Dezvoltarea algoritmilor randomizați prin Test-Driven Development (TDD)

TDD este o tehnică folosita în aplicații cu algoritmi determiniști, în care intrarea și rezultatul așteptat sunt cunoscute.

1. Introducere

Rezultatul unui sistem software care este întotdeauna anticipat indică un ***algoritm determinist***. Cu toate acestea, există algoritmi care pot produce diverse rezultate corecte pentru aceeași intrare, cunoscuți sub numele de ***algoritmi randomizați***. Aceștia se bazează pe decizii aleatorii în timpul execuției.

Algoritmii randomizați sunt folosiți pentru a rezolva probleme cu mai multe soluții corecte sau acceptabile (de exemplu: rezolvarea problemelor dificile de aproximare). Ei utilizează funcții care returnează numere pseudo-aleatoare sau elemente aleatorii dintr-un set dat. În unele situații, când funcțiile aleatoare sunt folosite doar de câteva ori, testarea automată poate fi fezabilă prin simularea mai multor valori returnate. Totuși, în cazul în care funcția aleatoare este apelată de mai multe ori sau de un număr variabil de ori, simularea tuturor acestor apeluri nu este practică.

Prin urmare, nu este practic să simulăm toate aceste căi pentru a testa un algoritm randomizat cu mai multe alegeri aleatorii, chiar si preupunand ca ar fi, ele nu ar fi cu adevărat aleatoare.

TDD este o tehnică de dezvoltare a software-ului în care testele sunt dezvoltate înaintea codului, în cicluri scurte și incrementale. Această tehnică propune ca dezvoltatorul să creeze întâi un test eronat și apoi să implementeze o mică parte de cod pentru a satisface setul actual de teste. Apoi, codul este refactorizat, dacă este necesar, pentru a oferi o structură și arhitectură mai bună pentru soluția actuală.

a) rezultatele pentru fiecare execuție pot fi diferite pentru aceleași intrări, ceea ce face dificilă validarea valorii returnate
b) obținerea unei valori de retur valide pentru o execuție a unui caz de test nu înseamnă că valoarea de retur validă va fi livrată și în următoarele execuții
c) deciziile aleatorii și numărul lor de posibile căi fac ca returnarea rezultatelor fixe pentru aceste decizii să nu fie viabilă
d) este dificil să se execute un test eșuat anterior cu aceleași decizii aleatorii luate în execuția sa anterioară

2. Background

1.1 Algoritmi randomizati

a) Algoritmi Monte Carlo: definesc o clasă de algoritmi care rulează în general rapid și returnează răspunsul corect cu o probabilitate mare, deși ar putea returna un răspuns greșit. Algoritmii de aproximare, cum ar fi Algoritmii Genetici, sunt variații ale algoritmilor Monte Carlo

b) Algoritmi Las Vegas: sunt acei algoritmi care întotdeauna returnează un răspuns corect, dar timpul de execuție este aleator, finit

Algoritmul utilizat în studiul pentru a verifica viabilitatea abordării noastre este un algoritm de tip Las Vegas care întotdeauna oferă un răspuns corect, cu executie in timp aleatoriu. Folosind un graf ca intrare, algoritmul aplică o secvență aleatorie de transformări, iar rezultatul ar trebui să fie un graf care să aibă anumite metrici specifice. Cu toate acestea, abordarea propusă în acest studiu poate fi aplicată ambelor clase de algoritmi randomizați. (Detalii aprofundate in capitolul 5)

2.1 Algoritmi randomizati

a) Algoritmi Monte Carlo: definesc o clasă de algoritmi care rulează în general rapid și returnează răspunsul corect cu o probabilitate mare, deși ar putea returna un răspuns greșit. Algoritmii de aproximare, cum ar fi Algoritmii Genetici, sunt variații ale algoritmilor Monte Carlo

b) Algoritmi Las Vegas: sunt acei algoritmi care întotdeauna returnează un răspuns corect, dar timpul de execuție este aleator, finit

Algoritmul utilizat în studiul pentru a verifica viabilitatea abordării noastre este un algoritm de tip Las Vegas care întotdeauna oferă un răspuns corect, cu executie in timp aleatoriu. Folosind un graf ca intrare, algoritmul aplică o secvență aleatorie de transformări, iar rezultatul ar trebui să fie un graf care să aibă anumite metrici specifice. Cu toate acestea, abordarea propusă în acest studiu poate fi aplicată ambelor clase de algoritmi randomizați. (Detalii aprofundate in capitolul 5)

2.2 Test-driven development(TDD)

TDD este o tehnică de dezvoltare și design a codului în care codul de testare este creat înainte de codul de producție. În practica TDD, dezvoltatorul alege o cerință pentru a determina pe ce se concentreaza testele, apoi scrie un caz de test care definește modul în care acea cerință ar trebui să funcționeze din punctul de vedere al clientului clasei. Deoarece această cerință nu a fost încă implementată, se așteaptă ca noul test să eșueze. Următorul pas este să se scrie cea mai mică cantitate posibilă de cod pentru a implementa noua cerință verificată de test. În acest punct, testul adăugat, precum și toate celelalte teste existente anterior, sunt așteptate să ruleze cu succes. Dupa, codul poate fi restructurat astfel încât să poată fi continuu evoluat și îmbunătățit.

2.3 JUnit framework

JUnit este un open-source framework pentru Java utilizat pentru crearea de teste unitare și automatizarea acestora în practica dezvoltării conduse de teste (TDD). Acesta oferă puncte de extensie importante, cum ar fi Runner și Rule, care permit adăugarea de noi funcționalități și comportamente la testarea unitară.

a) clasa ***Runner*** este responsabilă pentru executarea metodelor de test dintr-o clasă de testare, iar clasele pot fi extinse pentru a crea runneri personalizați. Utilizarea @RunWith permite specificarea unui runner personalizat într-o clasă de testare
b) ***Rule*** este un alt punct de extensie care permite adăugarea de comportamente înainte și după execuția fiecărui test. Aceasta oferă o modalitate flexibilă de a gestiona resurse sau de a efectua acțiuni suplimentare în cadrul testelor

3. Research

Cercetarea a fost inițiată în timpul dezvoltării unui algoritm randomizat la *Institutul Național de Cercetare Spațială*. Dezvoltatorul era familiarizat cu tehnica TDD, dar nu știa cum să o aplice în contextul respectiv.

Studiile **single-subject** sunt o modalitate de a oferi un suport de incredere in ingineria software, în care doar un singur subiect efectuează sarcinile din studiu. Aceste studii sunt de obicei analizate prin determinarea dacă efectele tratamentului sunt vizibile și, prin urmare, nu necesită analize statistice elaborate.

În acest experiment **single-subject**, abordarea a fost aplicată în implementarea algoritmului care implică **decizii multiple aleatoare** în executare. Rezultatele acestei implementări folosind aceasta abordarea au fost comparate cu același algoritm implementat anterior **în mod tradițional**, folosind același limbaj de programare și fără teste unitare.

Acest tip de abordare a fost ulterior folosit în tehnici mai ample, ceea ce a dus la apariția unui framework Java numit ***ReTest*** care facilitează generarea de seed-uri, repetarea execuției testelor și suportul pentru re-executarea testelor eșuate anterior.

În acest studiu, 10 participanți au implementat un algoritm simplu folosind ***ReTest*** și urmând o procedură stabilită pentru a garanta aplicarea TDD-ului. La sfârșitul activității, un chestionar cu întrebări a fost utilizat pentru a colecta opinia dezvoltatorilor care au participat în legătură cu experiențele lor în ceea ce privea implementarea cu ajutorul acestui framework.

Capitolul 4 – Automatizarea testelor pentru logica randomizată

Atunci când vine vorba de testarea aplicațiilor, se folosesc în special algoritmi determiniști - inputul si output-ul așteptat sunt cunoscute. Printre altele se numără si tehnicile care determină un set de teste necesare care pot acoperi întregul domeniu de input-uri – ***Combinational Testing***. Totuși, când vine

vorba de logica randomizată, această metodă devine ineficientă deoarece același input poate returna mai multe valori. Cu alte cuvinte, un test efectuat anterior cu succes poate eșua la apariția aceluiași output într-un test efectuat ulterior.

Pentru automatizarea în logica randomizată s-a folosit o abordare alcătuită din 3 etape:

- **Asertare a Caracteristicii Deterministe** (*Deterministic Characteristic Assertion*) - pentru validarea rezultatului returnat de algoritm
- **ReTest** (*Re-Test with Different Seeds*) - pentru rularea repetată a testelor și obținerea mai multor rezultate
- "**Reciclarea**" (*Recycle Failed Seeds*) - pentru retestarea valorilor care au eșuat în testele anterioare

4.1. Asertare a Caracteristicii Deterministe (Deterministic Characteristic Assertion)

Această etapă propune utilizarea unor criterii specifice pentru a validarea rezultatelor obținute în urma rulării, dar și a corectitudinii algoritmului. În teorie, aceasta este dată de output - dacă acesta corespunde cu input-ul dat, atunci spunem că algoritmul este corect având în vedere "specificația" dată.

Ulterior, vor fi selectate doar soluțiile fundamentale care pot determina corectitudinea algoritmului cu scopul de a identifica regulile de bază, fără a modifica fluxul de rulare. În contextul algoritmilor determiniști, va fi nevoie de câte un set de verificări pentru fiecare valoare posibilă.

O metodă eficientă pentru a implementa acest proces este divizarea algoritmului in mai mulți sub-algoritmi pentru care se vor face testele necesare.

4.2. Re-Test with Different Seeds

Scopul acestei etape este de a crește range-ul de testare. Mai exact, algoritmul trebuie modificat astfel încât acesta va putea primi un obiect (sământă) capabil să genereze intrări pseudo-aleatoare, asupra cărora se vor rula testele definite la punctul 4.1..

O astfel de tehnică este prezentă în cadrul mai multor limbaje de programare. Sământa este adesea definită printr-un integer care funcționează ca o cheie primară. Atunci când algoritmul primește o sământă, acesta va începe testele pe toate rezultatele aleatoare posibile pornind de la obiectul primit. Cu cât se efectuează mai multe verificări, cu atât va crește aria de acoperire a valorilor valide.

În cazul programelor complexe, atunci când mulțimea semințelor este numeroasă, se recomandă creșterea numărului de teste efectuate pe fiecare set. Totuși, această tehnică nu garantează acoperirea tuturor posibilităților.

4.3. Recycle Failed Seeds

Reamintim că unele teste efectuate în cadrul pasului anterior pot eșua la o eventuală reverificare. Astfel de rateuri sunt aproape imposibil de reprodus și îngreunează utilizarea unor tehnici moderne în caitatea software-ului – exemplu: ***Teste de Regresie***.

Astfel de tehnici devin ineficiente deoarece verificările pot deveni foarte ample pe parcursul procesului de retestare. Drept soluție, mai mulți cercetători au sugerat ca un singur subset de teste –

ales după caracteristicile sistemului software - să fie selectat si optimizat pentru a minimiza timpul de execuție.

În final, s-a propus divizarea domeniului de intrare în două subseturi unde:

- primul subset este o proiecție a cazurilor de test
- al doilea subset este dat de testele care au eșuat, folosit pentru a diminua radical inputurile ce necesită o reverificare

4.4. Exemplu

Considerăm un algoritm care generează un vector cu n poziții, având valori pseudo-aleatoare cuprinse între -10 și 10, cu proprietatea că suma acestora este 0.

Notăm

- n = dimensiunea vectorului
- e = numărul de elemente
- P = posibilități

Astfel obținem: $p = e^n$

Presupunem că n = 3. Știind că n = 21 => p = 9261

Fiecare posibilitate p trebuie verificată, ceea ce duce la o creștere exponențială a numărului teste. Mai jos este o reprezentare vizuală a implementării celor 3 etape.

1. *AssertValueInterval()* și *assertSumEqualZero()* (corespunzătoare primei etape) determină caracteristicile vectorului.
2. Iterarea prin vector asigură o acoperire cât mai mare a valorilor posibile, specific celei de-a doua etape
3. Ultima etapă este implementată prin metoda saveFailureSeed care păstrează valorile testelor care au eșuat pentru a fi reverificate

```java
@Test
private void testFixedCharacteristics(){
  int n = 2;

  //First set of the input domain.
  //Number of repetitions.
  int numberRepetitions = 100;
  for(int i=1;i<=numberRepetitions;i++){
    //New seed is introduced based on the computer clock.
    long new_seed = System.currentTimeMillis();
    Random rSeed = new Random(new_{s}eed);
    int[] result =
        ArrayFactory.generateArrayBasedRandomSeedWithSumZero
        (rSeed, n);

    if(assertValueInterval(result, n)==false){
    saveFailureSeed(new_{s}eed);
    };
    if(assertSumEqualZero(result, n){
      saveFailureSeed(new_{s}eed);
    };
  }

  //Second set of the input domain.
  long[] allSeeds = getAllFailuresSeeds();
  for(int i=0;i<=allSeeds.length-1;i++){


    //Get failed seed.
    long fail_{s}eed = allSeeds[i];

    //A pseudo-random input is generated based on failed seed
    Random rSeed = new Random(fail_{s}eed);
    int[] result =
        ArrayFactory.generateArrayBasedRandomSeedWithSumZero
        (rSeed, n);

    assertValueInterval(result, n);
    assertSumEqualZero(result, n);
  }
}

private void assertValueInterval(int[] arr, int arraySize) {
  int result = 0;
  //verify if value are between -10 and 10
  for (int i = 0; i < arraySize; i++) {
    assertTrue(arr[i] >= -10 && arr[i] <= 10);
  }
}

private void assertSumEqualZero(int[] arr, int arraySize) {
  int result = 0;
  //verify if sum equals 0
  for (int i = 0; i < arraySize; i++) {
    result = result + arr[i];
  }

  //verify the sum
  assertEquals(0, result);
}
```

Capitolul 6 – ReTest: framework de testat algoritmi randomizati

Capitolul este dedicat descrierii și evaluării framework-ului ReTest, un instrument dezvoltat pentru a facilita utilizarea TDD (Test-Driven Development) în contextul algoritmilor randomizați. Acest framework este conceput pentru a ajuta dezvoltatorii să gestioneze seed-urile utilizate pentru generarea datelor aleatoare în testele lor, permițând astfel repetarea testelor cu scenarii specifice care au eșuat anterior, pentru a verifica corectitudinea soluțiilor adoptate după modificările codului.

ReTest este integrat cu JUnit și permite dezvoltatorilor să scrie teste care pot fi executate multiple ori cu diferite seed-uri aleatoare. Aceasta oferă un control mai bun asupra procesului de testare și asigură că testele pot acoperi o gamă mai largă de scenarii posibile, ceea ce este crucial pentru algoritmi care depind de decizii aleatoare.

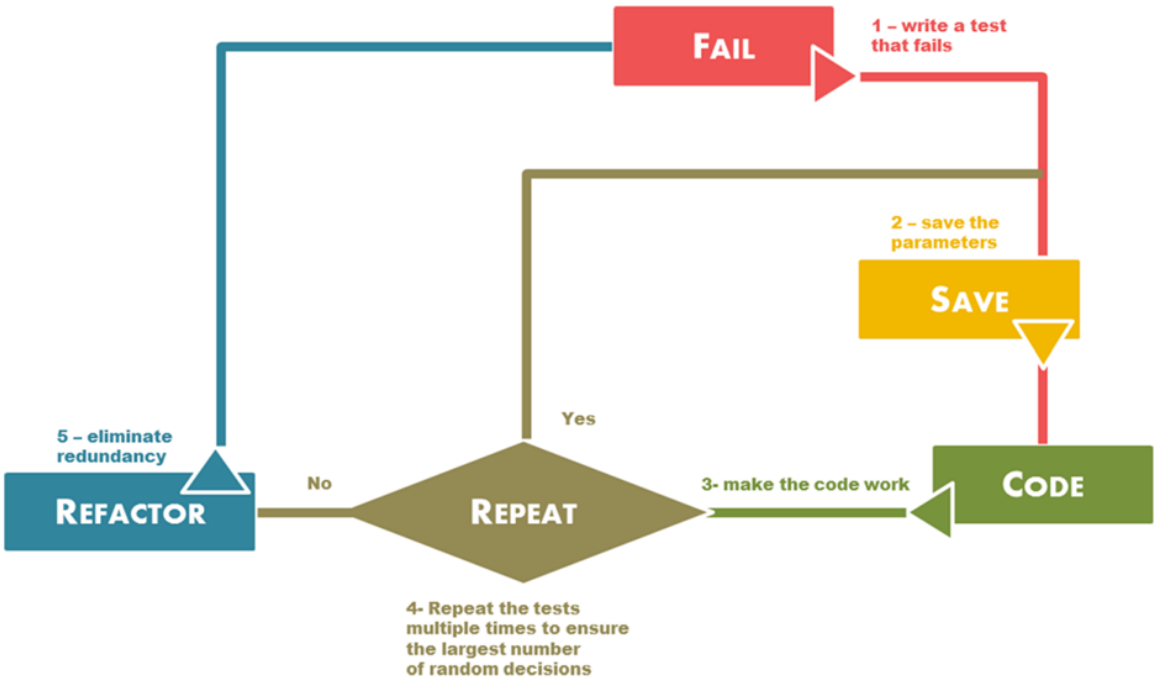

Fig. 6 Adaptation of TDD to ReTest

Un aspect central al cadrului ReTest este setul de functii pe care le introduce pentru a simplifica scrierea testelor. Printre acestea se numără:

- @ReTest: specifică numărul de repetări ale unui test cu diferite seed-uri.
- @SaveBrokenTestDataFiles și @LoadTestFromDataFiles: permit salvarea scenariilor de test eșuate pentru a fi reutilizate în testările viitoare, facilitând astfel testele de regresie.

Utilizarea framework-ului ReTest este exemplificată printr-un caz în care un dezvoltator trebuie să testeze un algoritm care generează un array de numere aleatoare cu suma totală zero. Prin utilizarea ReTest, dezvoltatorul poate configura testul să ruleze multiple ori automat, salvând scenariile care eșuează pentru analiză și corecții ulterioare. Acest lucru este deosebit de util în cazul dezvoltării de software, unde comportamentul non-determinist al algoritmilor poate face dificilă identificarea și reproducerea erorilor.

Capitolul subliniază importanța unui instrument ca ReTest în modernizarea practicilor de testare pentru software-ul care utilizează algoritmi randomizați. Prin automatizarea aspectelor repetitive ale testării și prin facilitarea testării de regresie, ReTest ajută la îmbunătățirea calității software-ului și la eficientizarea procesului de dezvoltare.

Capitolul 7 – ReTest framework: experienta dezvoltatorilor

Acest capitol prezintă un studiu realizat pentru a evalua experiența dezvoltatorilor de software în utilizarea abordării TDD în dezvoltarea algoritmilor randomizați, sprijinită de framework-ul ReTest, în cadrul cursului de proiectare a software-ului la INPE.

În acest sens, au fost investigate întrebările de cercetare privind eficiența aplicării abordării propuse de către dezvoltatorii mai puțin experimentați, principalele dificultăți întâmpinate, avantajele și dezavantajele acestei abordări, precum și utilitatea percepută de către dezvoltatori.

Zece participanți au fost selectați dintre studenții înscriși la cursurile de proiectare software la INPE. Pentru a participa, aceștia trebuiau să aibă experiență în programare în Java și să cunoască adnotările de cod. Acestia au fost împărțiți în două grupuri: cei cu experiență sub 10 ani și cei cu experiență de 10 ani sau mai mult. În ceea ce privește TDD-ul, majoritatea aveau experiență, dar toți au primit pregătire suplimentară în cadrul cursului, constând în 16 ore de teorie și exerciții practice.

Pentru a pregăti activitățile studiului, au fost prevazute două cerințe cheie: (a) un algoritm ușor de înțeles și de dezvoltat; și (b) asigurarea faptului că toți participanții execută sarcina bazată pe conceptele TDD. După perioada de pregătire, participanții au avut două săptămâni pentru a realiza sarcina de implementare a unui algoritm randomizat.

Sarcina de implementare se baza pe faptul ca participanții trebuiau să dezvolte o funcție folosind TDD și abordarea propusă cu framework-ul ReTest, care să genereze un tablou de "n" poziții, cu numere aleatoare între -10 și 10, având suma totală a elementelor mereu egală cu zero (0). Pentru a asigura respectarea pașilor minini TDD, participanților li s-a prezentat un ghid simplu, incluzând teste inițiale, implementarea metodei, introducerea testelor suplimentare și actualizarea implementării în funcție de acestea.

Rezultatele arata ca desi timpul de finalizare a fost puțin mai scurt pentru dezvoltatorii mai experimentați, analiza statistică prin testul t-Test nu a relevat diferențe semnificative între grupuri. Acest lucru sugerează că utilizarea framework-ului ReTest nu necesită o experiență extensivă în programare, deoarece toți participanții au finalizat cu succes sarcinile lor, indiferent de nivelul de experiență.

În ceea ce privește ușurința de utilizare, 60% din participanți au declarat că framework-ul a fost ușor de utilizat, în timp ce 40% dintre aceștia au indicat o dificultate medie în utilizarea acestuia. În plus, cinci participanți au raportat diverse dificultăți. Rapoartele lor au fost analizate și dificultățile lor au fost clasificate ca fiind legate de framework, de algoritm sau de mediul de dezvoltare integrat (IDE).

Majoritatea participantilor apreciaza posibilitatea de a efectua teste de regresie pe algoritmi randomizați (90%). În ceea ce privește dezavantajele, doar doi participanți au raportat dificultăți, inclusiv dificultatea în depanarea codului și creșterea timpului de rulare a testelor.

La finalul chestionarului, participanții au fost întrebați despre utilitatea framework-ului. Deși 40% dintre participanți cred că ar putea realiza sarcina propusă fără ajutorul ReTest, 90% au afirmat că framework-ul i-a ajutat foarte mult în sarcina de implementare. Unii au evidențiat, de asemenea, utilitatea framework-ului pentru alte tipuri de algoritmi.

Capitolul 8 – Discutii

Până în prezent, au fost realizate două studii pentru a evalua aceasta abordare. În primul studiu, am evaluat fezabilitatea acesteia printr-un experiment cu un singur subiect, care a implicat un algoritm randomizat pentru crearea și evoluția graficelor dinamice.

În cadrul experimentului cu un singur subiect, principalele provocări și limitări sunt asociate cu amenințările interne și externe la validitate. Amenințările la validitatea internă privesc analiza dacă tratamentul cauzează efectul. Amenințările identificate în acest studiu se referă la lipsa analizei statistice și la efectul învățării subiecților în experiment.

Analiza statistică nu se aplică de obicei în experimentele cu un singur subiect din cauza lipsei replicărilor. Cu toate acestea, un design A-B-A ar putea oferi dovezi mai puternice decât un design A-B. În plus, rezultatele experimentului ar putea fi afectate de efectul învățării, deoarece participantul a implementat același algoritm de două ori, iar în cea de-a doua oară problema era deja familiară.

Amenințările la validitatea externă sunt legate de generalizarea rezultatelor. Efectuarea unui experiment cu un singur subiect ne-a permis să evaluăm abordarea noastră în fața unei probleme reale și complexe din domeniul învățării automate.

În al doilea studiu, a fost evaluata experiența dezvoltatorilor în utilizarea framework-ului ReTest. Utilizarea corectă a framework-ului necesită cunoștințe în aplicarea TDD și a modelelor de proiectare, iar aceste cerințe au fost îndeplinite prin pregătirea efectuată. Prin acest studiu, am constatat că majoritatea dezvoltatorilor au recunoscut utilitatea framework-ului, raportând mai multe beneficii decât dezavantaje.

Capitolul 9 – Concluzii

Acest articol prezintă o abordare care permite utilizarea TDD în dezvoltarea și proiectarea aplicațiilor care implică algoritmi randomizați, în special atunci când sunt necesare mai multe alegeri aleatorii în timpul execuției. Abordarea propusă extinde TDD-ul pentru a permite aplicarea sa în algoritmi cu caracteristici non-deterministe, bazată pe un set de modele de software pentru a ghida aplicarea acestei abordări. De asemenea, a fost dezvoltat un framework de testare numit ReTest pentru a susține astfel de algoritmi în limbajul de programare Java.

Două studii au fost realizate pentru a evalua fezabilitatea abordării propuse aplicate într-o implementare a algoritmului care implică mai multe decizii aleatorii în secvență, și pentru a evalua utilizarea framework-ului ReTest din punctul de vedere al dezvoltatorilor.

În primul studiu, rezultatele experimentului cu un singur subiect sugerează că este posibil să se găsească mai multe bug-uri cu abordarea propusă, deoarece modelele lor permit utilizarea TDD în algoritmi randomizați sau non-determiniști, ceva ce până acum nu fusese utilizat. În al doilea studiu, rezultatele sugerează că framework-ul ReTest este ușor de utilizat și util în opinia majorității participanților care l-au utilizat. În ciuda dificultăților, toți participanții au reușit să finalizeze sarcina folosind abordarea propusă pe baza documentației furnizate.

Este important să subliniem că rezultatele ambelor studii nu sunt concluzii definitive din cauza limitărilor și amenințărilor la validitate. Până în prezent, rezultatele au fost satisfăcătoare și au indicat că strategia este destul de promițătoare. Cu toate acestea, sunt necesare investigații suplimentare pentru a verifica dacă astfel de constatări pot fi extinse la cazurile care implică (i) sarcini mai complexe care reproduc mai bine nevoile și realitatea dezvoltatorilor; (ii) un alt tip de algoritmi randomizați în alte domenii de cercetare; și (iii) măsurarea variabilelor obiective pentru a evalua mai bine framework-ul ReTest.