Automatyczne generowanie kodu

Wstęp – dla kogo to jest?

Automatyczne generowanie kodu ma zastosowanie w dwóch zasadniczych przypadkach: gdy developer chce szybko napisać dużą część powtarzalnego kodu lub gdy implementujemy duży system z udziałem wielu osób i przewidziany do wieloletniego rozwoju i utrzymania.

Development bez użycia automatycznego generowania kodu - przykład

Żeby zainspirować wyobraźnię czytelnika przedstawię sytuację z jaką możemy się spotkać nie stosując automatycznej generacji kodu w najbardziej powszechnym miejscu jakim jest dostęp do bazy danych (zwany najczęściej O/R mapping'iem). W tej sytuacji oprogramowujemy dostęp do bazy danych przy pomocy klasycznego JDBC. Mamy kawałek skryptu SQL który pobiera nam dane z tabeli „`klient`”, w tym pole o nazwie "`adres`":

```
q = "SELECT … , adres, ... FROM klient WHERE id = 1";
```

Następnie z wyników zapytania pobieramy zwartość i przepisujemy do zmiennej lokalnej:

```
String adres = rs.getString("adres");
```

Dodajmy do tego jeszcze fakt, że pole o nazwie "`adres`" jest używane jeszcze w kilku innych tabelach, np. "`zamówienie`", "`dostawca`", "`punkt_odbiorczy`", itp.

Przypuśćmy teraz że po roku działania aplikacji musimy wdrożyć ją w innym kraju, gdzie zwyczajowo używa się dwóch linii na pole adresu. Osoba odpowiedzialna za bazę danych wykonuje zmianę w modelu bazy wszystkich wystąpień pola "`adres`" na "`adres_linia_1`" i "`adres_linia_2`", przygotowuje odpowiednie skrypty migracyjne a następnie przekazuje zmianę do oprogramowania programiście. W większych zespołach najpewniej będzie to inna osoba niż autor oryginalnego kodu, więc najlepsze co może taka osoba zrobić to metodycznie wyszukać wszystkie wystąpienia w kodzie napisów "`adres`" i cierpliwe je zamieniać na obsługę dwóch nowych pól.

Niestety takie podejście niesie za sobą bardzo poważne zagrożenia:

1. Niekompletność: może się zdarzyć że developer, jako że też jest człowiekiem i ma swoje gorsze i lepsze dni, może nie znaleźć wszystkich wystąpień, szczególnie jeżeli poprzednik zostawi mu pułapkę w postaci:

   ```
   q = "SELECT … , adr" + "es, ... FROM klient WHERE id = 1";
   ```

2. Nadmiarowość: developer rozpędzi się i zmieni kod w miejscu gdzie model bazy nie uległ zmianie.
3. O ewentualne pomyłce dowiemy się nie w momencie kompilacji kodu, ale w czasie testów funkcjonalnych lub - bardzo często - po wdrożeniu produkcyjnym od rozgniewanego klienta.

Development z użyciem automatycznego generowania kodu - przykład

Zastosowanie automatycznej generacji kodu pozwala nam wykorzystać zaletę płynącą z kompilacji

kodu przed uruchomieniem. Dzięki temu możliwe jest wcześniejsze dostrzeżenie błędu przez developera i uniknięcie kompromitacji przed klientem.

Automatyczne wygenerowanie kodu dostępu do bazy danych powinno tworzyć nam zarówno obiekty reprezentujące rekordy w bazie danych jak i klasy umożliwiające łatwy dostęp do nich. Analogiczne fragmenty kodu przedstawione w powyższym przykładzie przed zmianą modelu będą wyglądać następująco:

```
Klient k = klientDAO.getByPK(1);
String adres = k.getAdres();
```

Po zmianie w modelu pola "`adres`" na "`adres_linia_1`" oraz dodaniu nowego "`adres_linia_2`" i ponownym wygenerowaniu kodu dostępu do bazy danych wszystkie miejsca gdzie było odwołanie do pola "`adres`", czyli druga linia w powyższym kodzie, przestaną się kompilować. Developer, nawet mało doświadczony i po nieprzespanej nocy, dostanie wszystko na tacy, w szczególności dlatego że współczesne środowiska developerskie (np. Eclipse) oznaczą nam błąd tak wyraźnie jakbyśmy napisali w edytorze tekstu "gura" zamiast "góra"...

Ergonomia pracy

Pisząc o środowiskach developerskich nie sposób nie wspomnieć o jeszcze jednym ułatwieniu bardzo podnoszącym efektywność pracy programisty: funkcja automatycznego dopełniania nazw metod, która jest dostępna np. w Eclipse po wciśnięciu kombinacji klawiszy Crtl-Space: wystarczy że w powyższym przykładzie programista napisze "`k.getA`" i użyje dopełniania a edytor sam dopisze brakujący fragment "`dres()`". W tym wypadku zysk nie jest duży, ale jeżeli mamy w bazie danych kolumnę o nazwie "`care_of_edit_for_one_time_addr`" (to nie żart...) to jej ręczne bezbłędne przepisanie z wydrukowanego schematu bazy danych za pierwszym razem jest w zasadzie niemożliwe.

Implementacja

Najprostszym sposobem generowania gotowego kodu jest znalezienie gotowej biblioteki która za nas to zrobi. Szukając gotowego rozwiązania trzeba zwrócić uwagę czy:

- jest ono popularne, ma wiele użytkowników i ewentualne wsparcie: forum, listy dyskusyjne
- będzie ono dostosowane do naszego systemu budowania aplikacji - jeżeli korzystamy np. ant czy maven'a
- jest na bieżąco poprawiane i rozwijane.

Wybierając gotową bibliotekę trzeba mieć świadomość że będzie to dłuższa znajomość, niemal jak małżeństwo... W obszarach tak krytycznych jak O/R-mapping trzeba będzie ją dobrze poznać, zrozumieć, a ewentualny rozwód i wymiana na inny (młodszy...) model będzie bardzo trudna lub wręcz niemożliwa.

Wdrażając bibliotekę automatycznie generującą kod trzeba pamiętać o podstawowych zasadach:

- wygenerowany kod **nie może być umieszczany w repozytorium** (np. CVS, SVN), tylko przy każdej iteracji budowania musi być tworzony ze źródeł (model bazy, pliki konfiguracyjne) - w przeciwnym wypadku mielibyśmy do czynienia z "dwoma źródłami prawdy", które prędzej czy później okażą się sprzeczne,
- wygenerowany kod **nie może być edytowany ręcznie** - to powinno być zrozumiałe, ponieważ przy powtórzeniu iteracji generowania nasze zmiany zostaną zamazane.

Obszary zastosowań

Generowanie kodu powinno stosować się wszędzie gdzie jest to możliwe i/lub są do tego stosowne narzędzia. Sugestią do generowania kodu jest istnienie w aplikacji kontraktu zapisanego w postaci jakiegoś pliku. Takim kontraktem może być:

- model bazy danych (kontrakt z architektem systemu),
- model wymiany danych przez WebService zapisany jako WSDL (kontrakt z systemem zewnętrznym),
- pliki konfiguracyjne (kontrakt z projektantem aplikacji)
- klucze plików lokalizacyjnych (kontrakt z osobami tłumaczącymi wersje językowe).

Baza danych (O/R-mapping)

To zastosowanie zostało częściowo opisane w przykładzie wprowadzającym. Jest to najczęściej spotykany obszar gdzie używa się generacje kodu, ponieważ:

- większość aplikacji pisanych w Javie korzysta z bazy danych,
- baza danych ma dużo tabel i kolumn więc ręczne tworzenie kodu dostępowego (np. przez JDBC) jest bardzo czasochłonne
- baza danych rozwijanej aplikacji podlega częstym zmianom
- w większych projektach inna osoba jest odpowiedzialna za zmiany w modelu bazy danych (architekt) oraz za zmiany w kodzie aplikacji (programista), przez co może dojść do niedopowiedzeń i pomyłek przy prowadzaniu zmian
- baza danych jest krytyczna dla działania aplikacji a nawet drobna literówka w kodzie powoduje wystąpienie sytuacji wyjątkowej (dobrze znany `SQLException`) i błąd biznesowy aplikacji

Powstało wiele bibliotek do komunikacji z bazą danych przez mapowanie rekordów na obiekty (ang. O/R-*mapping*), jednak należy zwrócić uwagę że niewiele z nich generuje kod Java. Ponadto rozważając wybór takiej biblioteki trzeba uważnie sprawdzić listę wspieranych baz danych, szczególnie jeżeli korzystamy lub mamy zamiar korzystać z mniej popularnego produktu.

Hibernate

Aby pokazać że niekażda biblioteka O/R-mappingowa spełnia wszystkie warunki stawiane automatycznemu generowaniu kodu zacznę od opisu możliwości Hibernate (http://www.hibernate.org) - chyba najpowszechniej używanej bibliotece do O/R-mapping'u w Javie. Hibernate jest bez wątpienia produktem dopracowanym, bardzo popularnym, intensywnie rozwijanym, posiadającym wręcz doskonałe wsparcie. Niestety jest jeden problem: strategicznym zamysłem przy tworzeniu Hibernate było zapewnienie mapowania już napisanych klas w Javie na rekordy w bazie a nie generowanie ich automatycznie. Faktem jest że generowanie obiektów POJO (*ang. Plain Old Java Object* - proste obiekty odpowiadające strukturze rekordu z bazy) jest możliwe przy pomocy dodatkowej biblioteki "z rodziny" Hibernate o nazwie 'hibernate-tools" jednakże nie można uniknąć wrażenia że ten temat jest nisko priorytetowy: rozwój tej biblioteki nie zawsze nadąża za główną biblioteką hibernate-core (co było dobrze widoczne przy przejściu z Java 1.4 na 1.5). Ponadto Hibernate nie umożliwia tworzenia zapytania w oparciu o wygenerowane stałe odpowiadające nazwom tabel i kolumn.

Przykład zapytania pobierającego rekord po kluczu głównym oraz rekordów spełniających dwa warunki (nazwa ulicy i status aktywny):

```
Session session = HibernateUtil.getSessionFactory().getCurrentSession();
```

```
Klient k = (Klient) session.get(Klient.class, 1);
String adres = k.getAdres();

String q = " FROM " + Klient.class.getName() + " AS klient " +
            " WHERE klient.adres = :adres AND klient.aktywny = : aktywny ";
List<Klient> result = (List<Klient>)session.createQuery(q).
            setString("adres", "Marszalkowska").
            setBoolean("aktywny", true).list();
```

Jak widać na tym przykładzie, problem ze zmianą pola „`adres`" na „`adres_linia_1`" i „`adres_linia_2`" zostałby przez kompilator znaleziony tylko częściowo, tzn. przestałaby się kompilować linia 3 zawierająca „`k.getAdres()`" (bo już nie ma takiej metody), jednakże zapytanie zapisane w zmiennej `q` ciągle byłoby prawidłowe i to programista ręcznie musiałby znaleźć i poprawić wszystkie takie wystąpienia.

Torque

Torque (http://db.apache.org/torque/) to produkt bliższy naszym oczekiwaniom: głównym zamysłem jego twórców było właśnie generowanie kodu klas POJO oraz DAO (*ang. Data Access Object* - obiekt dostępu do danych) - wygenerowane przez Torque klasy tego typu mają niezbyt szczęśliwy sufiks "Peer" (*ang. wypatrywać*).

Torque składa się z dwóch części, pierwsza służąca do generowania obiektów, druga używana w aplikacji do korzystania z nich. Biblioteka ta posiada możliwość tworzenia zapytań w oparciu o wygenerowane stałe odpowiadające nazwom tabel i kolumn, dzięki czemu zapewniamy sobie gwarancje że zmiana nazw kolumn czy tabel spowoduje błędy kompilacji dzięki którym łatwo obsłużymy ją w kodzie.

Minusem Torque jest fakt że jest to biblioteka bardzo stara, utworzona wcześniej jako część większego frameworku Turbine. Jej kod źródłowy jest nie najwyższej jakości a niektóre wzorce bardzo niewygodne, począwszy od statycznych metod w oparciu o który działają wygenerowane klasy Peer, co praktycznie uniemożliwia mockowanie i testy jednostkowe.

Przykład zapytania pobierającego rekord po kluczu głównym oraz rekordów spełniających dwa warunki (nazwa ulicy i status aktywny):

```
Klient k = KlientPeer.retrieveByPK(1);
String adres = k.getAdres();

Criteria crit = new Criteria();
crit.add(KlientPeer.ADRES, "Marszalkowska").and(KlientPeer.AKTYWNY, true);
List<Klient> result = KlientPeer.doSelect(crit);
```

Jak widać na tym przykładzie, problem ze zmianą pola „`adres`" na „`adres_linia_1`" i „`adres_linia_2`" zostałby przez kompilator znaleziony we wszystkich miejscach, tzn. przestałaby się kompilować linia 2 zawierająca „`k.getAdres()`" (bo już nie ma takiej metody) oraz linia 5 zawierająca „`KlientPeer.ADRES`" ponieważ nie ma już takiej stałej – programista poprawiając błędy kompilacji będzie mieć pewność że nowe zapytania SQL będą poprawne.

XML

W przypadku XML'a mamy podobną sytuację. Załóżmy że integrujemy się z jakimś zewnętrznym systemem, umawiamy się że dane będą wymieniane jako pliki w formacie XML. Sposobów na wygenerowanie plików XML są dziesiątki, począwszy od System.out.println(...) aż do tworzenia

reprezentacji w pamięci obiektów DOM (*ang. Document Object Model*). Podobnie jest z czytaniem plików XML: mamy do dyspozycji reprezentacje DOM, interfejs SAX (*ang. Simple API for XML*) – szczególnie przydatny w przypadku dużych plików które mogą w całości nie zmieścić się w pamięci, itp. Niestety w przypadku gdy zmieni się kontrakt (zapisany w dokumencie DTD lub Schema) to aplikacja będzie nadal się kompilować bez przeszkód a o ewentualnym błędzie dowiemy się dopiero po uruchomieniu aplikacji.

Zabezpieczeniem przed takim błędem jest wykorzystanie biblioteki generującej kod Java na podstawie kontraktu. Przykładem takiej biblioteki jest JAXB (ang. *Java Architecture for XML Binding*, https://jaxb.dev.java.net/). Na podstawie kontraktu zapisanego w pliku schema generuje pliki Java które następnie są wypełniane danymi i przekształcane w postać strumienia XML.

Część definicji w pliku schema mogłaby wyglądać tak:

```
<xsd:element name="klienci" type="t:klienci_type" />

<xsd:complexType name="klienci_type">
    <xsd:sequence>
        <xsd:element name="klient" type="t:klient_type"
            minOccurs="0" maxOccurs="unbounded" />
    </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="klient_type">
    <xsd:attribute name="id" type="t:positive_integer" use="required" />
    <xsd:attribute name="imie" type="t:text255" use="required" />
    <xsd:attribute name="nazwisko" type="t:text255" use="required" />
    <xsd:attribute name="status" type="t:statusKlienta" use="required" />
    <xsd:attribute name="adres" type="t:text255" use="required" />
    <xsd:attribute name="birthdate" type="t:dateYYYYMMDD" use="required" />
</xsd:complexType>
```

Na podstawie takiej definicji biblioteka jaxb wygeneruje nam obiekty POJO z których możemy skorzystać w przykładowy sposób:

```
KlientType klient1 = factory.createKlientType();
[...]
klient1.setStatus(StatusKlientaEnum.T);
klient1.setAdres("Marszałkowska");

KlientType klient2 = factory.createKlientType();
[...]
klient2.setStatus(StatusKlientaEnum.N);
klient2.setAdres("Al. Niepodległości");

Klienci klienci = factory.createKlienci();
klienci.getKlient().add(klient1);
klienci.getKlient().add(klient2);

FileOutputStream out = new FileOutputStream(new File("/tmp/klienci.xml"));
marshaller.marshal(klienci, out);
```

W wyniku działania programu w pliku klienci.xml znajdziemy treść:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<klienci>
    <klient adres="Marszałkowska" birthdate="20090730" id="1"
      imie="Marek" nazwisko="Abacki" status="T"/>
    <klient adres="Al. Niepodległości" birthdate="20090730" id="2"
```

```
      imie="Krzysztof" nazwisko="Babacki" status="N"/>
</klienci>
```

Jak można zauważyć w tym przykładzie, wygenerowane obiekty POJO przypilnują za nas zgodność struktury pliku, zgodność typów argumentów (np. format daty) a nawet tego czy typy wyliczeniowe są zgodne (klasa `StatusKlientaEnum` zawiera stałe z dozwolonej przestrzeni). Jak łatwo się domyślić, zmiana w pliku schema w definicji klienta pola „`adres`" na „`adres_linia_1`" i „`adres_linia_2`" szybko doprowadzi do błędu kompilacji.

Pliki konfiguracyjne

Większość frameworków i aplikacji korzysta z plików konfiguracyjnych – przykładem może tu być Struts z plikiem `struts-config.xml` zawierający definicje formularzy. Fragment takiego pliku może wyglądać następująco:

```
<form-bean name="klientForm" type="org.apache.struts.action.DynaActionForm">
    <form-property name="imie"     type="java.lang.String" />
    <form-property name="nazwisko" type="java.lang.String" />
    <form-property name="adres"    type="java.lang.String" />
    <form-property name="aktywny"  type="java.lang.Boolean" />
</form-bean>
```

Standardowy kod pobierający dane z formularza wygląda podobnie do tego:

```
Klient klient = new Klient();
klient.setImie(form.getString("imie"));
klient.setNazwisko(form.getString("nazwisko"));
klient.setAdres(form.getString("adres"));
klient.setAktywny((Boolean)form.get("aktywny"));
```

Jak widać na przykładzie, odwołujemy się do nazw pól formularzy zdefiniowanych w pliku XML poprzez ich nazwy wpisując je „z palca". Zmiana definicji formularza (np. zmiana pola „`adres`" na „`adres_line_1`" nie spowoduje błędu kompilacji ale aplikacja przestanie działać.

Wykorzystując proste narzędzie do generowania kodu możemy na podstawie pliku XML wygenerować sobie interfejsy ze stałymi zawierającymi nazwy pól formularza dzięki czemu kod dostępu do tych pól będzie wyglądał następująco:

```
Klient klient = new Klient();
klient.setImie(form.getString(klientFormC.imie));
klient.setNazwisko(form.getString(klientFormC.nazwisko));
klient.setAdres(form.getString(klientFormC.adres));
klient.setAktywny((Boolean)form.get(klientFormC.aktywny));
```

Po zmianie definicji w pliku XML i przebudowaniu aplikacji powyższy kod przestanie się kompilować i programista będzie musiał nanieść poprawki. Ponadto dużo łatwiej znaleźć jest odniesienia do tego samego elementu: załóżmy że szukamy miejsc w kodzie gdzie są odniesienia do pola „`adres`" w formularzu klienta. Jeżeli będziemy tekstowo szukać wszystkich wystąpień napisu „`"adres"`" to znajdziemy również odniesienia do pola „adres" w formularzu kontrahenta, formularzu dostawcy, itd. a także inne przypadkowe wystąpienia niezwiązane z formularzami. Jeżeli jednak poszukamy odwołań do stałej `klientFormC.adres` to otrzymamy dokładnie żądany wynik

Nie spotkałem się z gotową biblioteką generującą takie stałe dla frameworku Struts, ponadto każdy framework i aplikacja mają swoje własne formaty plików konfiguracyjnych, także takie generatory

najlepiej napisać samemu. Nakład samodzielnej pracy na napisanie pierwszego takiego generatora w postaci taska Ant'a lub plugina Maven'a jest nie większy niż niż dzień, kolejne to już kwestia pojedynczych godzin.

Pliki lokalizacyjne

Kolejnym miejscem w którym warto rozważyć generacje kodu są klucze z plików lokalizacyjnych. Załóżmy że mamy plik validations.properties zawierający wiersze:

```
klient_form.adres.required=Pole "Adres" jest wymagane
klient_form.adres.maxlength=Pole "Adres" nie może zawierać więcej niż {0}
znaków.
```

Następnie w kodzie aplikacji walidującym dane wprowadzone do formularza mamy następujące odniesienia do kluczy lokalizacyjnych:

```
String adres = form.getString("adres");
if("".equals(adres.trim())) {
    errors.add("validations", "klient_form.adres.required");
}
if(adres.trim().length() > 255) {
    errors.add("validations", "klient_form.adres.maxlength", 255);
}
```

Jeżeli ktoś usunie wspomniane wpisy w pliku albo zmieni ich klucze to aplikacja przy próbie prezentacji komunikatu klientowi wygeneruje błąd.

Zastosowanie prostego generatora interfejsów ze stałymi na podstawie pliku propertiesów pozwoli nam zastąpić powyższy kod tak:

```
String adres = form.getString("adres");
if("".equals(adres.trim())) {
    errors.add(validationsC.klient_form.adres.required);
}
if(adres.trim().length() > 255) {
    errors.add(validationsC.klient_form.adres.maxlength, 255);
}
```

Usunięcie wpisu w pliku lokalizacyjnym spowoduje że aplikacja przestanie się kompilować a IDE pokaże nam klucze do których się odwołujemy w kodzie a nie ma ich w pliku. Takie rozwiązanie ułatwi nam znacząco również refaktoringi, np. przywoływaną wyżej zmianę pola „`adres`” na „`adres_line_1`” i „`adres_line_2`”.

Nie spotkałem się z gotową biblioteką generującą takie stałe, jednak - podobnie jak w przypadku plików konfiguracyjnych - nakład samodzielnej pracy na napisanie jej w postaci taska Ant'a lub plugina Maven'a jest nie większy niż niż dzień.

Podsumowanie

Generowanie kodu znacząco ułatwia pisanie i dalszy rozwój aplikacji, taki kod jest również nieoceniony w przypadku większych refaktoringów czy szukania wielu odniesień w kodzie do jednego elementu.

Zachęcam każdego do spojrzenia na własną aplikacje w poszukiwaniu umieszczonych w kodzie stałych napisowych i zastanowienia się co one reprezentują i czy nie powinny zostać zastąpione

stałymi generowanymi. Idealnym stanem byłaby sytuacja gdyby aplikacja zawierała wyłączne odniesienia do generowanych stałych - bez samodzielnego definiowania w kodzie stałych napisowych.

Plusy

- brak konieczności pisania oczywistego i powtarzalnego kodu (np. obiekty POJO reprezentującego wiersze z bazy danych)
- spójność kodu i powtarzalność wzorców
- błędy znajdowane na etapie kompilacji
- wykorzystanie auto-uzupełniania w IDE (Crtl-Space w Eclipse)
- łatwiejsze refaktoringi
- łatwiejsze znajdowanie wielu odniesień do tego samego elementu (Crtl-Shift-G w Eclipse)

Minusy

- uzależnienie od dodatkowych bibliotek zewnętrznych: konieczność aktualizacji zależności, możliwość występowania błędów,
- trudniejsze wprowadzenie nowego developera do projektu
- bardziej skomplikowany proces budowania aplikacji
- konieczność tworzenia własnych generatorów do specyficznych elementów (np. pliki konfiguracyjne)