

Testarea programelor orientate pe obiecte

21 noiembrie 2011

Probleme în testarea orientată pe obiecte [Binder]

Fiecare *nivel de ierarhie* creează *alt context* pentru elementele moștenite:
⇒ corectitudinea superclasei nu garantează pe cea a clasei derivate

Se comportă corect metodele superclasei în contextul subclasei?

Exemplu, pentru o clasa B care mosteneste o metoda `m` din A

1. putem să oitem complet re-testarea lui `B.m` ?
2. sunt de ajuns cazurile de test pentru `A.m` ?
3. trebuie cazuri de test noi ?

Liskov Substitution Principle

subclasa poate fi folosita oriunde in locul superclasei

$$\text{pre}(m, \text{Class}) \Rightarrow \text{pre}(m, \text{SubClass})$$
$$\text{post}(m, \text{SubClass}) \Rightarrow \text{post}(m, \text{Class})$$
$$\text{inv}(\text{SubClass}) \Rightarrow \text{inv}(\text{Class})$$

Dar: trebuie sa *cunoaştem* invariantii pentru a-i verifica

Ca minim: analizăm ce membri sunt modificati

Exemplul clasic dreptunghi - patrat

```
public class Rectangle {
    private int height; private int width;
    public void setHeight(int value) { this.height = value; }
    public void setWidth(int value) { this.width = value; }
    public int getArea() { return this.height * this.width; }
}
```

```
public class Square extends Rectangle {
    public void setHeight(int value) { super.setHeight(value);
                                     super.setWidth(value); }
    public void setWidth(int value) { super.setWidth(value);
                                     super.setHeight(value); }
}
```

Probleme în testarea orientată pe obiecte (cont.)

Interacțiunile dintre *apeluri* și *starea* obiectului sunt complexe

Există interacțiuni nedorite între metode ?

Polimorfismul și legarea dinamică cresc numărul de căi de execuție

îngreunează analiza statică pentru determinarea căilor prin cod

```
void foo(A obj) { obj.m(); }
```

poate fi de fapt metoda *m* pentru oricare subclasa a lui *A*

Încapsularea limitează *observabilitatea* stării la testare

Legarea dinamică crește potențialul de neînțelegere și erori

Erori de interfață sunt favorizate de existența multor componente mici

Controlul stării obiectelor e dificil, fiind *distribuit* în tot programul

Specific și probleme în testarea OO (cont.)

[McGregor&Sykes] Datorate noțiunilor fundamentale de limbaj:

Obiecte

ascund informația ⇒ îngreunează observarea stării în testare

au *stare* persistentă ⇒ dacă e inconsistentă cauzează erori

au o durată de viață ⇒ erori prin construirea/distrugerea inoportună

Mesaje / metode ⇒ importante în testarea *interacțiunii* obiectelor

pot fi apelate în stări nepotrivite ale obiectului

au parametri (folosiți/actualizați): se află în starea potrivită?

implementează corect interfețele dorite? (vezi probleme de subtip)

Interfață = specificație comportamentală

Două abordări privind condițiile de funcționare corectă:

bazată pe contract: le presupune / *programare defensivă*: le verifică

⇒ influențează complexitatea implementării și testării:

simplifică/complică testarea clasei/testarea de integrare

Obs: programarea defensivă verifică și rezultatele (deși în practică adesea serverul/receptorul e considerat de încredere, doar clientul nu)

Specific și probleme în testarea OO (cont.)

Clasa

specificație: pre/postcondiții de metode, invarianți de clasă ⇒ testate!

Specificația trebuie și ea *validată* !

implementare: potențial de erori prin:

Constructori/destructori (inițializare/(de)alocare incorectă)

Colaborare inter-clase: membri/parametri obiecte pot avea erori

Un client are mijloacele de a verifica condițiile necesare?

Moștenirea

Poate propaga erorile la descendenți ⇒ oprite prin testarea la timp

Impune aspectul tipic de cod OO (multe apeluri, puține prelucrări, metode scurte) ⇒ acoperirea de cod/decizie e puțin relevantă

Oferă un mecanism de re folosire a testelor, din super- în subclasă

Testarea poate detecta moștenirea doar pentru re folosirea codului (fără a fi o specializare, adică a moșteni specificată)

Specific și probleme în testarea OO (cont.)

Polimorfismul

Testarea trebuie să verifice principiul substituției. În subclasă metodele au precondiții mai slabe/postcondiții mai puternice invariantul implică cel al clasei de bază (e mai puternic)

Din perspectiva stărilor observabile (prin program / test):

Subclasa păstrează toate stările observabile și tranzițiile între ele

Poate adăuga tranziții (comportament suplimentar)

Poate adăuga stări observabile ca sub-stări ale celor inițiale

Problema yo-yo: dificultatea înțelegerii (\Rightarrow testării) secvenței de apeluri

\Rightarrow eroare probabilă: apelul versiunii greșite de metodă din ierarhie

Abstracția în ierarhia de clase \Rightarrow reflectată în teste (general \rightarrow specific)

Axiome de testare

[Weyuker '86,'88], aplicate în contextul OO de [Perry & Kaiser '90]
Antiextensionalitate: Implementări diferite la aceeași funcționalitate pot necesita suite de test diferite.

1) O metodă redefinită necesită (și) alte teste (depinzând de cod)

2) *Aceași* metodă moștenită necesită teste în funcție de clasă!

Ex: A: +m(), +n() B: +m() C: +n() și m apelează n()

⇒ C::m moștenește B::m dar apelează *alt* n() ⇒ cere alte teste!

Antidecompoziție: Un set de teste adecvat pentru un program nu e neapărat adecvat pentru o componentă a lui.

(ea poate fi exercitată în alt context decât programul respectiv)

⇒ Testarea adecvată a unui client nu e suficientă pentru serverele lui!

(clientul ar putea folosi doar o parte din funcționalitate)

⇒ Derivând dintr-o clasă testată trebuie re-testate metodele moștenite!

(codul adăugat poate interacționa cu starea ⇒ cu metodele moștenite)

Axiome de testare (cont.)

Anticompoziție: Un set de teste adecvat pentru componente nu e neapărat suficient pentru combinația lor.

adevărată și pentru combinație secvențială:

p căi de test în P și q căi de test în Q produc $p \cdot q > p + q$ căi în $P; Q$
cu atât mai mult când execuția alternează repetat între P și Q

⇒ Testarea de modul nu poate substitui testarea de integrare!

⇒ O metodă testată în clasa de bază nu e suficient testată în clasa derivată (pentru că poate fi compusă în alte feluri)

General Multiple Change Programe care au același flux de control dar alte valori / operații necesită suite de test diferite.

Exemple de erori: Încapsulare

Exemplu: clasă mulțime cu metode de

```
add(element) // condiție: element nu e în mulțime
              // generează excepție Duplicate în caz contrar
remove(element)
```

Testare: două `add(x)` consecutive generează excepție
dar totuși elementul e adăugat a doua oară

⇒ eroare descoperită doar cu `2 × add`, `2 × remove`

mai dificil decât dacă starea obiectului ar fi direct observabilă

Exemple de erori: Moștenire

Problema: în realizarea unei clase trebuie înțelese detaliile și convențiile de reprezentare ale tuturor claselor de bază pentru a fi siguri de o implementare corectă.

⇒ *Moștenirea slăbește încapsularea*

Două clase mari de probleme:

1) inițializarea

– ex. dacă se uită execuția corectă a inițializării pentru superclasă

2) omiterea redefinirii unor metode ținând cont de specificul clasei

– ex. metode de copiere, sau `isEqual`

Modele de eroare în testarea OO [Offutt]

Folosire inconsistentă ca tip

Deriv e folosită inconsistent și ca *Base* (chiar fără redefiniri)

Ex: *Stack* (acces la un capăt) implementat din *Vector* (acces arbitrar)

folosirea `Vector::removeAt(idx)` pe *Stack* violează invariantul clasei

Cauza: eroare de proiectare. Detecție: testarea invariantilor de clasă

Erori de definiție de stare

1) Metodele derivate interacționează diferit cu starea obiectului

Detecție: verificare că metodele definesc/folosesc aceiași membri

2) Redefinirea locală a unui membru (ascunde membru omonim moștenit)

dar metodele moștenite accesează membrul vechi \Rightarrow inconsistență

3) Metodă redefinită face alt calcul asupra aceluiași membru

\Rightarrow inconsistență a stării în raport cu specificația (moștenită)

Erori de constructor

Apel virtual în constructor \Rightarrow în derivat, acces la stare neinițializată

... și altele (anomalii de vizibilitate, etc.)

Particularități ale testării OO

Nivele: intra- și inter-metodă, intra- și inter-clasă

Problema vizibilității (limitată de încapsulare):

- expandarea explicită în sursă a ierarhiei de clase (flattening)

- suport de limbaj/implementare pentru accesul de către clasa de test

- folosirea de metode accesori pentru observarea stării

Polimorfismul: necesită instanțierea prin test a tuturor subtipurilor
posibile pentru un obiect declarat dintr-un tip de bază

- analiză statică pentru determinarea tuturor posibilităților

Testarea bazată pe *flux de date*

Sunt importante datele transmise/starea modificată; acoperirea de
cod/decizie dă informație redusă pe corpuri mici de metodă

Cuplajul: definit prin perechi *def-use* între metode

i.e. un membru definit (scris) în `m1()` și folosit (citit) în `m2()`

folosit pentru a selecta metodele care sunt testate împreună

Testarea ierarhiilor de clase

Distingem: teste pornind de la *specificație* sau *implementare* (cod)

S: noi teste pentru metode vechi, la modificarea specificației

S: postcondiții/invarianți noi pentru teste vechi, în clase derivate

I: noi teste pentru metode noi, în funcție de criteriul de acoperire dorit

Exemple:

Modificare $m()$ în superclasă: re-testare $m()$ + metode dependente;
re-testare $m()$ în contextul subclaselor

Modificare subclasă: retestare metode moștenite care pot interacționa

Suprascriere $m()$: augmentare teste `Base::m` pt. acoperire adecvată

Suprascriere $m()$ folosită de `Base::n`: test n în subclasă

Modificare interfață (clasă abstractă): retestarea întregii ierarhii!

Tipare de testare OO [Binder]

La nivel de *metodă*

Category/Partition (analiză I/O, partiționare/echivalență)

Combinational Function Test (acoperire a condițiilor)

Recursive Function Test

Polymorphic Message Test (client al unui server polimorfic)

La nivel de *clasă*

Invariant Boundaries

Nonmodal Class Test (clasă fără constrângeri de secvențiere)

Modal Class Test (clasă cu constrângeri de secvențiere)

Quasi-Modal Class Test (constrângeri dependente de stare)

Pentru *componente reutilizabile*

Abstract Class Test (interfață)

Generic Class Test (parametrizată)

New Framework Test

Popular Framework Test (modificări în cadru de aplicație intens folosit)

Exemplu: Polymorphic Message Test

– pentru un apel de metodă virtuală (într-un client), testează toate clasele posibile la care s-ar putea face apelul

Necesitate / erori posibile:

- precondiții incorecte de apel pentru anumite subclase
- apel (prin pointer incorect) la clasa neintentionată
- modificarea ierarhiei de clase

Procesul de legare dinamică \simeq ramificare în cod

\Rightarrow acoperirea tuturor instanțelor posibile \simeq branch coverage

Nonmodal Class Test

clasă ne-modală = acceptă orice mesaj (apel) în orice stare
ex. DateTime acceptă orice secvență de get/set (use/def)

Tipuri de comportament de test

- define-operation: set pentru intrare validă / verifică răspuns
- define-exception: set pentru intrare invalidă / verifică răspuns
- define-exception-corruption: stare nu e coruptă după excepție
- use-exception-test: se revine normal după utilizare
- use-correct-return: se revine cu valoarea corectă după utilizare
- use-corruption: obiectul nu e corupt după utilizare

(Quasi-)Modal Class Test

Modal Class Test:

clasă cu constrângeri permanente/fixe privind ordinea operațiilor
– se creează un *model* cu stările obiectului și tranzițiile între ele

Probleme:

- tranziție lipsă: o operație e respinsă într-o stare validă
- acțiune incorectă: răspuns incorect pentru stare/metodă dată
- stare rezultată invalidă: metoda produce tranziție în stare incorectă
- stare rezultată coruptă
- mesaj acceptat când ar trebui respins

Quasi-modal class test

clasă unde constrângerile la mesaje se schimbă odată cu starea clasei
ex. clasele de tip container / colecție (stivă plină/goală, etc.)

Tipic: am dori acoperire tip $N+$ (orice mesaj în orice stare)

Testarea la nivel de clasă

Abordarea *Small Pop*

- scriere clasă, scriere teste, rulare (fără alte detalieri/intermedieri)
- valabilă pentru clase simple în contexte stabile

Abordarea *Alpha-Omega* – se trece obiectul de la creare la distrugere, prin fiecare metodă

- constructori
- accesorii (get)
- predicate
- modificatori (set)
- iteratori
- destructori