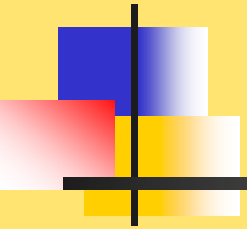
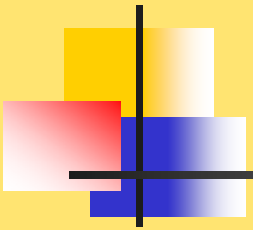


DESIGN BY CONTRACT (DBC) I ORACLE PL/SQL

Zlatko Sirotić
Istra informatički inženjering d.o.o.
Pula

There are two ways of constructing a software design. One way is to make it so simple that there are obviously no deficiencies. And the other way is to make it so complicated that there are no obvious deficiencies.
— Sir C.A.R. (Tony) Hoare

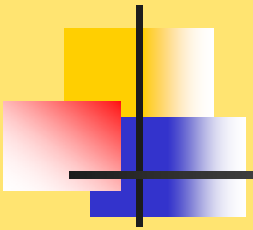
Debugging is twice as hard as writing the program, so if you write the program as cleverly as you can, by definition, you won’t be clever enough to debug it.
— Kernighan's Law; B.W.Kernighan



Teme

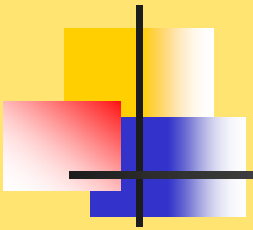
- apstraktni tipovi podataka (Abstract Data Types - ADT)
- Eiffel i Design By Contract (DBC)
- DBC i obrada iznimaka
- DBC i nasljeđivanje
- DBC i drugi programski jezici
- Oracle PL/SQL – osnovne OO(PL) osobine
- DBC I Oracle PL/SQL

Uvod



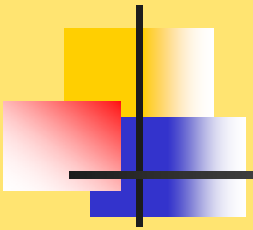
- 60-te i prva polovina 70-ih godina prošlog stoljeća predstavljaju razdoblje burnog razvoja teorije (i prakse) programskih jezika
- u tom razdoblju intenzivno su proučavane i **formalne (matematičko-logičke) metode za dokazivanje korektnosti programa**: R.W.Floyd, C.A.R.Hoare, E.W.Dijkstra, N.Wirth
- te metode nisu uhvatile dubljeg korijena u praksi programiranja; vjerojatni razlozi: spor (i skup) hardver, želja da softver što prije bude gotov, složenost tadašnjih formalnih metoda; no, daljnji razvoj formalnih metoda ipak nije stao
- Bertrand Meyer je prije dizajniranja jezika Eiffel sudjelovao u razvoju formalnih specifikacijskih jezika Z i M i na bazi tih iskustava i svojih originalnih ideja razvio je metodu **Design by contract** (DBC) i ugradio je u svoj programski jezik **Eiffel**

Apstraktni tipovi podataka (Abstract Data Types - ADT)



- **apstraktni tipovi podataka** (ADT) predstavljaju teorijsku (matematičku) osnovu za DBC metodu, pa i za cijeli Eiffel jezik
- ADT omogućavaju da specificiramo softverski sustav na precizan i kompletan (koliko je to potrebno) način bez prespecifikacije (overspecification), tj. bez ulaženja u detalje softverskog rješenja
- drugim riječima, ADT omogućavaju da precizno definiramo što želimo dobiti, a da ne moramo ulaziti u detalje kako to postići – to je tzv. **aplikativni**, a ne **imperativni** pristup
- kod objašnjavanja ADT-ova često se kao primjer uzima stog (stack); stog je objekt koji sadrži elemente (druge objekte) i puni se i prazni na LIFO način (last-in, first-out), što znači da će onaj element koji smo zadnji stavili na stog biti prvi koji ćemo uzeti

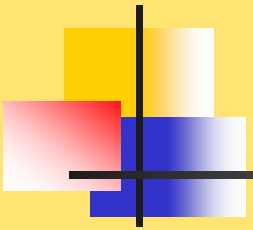
Apstraktni tipovi podataka (Abstract Data Types - ADT)



Neovisno od toga kako stog fizički realiziramo (npr. kao niz ili povezanu listu), stog uobičajeno ima sljedeće operacije:

- naredbu put kojom stavljamo neki element na vrh stoga
- naredbu remove kojom mičemo element sa vrha stoga (ako nije prazan)
- upit item kojim čitamo element koji je na vrhu stoga (ako nije prazan)
- upit empty kojim tražimo da li je stog prazan
- operaciju kreiranja make kojom kreiramo (inicijalno prazan) stog

Apstraktni tipovi podataka (Abstract Data Types - ADT)

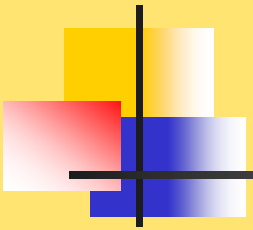


- ADT specifikacija se sastoji od četiri dijela – tipovi, funkcije, aksiomi, pretkondicije

- **TIPOVI** – tu se jednostavno navode svi tipovi koji se koriste u ADT specifikaciji; često se koristi tzv. formalni generički parametar, kao npr. u STACK [G], što znači da stog može sadržavati elemente bilo kojeg tipa, a taj tip označavamo sa G

- **FUNKCIJE –** tu se navode operacije (funkcije) koje se primjenjuju na instancu ADT-a; ne navodi se kako funkcija radi, već samo kako izgleda, tj. kakva je njena signatura
- primjer funkcije:
  put: STACK [G] X [G] → STACK [G]
  s lijeve strane strelice navode se tipovi argumenata, a s desne tip rezultata; npr. funkcija put kao argumente uzima stog i (neki) element, a kao rezultat vraća opet (drugi) stog

Apstraktni tipovi podataka (Abstract Data Types - ADT)



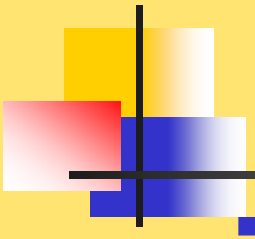
- funkcija koja ne može za vrijednost argumenta uzeti sve elemente iz skupa koji čini tip argumenta naziva se **parcijalnom funkcijom** (označava se prekriženom strelicom); npr. ne možemo pročitati element iz praznog stoga, pa je funkcija item ovako definirana
  item: STACK [**G**] **-/→** [**G**]

- **funkcije-kreatori** kreiraju instancu ADT-a; nemaju argumente, već imaju samo rezultat, a rezultat je ADT tipa, npr.
  make: → **STACK** [**G**] ili skraćeno **make: STACK** [**G**]

- **funkcije-upiti:** ADT tip javlja se samo kao argument, npr.
  item: STACK [**G**] **-/→** [**G**]

- **funkcije-naredbe:** ADT tip javlja i sa lijeve i sa desne strane strelice - naredbe zapravo mijenjaju instancu ADT-a, npr.
  put: STACK [**G**] **X** [**G**] → **STACK** [**G**]

Apstraktni tipovi podataka (Abstract Data Types - ADT)



- **AKSIOMI** – njima se implicitno definira na koji način se funkcije moraju ponašati, tj. kojim zahtjevima moraju udovoljiti, npr.

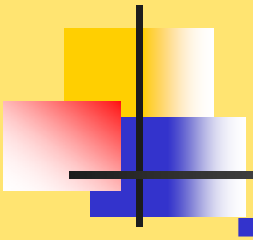
  za svaki x: G, s: STACK [G] vrijedi
  A1: item (put (s, x)) = x

  znači da kad stavimo (sa put) neki element x (tipa G), na stog s (tipa STACK [G]) i nakon toga pročitamo (sa item) novodobiveni stog, dobit ćemo kao rezultat element x (drugačije rečeno, ono što smo stavili na stog bit će na vrhu stoga)

- **PRETKONDICIJE** - služe za definiranje domene parcijalnih funkcija; za stog imamo sljedeće dvije pretkondicije (jer imamo dvije parcijalne funkcije)

  remove (s: STACK [G]) require not empty (s)
  item (s: STACK [G]) require not empty (s)

Apstraktni tipovi podataka (Abstract Data Types - ADT)

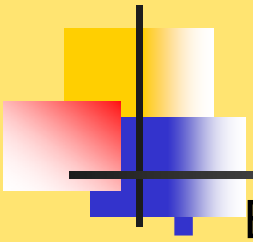


- **TIPOVI: STACK [G]**
- **FUNKCIJE:**
  put: STACK [G] X [G] → STACK [G]
  remove: STACK [G] -/→ STACK [G]
  item: STACK [G] -/→ [G]
  empty: STACK [G] → BOOLEAN
  make: STACK [G]

- **AKSIOMI: Za svaki x: G, s: STACK [G]**
  A1: item (put (s, x)) = x
  A2: remove (put (s, x)) = s
  A3: empty (make)
  A4: not empty (put (s, x))

- **PRETKONDICIJE:**
  remove (s: STACK [G]) require not empty (s)
  item (s: STACK [G]) require not empty (s)

Eiffel i Design By Contract (DBC)

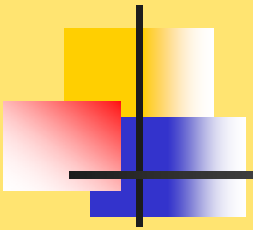


- Bertrand Meyer je dizajnirao Eiffel 1985. godine; Eiffel je od početka podržavao **višestruko nasljeđivanje**, **generičke klase**, **obradu iznimaka**, i (naravno) **Design by contract**; pretprošle godine (2005.) donesen je ECMA (367) standard, prošle godine i ISO standard

- DBC se zasniva na ideji da svaka metoda (procedura ili funkcija), uz "standardni" programski kod, treba imati još dva dodatna dijela - **pretkondiciju** (precondition) i **postkondiciju** (postcondition); klasa treba imati još jedan dodatni dio - **invarijantu** (invariant)

- **ugovor** (contract) se zasniva na tome da pozvana metoda "traži" od svog pozivatelja da zadovolji uvjete definirane u pretkondiciji plus uvjete definirane u invarijanti, a pozvana metoda se tada "obvezuje" da će na kraju zadovoljiti uvjete definirane u postkondiciji plus uvjete definirane u invarijanti

Eiffel i Design By Contract (DBC)

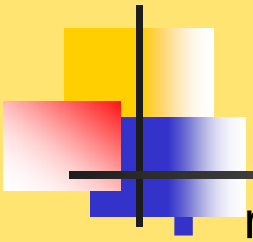


- pretkondicije, postkondicije i invarijante su **aplikativni** dijelovi programa (ne mogu mijenjati ponašanje programa), a ne **imperativni** (procedure i funkcije, tj. metode)

- DBC je suprotna od tzv. defanzivnog programiranja, koje zagovora da se u svim mogućim trenucima pokušava što više provjeriti

- Eiffel za specifikaciju DBC elemenata koristi ključne riječi **require** za označavanje pretkondicije, **ensure** za postkondicije i **invariant** za invarijante klasa (navedene naredbe su najvažnije za DBC podršku, ali Eiffel ih ima još)

Eiffel i Design By Contract (DBC)



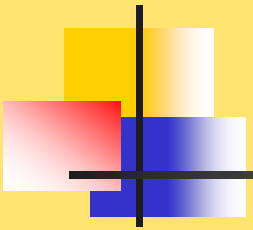
- može se postaviti pitanje utjecaja izvršenja pretkondicija, postkondicija i invarijanti na brzinu izvođenja programa; nažalost, utjecaj postoji, a naročito je “skupa” provjera invarijanti, koje se moraju izvršavati **i na početku i na kraju poziva svake metode** (zbog postojanja referenci)
- stoga se u Eiffel-u može odrediti **nekoliko stupnjeva provjere**; najslabija provjera (ali sa najmanjim negativnim utjecajem na brzinu izvođenja) je provjera samo pretkondicija (ta se provjera obično ostavlja i nakon završetka faze testiranja, tj. ostaje u produkcijskom kodu)
- pretkondicije, postkondicije i invarijante korisne su čak i kada nisu realizirane kao programski kod, nego samo kao komentar, jer **poboljšavaju dokumentiranost** izvornog koda
- Eiffel nema operatore univerzalnog i egzistencijalnog kvantifikatora iz predikatnog računa, pa je neke uvjete moguće izraziti ili samo kao komentar, ili pozivati funkcije iz “standardnog” (imperativnog) programskog koda

Eiffel i Design By Contract (DBC)

- metoda put prikazana je kroz tzv. **kratki oblik klase,** u kojem se ne prikazuje izvršni kod metoda, već samo njihovo sučelje (interface); vidimo da Eiffel omogućava da se u postkondicijama koristi riječ **old**, čime dobijemo staru vrijednost atributa**:**

```
-- dodaj x na vrh stoga
put (x: G) is
  require
    not_full: not full
  ensure
    not_empty: not empty
    added_to_top: item = x
    one_more_item: el_count = old el_count + 1
  end
```

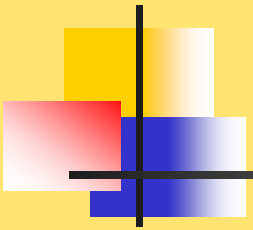
Eiffel i Design By Contract (DBC)



- slijede invarijante klase:

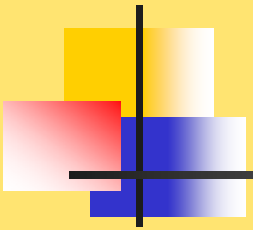
```
invariant
  count_non_negative: 0 <= el_count

  count_bounded: el_count <= capacity

  empty_if_no_elements: empty = (el_count = 0)
```

Eiffel i Design By Contract (DBC)



- kad bismo pogledali kompletan programski kod kratkog oblika klase i usporedili ga sa ADT specifikacijom, vidjeli bismo sljedeće:
- **pretkondicije** iz ADT specifikacije realiziraju se (u programskom kodu) kao **pretkondicije** odgovarajućih metoda
- **aksiomi** koji se odnose na funkcije-naredbe odnosno funkcije-kreatore realiziraju se kao **postkondicije** odgovarajućih metoda-procedura odnosno metoda-konstruktora
- **aksiomi** koji se odnose isključivo na funkcije-upite (ali, takvih aksioma ovdje nema) realiziraju se kao **postkondicije** odgovarajućih metoda-funkcija ili kao **klauzule u invarijantama** klase

Eiffel i Design By Contract (DBC)



- iz kratkog oblika klase mogli bismo vidjeti i da aksiom

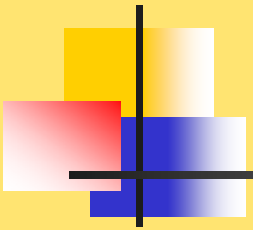
  A2: remove (put (s, x)) = s

  nije nigdje realiziran u programskom kodu

- razlog je taj što Eiffel podržava samo algebru sudova (proširenu sa konceptom **old**) a ne i predikatni račun

- no, ipak se moglo realizirati taj aksiom pomoću odgovarajuće (programske) funkcije, ali uz opasnost uvođenja imperativnih elemenata (funkcija, procedura) u aplikativni svijet pretkondicija, postkondicija i invarijanti

Eiffel i Design By Contract (DBC)

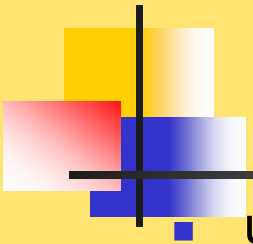


- također, mogli bismo primijetiti da u programskom kodu postoje neke pretkondicije, neke postkondicije ali i neke invarijante (u konkretnom slučaju – sve tri) koje nemaju svoj "izvor" u ADT specifikaciji

- one su nastale na temelju konkretne implementacije programskog rješenja (u konkretnom slučaju stog je realiziran pomoću niza), tj. nisu definirane u ADT specifikaciji

- invarijante koje su nastale na temelju implementacije (u našem slučaju sve tri invarijante) nazivaju se **implementacijskim invarijantama**

DBC i obrada iznimaka



- u toku izvođenja programa može doći do događaja koji zovemo **iznimka** (exception); ako do iznimke dođe u pozvanoj metodi i iznimka nije u njoj obrađena, tada poziv završi sa neuspjehom, što dovodi do iznimke u metodi koja ju je pozvala

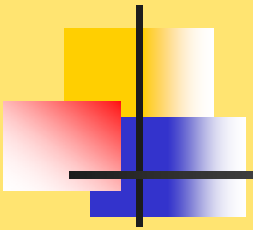
- Eiffel-ov model obrade iznimaka:

  - **pokušati ponovno** (koristeći alternativnu strategiju): moramo u **rescue** dijelu (dijelu za obradu iznimaka) metode koristiti naredbu **retry**, kojom vraćamo izvršavanje na početak metode

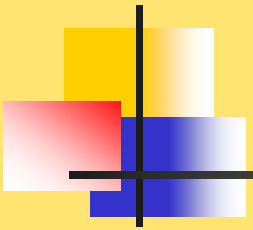
  - "organizirana panika": vratiti stanje klase tako da vrijede invarijante klase i onda proslijediti grešku dalje

  - **"lažni alarm"**: ponekad je moguće nastaviti bez obzira na grešku

DBC i obrada iznimaka



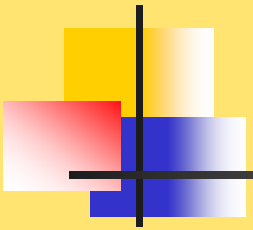
- u Eiffel-u vrijedi - **pozvana metoda ne može obraditi iznimku koja se desila zbog nezadovoljavanja pretkondicije**
- za iznimku koja se desila u pretkondiciji "krivac" je metoda-pozivatelj
- za iznimku koja se desila u postkondiciji "krivac" je metoda-izvršitelj



DBC i nasljeđivanje

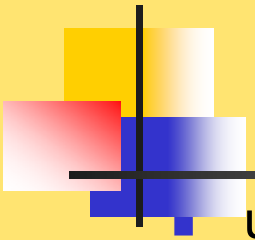
- nasljeđivanje je jedna od tri osnovne operacije računa klasa - ostale dvije su agregacija i generičnost
- metoda iz nadklase može se u podklasi **nadjačati** (drugom) metodom
- Eiffel omogućava i da se atribut nadjača atributom, koji mora biti ili istog tipa ili mora biti podtipa (to je tzv. kovarijanca - covariance) ili da se funkcija bez parametara nadjača atributom (isto mora vrijediti kovarijanca)

DBC i nasljeđivanje



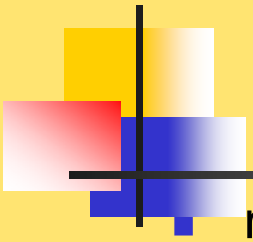
- nadjačane metode ne mogu imati bilo kakve pretkondicije ili postkondicije
- **nadjačana metoda mora imati jednaku ili slabiju pretkondiciju,** tj. može zahtijevati od metode koja ju je pozvala ili isto ono što zahtijeva metoda nadklase, ili manje od toga
- **nadjačana metoda mora imati jednaku ili jaču postkondiciju,** tj. mora osigurati barem ono što je osiguravala metoda nadklase, ili više od toga
- zato Eiffel za definiranje pretkondicije u nadjačanoj metodi koristi oblik **require else** (umjesto **require**), a za definiranje postkondicije u nadjačanoj metodi koristi **ensure then** (umjesto **ensure**)

DBC izvan Eiffel-a



- u jednoj staroj verziji specifikacije jezika Java (1994. godine), zadnjoj koju je J.Gosling (glavni autor jezika Java) pisao sam, postojala je podrška za pretkondicije, postkondicije i invarijante; zbog potrebe za što bržim izlaskom na tržište, Gosling je odlučio da to izbaci iz specifikacije jezika Java
- na početku Java nije imala čak niti naredbu **assert** (sličnu Eiffel naredbi **check**) dok je C++ jezik imao tu naredbu; Java je u verziji JDK 1.4 ipak dobila naredbu assert
- postoje različiti pristupi za realizaciju DBC-a u Javi; većina se starijih pristupa (prije uvođenja naredbe assert) temelji na pretprocesorskim tehnikama, tj. uvodi se "proširenje" jezika Java (najčešće kao komentar), a izvorni programi pisani u "proširenom" jeziku prvo prolaze kroz pretprocesor, koji onda taj izvorni kod pretvori u "običan" Java izvorni kod; slične pretprocesorske tehnike koristili su i prvi C++ "prevodioci", zapravo pretprocesori koji su C++ kod pretvarali u C kod, a dalje se koristio C prevodilac

DBC izvan Eiffel-a



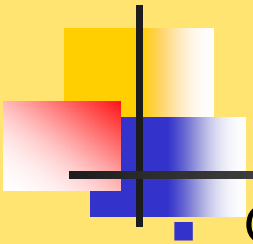
- nakon uvođenja assert naredbe, pojavili su se i drugačiji pristupi (npr. Kopi Java compiler), koji su omogućili da se izvorni Java kod (koji uključuje podršku za DBC) direktno prevede u byte kod

- u verziji 5.0 uvedene su u jezik Java anotacije (**annotations**); one omogućavaju uvođenje u program dodatnih informacija koje su testirane i verificirane od strane prevodioca; te nove mogućnosti iskoristio je jedan od najnovijih projekata na području uvođenja DBC metode u jezik Java - Contract4J (Design by Contract for Java)

- i jezik UML (danas de-facto standard za modeliranje OO sustava) podržava DBC od 1999., zahvaljujući OCL (Object Constraint Language) formalnom jeziku

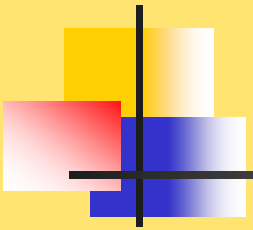
- nažalost, Oracle PL/SQL nema za sada ništa slično Java naredbi assert ili Java anotacijama

Oracle PL/SQL – osnovne OO(PL) osobine



- Oracle je u sklopu baze 6.0 napravio novi programski jezik - PL/SQL (Procedural Language extensions to the Structured Query Language), kao proceduralnu nadopunu (deklarativnog) jezika SQL
- Oracle je napravio PL/SQL na temelju programskog jezika ADA 83
- Oracle je 1997. objavio 8.0 verziju baze i nazvao ju objektno-relacijskom; ta verzija baze (niti verzija 8i) nije imala neke važne objektne mogućnosti kao što su npr. nasljeđivanje, nadjačavanje metoda, polimorfizam
- njih je Oracle uveo 2001. godine u bazi 9i

Oracle PL/SQL – osnovne OO(PL) osobine



- PL/SQL za sada ne podržava:
- **privatne metode i atribute**
- **reference na tranzijentne objekte**, u smislu da ako klasa K1 ima atribut k2 koji se odnosi na klasu K2, taj atribut može (ako to programer želi) sadržavati referencu na objekt klase K2, a ne cijeli (pod)objekt; zbog nepostojanja referenci u PL/SQL-u se invarijante ne bi morale pozvati na početku, već samo na kraju metoda (ali pozivamo ih i na početku - da kod bude sličan onome u Eiffel-u)
- **višestruko nasljeđivanje** (no, to ne podržava niti Java)
- **generičke klase** (no, i Java ih podržava tek od verzije 5.0)

Oracle PL/SQL – osnovne OO(PL) osobine

- PL/SQL klasa se (barem za sada) može definirati samo kao objekt baze; ima dva dijela - specifikaciju i tijelo; prikazana je specifikacija klase osoba koja ima dva atributa i tri metode (jedna je konstruktor):

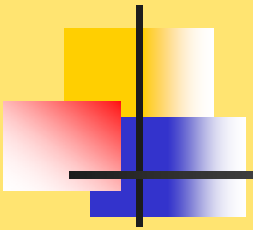
```
CREATE OR REPLACE TYPE osoba AS OBJECT (
  ime VARCHAR2 (20),
  tezina_u_kg NUMBER,
  CONSTRUCTOR FUNCTION osoba
    (p_ime VARCHAR2, p_tezina_u_kg NUMBER)
     RETURN SELF AS RESULT,
  MEMBER PROCEDURE prikazi_podatke,
  MEMBER FUNCTION tezina_u_funtama RETURN NUMBER
) NOT FINAL;
```

Oracle PL/SQL – osnovne OO(PL) osobine

- prethodna klasa nije finalna, pa možemo napraviti njenu podklasu; npr. u sljedećoj podklasi gurman uvodimo novi atribut, mijenjamo konstruktor, nadjačavamo metodu prikazi_podatke i uvodimo novu (finalnu) metodu jedi_puno_i_dobro :

```
CREATE OR REPLACE TYPE gurman UNDER osoba (
  omiljena_hrana VARCHAR2(20),
  CONSTRUCTOR FUNCTION gurman (
    p_ime VARCHAR2, p_tezina_u_kg NUMBER,
    p_omiljena_hrana VARCHAR2)
   RETURN SELF AS RESULT,
  OVERRIDING MEMBER PROCEDURE prikazi_podatke,
  FINAL MEMBER PROCEDURE jedi_puno_i_dobro
) NOT FINAL;
```

DBC i Oracle PL/SQL

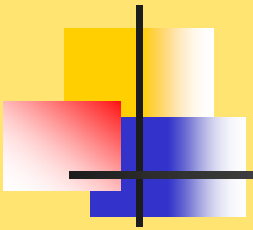


- jednu jednostavnu (i primitivnu) realizaciju DBC metode u jeziku PL/SQL prikazat ćemo također na primjeru klase stack

- prvo ćemo morati na bazi kreirati tip array_t (niz) koji će nam kasnije trebati za deklaraciju atributa implement u klasi stack:

```
CREATE OR REPLACE TYPE array_t AS TABLE OF INTEGER;
```

DBC i Oracle PL/SQL



- napravit ćemo i pomoćni paket dbc koji će služiti za prikaz poruke o grešci i za definiranje razine provjere:

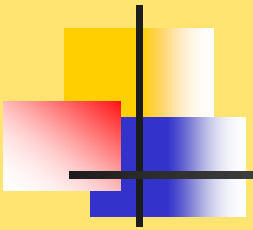
```
c_no_check CONSTANT INTEGER                    := 0;

c_check_preconditions CONSTANT INTEGER         := 1;

c_check_pre_postconditions CONSTANT INTEGER    := 2;

c_check_pre_post_invariants CONSTANT INTEGER := 3;
```

DBC i Oracle PL/SQL

- pomoćni paket dbc (samo specifikacija, bez konstanti):

```
CREATE OR REPLACE PACKAGE dbc AS
  FUNCTION check_preconditions RETURN BOOLEAN;

  FUNCTION check_pre_postconditions RETURN BOOLEAN;

  FUNCTION check_pre_post_invariants RETURN BOOLEAN;

  PROCEDURE set_level (p_level INTEGER);

  PROCEDURE display_error (p_error VARCHAR2);
  …
END;
```

DBC i Oracle PL/SQL

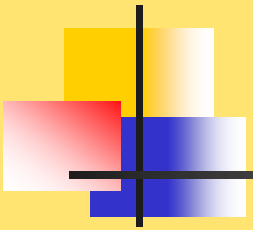


- sada možemo kreirati specifikaciju i tijelo klase stack
- budući da PL/SQL nema privatnih atributa i metoda, svi atributi i metode su javni, pa i implementacijski atribut implement (on je niz)
- osim toga, budući da PL/SQL ne podržava generičke klase, ovdje stog neće moći imati elemente bilo kojeg tipa, nego samo elemente određenog tipa - npr. INTEGER
- paket će sadržavati (i) tri funkcije koje će implementirati tri invarijante klase, te proceduru koja će (ako je razina provjere postavljena na 3) provjeravati da li sve te funkcije vraćaju TRUE

DBC i Oracle PL/SQL

```
MEMBER FUNCTION count_non_negative RETURN BOOLEAN…

MEMBER FUNCTION count_bounded RETURN BOOLEAN…

MEMBER FUNCTION empty_if_no_elements RETURN BOOLEAN…

MEMBER PROCEDURE check_invariants IS BEGIN
  IF NOT dbc.check_pre_post_invariants THEN RETURN;…
  IF NOT count_non_negative THEN
    dbc.display_error ('INVARIANT count_non_…’);…
  IF NOT count_bounded THEN
    dbc.display_error ('INVARIANT count_bounded');…
  IF NOT empty_if_no_elements THEN
    dbc.display_error ('INVARIANT empty_if_…');…
END;
```

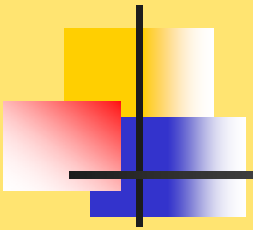
DBC i Oracle PL/SQL



- U nastavku prikazujemo primjer jedne procedure - procedura put
- nažalost, PL/SQL nema Eiffel-ov konstrukt **old**, pa jedna klauzula iz Eiffel postkondicije ovdje nije realizirana

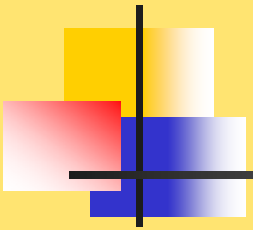
```
put (x: G) is
  require
    not_full: not full
  ensure
    not_empty: not empty
    added_to_top: item = x
    one_more_item: el_count = old el_count + 1
  end
```

DBC i Oracle PL/SQL



```
MEMBER PROCEDURE put (x INTEGER) IS BEGIN
  IF dbc.check_preconditions AND full THEN
    dbc.display_error('put - PRE');
  END IF; -- pretkondicija
  check_invariants; -- ulazni poziv invarijanti
  el_count := el_count + 1; -- početak glavnog koda
  implement(el_count) := x; -- kraj glavnog koda
  IF dbc.check_pre_postconditions
    AND (empty OR item <> x) -- nema Eiffel-ov old !
  THEN
    dbc.display_error ('put - POST');
  END IF; -- postkondicija
  check_invariants; -- izlazni poziv invarijanti
END;
```

Zaključak



- iako je DBC metoda stara već 20-tak godina, izgleda da je tek zadnjih godina počela privlačiti veću pažnju - npr. za Javu se razvilo dosta alata za podršku DBC metode

- DBC može povećati kvalitetu programskog koda, što je nužan uvjet za višestruku iskoristivost koda, čime se povećava produktivnost u razvoju softvera (sama primjena OOPL jezika nije donijela obećanu višestruku iskoristivost koda); DBC metoda može značajno doprinijeti povećanju kvalitete čak i kad ju primijenimo samo kao metodu za analizu i za dokumentiranje

- bilo bi odlično kada bi dizajneri programskih jezika uveli direktnu podršku za DBC u svoje jezike; međutim, ne moramo čekati dizajnere jezika - u bilo kojem programskom jeziku moguće je realizirati barem pretkondicije i postkondicije, a u OOPL jezicima mogu se realizirati i invarijante klase

The most likely way for the world to be destroyed,
most experts agree, is by accident.
That's where we come in; we're computer professionals.
We cause accidents.

- Nathaniel Borenstein