadku gdy architektura wytwarzanego systemu zakłada wykorzystanie współpracujących technologii wielu różnych dostawców sytuacja ulega znacznej komplikacji. Należy również zwrócić uwagę na fakt, że o ile odpowiednio stosowana koncepcja luźnego powiązania zazwyczaj przynosi wymierne korzyści, to w przypadku składnic danych (serwery baz danych, serwery usług katalogowych) możliwość jej zastosowania może być znacznie ograniczona. Dzieje się tak w przypadku, gdy dany system obejmuje rozległy obszar merytoryczny, w którym dane mają być składowane. O ile logikę biznesową zwykle można łatwo wydzielić i pogrupować w podobszary merytoryczne udostępniane innym systemom jako grupy usług, to w przypadku skomplikowanego schematu danych trudno jest go podzielić na niezależne podschematy bez utraty kontroli nad spójnością niektórych danych. Nie zmienia to faktu, że pewne grupy usług będą zazwyczaj korzystać tylko z pewnego podobszaru całości schematu, a inne z innego podobszaru. Schemat składnic danych ewoluuje wraz z całym systemem, a jego defini-

cja jest szczególnie podatna na powstawanie niespójności między wersją wytworzoną a wdrożoną, dlatego powinien on tym bardziej podlegać kontroli spójności w środowisku wdrożeniowym. Co więcej, ponieważ w „zakresie zainteresowania" poszczególnych modułów leżą różne obszary całości schematu, więc ze względów wydajnościowych i użyteczności (niespójność obszaru leżącego poza „zainteresowaniem" danego modułu nie powinna wpływać na jego funkcjonowanie) pożądana jest możliwość weryfikacji spójności tylko wybranych obszarów schematu.

Sytuacja komplikuje się znacznie w chwili, gdy rozważania zaczynają dotyczyć nie systemu operacyjnego funkcjonującego na pojedynczym komputerze, a usług rozmieszonych w rozproszonym środowisku heterogenicznych węzłów, korzystających nierzadko z kilku niezależnych składnic danych.

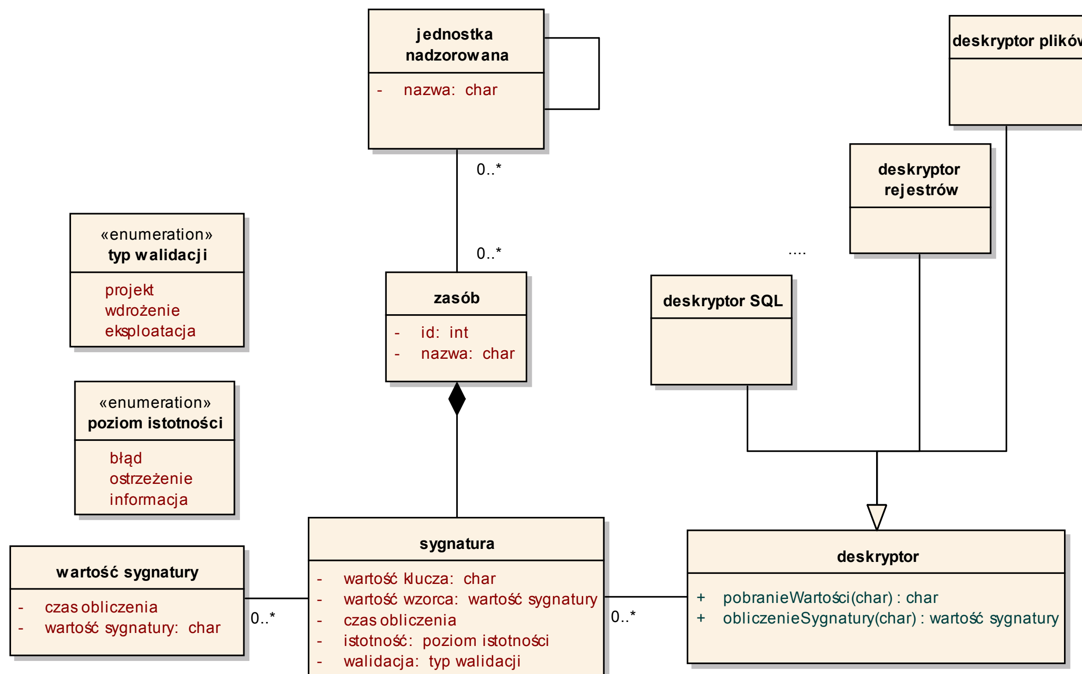
2.2. Model pojęciowy i koncepcja rozwiązania

Przeciwdziałanie zagrożeniom wyszczególnionym w poprzednim podpunkcie może zostać zrealizowane przez implementację i wdrożenie mechanizmu kontroli spójności artefaktów wdrożeniowych. Ponieważ, w ogólnym przypadku, system złożony jest z wielu składowych realizowanych w różnych technologiach, zatem konieczne jest sprawdzanie każdej ze składowych z uwzględnieniem jej specyfiki. Z każdą wersją systemu konieczne jest zapamiętanie wzorcowej sygnatury każdej ze składowych, aby później, w środowisku wdrożeniowym, możliwe było jej porównanie z sygnaturami obliczonymi dla składowych zastanych.

Na rys.1 przedstawiono diagram klas wprowadzający w dziedzinę problemową.

Najmniejszym elementem podlegającym kontroli spójności jest jednostka nadzorowana. Stanowi ona arbitralnie wydzielony i dobrze zdefiniowany fragment całości wdrożonego systemu. Kluczem do takiego wydzielenia mogą być potrzeba kontroli fragmentu całości infrastruktury na potrzeby pojedynczego modułu operującego tylko na tym fragmencie oraz potrzeba możliwości powtórnego użycia – raz zdefiniowana jednostka nadzorowana może być weryfikowana na potrzeby wielu składowych systemu. Powtórne użycie jednostki nadzorowanej może być również wykorzystane dla definicji hierarchicznych, gdzie mniejsze jednostki nadzorowane mogą być grupowane w konglomeraty.

Jednostka nadzorowana może składać się z kilku zasobów. Zwykle, choć nie jest to regułą, jednostka nadzorowana będzie pewnym merytorycznym fragmentem dziedziny problemowej, którą obejmuje system, zaś jej zasoby stanowić będą odzwierciedlenie tego fragmentu w poszczególnych warstwach/składowych technologicznych systemu (bazy danych, silniki procesów biznesowych, warstwa prezentacji itd.).



Rys. 1. Diagram klas – model dziedziny
Fig. 1. Class diagram – domain model

Przykładowo, jednostka „raporty kontrahentów" mogłaby zawierać jednostki „raporty nabywców" oraz „raporty sprzedawców", tak aby nie było potrzeby powielania tych samych powiązań z zasobami.

Chcąc opisać zasób, należy zdefiniować opisujące go sygnatury. Przykładowo, aby opisać strukturę pojedynczej tabeli bazy danych, najwygodniej będzie zdefiniować osobne sygnatury dla listy kolumn z typami wyzwalaczy, referencji itd. Każdej sygnaturze może być przypisany poziom istotności – analogicznie do bibliotek rejestrowania zdarzeń – przykładowo, poziom „błąd" oznacza, że stan danego fragmentu infrastruktury uniemożliwi poprawne funkcjonowanie modułu, który powiązano z daną jednostką nadzorowaną – w przykładowym przypadku bazy danych może się tak stać, np. dla niezgodnego z oczekiwanym typu kolumny. Z kolei poziom „ostrzeżenie" oznacza, że niezgodność nie jest krytyczna, ale może wpływać w jakiś sposób na realizację założonej funkcjonalności – przykładowo może to być niezgodna ze wzorcową budowa indeksu dla danej tabeli itd.

Drugą klasę kategorii sygnatury stanowi „typ walidacji". Wartość „projekt" oznacza, że odcisk wzorcowy może zostać określony już w środowisku wytwórczym i będzie niezmienny niezależnie od procesu wdrożenia – tak więc dla każdego wdrożonego systemu będzie dokładnie taki sam. Wartość „wdrożenie" oznacza, że odcisk wzorcowy może być określony dopiero na etapie wdrożenia, ale musi być niezmienny w czasie dalszej eksploatacji systemu – w przykładowym przypadku bazy danych może tak być dla niezmiennych słowników, których zawartość zależna jest jednak od miejsca wdrożenia systemu itd.

Klasa „wartość sygnatury" stanowi dokumentację (ang. *log*) wyliczonych wartości sygnatur i ma zastosowanie diagnostyczne – dokumentując wartości sygnatur wraz z czasem ich obliczenia, można wnioskować o czasie i powodzie powstania rozbieżności z sygnaturą wzorcową.

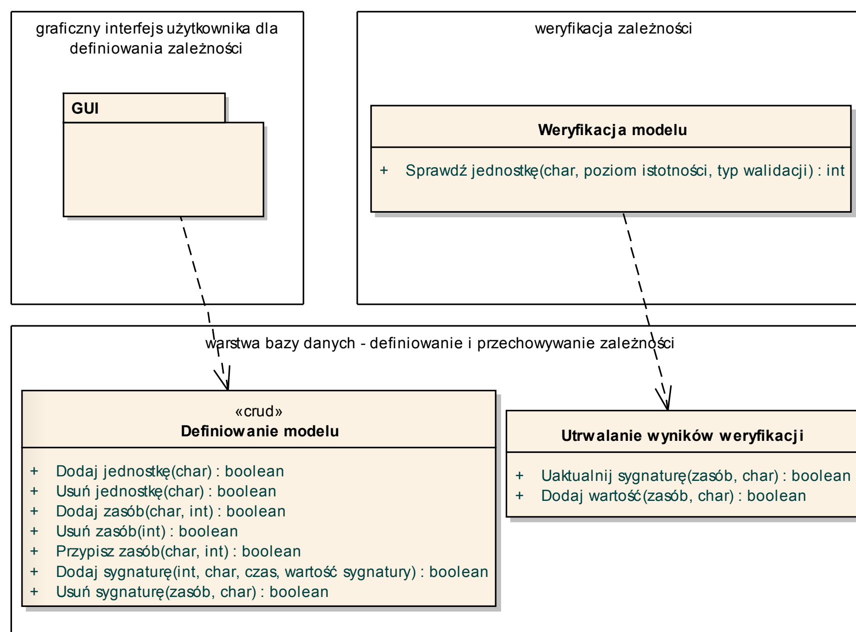
Każda sygnatura powiązana jest z deskryptorem, który stanowi „dostawcę" usług pobrania wartości opisywanej przez klucz sygnatury oraz obliczania odcisku sygnatury. Należy zwrócić uwagę, że dla każdej technologii (relacyjna baza danych, LDAP, interpretowane kody źródłowe, pliki wykonywalne aplikacji, pliki wykonywalne kontenerów) implementacja interfejsu deskryptora zazwyczaj będzie odmienna.

3. Implementacja

Implementacja przykładowego systemu kontroli spójności wdrożeniowej, oparta na koncepcjach opisanych w poprzednim punkcie, podzielona została na dwie podstawowe warstwy:

- warstwa definiowania zależności wdrożeniowych (używana w czasie wytwarzania oprogramowania – ang. *design time*),
- warstwa kontroli zależności wdrożeniowych (używana przez oprogramowanie działające już w środowisku wdrożeniowym – ang. *run time*).

Zostało to zobrazowane diagramem na rys. 2.



Rys. 2. Architektura logiczna systemu
Fig. 2. System logical architecture

Dla ułatwienia definiowania zależności projektant może posłużyć się opcjonalnie pakietem udostępniającym mu tę funkcjonalność w postaci interfejsu graficznego.

Algorytm realizowany przez metodę sprawdzającą zgodność jednostki jest bardzo prosty. Mając za parametry identyfikator jednostki, poziom istotności oraz typ walidacji, metoda sprawdzająca iteruje ona po wszystkich sygnaturach zasobów przypisanych do danej jednostki, dla zgodnych typów walidacji i poziomu istotności nie wyższego niż zadany, i dla każdej sygnatury, za pomocą specyficznej implementacji deskryptora, oblicza sygnaturę dla podanego klucza. Następnie porównuje ją z wartością wzorcową i, w przypadku wystąpienia niezgodności, zgłasza odpowiedni komunikat.

Poniżej, dla lepszego zobrazowania zagadnienia, przedstawiono przykład implementacji deskryptora dla bazy danych Oracle w prostym zastosowaniu. Załóżmy, że projektant pewnej jednostki programowej obsługującej słownik pracowników chce, aby przed jej każdorazowym uruchomieniem w środowisku wdrożeniowym odbyło się sprawdzenie, czy struktura tabeli jest dokładnie taka jak opisana wyrażeniem:

```
create table pracownicy (id integer, nazwisko varchar2(40), imie varchar2(20));
```

Wtedy definiuje model zawierający jednostkę „pracownicy", do której będzie przypisany zasób „tabela pracowników", który będzie zawierał sygnaturę o wartości klucza:

```
select column_name||data_type||data_length c from all_tab_columns where
TABLE_NAME='PRACOWNICY'
```

Wzorcowa wartość sygnatury, obliczona za pomocą deskryptora, wyniesie wtedy:

```
9305C1913F7F553795008376A3C528BD
```

Przykładowy kod źródłowy deskryptora przedstawiono poniżej:

```
function GetSignature( pid sealadm.signatures.id%type ) return varchar2 is
  stm varchar2;
  val varchar2;
begin
  select s.key into stm from sealadm.signatures s where s.id = pid;
  execute immediate
  '
  select
  dbms_crypto.Hash (
  utl_raw.cast_to_raw (
                 rtrim (xmlagg (xmlelement (e,
                        (t.c)
                 || '',,'')).extract (''//text()''), '','')
  )
  ,2)

  from
  ( ' ||
    stm
  || ') t

  '
  into  val ;
  return val;
end;
```

Jak wynika z powyższego przykładu, dla weryfikacji zgodności struktur bazy danych wykorzystywana jest zawartość słowników systemowych bazy danych Oracle. Jest to podejście

skrajnie przeciwne do np. wykorzystania interfejsów API ODBC. Z jednej strony wyklucza to przenaszalność rozwiązania na inne platformy baz danych, z drugiej jednak pozwala na szczegółowe uwzględnienie specyfiki platformy Oracle, a w szczególności na kontrolę rozszerzeń wersji enterprise edition. Dla obliczenia wartości sygnatury w powyższym przykładzie wykorzystano funkcję mieszającą (ang. *hash function*), pochodzącą z pakietu dbms_crypto, nic nie stoi jednak na przeszkodzie, żeby w przypadku potrzeby podniesienia poziomu wiarygodności, wykorzystać mechanizm podpisywania zawartości.

4. Podsumowanie

W artykule przedstawiono zagrożenia, na które narażone są aplikacje funkcjonujące w środowiskach korporacyjnych, których różne obszary podlegają niezależnym cyklom wytwarzania oprogramowania, a szczególnie w sytuacji, gdy środowisko wdrożeniowe nie jest bezpośrednio administrowane przez dostawców oprogramowania. Opracowano model zależności, którego zdefiniowanie i utrzymywanie dla każdego z wytwarzanych systemów pozwoli na kontrolę ich spójności w środowiskach wdrożeniowych. Przykładowa implementacja zaproponowanego rozwiązania zrealizowana jest w zakresie definiowania modelu zależności oraz kontroli spójności dla bazy danych Oracle. Jest ona praktycznie wykorzystywana przez system opisany w [1] podczas procesu instalacji i aktualizacji składowych oprogramowania oraz podczas uruchamiania wykonywalnych składowych części bazodanowej (enkapsulowanych w pakietach). Zastosowane podejście pozwoliło na kontrolę zgodności infrastruktury bazy danych z dokładnością do opcji enterprise edition.

BIBLIOGRAFIA

1. Wyciślik Ł.: Zarządzanie instalacją i aktualizacją zależnych systemów aplikacyjnych w środowiskach korporacyjnych. Studia Informatica Vol. 32, No. 2B (97), Wydawnictwo Politechniki Śląskiej, Gliwice 2011.
2. Austin C., Pawlan M.: Advanced Programming for the Java 2 Platform. November 1999.
3. Oracle Database Administrator's Guide 11g Release 1 (11.1). March 2008.
4. Oracle Database Security Guide 11g Release 1 (11.1). November 2011.

Wpłynęło do Redakcji 10 stycznia 2012 r.

Abstract

The article presents the risks faced by applications running in enterprise environments, especially when deployment infrastructures are managed and tuned by entities not involved in software development process. This situation brings risk of integrity loss of some deployment artifacts. The article presents possible sources of problems, the concept of how to prevent them and the exemplary implementation based on Oracle DBMS.

Adres

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