Erori frecvente în modelarea POO

7 aprilie 2008

Programarea orientată pe obiecte este o paradigmă de programare care foloseşte obiecte şi interacţiunile lor pentru conceptualizarea aplicaţiilor pentru calculator. Folosind tehnici precum polimorfismul, moştenirea, încapsularea şi modularitatea, POO a deschis nenumărate porţi pentru dezvoltarea software - a condus către programe mai structurate, mai uşor de înţeles şi mai complexe. Odată cu această paradigmă au apărut şabloanele de design (design patterns) care au ajutat foarte mult înţelegerea unor fenomene şi găsirea unor soluţii pentru problemele complexe mai des întîlnite în viaţa reală. Pe de altă parte această sofisticare a împins programatorii către greşeli de programare mult mai complexe, pe măsura noilor nevoi software.

Studierea acestor greşeli este foarte importantă pentru înţelegerea conceptelor programării pe obiecte - foarte multe greşeli sunt întîlnite la programatorii începători, dar poate cea mai gravă eroare se găseşte la programatorii avansaţi şi mai ales la designerii de sisteme software complexe, şi anume:

1 Overdesign-ul sau modelarea excesivă a detaliilor

Overdesign-ul este supraîncărcarea designului software cu prea multe detalii. Controlul fiecărui detaliu din faza de design este un ideal care de cele mai multe ori creează probleme. De ce?

În primul rînd pentru că foarte des, problemele puse în lumea reală afectează detaliile din design. Foarte des însă, schimbarea design-ului e mult mai costisitoare decît simpla schimbare a codului scris de programator.

Supraîncărcarea cu detalii va duce la concepte mult prea complexe şi vor exista întotdeauna probleme de comunicare cu implementatorii.

Rezolvarea problemelor din stadiul de design este, cu alte cuvinte, contraproductivă. Aşa cum rolul design-pattern-urilor nu este de a crea bucăţi de cod gata de aplicat, rolul design-ului nu este de a crea soluţii ci de a crea cadrul pe care se vor construi soluţiile.

Overdesign-ul este cea mai costisitoare greşeală din programarea orientată pe obiecte. Spre deosebire de celelalte greşeli care sunt cel mai des locale şi deci uşor de modificat, overdesign-ul afectează întregi proiecte, făcînd imposibilă portarea codului rezultat, refolosirea componentelor, CHIAR dacă iniţial, componentele

au fost gîndite pentru refolosire. Overdesign-ul împinge către cuplarea prea puternică între componente făcîndu-le interdependente.

Cine face această greşeală? De obicei analiştii care pierd contactul cu realitatea dar vor să se simtă implicaţi puternic în proiect, sau managerii de proiect care doresc să controleze totul pînă în cel mai mic detaliu. Cel mai des, cei implicaţi generează complexitate suplimentară din cauza incapacităţii de a abstractiza problemele.

Ce soluţie alternativă există? Cel mai probabil simplificarea design-ului, şi crearea de module independente în loc de module interdependente.

2 Abuzul de şabloane de design

Din aceeaşi categorie, dar un pic nuanţată problema: abuzul de design patterns poate afecta puternic calitatea codului şi eficienţa sa. Problema aceasta provine din faptul că problemele nu sunt identificate corect iar soluţiile, deşi corecte în sine, nu fac altceva decît să producă supraîncărcarea codului aplicaţiei.

Exemplu folosirea şablonului Singleton poate fi foarte utilă într-o aplicaţie complexă pentru protejarea unei conexiuni cu baza de date, dar la un moment dat este nevoie să se porteze aplicaţia pe un alt sistem - tot codul însă se va baza pe acel obiect Singleton care probabil va fi adus ca dependenţă suplimentară. Rezultatul: codul va trebui modificat, şi vor trebui menţinute două versiuni diferite a aceluiaşi soft.

Cine face această greşeală? De obicei programatorii care tocmai au învăţat design patterns fără a înţelege problemele pe care şabloanele le vor rezolva, şi mai ales problemele pe care şabloanele le vor introduce. Cel mai des, aplicarea unui design pattern va duce la explicaţii gen: „s-a folosit pattern-ul X" în loc de explicarea detaliilor de implementare. Şi, probabil, detaliile de implementare vor rămîne un mister şi pentru programator care va aplica pattern-ul într-un anumit fel pentru că „aşa se face", nefiind capabil să-şi argumenteze, explice şi în cele din urmă să menţină propriul cod.

Cum se poate rezolva această problemă? Cel mai des, rezolvarea vine odată cu experienţa - un design pattern trebuie înţeles înainte de a fi aplicat. Nu are nici un sens ca un programator să penduleze între „Abstract Factory", „Factory Method" sau „Builder", cît timp nici un şablon nu se potriveşte 100% pe problema de implementat. Adesea aplicarea unui design pattern în aceste cazuri face codul mai greu de înţeles şi de folosit din exterior, doar pentru că rigorile şablonului nu permit anumite scurtături care să uşureze utilizarea. Soluţia: întotdeauna rezolvarea problemei e mai importantă decît aplicarea unui pattern.

3 Probleme privind conceptele de bază OOP

Aceasta e probabil una din cele mai frecvente greşeli din programare, şi afectează programatorii indiferent de limbajul folosit. Există nenumărate feluri

în care se manifestă; poate cele mai importante ar fi:

3.1 Copy-Paste programming

Foarte des se întîlneşte aceeaşi secvenţă de cod este întîlnită în mai multe locaţii în program. Cel mai des, secvenţa care trebuie copiată cu copy-paste ar trebui să fie făcută metodă a clasei pe care o afectează sau a clasei care o apelează, în funcţie de felul în care acţionează. Cel mai des, copy-paste programming încearcă să compenseze neînţelegerea ierarhiei de obiecte a aplicaţiei.

C&P reprezintă o rezolvare foarte simplă a problemelor locale, însă devine o problemă cînd vine vorba de întreţinerea codului - mai ales că foarte multe secvenţe Copy-Paste sunt mascate prin inserturi şi modificări locale specifice locului din care sunt apelate

3.2 Marele switch / if-urile imbricate

Foarte des programatorii care nu au încredere în ierarhia de obiecte pe care o folosesc generează cod de forma:

```
if (x=="a")
  doSomethingForA();
else if (x=="b")
  doSomethingForB();
else ...
else doSomethingForZAndAnythingElse();
```

sau, un switch enorm în care codul în mare parte este similar, precum:

```
class A {
public:
  virtual void method();
};

class B : public A {
public:
  void method();
}

class C : public A {
public:
  void method();
}

switch (x) {
   case 0: {
     C* ptr = new C;
     ptr->method();
```

```
        doSomething();
        ptr->method();
        delete ptr;
        break;
      }
      case 1: {
        B* ptr = new B;
        ptr->method();
        doSomething();
        ptr->method();
        delete ptr;
        break;
      }
      default: {
        A* ptr = new A;
        ptr->method();
        doSomething();
        ptr->method();
        delete ptr;
      }
    }
```

Codul de mai sus se poate rescrie foarte bine prin:

```
A* ptr;
switch (x) {
  case 0: ptr = new C; break;
  case 1: ptr = new B; break;
  default: ptr = new A;
}
ptr->method();
doSomething();
ptr->method();
delete ptr;
```

Ignorarea ideii de moştenire şi derivare este cea care generează aceste efecte. De multe ori, aparentele diferenţe din metoda 'doSomething' de pe fiecare caz în parte ar putea crea probleme - ele ar putea fi tratate

3.3 Cast-uri inutile şi abuzul RTTI

Uneori, ierarhiile mari de obiecte se bazează pe crearea unei clase 'Object' pe care se construieşte întreaga ierarhie. Acest lucru permite uniformizarea accesului la obiecte, şi e foarte util pentru procedee precum message-passing, signal-slot. De obicei, cînd un astfel de design este folosit se creează premizele pentru folosirea unui sistem tip RTTI - Real Time Type Information. În linii

mari, un sistem tip RTTI ataşează fiecărui obiect o metodă „IsA", metodă care poate fi folosită pentru a scrie cod de tipul:

```
void RedrawEvent (Object *obj) {
  if (obj->IsA()=="BUTTON") {
    Button *btn = (Button *)obj;
    btn->draw();
  } else if (obj->IsA()=="DIALOG") {
    Dialog *dlg = (Dialog *)obj;
    dlg->draw();
  }
}
```

Din nou, sursa acestei greşeli e neînţelegerea ideii de derivare şi a ierarhiei pe care designerul a conceput-o. Problema vizibilă e duplicarea codului, faptul că probabil un RedrawEvent nu vine pentru un obiect generic ci pentru un obiect care poate fi desenat (deci posibil un Widget *). Dar probabil eroarea cea mai mare în acest caz e faptul că un nou tip (de exemplu MessageBox, cu IsA()=="MSGBOX") nu va fi acceptat de metoda aceasta de Redraw.

Casturile inutile provin cel mai des din neînţelegerea layerelor ierarhiei de clase folosite, şi de fapt sugerează faptul că fie interfaţa clasei curente e deficitară, fie posibilităţile bibliotecii folosite sunt prea limitate.

C++ oferă operatori speciali de cast: const_cast, reinterpret_cast, dynamic_cast, static_cast - consider însă că folosirea acestor operatori este simptomatică pentru erori în concepţia ierarhiilor de obiecte, şi codul care se bazează pe astfel de operatori ar trebui reevaluat.

3.4 Încălcarea ierarhiilor (layer breaking)

Foarte des este mult mai uşor pentru un anumit obiect să apeleze o metodă a obiectului părinte, cel care l-a creat. Această situaţie nu este deloc rară cînd ne gîndim la ierarhii vaste de obiecte precum structurile de date folosite în GUI design. Chiar dacă bibliotecile respective sunt complete în sine, şi modul lor de operate e bine definit, întotdeauna programatorii au nevoie să rupă ordinea internă pentru accesarea unor date şi metode din locuri neaşteptate din ierarhia de obiecte.

Exemplu:

```
(Widget *)((Application::App*)(this->GetRoot())->GetFirst())->SetFocus();
```

Acest gen de probleme se manifestă în special în cazul în care programatorii nu sunt obişnuiţi cu o bibliotecă complexă pe care o utilizează şi asupra cărora nu au nici un fel de control. Căutarea de soluţii pentru problemele aparente începe întotdeauna cu un workaround, iar design-ul aplicaţiei este de cele mai multe ori deficitar. Problema mai apare şi atunci cînd se evită rezolvarea problemelor de fond şi se rezolvă (uneori justificat) problema temporar apărută.

Acest gen de abordare rezolvă problema imediată şi introduce probleme de fond mai dificile, precum legarea de anumite versiuni ale unor biblioteci, sau

probleme mai directe precum cast-ul defectuos aşa cum am prezentat anterior. În cazul enunţat mai sus cast-ul „Widget *" este inutil, problema este însă legarea de faptul că primul widget din aplicaţie nu este neapărat cel dorit. De asemenea, la un update al bibliotecii folosite am putea găsi un nivel nou de indirectare, codul trebuind să fie rescris în forma:

```
((Application::App *)(this->GetRoot())->GetActiveScene()->GetFirst()->SetFocu
```

Cum se poate rezolva această problemă? Întotdeauna înţelegerea mai bună a uneltelor folosite ajută foarte mult - în al doilea rînd e nevoie de o disponibilitate mai mare de a accepta regulile impuse de bibliotecile folosite.

3.5 Problema cerc/elipsă

O altă problemă des întîlnită se datorează simplificării exagerate a modelului obiectual. Să luăm, spre exemplu, două clase de obiecte: cercul şi elipsa, şi să identificăm cum ar fi corect să implementăm derivarea obiectelor.

Prima variantă ar fi să derivăm elipsa din cerc - elipsa mai adaugă funcţionalitate la ideea de cerc, deci logic ar fi ca elipsa să fie derivată din cerc. Funcţionalitatea adăugată ar fi o nouă valoare pentru a doua axă a elipsei şi unghiul cu care se înclină elipsa. Problema majoră a acestui model este că, de fapt, nici un pic de funcţionalitate din cerc nu va fi folosită în codul pentru elipsă, şi deci utilitatea acestui tip de derivare ar fi extrem de limitată.

O a doua variantă, mult mai viabilă, este să implementăm cercul ca un caz special de elipsă, şi deci să derivăm cercul din elipsă. Conceptual, derivarea cercului din elipsă este absolut corectă - relaţia „IsA" este respectată („a circle is an ellipse"). Dacă obiectul cerc este imutabil, atunci modelul cerc-elipsă ales este cît se poate de corect.

Dacă însă obiectul suportă mutaţii (de exemplu se implementează pentru elipsă o metodă Stretch (XFactor, YFactor)) care să modifice datele obiectului, cercul va fi transformat într-o formă geometrică ce nu este cerc.

În acest caz, soluţia este regîndirea eşafodajului pentru ierarhia de obiecte. Regula simplă ar fi ca Cerc şi Elipsă să se afle pe acelaşi nivel derivare dintr-o formă geometrică generică ce să ofere, de exemplu, primitivă 'Draw'. O altă variantă ar fi ca astfel de derivare să se asigure că obiectul este imutabil; în acest caz o metodă Stretch pentru cerc ar returna un nou obiect de tip elipsă, în loc să afecteze obiectul cerc.

4 Dezechilibrarea ierarhiilor obiect-orientate

Această problemă se manifestă în design-urile în care o singură clasă monopolizează întreaga procesare în timp ce celelalte clase doar încapsulează date. Clasa 'principală' creşte cu noi funcţionalităţi, acaparînd întreaga arhitectură, coordonînd toată funcţionalitatea aplicaţiei.

Acest timp de abordare produce clase cu foarte mulţi membri şi foarte multe metode, din care e foarte posibil ca foarte multă funcţionalitate să fie inutilă.

Problema cea mai mare creată de dezechilibrarea ierarhiei de obiecte o reprezintă faptul că portarea într-un alt proiect va trage după sine foarte mult cod şi foarte multă funcţionalitate inutilă, fiind în acelaşi timp foarte dificil de refactorizat.

Clasa monolit este cel mai des expresia absenţei de design obiect-orientat şi se manifestă cel mai des la programatorii proveniţi dintr-un mediu care pune mult accent pe programarea procedurală (de exemplu programatorii de C care încep să înveţe C++).

Soluţia pentru această problemă este împărţirea responsabilităţilor şi ruperea clasei monolit în mai multe clase independente, izolînd codul independent - de cele mai multe ori clasele monolit oferă mai multe seturi funcţionale independente sau foarte puţin cuplate; identificarea şi izolarea claselor de funcţionalitate este o necesitate în acest caz. Foarte des, defalcarea monolitului va duce la eliminarea codului inutil şi la o structură internă mult simplificată şi mult mai flexibilă

5 Folosirea tehnologiilor nepotrivite

Foarte des, restricţiile în materie de tehnologii pot duce la rezultate dezastruoase. Un exemplu clasic sunt restricţiile impuse de sistemele embedded - foarte puţină memorie şi procesor foarte lent - care împinge decizia folosirii unui limbaj cît mai simplu (C) în detrimentul echivalentului de nivel înalt (C++). La baza acestei decizii pot sta argumente foarte puternice dar nu neapărat fundamentate în realitate precum: limbajul de nivel înalt produce cod mai lent, foloseşte mai multă memorie.

În cazul expus, alegerea limbajului C în detrimentul lui C++ îngroapă orice bune intenţii din partea programatorilor şi a designerilor. În momentul în care se încearcă crearea unor ierarhii logice de obiecte în C apar nenumărate probleme precum codul ilizibil, complexitate sporită pentru a compensa lipsa de suport pentru OOP a limbajului, concentrarea pe probleme minore de optimizare locală în loc de a avea în vedere întreaga problemă. Efectele vizibile imediate sunt

- creşterea nejustificată a numărului de linii de cod: o 'clasă' de tip listă cere cca. 50-100 de linii de cod în C++ şi 300-500 loc. în C
- creşterea numărului de caractere pe linie: comparaţi apelul *this->GetFirst()->SetFocus()* cu *Widget *pWidget = scene_get_first_widget (_this); widget_set_focus (pWidget);*
- necesitatea de a cunoaşte modul de funcţionare al 'claselor': de exemplu codul de mai sus s-ar putea scrie în C în felul următor:
  *Widget *pWidget = scene_get_first_widget (_this);*
  widget_set_focus (pWidget);
  scene_notify_focus_set (_this, pWidget);
- conţinerea întregii ierarhii de obiecte în numele funcţiilor nu este rar întîlnită în aceste cazuri, mascînd suprascrierea de funcţii:
  messagebox_dialog_widget_on_key_pressed();

Un exemplu de astfel de ierarhie scăpată de sub control este GTK+/GNOME, un desktop environment complex şi biblioteca pe care se bazează. Pentru mai multe detalii vezi http://www.gnome.org/ şi http://www.gtk.org/ . Chiar dacă reguli stricte sunt folosite pentru păstrarea ordinii în această ierarhie, programele devin foarte complexe, şi foarte greu lizibile. Din fericire există interfeţe C++ care înglobează funcţionalitatea C în apeluri C++, ascunzînd o parte din complexitate (http://www.gtkmm.org/).

Această problemă nu se manifestă numai în cazul limbajelor de programare, ci şi în cazul diverselor tehnologii. Cei care fac greşeala de a adopta o tehnologie sunt cel mai des victime ale marketing-ului puternic venit din partea proponenţilor acelei tehnologii, fie numele ei COM, CORBA, Windows CE sau Linux. Întotdeauna alegerea trebuie făcută în funcţie de necesităţi, şi nu neapărat din cauza orientărilor politice (în acest caz e de preferat să se refuze un proiect decît să se folosească tehnologia greşită). Genul acesta de abuz se numeşte în termeni comuni „Ciocanul de Aur" avînd ca sursă proverbul: „Cînd singura ta unealtă este un ciocan, toate lucrurile din jurul tău ţi se vor părea cuie"

Soluţia e, aşadar, să se aibă mereu în vedere folosirea uneltelor potrivite pentru problema de rezolvat.