Universitatea Tehnică a Moldovei

Facultatea Calculatoare Informatică şi Microelectronică

Departamentul Ingineria Software și Automatică

**Referat**

La disciplina „Testarea Produselor Program”

Tema: “**Testarea Unitară**”

A efectuat: st. gr. SI-211 A. Chihai

A verificat: dr. conf.univ. A. Prisacaru

**Chișinău – 2024**

Cuprins

[Introducere 3](#_Toc166867542)

[1. Conceptele de bază ale testării unitare 4](#_Toc166867543)

[2. Scopul și beneficiile testării unitare 5](#_Toc166867544)

[3. Implementarea testelor unitare în procesul de dezvoltarea software 6](#_Toc166867545)

[4. Impactul negativ al lipsei dezvoltării bazate pe teste (TDD) în dezvoltarea software 7](#_Toc166867546)

[5. Exemplu practice de testare unitară 8](#_Toc166867547)

[Concluzie 12](#_Toc166867548)

# Introducere

Testarea unitară reprezintă un pilon fundamental în dezvoltarea software-ului modern, jucând un rol crucial în asigurarea calității și fiabilității acestuia. O testare unitară bine realizată poate identifica erori și probleme în codul aplicației încă de la începutul ciclului de dezvoltare, ceea ce duce la reducerea costurilor și a timpului necesar pentru remedierea acestora. În acest context, testarea unitară devine esențială pentru echipele de dezvoltare care urmăresc să livreze produse software de înaltă calitate, într-un mod eficient și sustenabil.

Conceptul de testare unitară se bazează pe ideea de a testa fiecare componentă individuală a unei aplicații software în izolare, pentru a verifica dacă funcționează conform specificațiilor și așteptărilor. Aceste componente, cunoscute sub numele de "unități", pot fi funcții, clase sau module care îndeplinesc o anumită funcționalitate în cadrul aplicației. Prin testarea fiecărei unități în mod separat, dezvoltatorii pot identifica și corecta erori la nivel de unitate, fără a afecta alte părți ale aplicației.

Există mai multe beneficii ale testării unitare în dezvoltarea software-ului. În primul rând, testarea unitară contribuie la creșterea calității codului, deoarece ajută la identificarea și remedierea erorilor într-un stadiu incipient al dezvoltării. Acest lucru duce la reducerea numărului de erori și la creșterea stabilității și fiabilității aplicației. În plus, testarea unitară facilitează refactoring-ul și îmbunătățirea continuă a codului, deoarece oferă o siguranță că modificările efectuate nu vor introduce erori în funcționalitatea existentă.

Un alt avantaj al testării unitare este reprezentat de eficiența în timp și resurse pe care o aduce în procesul de dezvoltare. Identificarea și remedierea erorilor într-un stadiu incipient al dezvoltării poate economisi timp și resurse considerabile care altfel ar fi necesare pentru diagnosticarea și repararea acestora în etapele ulterioare ale proiectului. De asemenea, testarea unitară poate îmbunătăți comunicarea și colaborarea în cadrul echipei de dezvoltare, deoarece oferă un set comun de criterii pentru evaluarea calității codului și a funcționalității aplicației.

În concluzie, testarea unitară reprezintă o practică esențială în dezvoltarea software-ului modern, contribuind la îmbunătățirea calității, fiabilității și eficienței procesului de dezvoltare. Acest raport explorează în detaliu conceptele, beneficiile și practicile recomandate în domeniul testării unitare, oferind o perspectivă comprehensivă asupra acestui aspect critic al dezvoltării software-ului.

# Conceptele de bază ale testării unitare

Testarea unitară reprezintă o metodă de testare a componentelor individuale (unități) ale unei aplicații software pentru a verifica dacă acestea funcționează conform specificațiilor și așteptărilor. Pentru a înțelege mai bine testarea unitară, este important să explorăm câteva concepte de bază care stau la baza acestei practici esențiale în dezvoltarea software-ului.

* **Unitatea de testare**

Unitatea de testare reprezintă cea mai mică parte a unei aplicații software care poate fi testată în mod izolat. Aceasta poate fi o funcție, o metodă, o clasă sau chiar un modul întreg, în funcție de nivelul de granularitate dorit pentru testare. Scopul testării unitare este de a verifica fiecare unitate în parte, pentru a asigura că aceasta funcționează corect și produce rezultatele așteptate.

* **Fixture**

Fixture reprezintă un set predefinit de date de intrare și condiții inițiale necesare pentru a putea efectua un test unitar. Aceste date și condiții sunt folosite pentru a pregăti mediul de testare înainte de rularea testului propriu-zis. Fixture-ul asigură un context controlat pentru testare și ajută la reproducerea testelor în mod consistent.

* **Aserțiuni (assertions)**

Aserțiunile reprezintă afirmații care verifică dacă rezultatul obținut în urma rulării unui test unitar corespunde cu rezultatul așteptat. Aserțiunile sunt esențiale în testarea unitară, deoarece permit validarea comportamentului unităților testate și identificarea eventualelor erori sau devieri.

* **Dubluri de testare (test doubles)**

Dublurile de testare sunt obiecte care înlocuiesc alte obiecte din aplicație în timpul testării unitare, pentru a simula comportamentul acestora și a izola unitatea de testat. Există mai multe tipuri de dubluri de testare, cum ar fi stub-urile, mock-urile și spy-urile, fiecare având un rol specific în simularea interacțiunilor cu alte componente.

* **Test-driven development (TDD)**

Test-driven development este o metodologie de dezvoltare software care se bazează pe scrierea testelor unitare înainte de implementarea efectivă a funcționalității. Acest proces inversează paradigma tradițională a dezvoltării software, punând accentul pe testare și validare încă de la începutul dezvoltării unei aplicații.

* **Integrare continuă (continuous integration)**

Integrarea continuă este o practică în dezvoltarea software-ului care presupune integrarea și testarea automată a codului în mod regulat, de obicei de mai multe ori pe zi. Scopul integrării continue este de a identifica și rezolva conflicte și erori într-un stadiu incipient al dezvoltării, facilitând astfel livrarea frecventă și consistentă a produsului software.

Aceste concepte de bază reprezintă fundamentul testării unitare și sunt esențiale pentru înțelegerea și aplicarea corectă a acestei practici în dezvoltarea software-ului modern. Prin utilizarea lor adecvată, dezvoltatorii pot asigura calitatea și fiabilitatea aplicațiilor lor, contribuind la succesul și satisfacția utilizatorilor finali.

# Scopul și beneficiile testării unitare

Scopul testării unitare este de a asigura că fiecare unitate individuală a codului sursă funcționează corect și produce rezultatele așteptate. Testarea unitară vizează identificarea erorilor și a problemelor într-un stadiu incipient al dezvoltării, reducând astfel costurile și eforturile necesare pentru remedierea acestora mai târziu în ciclul de dezvoltare. Prin testarea fiecărei unități în mod izolat, dezvoltatorii pot verifica corectitudinea și fiabilitatea fiecărei componente a aplicației, asigurând astfel calitatea și stabilitatea întregului sistem.

Testarea unitară reprezintă un element esențial al procesului de dezvoltare software, oferind numeroase beneficii care contribuie la îmbunătățirea calității, eficienței și fiabilității aplicațiilor. Această practică presupune testarea fiecărei unități individuale ale codului sursă în mod izolat, pentru a verifica corectitudinea și funcționalitatea acestora conform specificațiilor. Iată câteva dintre cele mai importante beneficii ale testării unitare:

* **Identificarea și prevenirea erorilor în stadii incipiente:**

Una dintre cele mai mari avantaje ale testării unitare este capacitatea sa de a identifica și corecta erori în codul sursă încă din faza de dezvoltare. Prin testarea fiecărei unități în mod izolat, dezvoltatorii pot identifica și remedia problemele într-un stadiu incipient, evitând astfel costurile și complexitatea remedierii acestora în etapele ulterioare ale proiectului.

* **Îmbunătățirea calității codului:**

Testarea unitară promovează scrierea unui cod mai clar, mai modular și mai ușor de întreținut. Dezvoltatorii sunt încurajați să își îmbunătățească practicile de programare, ceea ce duce la creșterea calității și stabilității aplicației.

* **Facilitarea refactoring-ului:**

Refactoring-ul reprezintă o practică esențială în dezvoltarea software-ului, care presupune îmbunătățirea continuă a structurii și calității codului fără a afecta funcționalitatea acestuia. Testele unitare oferă o siguranță că modificările efectuate în timpul refactoring-ului nu vor introduce erori în funcționalitatea existentă.

* **Economisirea timpului și a resurselor:**

Deși testarea unitară poate părea consumatoare de timp la început, ea poate economisi resurse considerabile pe termen lung. Identificarea și remedierea erorilor într-un stadiu incipient al dezvoltării poate evita problemele complexe și costisitoare care ar putea apărea mai târziu în proces.

* **Creșterea încrederii în codul sursă:**

Având un set complet de teste unitare care acoperă diferite scenarii și funcționalități, dezvoltatorii pot avea încredere că modificările aduse codului nu vor afecta funcționalitatea existentă. Acest lucru crește nivelul de încredere în calitatea și fiabilitatea codului sursă.

* **Facilitarea colaborării în echipă:**

Testarea unitară încurajează o colaborare mai strânsă între membrii echipei de dezvoltare. Toți membrii pot contribui la scrierea și întreținerea testelor unitare, asigurând astfel că toată lumea înțelege și respectă standardele de calitate ale proiectului.

* **Suport pentru dezvoltarea bazată pe teste (TDD):**

Testarea unitară este un element central al dezvoltării bazate pe teste (TDD), o metodologie care promovează scrierea testelor înainte de implementarea efectivă a codului. Acest proces inversat încurajează dezvoltatorii să scrie cod mai simplu, mai modular și mai ușor de testat.

# Implementarea testelor unitare în procesul de dezvoltarea software

Testarea unitară reprezintă un aspect crucial al dezvoltării software moderne, având un impact semnificativ asupra calității, fiabilității și eficienței aplicațiilor. Pentru a implementa și gestiona testele unitare în mod eficient în cadrul unui proces de dezvoltare, este necesară o abordare structurată și bine definită. Iată câteva aspecte cheie ale implementării și gestionării testelor unitare:

* **Definirea obiectivelor și a acoperirii testelor:**

Înainte de a începe să scriu teste unitare, este important să definesc obiectivele testelor și să determin care părți ale codului meu sursă trebuie acoperite de acestea. O acoperire cât mai cuprinzătoare a testelor va asigura că toate funcționalitățile și scenariile critice sunt testate corespunzător.

* **Selectarea framework-ului de testare potrivit:**

Există o varietate de framework-uri de testare disponibile pentru diferite limbaje de programare și platforme. Este important să selectez un framework care să se potrivească nevoilor și cerințelor proiectului meu și care să ofere funcționalitățile necesare pentru scrierea și rularea testelor unitare în mod eficient.

* **Scrierea testelor unitare:**

După ce am definit obiectivele și am selectat framework-ul de testare, următorul pas este să scriu teste unitare efective. Testele ar trebui să acopere diferite scenarii și funcționalități ale aplicației mele, verificând atât cazurile de utilizare corecte, cât și cele eronate.

* **Integrarea testelor în procesul de construire automată:**

Pentru a asigura că testele mele sunt rulate în mod regulat și automat, este recomandabil să le integrez în procesul meu de construire automată (CI/CD). Astfel, fiecare modificare a codului meu sursă va fi testată automat, asigurând că nicio modificare nu afectează funcționalitatea existentă.

* **Gestionarea și întreținerea testelor:**

Pe măsură ce proiectul meu evoluează, este important să gestionez și să întrețin testele unitare. Aceasta include actualizarea testelor pentru a reflecta modificările aduse codului meu sursă, rezolvarea problemelor și îmbunătățirea testelor pentru a crește acuratețea și eficiența acestora.

* **Analiza rezultatelor testelor:**

Analiza rezultatelor testelor unitare este esențială pentru identificarea erorilor și a potențialelor probleme în codul meu sursă. Este important să evaluez rezultatele testelor în mod regulat și să iau măsuri pentru remedierea problemelor identificate.

# Impactul negativ al lipsei dezvoltării bazate pe teste (TDD) în dezvoltarea software

Lipsa utilizării TDD poate avea un impact semnificativ asupra procesului de dezvoltare și asupra calității produsului final. În absența TDD, există riscul de a scrie cod care poate fi mai dificil de testat, mai puțin flexibil și mai susceptibil la erori. Aceste probleme pot duce la o scădere a calității și fiabilității aplicației, precum și la o creștere a costurilor și eforturilor necesare pentru remedierea erorilor și gestionarea schimbărilor.

Dacă nu folosim dezvoltarea bazată pe teste (TDD), putem întâmpina mai multe probleme și dificultăți în procesul de dezvoltare a software-ului. Iată câteva dintre acestea:

* **Calitatea codului:** Fără TDD, există o tendință de a scrie cod care poate fi mai dificil de testat și mai susceptibil la erori. Lipsa testelor scrise în prealabil poate duce la o acoperire inadecvată a scenariilor de testare și la introducerea unor bug-uri neașteptate.
* **Reacția la schimbări:** TDD promovează scrierea de cod modular și flexibil, care poate fi ușor modificat și extins. Fără TDD, codul poate deveni rigid și greu de modificat, ceea ce face dificilă gestionarea schimbărilor și adaptarea la cerințele în continuă schimbare ale proiectului.
* **Refactoring-ul:** TDD facilitează refactoring-ul continuu al codului pentru a îmbunătăți structura și claritatea acestuia. Fără testele unitare scrise în prealabil, refactoring-ul poate fi mai dificil și mai riscant, deoarece modificările aduse pot afecta funcționalitatea existentă fără a fi detectate.
* **Încrederea în cod:** TDD oferă o încredere crescută în codul sursă, deoarece fiecare modificare este validată printr-un set complet de teste. Fără TDD, încrederea în calitatea și fiabilitatea codului poate fi mai scăzută, ceea ce poate duce la mai mult timp și resurse necesare pentru testare manuală și depanare ulterioară.
* **Complexitatea testării ulterioare:** Fără TDD, testarea ulterioară a codului poate fi mai dificilă și mai laborioasă, deoarece trebuie să scriem teste pentru un cod care nu a fost proiectat inițial pentru testabilitate. Acest lucru poate duce la un proces de testare ineficient și la creșterea riscului de a introduce erori în timpul testării.
* **Dificultăți în identificarea și izolarea erorilor:** Lipsa testelor unitare poate face mai dificilă identificarea și izolarea erorilor în codul sursă. Fără un set cuprinzător de teste, identificarea cauzelor erorilor și reproducerea acestora poate fi o sarcină laborioasă și consumatoare de timp.
* **Costuri mai mari și întârzieri în livrare:** Erorile descoperite într-un stadiu avansat al dezvoltării pot necesita eforturi semnificative pentru remediere, ceea ce poate duce la întârzieri în livrarea produsului și la creșterea costurilor de dezvoltare.
* **Impact negativ asupra experienței utilizatorilor:** Erorile și problemele descoperite în produsul final pot avea un impact negativ asupra experienței utilizatorilor și pot afecta reputația și imaginea companiei pe piață.
* **Lipsa încrederii în calitatea produsului:** Lipsa unui proces robust de testare poate afecta încrederea clienților și utilizatorilor în calitatea produsului, ceea ce poate afecta adoptarea și succesul acestuia pe piață.
* **Dificultăți în mentenanță și evoluție:** Codul care nu este testat corespunzător poate deveni greu de întreținut și de evoluat pe termen lung, deoarece modificările aduse pot introduce erori neașteptate în funcționalitatea existentă.

# Exemplu practice de testare unitară

Această secțiune prezintă clasa SalaryCalculator concepută pentru a calcula salariul săptămânal al unui angajat, luând în considerare orele minime de lucru, plata pentru orele suplimentare și intrările invalide ale numărului de ore lucrate.

**1. Ore Minime de Lucru:**

Conceptul de ore minime de lucru asigură un venit de bază pentru angajații care lucrează săptămâna standard de lucru. În această implementare, constanta MIN\_HOURS\_PER\_WEEK definește numărul minim de ore necesare pentru a primi salariul de bază complet.

**2. Plata pentru Orele Suplimentare:**

Orele suplimentare se referă la orele lucrate peste programul standard de lucru. Acest calculator include plata pentru orele suplimentare pentru orele care depășesc limita MAX\_HOURS\_PER\_WEEK. Bonusul pentru orele suplimentare este calculat ca 130% din rata orară standard a angajatului.

**3. Gestionarea Orelor Invalide:**

Metoda isValidNrOfHours se asigură că numărul de ore lucrate furnizat se încadrează în intervalul valid (de la 0 la MAX\_HOURS\_PER\_WEEK). Dacă numărul de ore lucrate nu se încadrează în acest interval, metoda calculateSalary aruncă o excepție RuntimeException cu un mesaj corespunzător.

**4. Logica de Calcul al Salariului:**

Metoda calculateSalary efectuează calculul principal al salariului pe baza următoarei logici:

* Inițial verifică validitatea numărului de ore lucrate folosind isValidNrOfHours.
* Dacă orele sunt valide, verifică dacă sunt îndeplinite orele minime de lucru folosind hasMinimumHours.
  + Dacă angajatul îndeplinește orele minime:
    - Salariul de bază este calculat ca MIN\_HOURS\_PER\_WEEK \* hourlyRate.
    - Dacă există ore suplimentare (hasWorkedOvertime), plata pentru orele suplimentare se adaugă la salariul de bază folosind calculateOvertimeSalary.
  + Dacă angajatul lucrează mai puțin decât orele minime, salariul este calculat pur și simplu ca numărul total de ore lucrate înmulțit cu rata orară (workedHours \* hourlyRate).

public class **SalaryCalculator** {  
 private int MIN\_HOURS\_PER\_WEEK = 40;  
 private int MAX\_HOURS\_PER\_WEEK = 55;  
 private double hourlyRate;  
 private double workedHours;  
 private double salary;  
  
 public SalaryCalculator(double hourlyRate, double workedHours) {  
 this.hourlyRate = hourlyRate;  
 this.workedHours = workedHours;  
 }  
  
 public double calculateSalary() {  
 if (!isValidNrOfHours()) {  
 throw new RuntimeException("Invalid number of hours");  
 }  
  
 if (hasMinimumHours()) {  
 salary = MIN\_HOURS\_PER\_WEEK \* hourlyRate;  
  
 if (hasWorkedOvertime()) {  
 salary += calculateOvertimeSalary();  
 }  
  
 } else {  
 salary = workedHours \* hourlyRate;  
 }  
  
 return salary;  
 }  
  
 private double calculateOvertimeSalary() {  
 return (workedHours - MIN\_HOURS\_PER\_WEEK) \* calculateOvertimeBonus();  
 }  
  
 private double calculateOvertimeBonus() {  
 return hourlyRate \* 130 / 100;  
 }  
  
 private boolean hasMinimumHours() {  
 return workedHours >= 40;  
 }  
  
 private boolean hasWorkedOvertime() {  
 return workedHours > 40;  
 }  
  
 private boolean isValidNrOfHours() {  
 return workedHours >= 0 && workedHours <= MAX\_HOURS\_PER\_WEEK;  
 }  
  
  
  
}

În contextul clasei SalaryCalculator, testarea unitară oferă mai multe beneficii:

**1. Detectarea erorilor din stadii incipiente:**

Testarea unitară permite identificarea erorilor logice și de calcul în codul SalaryCalculator la un stadiu incipient al dezvoltării. Prin rularea testelor unitare în timpul procesului de scriere a codului, dezvoltatorii pot depista problemele rapid și le pot remedia înainte ca acestea să afecteze funcționalitatea generală a programului.

**2. Asigurarea calității calculelor salariale:**

Calcularea precisă a salariilor este esențială pentru orice companie. Erorile din codul SalaryCalculator pot duce la plata incorectă a angajaților, ceea ce poate genera nemulțumiri și probleme legale. Testarea unitară ajută la verificarea calculelor pentru diferite scenarii de ore lucrate, sporuri și ore suplimentare, asigurând corectitudinea plăților.

**3. Îmbunătățirea mentenanței codului:**

Pe măsură ce codul evoluează și se adaugă noi funcționalități, testarea unitară existentă ajută la menținerea calității codului SalaryCalculator. Modificările ulterioare nu vor afecta în mod negativ calculele existente, deoarece testele unitare vor detecta orice regresii introduse accidental.

**4. Crește încrederea în cod:**

Testarea unitară riguroasă oferă dezvoltatorilor și utilizatorilor încredere în faptul că SalaryCalculator funcționează corect. Acest lucru este important pentru a evita erorile de calcul salarial care pot avea consecințe financiare și legale semnificative.

**5. Simplifică procesul de depanare:**

Atunci când apar probleme cu calculele salariale, testele unitare pot oferi indicii valoroase pentru identificarea sursei erorii. Dacă un test unitar eșuează, dezvoltatorul poate investiga scenariul specific testat pentru a găsi cauza erorii.

public class **SalaryCalculatorTest** {  
  
 @Test  
 public void testMinHoursPerWeek() {  
 **SalaryCalculator** salaryCalculator = new SalaryCalculator((double) 16, (double) 40);  
 double calculatedSalary = salaryCalculator.calculateSalary();  
 *assertEquals*(640, calculatedSalary, 0.01);  
 }  
  
 @Test  
 public void testOvertimeHours() {  
 **SalaryCalculator** salaryCalculator = new SalaryCalculator(20, 50);  
 double calculatedSalary = salaryCalculator.calculateSalary();  
 *assertEquals*(1060, calculatedSalary, 0.01);  
 }  
  
 @Test  
 public void testInvalidHoursPerWeek\_BellowMin() {  
 **SalaryCalculator** salaryCalculator = new SalaryCalculator(14, -4);  
 try {  
 salaryCalculator.calculateSalary();  
 *fail*("Expected RuntimeException for invalid hours");  
 } catch (**RuntimeException** e) {  
 *assertEquals*("Invalid number of hours", e.getMessage());  
 }  
 }  
  
 @Test  
 public void testInvalidHoursPerWeek\_ExceedMax() {  
 **SalaryCalculator** salaryCalculator = new SalaryCalculator(14, 70);  
 try {  
 salaryCalculator.calculateSalary();  
 *fail*("Expected RuntimeException for invalid hours");  
 } catch (**RuntimeException** e) {  
 *assertEquals*("Invalid number of hours", e.getMessage());  
 }  
 }  
}

# Concluzie

Testarea unitară este o practică esențială în dezvoltarea software-ului modern, oferind numeroase beneficii care contribuie la îmbunătățirea calității, fiabilității și eficienței aplicațiilor. Implementarea testelor unitare în procesul de dezvoltare software poate duce la:

* **Identificarea și remedierea erorilor în stadii incipiente:** Testarea unitară ajută la identificarea erorilor și problemelor în codul sursă încă din faza de dezvoltare, reducând costurile și eforturile necesare pentru remedierea acestora mai târziu în ciclul de dezvoltare.
* **Îmbunătățirea calității codului:** Testarea unitară promovează scrierea unui cod mai clar, mai modular și mai ușor de întreținut. Dezvoltatorii sunt încurajați să își îmbunătățească practicile de programare, ceea ce duce la creșterea calității și stabilității aplicației.
* **Facilitarea refactoring-ului:** Refactoring-ul reprezintă o practică esențială în dezvoltarea software-ului, care presupune îmbunătățirea continuă a structurii și calității codului fără a afecta funcționalitatea acestuia. Testele unitare oferă o siguranță că modificările efectuate în timpul refactoring-ului nu vor introduce erori în funcționalitatea existentă.
* **Economisirea timpului și a resurselor:** Deși testarea unitară poate părea consumatoare de timp la început, ea poate economisi resurse considerabile pe termen lung. Identificarea și remedierea erorilor într-un stadiu incipient al dezvoltării poate evita problemele complexe și costisitoare care ar putea apărea mai târziu în proces.
* **Creșterea încrederii în codul sursă:** Având un set complet de teste unitare care acoperă diferite scenarii și funcționalități, dezvoltatorii pot avea încredere că modificările aduse codului nu vor afecta funcționalitatea existentă. Acest lucru crește nivelul de încredere în calitatea și fiabilitatea codului sursă.
* **Facilitarea colaborării în echipă:** Testarea unitară încurajează o colaborare mai strânsă între membrii echipei de dezvoltare. Toți membrii pot contribui la scrierea și întreținerea testelor unitare, asigurând astfel că toată lumea înțelege și respectă standardele de calitate ale proiectului.
* **Suport pentru dezvoltarea bazată pe teste (TDD):** Testarea unitară este un element central al dezvoltării bazate pe teste (TDD), o metodologie care promovează scrierea testelor înainte de implementarea efectivă a codului. Acest proces inversat încurajează dezvoltatorii să scrie cod mai simplu, mai modular și mai ușor de testat.

Lipsa testelor unitare poate avea un impact semnificativ asupra procesului de dezvoltare și asupra calității produsului final. În absența TDD, există riscul de a scrie cod care poate fi mai dificil de testat, mai puțin flexibil și mai susceptibil la erori.

Pe scurt, testarea unitară este o investiție valoroasă în calitatea și fiabilitatea software-ului. Implementarea testelor unitare în mod eficient poate duce la o serie de beneficii semnificative, inclusiv o reducere a costurilor, o îmbunătățire a calității și o accelerare a timpului de livrare a produsului.