Propozycja Dynamicznych Ról i Programowania Przez Kontrakt dla języka Java

Aleksander Kosicki

Politechnika Warszawska
wydział Matematyki i Nauk Informacyjnych

10 maja 2009

Przedmiot prezentacji

Prezentacja w luźny sposób przedstawia kontekst i problematykę pracy magisterskiej pt. "Propozycja Dynamicznych Ról i Programowania Przez Kontrakt dla języka Java".

- Praca powstała na wydziale Matematyki i Nauk Informacyjnych Politechniki Warszawskiej, pod kierunkiem dr inż. Krzysztofa Kaczmarskiego.
- Udana obrona miała miejsce 24 lutego 2009.
- Praca zdobyła II nagrodę w XXVI edycji Ogólnopolskiego Konkursu PTI na najlepsze prace magisterskie z informatyki.
- Treść pracy można znaleźć w Internecie.

Krótka charakterystyka pracy

Celem niniejszej pracy jest propozycja oraz implementacja pewnych dwóch koncepcji z zakresu języków programowania.

- Pierwsza z koncepcji związana jest z "Programowaniem Przez Kontrakt", druga z "Dynamicznymi Rolami".
- Obie z koncepcji zostały osadzone w istniejącym języku programowania, tj. w Javie.
- Charakter pracy jest eksperymentalny.
- Podejście autora do omawianych w pracy zagadnień ewoluowało w czasie.

Zakres Pracy

Z perspektywy wytworzonych artefaktów, pracę można podzielić na następujące części:

- Część teoretyczno-opisowa, tj.:
  - rozważania ogólne
  - propozycje konkretnych rozwiązań
- Część praktyczną, tj.:
  - dostarczenie implementacji† dla danej propozycji
  - ~~przygotowanie środowiska programistycznego~~

† W wypadku pierwszej z koncepcji, implementacja dostarczona jest w postaci translatora, w wypadku drugiej — jako biblioteka.

Motywacja

- Główną motywacją dla poruszenia niniejszych zagadnień były dla autora własne doświadczenia praktyczne.
- Propozycje autora nie zwiększają możliwość obliczeniowych Javy, pozwalają natomiast na uzyskanie większej zwięzłości, czytelności i, przede wszystkim, umożliwiają dalszy "rozdział funkcjonalności" (Separation of Concerns).
- Cechą wspólną obu propozycji jest, w pewnym sensie, formalizacja cyklu życia obiektów.
- Mimo, że propozycje owe zostały przedstawione na tle Javy, dotyczą one w ogólności języków obiektowych.

Część I

Programowanie Przez Kontrakt

Jakość oprogramowania

- "Jakość oprogramowania" - termin używany na określenie bardzo szerokiego zagadnienia.
- "Jakość oprogramowania" nigdy nie jest wystarczająca.
- Braki w jakości często wychodzą na jaw, gdy oprogramowanie funkcjonuje już w środowisku produkcyjnym (patche, poprawki, service packi).
- Jednym z najistotniejszych aspektów "Jakość Oprogramowania" jest zgodność wykonania programu z pewnymi oczekiwaniami, np. z wymaganiami funkcjonalnymi.

Prosty przykład

Hello World — wersja popularna. Sprawdzona i działająca.

hello1.c

```
#include <stdio.h>

int main(int argc, char *argv[]){
    printf("Hello World");
    return 0;
}
```

Prosty przykład c.d.

Hello World — wersja mniej typowa, oparta na problemie Collatza.

hello2.c

```
#include <stdio.h>

int main(int argc, char *argv[]){
    unsigned very_long int seed;
    scanf("%d", &seed);
    while(seed != 1)
        seed = seed & 1 ? 3*seed + 1 : seed / 2;
    printf("Hello World");
    return 0;
}
```

Dwa problemy

Problem 1:

Czy program *hello2.c* faktycznie będzie zawsze wypisywał powitanie? Analiza statyczne tego programu nie może być, przynajmniej na dzień dzisiejszy, przeprowadzona w sposób automatyczny, wymaga natomiast zastosowania zaawansowanego aparatu matematycznego i wiąże się z rozwiązaniem tzw. Problemu Collatza.

Problem 2:

Skąd w ogóle wiadomo jakie jest zadanie programu *hello2.c*?

Dwa problemy c.d.

Dyskusja 1:

Być może Problem Collatza jest rozstrzygalny (czego nie udowodniono). Jeśli tak, prędzej czy później, zostanie on rozwiązany i tym samym będzie wiadomo czy program *hello2.c* działa prawidłowo. W przeciwnym wypadku...

Dyskusja 2:

Nie wiadomo. Wymagania funkcjonalne programu nie zostały nigdzie zapisane. Jedyne co można powiedzieć, to że "zadaniem programu jest imperatywne wykonanie instrukcji zapisanych w kodzie źródłowym".

Dwa problemy c.d.

Rozwiązanie 1:

Analiza statyczna niestety nie zawsze jest możliwa. Pewnym obejściem tego problemu jest po prostu przeprowadzanie *odpowiedniej ilości* różnego rodzaju testów.

Rozwiązanie 2:

Zapisywanie wymagań funkcjonalnych — szczególnie w wypadku języków imperatywnych, gdzie kod programu stwierdza "jak" ale już nie "co". Najlepiej w sposób formalny, tak aby możliwa była ich automatyczna weryfikacja podczas testów.

Wprowadzenie

- Formalna specyfikacja wymagań dostępna jest w różnych w stopniu, w różnych klasach języków.
- Np. wypadku języków opisu sprzętu (Hardware Description Languages) istnieją towarzyszące im języki weryfikacji sprzętu (Hardware Verification Languages), często o znacznie bogatszej od swoich odpowiedników składni (Verilog i SystemVerilog). Na ogół języki te mogą występować razem w obrębie tych samych jednostek kompilacji. Języki HVL mają silne podstawy matematyczne (logiki temporalne).
- W wypadku języków programowania, specyfikacja wymagań występuje na ogół w formie elementów “Programowania Przez Kontrakt”, a często w ogóle nie jest przez dany język wspierana.

Wprowadzenie

Programowanie Przez Kontrakt — termin ukuty przez Bertranda Meyera w trakcie prac nad językiem Eiffel.

Przykład specyfikacji kontraktu w języku D

```
double divide(divident a, divisor b)
in { assert b != 0; } body {
        return divident / divisor;
};
```

Typowe elementy kontraktów:

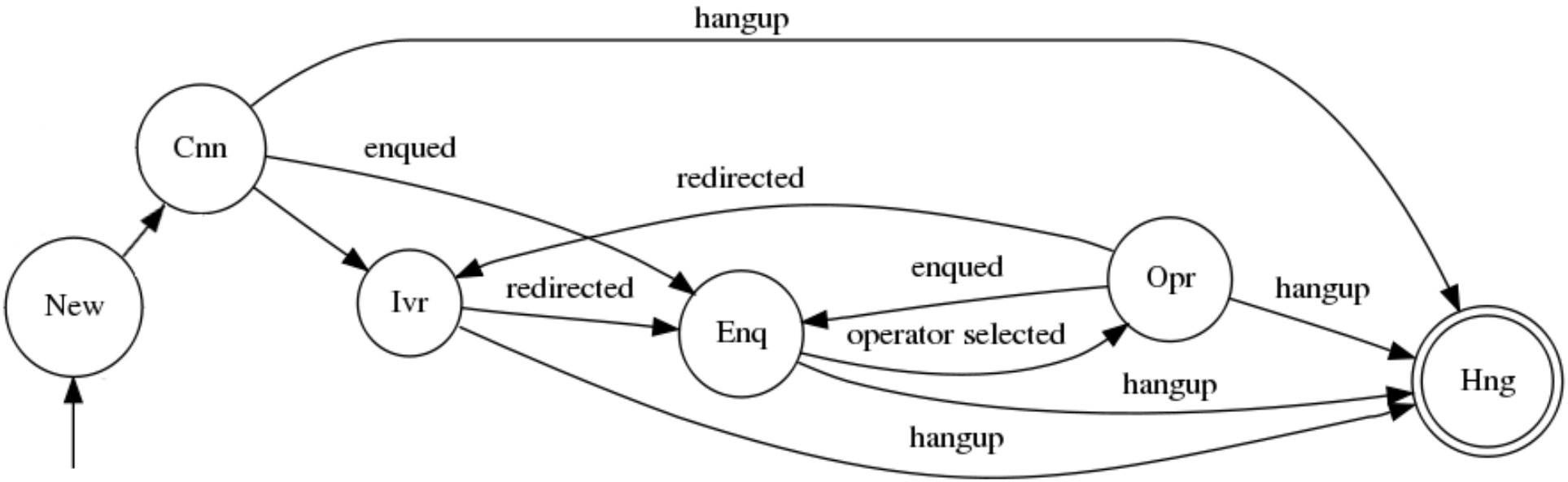
- warunki wstępne
- niezmienniki
- błędy
- warunki końcowe
- efekty uboczne
- asercje

- W Javie występują już pewne elementy kontraktów, jak np. asercje. Niestety język ten nie oferuje, i nie będzie w przewidywalnej przyszłości oferował, pełnej funkcjonalności PPK.

- Autor postanowił temu zaradzić i wzbogacić ten lubiany przez niego język o brakującą funkcjonalność PPK, dodając przy tym pewien unikalny element — **automaty skończone**:

Klasa jako automat skończony

Spostrzeżenie: niektóre klasy w językach programowania mogą realizować schemat działania automatu skończonego. Za przykład może tu posłużyć stworzona swego czasu przez autora klasa monitorująca połączenia centralki telefonicznej:



Fakt ten można wykorzystać przy kontroli kontraktów.

Propozycja autora

Propozycja PPK dla Javy - przykład definicji automatu.

```
public class AutomatonDemo {
    automaton {
        initial S0 : S1;
        accepting S1;
    }
    public void callMeOnce() {
        transient : S1;
    }
    public boolean wasAlreadyCalled() {
        return transient ? S1;
    }
}
```

Implementacja

- Zaproponowane przez autora mechanizmy PPK zostały osadzone w składni Javy za pomocą dodatkowych reguł gramatycznych.
- Gramatyka została zmodyfikowana w taki sposób aby powstały język był nadklasą Javy, przy zachowaniu tej samej semantyki odziedziczonych konstrukcji.

- Implementacja propozycji dostarczona jest w postaci translatora produkującego na wyjściu programy napisane w Javie. Translator został wygenerowany przy pomocy narzędzia ANTLR.
- Translator potrafi dodatkowo wykrywać błędy semantyczne, występujące w składni kontraktów.

Przykład schematu translacji

Fragment kodu przed translacją:

example.javab

```
protected int method () out (return * return >= 0){
  return 12;
}
```

I po translacji (na kolejnym slajdzie):

example.java

```
protected int method () {
  assert ((Example.this.automaton__Example ==
    Example.AutomatonEnumeration__Example.S3))
    ? true : true;
  int out_return__ = 0;
  try {
    return out_return__ = 12;
  } finally {
    final int out_return2 = out_return__;
    assert out_return2__ * out_return2__ >= 0;
    assert ((Example.this.automaton__Example ==
        Example.AutomatonEnumeration__Example.S3))
        ? true : true;
    }
}
```

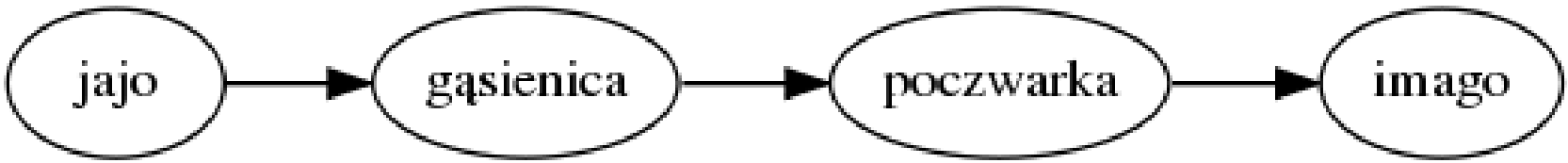
Część II

Dynamiczne Role

(W tej chwili powinna mijać 40 minuta prezentacji...)

Dynamiczna funkcjonalność

- Przykład 1 — **cykl rozwojowy motyla**.
- Przejście ze stadium gąsienicy do stadium imago nosi wymowną nazwę “zupełnego przeobrażenia”; Mimo że dany motyl potrafi w trakcie swojego życia zmienić zupełnie postać, cały czas pozostaje tym samym, w sensie identyfikacji, osobnikiem.



Dynamiczna funkcjonalność c.d.

- Przykład 2 — **system świadczący usługi**.
- Niech dany będzie pewien system. W systemie tym istnieją byty abstrakcyjnych usługobiorców, na rzecz których świadczone są usługi. Konkretna usługa wymaga konkretnego typu usługobiorcy. Abstrakcyjny usługobiorca może być klientem różnych usług, tj. mogą być z nim związane różne instancje konkretnego usługobiorcy.

Dynamiczna funkcjonalność c.d.

Problem — jak zamodelować komponent usługobiorcy?

- jako pojedynczą klasę?
- jako szereg klas związanych ze sobą pewnym wzorcem (Composite, Decorator)?
- w ogóle nie wyodrębniać komponentu?

Wady poszczególnych rozwiązań...

Dynamiczne Role - definicja

Dynamiczne role *są pewną koncepcją z pogranicza kontroli typów w językach programowania. Pojęcie to oznacza swego rodzaju mechanizm, który pozwala na dynamiczne przypisywanie danemu obiektowi wielu różnego rodzaju typów (i związanych z nimi wartości) jednocześnie. Obiekt ów nosi miano aktora, a jego reprezentacje, w postaci wartości o określonych typach, nazywane są rolami*

Można powiedzieć, że głównym celem użycia dynamicznych ról jest modelowanie bytów o dynamicznie zmieniającym się interfejsie:

Propozycja autora

- Funkcję ról pewnego aktora pełnią obiekty zwyczajnych klas języka Java, tzw. POJO.
- Funkcja aktora jest pełniona przez specjalną klasę `Actor`.
- Współdziałanie ról w obrębie danego aktora polega na synchronizacji odpowiednich pól.

```
Actor amphibian = ActorFactory.createActor();
Ship ship = amphibian.playRole( Ship.class );
ship.fleet( 10 );
Car car = amphibian.playRole( Car.class );
car.move( -15 );
assert ship.getPosition() == -5;
```

Implementacja

- Zaproponowany przez autora mechanizm Dynamicznych Ról został dostarczony w formie biblioteki.
- Składnia języka nie została zmieniona, jednakże jego semantyka w pewnym sensie tak.
- Biblioteka, w celu uzyskania żądanego działania, podczas ładowania klas ról tworzy odpowiadające im klasy agentów (proxy).
- "Pod maską" używana jest biblioteka Javassist, dzięki której nie było potrzeby bezpośredniej manipulacji bytecodem.

Wybrana bibliografia


A.V. Aho, R. Sethi, and J.D. Ullman. *Kompilatory: reguły, metody i narzędzia*. Wydawnictwa Naukowo-Techniczne, Warszawa, 2002.


J.E. Hopcroft, R. Motwani, and J.D. Ullman *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley, 2006.


J. Gosling, B. Joy, G. Steele, and G. Bracha. *Java (TM) Language Specification*. Addison Wesley Professional, 2005.


T. Parr. The Definitive ANTLR Reference: *Building Domain-specific Languages*. Pragmatic Bookshelf, 2007.

Pytania

Pytania?

Podziękowania

Dziękuję za uwagę