Testarea Unitară

Testarea este necesară și pentru programele scrise într-un limbaj "static" (pentru care se fac verificări de corectitudine la compilare), dar cu atât mai mult pentru programe scrise într-un limbaj dinamic și care nu beneficiază de verificările efectuate de un compilator.

În cursul dezvoltării unei aplicații, codul evoluează (fie datorită modificării cerințelor, fie prin reproiectare sau prin refactorizare), dar prin teste ne putem asigura că acest cod modificat continuă să respecte cerințele impuse (că este corect).

De ce Unit Testing?

1.Ajută dezvoltatorii să înțeleagă baza de cod și le permite să facă modificări rapide.

2.Testele unitare bune servesc drept documentație a proiectului.

3.Testele unitare ajută la reutilizarea codului. Reutilizând atât codul, cât și testele către un nou proiect. Modificând codul până când testele rulează din nou.

4.Testele unitare ajută la remedierea erorilor la începutul ciclului de dezvoltare și la reducerea costurilor.

Menționând aceste aspecte putem indica ce implică testarea unitară Figura 1:

- Teste unitare, care verifică fiecare unitate de program separat de celelalte (o unitate poate fi o metodă, o clasă, o secvență de apeluri de metode).
- Teste de integrare, care verifică interacțiunile dintre unitățile de program (testate separat).
- Teste de sistem, care verifică toata aplicația.
- Teste de acceptare, care verifică modul cum programul răspunde cerințelor beneficiarilor.

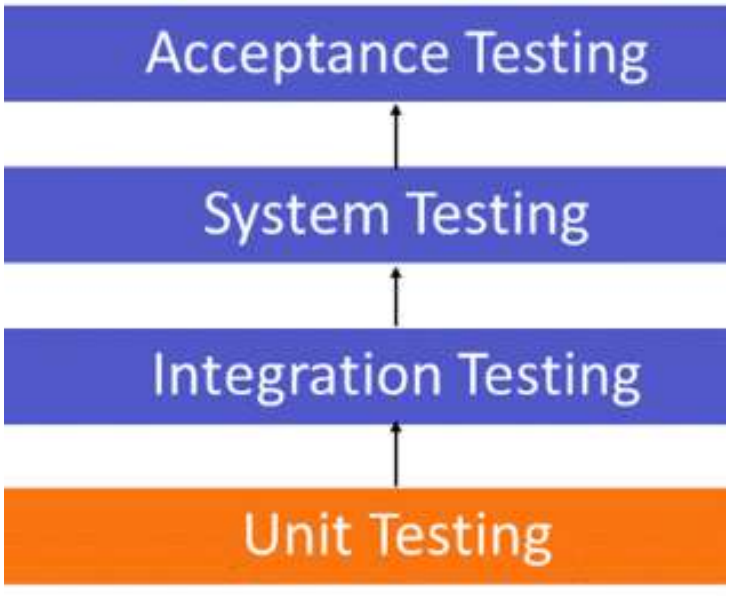

Fig. 1. Este indicat ce implică testarea unitară

Scrierea de teste unitare, înainte de scrierea codului efectiv, încurajează coeziunea și un cuplaj slab, în sensul că vom prefera pentru testare să definim clase și metode mai mici (care un singur rol în aplicație) și cu mai puține dependențe între ele, astfel ca să poată fi testate izolat fiecare.

Cuplajul unei unități de cod poate fi de două feluri:

- Unitatea testată depinde de alte unități
- Alte unități de cod depind de unitatea testată

Este posibil ca unitatea de care depinde secvența testată (o resursă) să nu fie încă disponibilă, să fie costisitor de utilizat sau să fie impredictibil în comportare. În aceste cazuri se folosesc obiecte surogat de tip "stub" sau "mock" în locul unui "colaborator" din aplicația reală.

Obiectul surogat poate corespunde unui singur obiect din aplicația finală, unei componente, unui nivel sau unui subsistem din aplicație.

Un obiect "stub" mimează parțial comportarea unui obiect real (indisponibil), prin transmiterea unui răspuns corespunzător la fiecare cerere primită (printr-un apel de metodă), fără a ține seama de succesiunea acestor cereri.

Un obiect "mock" mimează mai bine comportarea obiectului real, pentru că ține cont de secvența apelurilor și de numărul lor, deci simulează întreaga interacțiune cu obiectul testat.

Testarea unitară în Java este facilitată de existenta mai multe produse software de tip “Test Framework” dintre care mai folosite sunt JUnit, EasyMock, s.a.

În plus, cele mai importante IDE-uri (Eclipse, NetBeans, IDEA) facilitează efectuarea de teste prin crearea unui subdirector pentru metodele de test, printr-o comandă (opțiune) de testare aplicație (alta decât comanda “Run” pentru execuția aplicației) și prin afișarea în mod grafic a rezultatelor testelor. Atunci când testele eșuează, nu se afișează toată secvența de apeluri de metode care a condus la eroare “AssertionError” (așa cum se întâmplă în linia de comandă sau când se folosește un IDE mai simplu care nu are regim special pentru testare unitară).

Elementele de testare unitară, pun la dispoziție unelte pentru a înregistra și repeta teste, pentru ca testele unitare sa poată fi repetate ușor mai târziu (de regulă când se schimbă o parte din sistem),

astfel încât dezvoltatorul să fie convins ca noile modificări nu au stricat vechea funcționalitate. Acest lucru mai este cunoscut ca **testare regresivă**.

Conceptele testării unitare și testării regresive sunt destul de vechi, dar popularitatea lor a crescut brusc de curând, după apariția unei unelte de testare unitară pentru Java: JUnit.

Testarea unitară se referă la scrierea unor bucăți de cod, denumite cod de testare, care validează codul de producție. Testarea majorității aplicațiilor devin așadar automată.

Platforme de testare

- C++
  - CPPUnit
  - Boost.Testing library
  - CxxUnit
- Java
  - JUnit
  - TestNG
- .NET (C#, VB.NET, etc.)
  - NUnit

Teste unitare

- Fiecare test unitar (unit test) implementează un singur caz de testare;
- Unitățile sunt testate independent unele fata de altele;
- Pentru fiecare unitate se scriu, în mod uzual, mai multe teste unitare.

Testele unitare au câteva caracteristici importante:

- fiecare test validează un comportament din aplicație;
- rulează foarte repede, maxim în câteva minute;
- sunt foarte scurte și ușor de citit;
- rulează la apăsarea unui buton, fără configurări suplimentare.

Pentru a fi rapide, testele unitare folosesc adesea așa-numitele "duble de testare". La fel cum piloții de avioane învață într-un simulator înainte de a se urca în avion, testele unitare folosesc bucăți de cod care seamănă cu codul de producție, dar în realitate folosesc doar la teste. Stub-urile și mock-urile sunt cele mai întâlnite duble de testare, existând multe altele mai puțin folosite.

Un stub este o dublă de testare care întoarce valori. Stub-ul este similar cu o simulare foarte simplă: atunci când apeși un buton, apare o valoare.

Un mock este o dublă de testare care validează colaborarea între clase. Mock-ul validează apeluri de metode, cu anumiți parametri, de un anumit număr de ori. Din această cauză, un mock poate fi folosit și la validarea apelurilor de metode care nu întorc valori.

Majoritatea unităților de program supuse testelor depind de alți "colaboratori" care pot fi unități locale (alte metode din aceeași clasă), pot fi parametri ai metodelor testate sau pot fi chiar metode aflate în alt calculator. Pentru a testa unități cu dependențe de alte unități trebuie înlocuiți colaboratorii cu obiecte surogat ("stub" sau "mock"). Obiectele surogat se folosesc în testele unitare si vor fi înlocuite cu obiecte reale în testele de integrare.

Crearea de obiecte surogat în Java se poate face în două feluri:

- Manual, prin scrierea metodelor clasei surogat;
- Automat (dinamic) utilizând un "framework" ca EasyMock, JMock, ș.a.

Clasa surogat scrisă manual trebuie să aibă aceleași metode cu clasa pe care o înlocuiește; acest lucru se poate face prin extinderea clasei colaborator și redefinirea metodelor sau prin implementarea unei interfețe definite în scopul testării.

In Java, pentru a facilita trecerea de la teste unitare la teste de integrare se folosesc fie interfețe (pentru comunicarea cu obiectele surogat), fie obiecte de tip proxy, care interceptează cererile și le dirijează fie către obiecte surogat, fie către obiecte reale.

Schema următoare arată o interfață între codul testat și codul de care depinde sau codul surogat (care înlocuiește în teste codul real).

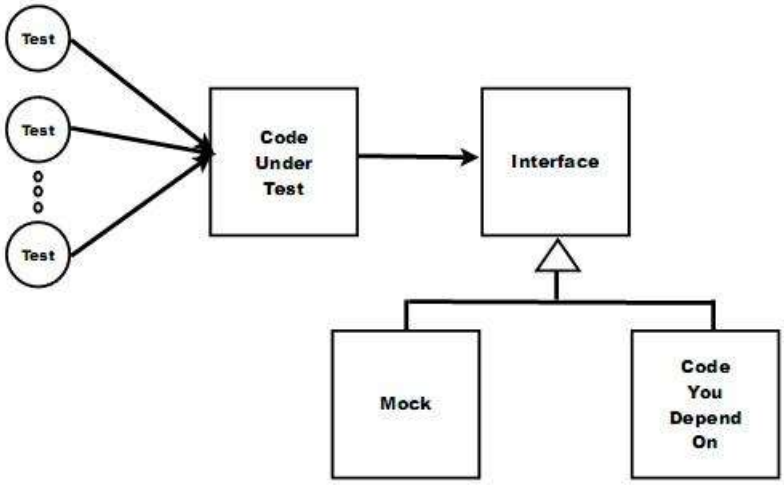

Fig. 2. Utilizare EasyMock în Java

EasyMock este un framework de testare pentru aplicații Java care generează dinamic clase si obiecte surogat si poate verifica ordinea apelurilor metodelor din obiectul surogat.

Dublele de testare pot fi create și folosind framework-uri speciale, cum ar fi mockito pentru Java (a fost portat și pe alte limbaje) sau moc pentru .NET.

Inițial dublele de testare erau folosite doar în locurile unde era foarte greu să controlezi sistemul sau unde testele erau încetinite de apeluri la sisteme externe. În timp, dublele de testare au ajuns să fie folosite în toate testele unitare, dând naștere metodei "mockiste" de testare unitară.

Testele unitare sunt scrise de programator, în timp ce implementează o funcționalitate. Din păcate, cel mai întâlnit mod de a scrie teste este cândva după ce a fost terminată implementarea. Rezultatul este că testele sunt scrise având în minte cum ar trebui să funcționeze codul și nu testarea lui.

Test „First Programming" este o metodă de a scrie teste care implică următorii pași:

- crearea unui design pentru implementarea funcţionalităţii;
- crearea minimului de cod necesar (compilabil, dacă limbajul folosit este compilat) pe baza design-ului;
- scrierea unuia sau mai multor teste care codează ceea ce trebuie să facă design-ul; testele vor pica în acest moment;
- implementarea codului care face testele să treacă.

Prin aplicarea testului „First Programming", programatorii se asigură că scriu teste unitare și că testează ceea ce ar trebui să rezolve, nu implementarea soluției.

Durează mai mult când scriu teste!

Studiile de caz și experiența personală a programatorilor a arătat că într-adevăr, timpul petrecut strict pe dezvoltarea unei funcționalități crește odată cu adoptarea testării unitare. Aceleași studii au arătat că timpul petrecut pe mentenanță scade rapid, arătând ca unit testing poate aduce o îmbunătățire netă în timpul de dezvoltare.

Acest fapt nu poate schimba percepția programatorului care trebuie să scrie mai mult cod. De aceea, programatorii presupun adesea că per total proiectul derulează mai încet din cauza testării automate.

Este bine ca „adopția" unit testing să se facă cu grijă, incremental, urmărind câteva puncte importante:

- Clarificarea conceptelor legate de unit testing înainte de a începe scrierea de teste.

  Programatorii trebuie să poată "mânui" fără teamă unelte precum: stub-uri, mock-uri, teste de stare, teste de colaborare, teste de contract. De asemenea, programatorii trebuie să înțeleagă ce cazuri merită și trebuie testate.

Greșeli comune

Câteva greșeli comune legate de unit testing sunt:

- Scrierea multor teste de integrare(care implică mai multe clase sau module) lente și fragile în detrimentul testelor unitare mici, rapide și ușor de întreținut
- Abandonarea dublelor de testare, sau folosirea lor în scopuri pentru care nu au fost create. Dublele de testare ajută la obținerea unor teste scurte și rapide.
- Numele testelor nu exprimă comportamentul testat. Numele testului poate da foarte multe informații atunci când testul pică.
- Folosirea intensivă a debugger-ului pe teste. Testele bine scrise vor spune imediat unde este problema în cazul în care pică. Debugging-ul este în continuare util în situații exotice.
- Cod de testare neîngrijit. Codul de testare este cel puțin la fel de important ca și codul de producție, și trebuie întreținut cu aceeași grijă.

Un programator utilizează în general un cadru UnitTest pentru a dezvolta cazuri de testare automate. Folosind un cadru de automatizare, dezvoltatorul codifică criteriile în test pentru a verifica corectitudinea codului. În timpul executării cazurilor de test, cadrul înregistrează cazurile de test nereușite. Multe cadre vor semnaliza și raporta automat, pe scurt, aceste cazuri de test nereușite. În funcție de gravitatea unui eșec, cadrul poate opri testarea ulterioară.

Fluxul de lucru al testării unitare este

1) Crearea cazurilor de testare

2) Revizuirea / reelaborarea

Avantajul testarii unitare:

Dezvoltatorii care doresc să afle ce funcționalitate oferă o unitate și cum să o folosească pot examina testele unitare pentru a obține o înțelegere de bază a API-ului unității.

- Testarea unității permite programatorului să refactorizeze codul la o dată ulterioară și să se asigure că modulul funcționează în continuare corect (adică testarea de regresie). Procedura constă în scrierea cazurilor de test pentru toate funcțiile și metodele, astfel încât ori de câte ori o modificare provoacă o eroare, aceasta poate fi identificată și remediată rapid.
- Datorită naturii modulare a testării unitare, putem testa părți ale proiectului fără a aștepta finalizarea altora.

Dezavantaje

Nu se poate aștepta ca testarea unității să surprindă fiecare eroare dintr-un program. Nu este posibil să se evalueze toate căile de execuție chiar și în cele mai banale programe.

- Testarea unitară prin natura sa se concentrează pe o unitate de cod. Prin urmare, nu poate detecta erori de integrare sau erori la nivel de sistem.

Se recomandă utilizarea testelor unitare împreună cu alte activități de testare.