Objektovo orientovaný návrh

Obsah:
Objektovo orientovaný vývoj softvéru
Abstraktné dátové typy
Design by Contract
Ako nájsť triedy + pár ďalších poznámok
Diagramy – najprv diagram tried

Objektovo orientovaný vývoj softvéru

chceme odpovedať na otázku: čo to znamená a ako ho robiť ?

Kde sme skončili minule ?

Robíme systém z modulov
**** slajd1 ****

Def: Objektovo orientovaný vývoj softvéru je vývoj softvérového produktu ako štruktúrovaného systému tried, t.j. (potenciálne parciálnych) implementácií abstraktných dátových typov.

Def: ADT je množina (matematických) objektov popísaná:
- **zoznamom funkcií aplikovateľných na všetky tieto objekty (funkcie môžu byť aj nulárne)**
- **vlastnosťami týchto funkcií**

(príklad stack)

FUNCTIONS
- **put: STACK [G] x G -> STACK [G]**
- **remove: STACK [G] +> STACK [G]**
- **item: STACK [G] +> G**
- **empty: STACK [G] -> BOOLEAN**
- **new: STACK [G]**

(o popise vlastností funkcií neskôr)

iný príklad môže byť: objednávka, faktúra, cestovný príkaz, používateľ, ...

Zatiaľ tieto funkcie môže mať hociaká štruktúra, napr. fronta (FIFO).

Chceme formálne popísať vlastnosti operácií. Nemožno ale explicitne: put = ... – pretože existuje mnoho implementácií (array up, array down, linked list (+ ďalšie, napr. dva stacky v poli ...)) - ktorú vybrať ?

AXIOMS
x: G, s: STACK [G]
- **item (put (s, x)) = x**
- **remove (put (s, x)) = s**
- **empty (new) = true**
- **not empty (put (s, x))**

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

Implicitný popis – netvrdíme, že zoznam funkcií je úplný, ani že funkcie nemajú iné vlastnosti (keď ich implementujeme, iste budú mať). Kľúčové sme ale vystihli. (Pojem úplnosti...)

Sila je v tom, že zachytíme podstatné vlastnosti bez prílišnej špecifikácie.

Toto je popis rozhraní modulov na abstraktnej úrovni.

Triedy potom urobíme tak, že vyberieme reprezentáciu a implementujeme features, ktoré budú spĺňať stanovené podmienky.

Súvisiace zmena – z aplikatívneho na imperatívny pohľad - namiesto operácií príkazy, ktoré menia objekt.

K definícii:
- **zaujímajú nás softvérové implementácie (nie samotné ADT) – sú to triedy**
- **implementácie sú potenciálne parciálne**
- **štruktúrovaný systém – relácie „klient" a „dedenie"**

Základné pojmy a mechanizmy (nástroje) OOSC (a teda aj OOD) sú: (Návrhár by ich mal poznať a používať. -- OO jazyk sám osebe nenúti programátora využívať výhody OO prístupu, len to umožňuje.)

P1: class (trieda) –implementovaný ADT (čiastočne alebo úplne). Má črty: atribúty a operácie (procedúry a funkcie). Abstraktné triedy (nie úplne definované).
(z pohľadu klienta hovoríme o operáciách – príkazoch a dotazoch, z pohľ. impl. o metódach – procedúrach a funkciách)

STACK
- **put, remove, item, empty (+ daj slajd!)**

triedy ako moduly: triedy sú jediné moduly (administratívne sa triedy môžu zoskupovať do zväzkov - clusters)
(niektoré jazyky majú vyššie celky – packages, units)

triedy ako typy: každý dátový typ je založený na triede

P2: vyvolanie operácie (feature call) je primárnym výpočtovým mechanizmom (client, supplier) (zaslanie správy) (čisté OO prostredie nemá šantenie s atribútmi)

s.put (100);

P3: skrývanie informácií – autor triedy má mať možnosť určiť, či je daná črta viditeľná všetkým, žiadnemu alebo vybraným klientom.

Na slajde: template <class G> class Stack

(P2,3 – vedie k minimálnym a explicitným interfejsom)

P4: dedičnosť – jednoduchá, viacnásobná (name clashes, repeated inheritance)

(účet: bežný, termínovaný,
geometrický objekt: kružnica, obdĺžnik, ...)

interface vs. implementation inheritance

WINDOW: GENERAL_WINDOW, MOTIF_WINDOW
ARRAYED_STACK: STACK, ARRAY

P5: polymorfizmus – entita môže obsahovať objekty viacerých typov (kompatibilných)

P6: dynamické viazanie – (geometricky_objekt g; g.plocha ();) – má zavolať operáciu aktuálneho objektu

P7: generické triedy (LIST [G] (template <class G> class List), neobmedzene/obmedzene generické)

P8: assertions (predpoklady) –
- **preconditions, postconditions, invarianty**

Načo sú: dosahovať spoľahlivosť, pri dokumentácii, pomoc pri testovaní a ladení. (Bude k tomu viac - DbC)

P9: exception handling (ošetrenie výnimiek)
- prostriedok na zotavenie z neočakávanej nenormálnej situácie (hw porucha, delenie nulou, chyba v sw)

Poznámky:
- **pri OO naberá na význame otázka správy pamäti**
- **otázka perzistentnosti**
- **seamlessness**

Konkrétne OOD v prvej etape (architektonický návrh) znamená popísanie sústavy abstraktných dátových typov – ide vlastne o návrh interfejsov medzi modulmi.

Design by Contract

(Stále sa venujeme otázke „Ako popísať rozhrania modulov ?“.)
(Teraz „trochu viac programátorsky“.)

Čo je to správnosť programu / správny program ?

x := y + 1
môže byť správne vzhľadom k špecifikácii:
- **zabezpeč aby x a y boli rôzne**

ale nie k špecifikácii:
- **zabezpeč aby x > 0**

správnosť je vždy relatívna – vzhľadom k špecifikácii

tvrdenie o správnosti kódu A má potom tvar: {P} A {Q}
P, Q sú predpoklady (assertions)
{False} A {Q}, {P} A {True} vždy platí.

Čo z toho vyplýva pre návrh:
- **pre-/post-conditions k operáciám (vo forme assertions)**
- **keď sa operácia napíše s pre-/post-cond., je to KONTRAKT (medzi klientom a supplierom) „keď ma zavoláš s PRE, ja sa zaväzujem, že sa vrátim s POST“.**
- **assertions = (label) + boolean expressions with few extensions („old“)**

****slajd – stack1****

Čo to prináša: jednoznačnosť, a tým pádom napr. pre suppliera – jednoduchšie spracovanie – nemusí testovať - ak je stack empty, môže si spraviť čo chce (zhodiť systém)
(čitateľnosť návrhu, jednoduchosť/čitateľnosť implementácie)

Nielenže nemusí, ale nesmie testovať na pre-condition!!
****slajd: sqrt****

Škodí to: klesá jednoduchosť.
Zložitosť je hlavným nepriateľom spoľahlivosti. Tisíce funkcií so (zbytočnými) kontrolami ...
(testuje to softvér – aspoň v debug móde)

****komplexný príklad STACK2****
Result = (count=capacity)
porovnaj s abs (Result^2 – x) <= tolerance
MFC tiež má asserty

Axiómy v ADT ===> pre/post cond. / inv.
(expressive power – dá sa riešiť komentárom, volaním funkcie) + niektoré veci v post-c sú implementačne závislé. *Pri vytváraní dokumentácie sa tieto vypustia.*

Pri návrhu treba zistiť, čo je nevyhnutné pre správne vykonanie funkcie a rozhodnúť sa, či danú vec dať alebo nedať do precondition. Potom sa podľa toho zariadiť.

- **Nie pre rozhranie človek-sw, sw-cudzí sw.**
- **Nie ako riadiaca štruktúra (porušenie assertions je vždy príznakom bugu).**

Design: tolerant vs. demanding – to je otázka.

Hlavný problém je: ako má supplier ošetriť chybu ?

empty stack: error message ?
klient vie: či to je naozaj chyba a ak áno, čo s ňou – zopakovať, postúpiť vyššie, vypísať chybovú správu.

Niekedy nemusí byť praktické dávať isté veci do preconditions, napr: riešenie sústavy lineárnych rovníc s pre: matica nie je singulárna (ale zistenie trvá zhruba toľko, čo vyriešenie).

Class invariant – pozri slajd
- **consistent_balance: deposits_list.total – withdrawals_list.total = balance**
- **musí platiť medzi volaniami funkcií**
- **(v podstate sa pridávajú do pre- aj post-conditions)**

check (assert) – niečo, čo má platiť, ale nie je to zrejmé z textu (príklad na slajde)

Použitie DbC:
- **pomoc pri písaní správneho softvéru (určenie zodpovednosti: jednoduchosť + dajú sa lepšie robiť úvahy o správnosti)**
- **pomoc pri dokumentovaní (extrakty)**
- **pomoc pri testovaní a ladení (runtime monitoring)**

- **čo sa týka zapnutia/vypnutia pri behu, závisí to od:**
  - **nakoľko veríme systému**
  - **nakoľko dôležitá je efektívnosť**
  - **nakoľko kritická je nezdetekovaná chyba**
- **pri vývoji áno**
- **pri behu sa to môže vypnúť, aj keď:**
  - **Hoare: ako ten, čo si dá dolu vestu, keď ide na more**
  - **50% strata pri preconditions**
  - **už keď sa dá von no-check, pribaliť aj checked build**
  - **má to zmysel aj v run-time, napr. kvôli chybám v HW a neodhaleným bugom**

Exceptions

Rozlíšenie úspešných a neúspešných volaní operácie.

(Keď sa metóde nepodarí naplniť kontrakt, musí to dať najavo - inak ako normálnym návratom.)

Možnosti ošetrenia:
- **retry**
- **failure (organized panic)**

At this moment: v pozadí ADT, pojmy OO, invarianty.

How to find the classes

Design Classes
- **analytické triedy**
- **čítanie kníh, design patterns**
- **rozmýšľanie :)**

Ideálna trieda...
- **má jasne asociovaný ADT**
- **podstatné meno / príd.meno**
- **má objekty (aspoň 1)**
- **má queries a commands (alebo aspoň funkcie produkujúce iné objekty (x+y))**

- **chyby:**
  - **My class performs ... (táto trieda tlačí výsledky, analyzuje vstup, ...) (ak aj trieda je OK, meno je zlé)**
  - **jedno-funkčné triedy**
  - **nerobiť dedenie priskoro (najskôr nájsť a pochopiť triedy)**
  - **spojené viaceré abstrakcie (nízka kohézia NSText)**

Návrh interfejsu triedy ďalej:

- **command-query separation – funkcie nemajú produkovať abstraktné side efekty (t.j. meniť hodnoty zistiteľné cez non-secret queries) (nie ako v C – getc, vracanie stavu)**
  - **príklad s random number (seq: make, next, value)**
- **rutiny s málo argumentami!**
  - **my_document.print (printer_name, paper_size, color_or_not, postscript_level, print_resolution)**
  - **operand – objekt, na ktorom rutina pracuje**
  - **option – spôsob práce**
  - **operandy sa nezvyknú podstatne meniť a nedá sa určiť ich default hodnota**
  - **argumenty rutiny by mali byť len operandy (options sa majú nastaviť a zisťovať zvlášť – možná je optimalizácia print_with_size_and_resolution)**
- **veľkosť samotnej triedy nie je problém, musí byť ale kohézna. (+ črty musia byť relevantné k danej abstrakcii, kompatibilné s ostatnými, neduplikované, držať invariant)**
- **triedy so stavom (LIST s vnútorným stavom zobrazujúcim kurzor)**

- **Dobre trafiť dedenie**
  - **CAR_OWNER (Is-a rule: nerob dedenie, pokiaľ nevieš zdôvodniť, že A „is-a“ B)**
  - **Výber medzi klientom a dedením**
    - **every sw engineer is an engineer**
    - **in every sw engineer there is an engineer**
    - **every sw engineer has an „engineer“ component**
  - **Rule of Change – nedaj dedenie, keď sa objekty vo vzťahu môžu vymieňať**
  - **Polymorphism Rule – daj dedenie, keď entity všeobecnejšieho typu majú ukazovať na objekty (aj) konkrétnejšieho typu**

- **taxomania – zbytočné robenie podtried (man – male, female – keď to netreba)**
- **typy dedenia**
  - **subtype**
  - **implementation inheritance (arrayed_stack: from stack, array)**