

LUCRARE ȘTIINȚIFICĂ
Testarea programului de gestionare a
unei biblioteci

Autor: Rogojanu Rebeca-Maria
Email: rebeca.rogojanu@student.upt.ro

30 Decembrie 2024

1. INTRODUCERE

În această lucrare am detaliat tehnicile de testare folosite pe un sistem de gestionare a unei bibliotecii. Am realizat diverse teste unitare – Unit Testing asupra clasei „Biblioteca”. Aplicația conține și dependențe pentru clasa ”Biblioteca”: „ICardBiblioteca” și „ICarte”.

Printre testele abordate în testarea software a programului am inclus tehnicile Domain Testing, Test Doubles și am utilizat framework-ul Moq. Așadar în cele urmează va fi prezentat programul de gestionare a bibliotecii, teste la limita pentru funcțiile din cadrul clasei „Biblioteca”, testarea comportamentului codului la excepții, teste prin injectare de parametrii, teste Stub, teste Mock, acoperirea testelor realizare prin analiza cu „Code Coverage” utilizând comanda „coverlet”.

2. PREZENTAREA PROGRAMULUI DE GESTIONARE A UNEI BIBLIOTECI

2.1 Descrierea claselor și a funcționalității

Programul de gestionare a bibliotecii este o aplicație software creată pentru organizarea activităților zilnice desfășurate în cadrul unei biblioteci. Utilă la organizarea resurselor, interacțiunii cu oamenii, oferă servicii de împrumut-retur, achiziționare de cărți și posibilitatea unui card de bibliotecă prin care se ține evidența cărților împrumutate și unor puncte bonus, ce pot fi folosite pentru achiziționarea unei cărți.

2.1.1 Clasa „Biblioteca”

Clasa este implementată în fisierul „Librarie.cs”, utilizează colecția Dictionary<string, ICarte> pentru a gestiona colecția de cărți și resursele din cadrul bibliotecii. Dictionary<TKey, TValue> este o colecție generică care stochează perechi cheie-valoare fără o anumită ordine. Cheia utilizată în cadrul colecției, de tip string, va fi reprezentată de concatenarea dintre titlul cărții și de autor (ex: „string cheie = carte.Titlu + carte.Autor”), deoarece o carte este reprezentată cel mai bine de titlu și autor, existând cărți cu același titlu. Valoarea colecției este un obiect de tip Carte, ce va fi prezentat mai jos.

Clasa are următoarele funcționalități:

- Funcția „AdaugăCarte(ICarte carte)” – permite adăugarea unei cărți noi în cadrul colecției. În cazul în care cantitatea cărții adăugate este zero/negativă

sau daca cartea există în colecție atunci funcția va returna „False”, verificând dacă cheia realizata din titlul și autorul cărții există deja în colecție (ex: `carti.ContainsKey(cheie)`);

- Funcția „`AdaugăCantitate(string cheie, int cantitate)`” – are rolul de a suplimenta numărul de exemplare pentru o carte existentă în bibliotecă. Se verifică în cadrul funcției dacă cantitatea e pozitivă și dacă cartea există.
- Funcția „`ImprumutăCarte(string cheie, ICardBiblioteca card)`” – permite unei cărți existente în bibliotecă, dacă numărul de exemplare este pozitiv, să fie împrumutată. Cartea fiind apoi adăugată în cadrul cardului de bibliotecă al deținătorului, prin apelarea funcției „`CarteImprumutata`” din clasa „`CardBiblioteca`”, pentru a se putea ține evidența și se decrementează cu unu numărul de exemplare pentru acea carte din bibliotecă.
- Funcția „`ReturneazăCarte(string cheie, ICardBiblioteca card)`” – permite unei cărți înregistrate în bibliotecă, să fie returnată. Cartea fiind apoi eliminată din cadrul cardului de bibliotecă al deținătorului, prin apelarea funcției „`ReturneazăCarte`” din clasa „`CardBiblioteca`”, pentru a se putea ține evidența și se incrementează cu unu numărul de exemplare pentru acea carte din bibliotecă.
- Funcția „`CumpărăCarte(string cheie, int numarExemplare, ICardBiblioteca card)`” – realizează cumpărarea cărții din bibliotecă prin decrementarea numărului de exemplare pentru acea carte și apelarea funcției „`AdaugaPuncteBonus()`” din cadrul clasei „`CardBiblioteca`”, în cazul în care condițiile sunt îndeplinite.
- Funcția „`CumparaCarteCuPuncte(string cheie, int pret, ICardBiblioteca card)`” – permite cumpărarea unei cărți cu punctele bonus de pe card dacă valoarea acestora este mai mare sau egală cu prețul cărții înmulțit cu zece, apoi se decrementează numărul de exemplare și se apelează funcția „`ScadePuncteBonus()`” din clasa „`CardBiblioteca`”, dacă condițiile au fost îndeplinite.

2.1.2 Interfața „`ICarte`”

Definiția interfeței se găsește în fișierul „`ICarte.cs`”. Conține trei componente: „`Titlu`” (titlul cărții, de tip `string`), „`Autor`” (autorul cărții, de tip `string`), „`NumarExemplare`” (numărul de exemplare pentru cartea respectivă, de tip `int`), „`AnPublicare`” (anul de publicare al cărții, de tip `uint`), „`Categorie`” (o categorie de încadrare a cărții).

2.1.3 Clasa „Carte”

Clasa „Carte” implementează interfața „ICarte”. Clasa reprezintă o dependență pentru clasa „Biblioteca”. Obiectul de tip „Carte”, ce conține informațiile despre carte este folosit în cadrul colecției din clasa „Biblioteca”, reprezintă resursa din bibliotecă. Proprietățile clasei sunt: „Titlu”, „Autor”, „NumarExemplare”, „AnPublicare”, „Categorie”.

Clasa conține doar un constructor ce primește argumentele necesare pentru a inițializa obiectul carte, atunci când acesta este creat în program.

2.1.4 Interfața „ICardBiblioteca”

Definiția interfeței „ICardBiblioteca” se găsește în fișierul „ICardBiblioteca.cs”. Aceasta e necesară pentru gestionarea cardului de membru al bibliotecii, prin intermediul căruia se realizează împrumutul cărților și strângerea punctelor bonus. Interfața conține următoarele proprietăți: „NumeDetinator” de tip string, „PuncteBonus” de tip intreg și o listă cu cărțile împrumutate „CartiImprumutate”.

Metodele definite în cadrul interfeței „ICardBiblioteca” sunt: „ContineCarte(string titlu, string autor)”, „CarteImprumutata(ICarte carte)”, „ReturneazaCarte(ICarte carte)”, „Puncte()”, „AdaugaPuncteBonus(int puncte)”, „ScadePuncteBonus(int puncte)”.

2.1.5 Clasa „CardBiblioteca”

Clasa „CardBiblioteca” implementează interfața „ICardBiblioteca”, prin care putem crea pentru fiecare membru al bibliotecii un card, utilizat în activitatea de împrumut/retur a unei cărți și pentru a monitoriza punctele bonus, acumulate în urma achiziționării unei cărți, punctele bonus putând fi folosite pentru a cumpăra cărți.

Metodele implementate în cadrul clasei au următoarele funcționalități:

- Funcția „ContineCarte(string titlu, string autor)” – returnează True dacă se găsește cartea dată ca argument, după titlu și autor, în lista de cărți împrumutate („List<ICarte> CartiImprumutate”);
- Funcțiile „CarteImprumutata(ICarte carte)” și „ReturneazaCarte(ICarte carte)” – au rolul de a adăuga, respectiv elimina cartea, dată ca argument, din lista de cărți împrumutate.
- Funcțiile „AdaugaPuncteBonus(int puncte)” și „ScadePuncteBonus(int puncte)” – permit adăugarea, respectiv scăderea, punctelor bonus pentru respectivul card de bibliotecă cu valoarea dată ca argument funcțiilor.

2.2 Dependența claselor

În programul nostru de gestionarea a unei biblioteci, relațiile de dependență între clase sunt esențiale. „Biblioteca” gestionează colecția de cărți, deci depinde direct de clasa „Carte”, utilizând instanțe ale clasei „Carte”. Clasa „Biblioteca” interconectează cu interfața „ICardBiblioteca”, implementată prin clasa „CardBiblioteca”, pentru a gestiona împrumuturile, returnările și punctele bonus acumulate. În momentul în care un membru returnează o carte, „Biblioteca” va actualiza informațiile (lista de cărți împrumutate) din cadrul obiectului de tip „CardBiblioteca”.

Clasa „CardBiblioteca” este legată de clasa „Carte” prin lista de obiecte de tip „Carte”, din cadrul clasei „CardBiblioteca” („List<ICarte> CartiImprumutate”), ce reprezintă cărțile împrumutate de un membru al bibliotecii.

Interacțiunea dintre interfața „ICarte” și „Carte”, la fel ca și în cazul interfeței „ICardBiblioteca” și „CardBiblioteca” este evidentă, interfața definește funcționalitățile și proprietățile pentru o carte, respectiv cu card de biblioteca, iar clasele oferă implementări ale acestor interfețe.

3. TEHNICIILE DE TESTARE UTILIZATE

Unit Testing sau testarea unitară presupune scrierea unui cod de testare, care validează codul aplicației noastre, testarea devenind automată. Aceasta presupune testarea părților mici din codul aplicației, cum ar fi metode și funcții, în izolare de restul codului, fiecare test validează un comportament al aplicației. Scopul este de a asigura că fiecare bucată de cod, componenta (clasa, metoda, funcție) funcționează cum este de așteptat și de a reduce numărul de greșeli pe care le putem face la scrierea codului.

Există mai multe software-uri automate de testare unitară disponibile pentru a ajuta la testarea unitară în testarea software-ului, în cazul aplicației am utilizat NUnit. NUnit este utilizat pe scară largă în cadrul de testare unitară pentru toate limbile .net. Este un instrument open source care permite scrierea manuală a scripturilor. Acceptă teste bazate pe date care pot rula în paralel.

Testele unitare folosesc bucăți de cod care seamănă cu codul din aplicație, în cazul nostru funcțiile din clasa „CardBiblioteca” și clasa „Carte”, dar în realitate folosesc doar la teste. Stub-urile și mock-urile sunt cele mai întâlnite duble de testare.

Am utilizat în cadrul testelor modelul AAA (Arrange-Act-Assert), un model folosit pentru aranjarea codului în metodele de test unitar. Modelul Arrange-Act-Assert, este considerat un model bun în practică, pentru crearea testelor într-un mod mai natural. Ideea

este de a dezvolta un test unitar în trei pași: Arrange (configurarea obiectelor de testare, pregătirea cerințelor pentru test), Act (apelează funcția ce se dorește testată, realizează acțiunea), Assert (verifică rezultatul).

În cazul testelor folosite pe parcursul aplicației am realizat un Arrange în funcția Setup(), în care inițializăm un obiect carte, cardbiblioteca, câteva obiecte de tip carte și adăugăm cărțile în lista de cărți din bibliotecă.

3.1 Domain Testing

Testarea domeniului este un tip de testare în care aplicația este testată pentru a se asigura că nu se acceptă valori nevalide sau în afara intervalului. Este testată ieșirea cu un număr minim de intrări pentru a vedea dacă sistemul acceptă intrarea în intervalul necesar sau nu. White Box Testing este un exemplu perfect de Domain Testing. Limitele pentru valori, obiecte, metode, definesc un domeniu.

Prin aceste teste analizăm fiecare punct din apropierea acestor limite, atât valorile de intrare valide, cât și cele invalide și un punct din interiorul intervalului. Tesăm orice domeniu care conține funcționalități de intrare și ieșire, introducem valoarea de intrare (de testare) și verificăm rezultatul.

În fișierul „Domain_UnitTesting.cs” se găsesc în total treizeci și unu de teste. Teste care verifică limitele și excepțiile fiecărei funcții.

Inițial am inițializat obiectul bibliotecă și restul obiectelor de care avem nevoie, cum am precizat mai sus. Urmând să creăm teste pentru fiecare funcție.

În cazul funcției „AdaugaCarte()” am realizat patru teste. Primul test, verifică cu valori valide dacă funcția adaugă cartea în lista de cărți și dacă cantitatea introdusă de noi pentru cartea testată este aceeași cu cea memorată pentru obiectul adăugat în listă. Al doilea test, verifică dacă la introducerea unei cărți deja existente în bibliotecă primim eroare, deoarece acea carte există. Al treilea test și al patrulea test verifică dacă în cazul unei cărți noi care are cantitatea zero, respectiv o valoare negativă, va returna false, adică că cartea nu a fost introdusă în bibliotecă, din cauza că nu avem cantitate mai mare ca zero. După aceste teste putem trage concluzia că funcția funcționează cum este așteptat.

Pentru funcția „AdaugăCantitate” am realizat 6 teste. Primul test, verifică cu valori valide, care nu sunt la limită, dacă funcția adaugă la cantitatea inițială, a unei anumite cărți există în bibliotecă, numărul de cărți cu care a fost suplimentat stocul. Al doilea și al treilea test verifică dacă pentru zero sau pentru cantitate negativă, nu se modifică cantitatea și se returnează excepția „Cantitatea trebuie să fie pozitivă”. Al patrulea verifică o valoare validă la limită, iar al cincelea test verifică dacă în cazul în care cartea dată ca input nu există, nu se modifică nimic și se returnează excepția „Cartea specificată nu există în bibliotecă”.

```
[Test]
[Category("Pass/Fail")]
[TestCase("Ion", "Liviu Rebreanu", 5)]
[TestCase("Casa Bantuita", "Shirley Jackson", -5)]
[TestCase("Casa Bantuita", "Shirley Jackson", 0)]
[TestCase("Casa Bantuita", "Shirley Jackson", 1)]
[TestCase("Napasta", "I. L. Caragiale", 3)]
0 | 0 references
public void AdaugaCantitateCarte(string titlu, string autor, int cantitate)
{
    cheie = titlu + autor;
    int cantitate_initiala = biblioteca.carti[cheie].NumarExemplare;
    //act
    biblioteca.AdaugaCantitate(cheie, cantitate);

    //assert
    Assert.AreEqual(cantitate_initiala + cantitate, biblioteca.carti[cheie].NumarExemplare);
}
```

Figura 1 - Testele pentru funcția „AdaugaCantitate()”

Pentru funcția „CarteaExista()”, care are ca input doar cheia, formată din titlu și autor, avem doar două teste. Un test în care cheia există în bibliotecă și returnează – true și un test în care cheia testată e invalidă și se returnează – false, funcția având comportamentul așteptat.

```
/*Testarea functiei CarteaExista()*/
[Test]
[TestCase("Casa BantuitaShirley Jackson")]
[TestCase("NapastaI. L. Caragiale")]
0 | 0 references
public void VerificareCarteaExista(string cheie) {
    bool val = biblioteca.CarteaExista(cheie);
    Assert.IsTrue(val);
}
```

Figura 2 - Testele pentru funcția „CarteaExista()”

Pentru funcția „ImprumutăCarte()” a fost nevoie de 4 teste. Un test în care se verifică dacă se poate împrumuta o carte existentă în bibliotecă, pentru care cantitatea nu era la limită și verificăm dacă cantitatea a fost decrementată și dacă cartea împrumutată a fost adăugată în lista de cărți împrumutate pentru acel card de membru. Al doilea test verifică același comportament pentru o valoare validă la limită. Al treilea test verifică pentru o carte neînregistrată în bibliotecă și se așteaptă să nu se facă modificări, iar la patrulea test verifică dacă o carte validă cu stocul zero, se poate împrumuta. După aceste teste s-a observat că funcția are comportamentul așteptat.

Funcția „ReturneazăCarte()” are o implementare asemănătoare cu funcția „ImprumutaCarte()”, dar necesită mai puține teste. Testăm dacă pentru o carte existentă în bibliotecă se poate realiza returul, numărul de exemplare pentru acea carte se incrementează și cartea se elimină din lista de cărți împrumutate. Al doilea test verifică că nu se poate realiza returul pentru o carte neînregistrată în bibliotecă.

Pentru funcția „CumparaCarte()” am realizat opt teste, care verifică excepțiile și limitele. Primul test verifică excepția în cazul în care cartea dată ca input e invalidă, nu se găsește în bibliotecă. Al doilea și al treilea test verifică dacă pentru un număr negativ, respectiv zero, de cărți ce se dorește a fi cumpărat nu se realizează cumpărarea și se aruncă excepția „Numărul de exemplare specificat este invalid”. Al patrulea test verifică pentru o cantitate introdusă ca input, mai mare decât numărul de exemplare pentru cartea respectivă, nu se realizează modificarea și se aruncă excepția „Nu sunt destule produse in stoc”. Iar al cincelea test verifică valoarea nevalidă, la limita superioară.

```
[Test]
[Category("Error Return")]
[TestCase("AnaEm Sava", 3)]
[TestCase("IonLiviu Rebreanu", -1)]
[TestCase("IonLiviu Rebreanu", 0)]
[TestCase("IonLiviu Rebreanu", 15)]
[TestCase("IonLiviu Rebreanu", 11)]
0 | 0 references
public void CarteCumparata_Error(string cheie, int cantitate)
{
    //act
    biblioteca.CumparaCarte(cheie, cantitate, cardBiblioteca);

    //assert
    Assert.IsFalse(biblioteca.carti.ContainsKey(cheie));
}
```

Figura 3 - Testele „False” pentru funcția „CarteCumparata()”

Următoarele trei teste verifică valori valide, la limită pentru o carte existentă: „1” limita inferioară acceptată, valoarea superioară acceptată și am verificat valorarea superioară validă minus unu, testând astfel o valoare din interval. După aceste teste putem concluziona că funcția are comportamentul așteptat.

```
[Test]
[Category("Pass")]
[TestCase("IonLiviu Rebreanu", 1)]
[TestCase("IonLiviu Rebreanu", 9)]
[TestCase("IonLiviu Rebreanu", 10)]
0 | 0 references
public void CarteCumparata(string cheie, int cant)
{
    int cantitate_initala = biblioteca.carti[cheie].NumarExemplare;
    //act
    biblioteca.CumparaCarte(cheie, cant, cardBiblioteca);

    //assert
    Assert.IsTrue(biblioteca.carti.ContainsKey(cheie));
    Assert.AreEqual(cantitate_initala - cant, biblioteca.carti[cheie].NumarExemplare);
    Assert.AreEqual(5 * cant, cardBiblioteca.Puncte());
}
```

Figura 4 – Testele „Pass” pentru funcția „CarteCumparata()”

Pentru ultima funcție „CumparaCarteCuPuncte()” am realizat șase teste. Un test verifică dacă pentru o carte invalidă se poate realiza acțiunea, dar se aruncă excepția „Cartea specificata nu exista în bibliotecă”. Al doilea test verifică dacă pentru o valoare validă, la limită inferioară de puncte necesare se realizează acțiunea de cumpărare. Al treilea test verifică dacă pentru o valoare mai mare decât limita inferioară se realizează acțiunea de cumpărare și de decrementare a numărului de exemplare din bibliotecă. Al patrulea test, verifică dacă pentru o valoare invalidă la limită (limita inferioară validă - 1) se realizează acțiunea și se aruncă excepția „Nu aveti suficiente puncte bonus pentru a cumpara aceasta carte”. Următorul test verifică dacă pentru o carte înregistrată în bibliotecă și puncte bonus suficiente, se realizează acțiunea dacă numărul de cărți este zero. Ultimul test verifică dacă se realizează acțiunea de cumpărare pentru o carte validă, cu număr de puncte suficient, dar numărul de exemplare fiind la limita validă. În urma acestor teste putem observa ca funcția funcționează cum este de așteptat.

```
[Test]
[TestCase("AnaEm Sava", 350, 30)]
[TestCase("IonLiviu Rebreanu", 150, 15)]
[TestCase("IonLiviu Rebreanu", 200, 15)]
[TestCase("IonLiviu Rebreanu", 149, 15)]
[TestCase("FluturiIrina Binder", 355, 20)]
[TestCase("De veghe in lanul de secaraJ.D.Salinger", 350, 15)]
0 references
public void CarteCumparataCuPuncte(string cheie, int puncte, int pret)
{
    CardBiblioteca card = new CardBiblioteca("Mara");
    card.PuncteBonus = puncte;
    int cantitate_initiala = biblioteca.carti[cheie].NumarExemplare;
    //act
    biblioteca.CumparaCarteCuPuncte(cheie, pret, card);

    //assert
    Assert.AreEqual(cantitate_initiala - 1, biblioteca.carti[cheie].NumarExemplare);
    Assert.AreEqual(puncte - pret * 10, card.PuncteBonus);
}
```

Figura 5 – Testele funcției „CarteCumparataCuPuncte()”

Putem observa că în urma testelor de mai sus, a fost testată funcționalitatea, comportamentul fiecărei funcții din clasa „Biblioteca” luându-se în considerare limitele, testarea valorilor valide din domeniu și testarea excepțiilor.

3.2 Test Doubles

Test Doubles presupune înlocuirea dependențelor pe care le are System Under Test-ul nostru, atunci când dorim să testăm în izolare. System Under Test este soft-ul pe care-l testăm. Printre tipurile de teste doubles în aplicația noastră am utilizat Stub și Mock.

Aceste teste sunt importante în testarea unitare pentru izolarea bucăților de cod testate și pentru verificarea că acestea interacționează corect cu dependențele sale.

3.2.1 Stub Testing

Prin Stub Testing, putem testa SUT-ul în izolare, simulând dependențele. O interfață stub simulează un obiect real folosind câteva metode. Așadar prin acest tip de testare înlocuim anumite părți ale aplicației care sunt greu de configurat sau de testat în izolare, deci Stub returnează răspunsuri predefinite la apeluri de metode utilizate.

În aplicația noastră folosim Stub care implementează interfețele „ICarte” și „ICardBiblioteca”, așadar folosim o altă clasă care seamănă cu clasa originală (dependința), cu scopul de a simplifica obiectul „Carte” și „CardBiblioteca” pentru testare.

Testele Stub se găsesc în fișierul „TestStub.cs”, unde am implementat și clasele stub pentru teste:

```
5 references
public class CardBibliotecaStub : ICardBiblioteca
{
    6 references | 4/4 passing
    public string NumeDetinator { get; set; }
    9 references | 2/2 passing
    public int PuncteBonus { get; set; }
    5 references
    public List<ICarte> CartiImprumutate { get; set; }

    4 references | 3/3 passing
    public CardBibliotecaStub(string nume)
    {
        NumeDetinator = nume;
        CartiImprumutate = new List<ICarte>();
        PuncteBonus = 0;
    }
    3 references | 1/1 passing
    public void CarteImprumutata(ICarte carte)
    {
        CartiImprumutate.Add(carte);
    }
    3 references | 1/1 passing
    public void ReturneazaCarte(ICarte carte)
    {
        CartiImprumutate.Remove(carte);
    }
    3 references | 2/2 passing
    public bool ContineCarte(string titlu, string autor)
    {
        return CartiImprumutate.Any(carte => carte.Titlu == titlu && carte.Autor == autor);
    }
    2 references | 1/1 passing
    public int Puncte() { return PuncteBonus; }
    3 references | 1/1 passing
    public void AdaugaPuncteBonus(int puncte)
    {
        PuncteBonus += puncte;
    }
    3 references | 1/1 passing
    public void ScadePuncteBonus(int puncte)
    {
        PuncteBonus -= puncte;
    }
}
```

Figura 6 – Implementarea interfeței „ICardBiblioteca” pentru Stub Test

```
8 references
public class CarteStub : ICarte
{
    21 references | 9/10 passing
    public string Titlu { get; set; }
    21 references | 9/10 passing
    public string Autor { get; set; }
    44 references | 26/33 passing
    public int NumarExemplare { get; set; }
    4 references | 2/2 passing
    public uint AnPublicare { get; set; }
    4 references | 2/2 passing
    public string Categorie { get; set; }

    7 references | 3/3 passing
    public CarteStub(string titlu, string autor, int numarExemplare, uint an, string categorie)
    {
        Titlu = titlu;
        Autor = autor;
        NumarExemplare = numarExemplare;
        AnPublicare = an;
        Categorie = categorie;
    }
}
```

Figura 7 - Implementarea interfeței „ICarte” pentru Stub Test

Ciclul unui test stub este următorul: configurăm datele (pregătim obiectul de testat), testăm funcționalitatea (Act) și verificăm stare (folosim Assert pentru a testa starea obiectului).

Pentru a utiliza clasele „CarteStrub” și „CardBibliotecaStub”, a fost nevoie sa modific Arrang-ul (fig. 8) astfel încat la inițializarea cărților și cardului să utilizez noile funcții:

```
public void Setup()
{
    //arrange
    biblioteca = new Biblioteca();
    cardBiblioteca = new CardBibliotecaStub("Flavia");
    cartel = new CarteStub("Ion", "Liviu Rebreanu", 10, 1920, "Roman social");
    cheie1 = cartel.Titlu + cartel.Autor;
    carte2 = new CarteStub("De veghe in lanul de secara", "J.D.Salinger", 1, 1951, "Fictiune");
    cheie2 = carte2.Titlu + carte2.Autor;
    carte3 = new CarteStub("Casa Bantuita", "Shirley Jackson", 3, 1959, "Roman gotic");
    cheie3 = carte3.Titlu + carte3.Autor;
    carte4 = new CarteStub("Fluturi", "Irina Binder", 0, 2013, "Fictiune");
    cheie4 = carte4.Titlu + carte4.Autor;
    biblioteca.carti.Add(cheie1, cartel);
    biblioteca.carti.Add(cheie2, carte2);
    biblioteca.carti.Add(cheie3, carte3);
    biblioteca.carti.Add(cheie4, carte4);
}
```

Figura 8 – Noul Arrange pentru teste Stub

În cele ce urmează o să prezint un test Stub din cadrul aplicației, pentru a se putea observa cum se utilizează noua clasa.

Un test stub pentru testarea comportamentului clasei „Biblioteca”, atunci când adăugăm o nouă carte are următoarea implementare (fig.9):

```
[Test]
[Category("Pass")]
// 0 references
public void AdaugareCarte()
{
    var carte = new CarteStub("Napasta", "I. L. Caragiale", 9, 1890, "Dramaturgie");
    cheie = "Napasta" + "I. L. Caragiale";
    //act
    bool val = biblioteca.AdaugaCarte(carte);

    //assert
    Assert.IsTrue(val);
    Assert.AreEqual(9, biblioteca.carti[cheie].NumarExemplare);
}
```

Figura 9 – Testarea clasei AdaugăCarte() pentru valoarea validă cu Stub

Mai sus am testat comportamentul clasei „Biblioteca” pentru metoda „AdaugăCarte”. Clasa depinde de clasa „ICarte”, dar noi vrem să testăm comportamentul, logica pentru metodele din clasa „Biblioteca”, fără a implica dependențele clasei „Carte”. Așadar am folosit „CarteStub” pentru a simplifica testul.

Avantajele testării Stub sunt:

- Permite testarea unei componente (metodele clasei „Biblioteca”) în izolare;
- Permite simplificarea utilizării obiectelor reale (dependențelor);
- Permite mai mult control asupra datelor de test folosite și funcționalității metodelor.

3.2.2 Mock Testing

Mock este o dublă de testare care validează colaborarea între clase, utilizat în testarea software pentru a imita comportamentul anumitor componente ale aplicației. Mock-urile funcționează similar altor duble în timpul fazei de exercițiu, deoarece trebuie să convingă SUT-ul că comunică cu clase autentice, cu toate acestea diferă în fazele de configurare și verificare.

Unit Tests cu Mock oferă o modalitate ușoară de a reduce dependențele nedorite atunci când scrieți teste unitare. 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.

Framework-ul utilizat în aplicație, folosit pentru .NET este Moq. Facilitează testarea componentelor cu dependențe. API-ul său este construit în jurul constructelor lambda pentru a se asigura că testele sunt scrise puternic, ușor de citit și rapid de scris.

Cicluul unui test mock este următorul: configurăm datele de testare, testăm funcționalitatea (Act), verificăm așteptările (dacă metodele au fost apelate) și verificăm starea (folosim Assert pentru a testa starea obiectului).

Avantajele testelor Mock sunt asemănătoare cu testele Stub. Mock permite testarea claselor în izolare, oferă control asupra funcționalităților și obiectelor simulate, permițând testarea mai multor comportamente. În plus, permite verificarea că anumite metode au fost apelate, putând astfel testa interacțiunile dintre componente.

O să prezint în continuare un test Mock din cadrul proiectului. Testăm comportamentul clasei „Biblioteca” atunci când împrumutăm o carte. Utilizăm funcția „ImprumutăCarte” din clasa „Biblioteca”, care interacționează cu obiecte de tip „ICardBiblioteca”, ce reprezintă cardurile de membru, pe baza cărora se ține evidența cărților împrumutate. În testul nostru utilizăm Mock Test, pentru a putea analiza comportamentul clasei „Biblioteca” la apelarea metodei „ImprumutaCarte” fără a avea grija dependențelor, obiectului real „CardBiblioteca”. În figura 10 este implementarea testului mock.

```
//Testare functie ImprumutaCarte()
[Test]
[Category("Pass")]
[TestCase("IonLiviu Rebreanu")]
//[TestCase("De veghe in lanul de secara J.D.Salinger")]
0 references
public void CarteImprumutata(string cheie)
{
    int cantitate_initiala = biblioteca.carti[cheie].NumarExemplare;
    Mock<ICardBiblioteca> cardMock = new Mock<ICardBiblioteca>();

    //act
    bool rezultat = biblioteca.ImprumutaCarte(cheie, cardMock.Object);

    //assert
    Assert.IsTrue(rezultat);
    Assert.AreEqual(cantitate_initiala - 1, biblioteca.carti[cheie].NumarExemplare);
    cardMock.Verify(_ => _.CarteImprumutata(It.Is<ICarte>(c => c.Titlu == "Ion" && c.Autor == "Liviu Rebreanu")), Times.Once());
}
```

Figura 10 – Testul Mock pentru metoda „ImprumutaCarte()”

În testul de mai sus am folosit framework-ul Moq pentru a crea mock-ul „cardMock” ce înlocuiește un obiect real al clasei „ICardBiblioteca”. Urmând să utilizăm mock-ul „cardMock” în apelarea funcției „ImprumutaCarte()”, astfel simulând că scenariul în care un membru al bibliotecii încercă să împrumute o carte.

La finalul testului, verificăm dacă metoda „CarteImprumutata” a fost apelată, iar dacă cartea din apel există și verificăm dacă rezultatul metodei apelate este true, iar cantitatea s-a actualizat în cadrul listei de cărți din clasa „Biblioteca”.

3.3 Code Coverage

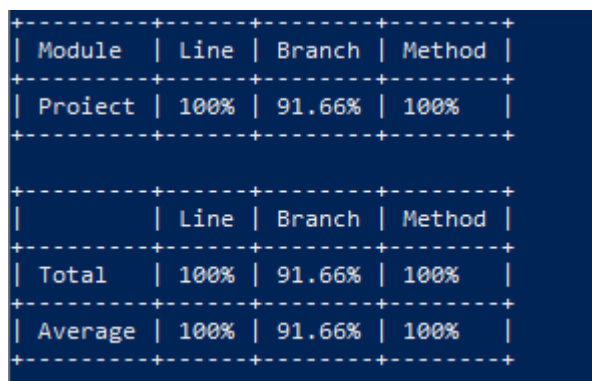
Code Coverage (Acoperirea codului) realizează raportul în care codul sursă a fost parcurs în timpul testării, cu scopul de a determina dacă aplicația a fost testată în mod adecvat.

Versiunea de Visual Studio Enterprise are funcția de Code Coverage ce permite să expună codul mort și părțile aplicației netestate sau insuficient testate. Se pot realiza analize asupra codului sursă și se pot indentifica testele unitare care trebuie reluate.

Aceste instrumente de acoperire a codului sunt utile în special în industriile pentru dezvoltarea încorporată a aplicațiilor critice pentru siguranță și securitate în care sistemele software nu pot eșua sau se vor pierde vieți.

În cazul proiectului, am instalat un pachet de tip Code Coverage, prin comandă: „dotnet tool install -g coverlet.console”. Urmând ca după ce rulat testele și am dat „build” la proiect să rulez comanda următoare: „coverlet .\bin\Debug\net6.0\TestProject1.dll --target "dotnet" --targetargs "test --no-build”.

După comanda de realizare a raportului am primit următorul rezultat (fig. 11):



| Module | Line | Branch | Method |
|---------|------|--------|--------|
| Proiect | 100% | 91.66% | 100% |
| Total | 100% | 91.66% | 100% |
| Average | 100% | 91.66% | 100% |

Figura 11 – Raport Test Code Coverage

În urma analizei de mai sus se poate observa, că toate metodele și toate liniile de cod au fost executate în timpul testelor.

Marea majoritate din ramurile condiționate au fost acoperite în timpul testelor, 91,66%. Acest scor arată ca unele cazuri se margine sau condiții rare nu sunt acoperite de teste. Și rămâne de analizat în continuare ramurile care nu au fost testate și să evaluăm dacă e necesar să adăugăm teste suplimentare pentru a acoperi aceste cazuri.

Consider că acest procent nu este mai mare din cauza că în momentul în care analizez cazurile în care un obiect aparține sau nu în colecția de cărți, primesc eroare de la colecție, înainte să analizez eu această eroare.

În concluzie, după acest raport e nevoie să analizez ramurile condiționale. Îmbunătățirea „branch coverage” poate duce la o mai mare încredere fiabilitatea și funcționalitatea codului. Dar raportul arată totuși o acoperire impresionantă a liniilor de cod și a metodelor, ceea ce este un semn bun pentru testele realizate.

4. CONCLUZII

În companii, Unit Testing-ul poate reduce semnificativ timpul petrecut cu repararea bug-urilor din aplicații, reducând încărcarea colegilor care se ocupă de suport și testare și permitând introducerea de noi funcționalități mai repede, ceea ce conduce la competitivitate crescută

Tehnicile Stub și Mock sunt metode comune și simple de a testa o aplicație, ajutând la reducerea complexității. Stub Testing este o tehnică eficientă în testarea unitară pentru a izola și a testa bucăți de cod în mod controlat. Stub simulează obiecte reale cu metode minime necesare pentru un test. Mock este folosit pentru a verifica dacă tehnicile și metodele sunt corect aplicate obiectelor și putem verifica că anumite metode au fost apelare. În plus, Mock este foarte util atunci când avem mai multe teste și fiecare test necesită un set unic de date, stub-urile sunt utilizate în principiu pentru un număr mai mic de teste și teste simple.

Code Coverage este un instrument important în dezvoltarea aplicațiilor software, este important să se echilibreze urmărirea unui scor mare cu dezvoltarea de teste care să abordeze cazurile reale. Code Coverage ajută la indentificarea codului mort, părțile aplicației netestate sau insuficient testate, indicând bucățile de cod unde pot exista erori nedetectate.

5. BIBLIOGRAFIE

- [1] Ovidiu Bantias, Domain Testing & Unit Testing, Universitatea Politehnica Timișoara, 2023.
- [2] Ovidiu Bantias, Test Doubles, Universitatea Politehnica Timișoara, 2023
- [3] Ovidiu Bantias, Code Coverage, Universitatea Politehnica Timișoara, 2023
- [4] <https://www.tutorialsteacher.com/csharp/csharp-dictionary>, accesat în data de 29.12.2023
- [5] <https://www.todaysoftmag.ro/article/377/din-unelte-artizanului-software-unit-testing>, accsat în data de 30.12.2023
- [6] <https://www.tutorialspoint.com/what-is-domain-testing-in-software-testing> , accesat în data de 30.12.2023

- [7] <https://learn.microsoft.com/en-us/visualstudio/test/using-code-coverage-to-determine-how-much-code-is-being-tested?view=vs-2022&tabs=csharp>, accesat în data de 01.01.2024
- [8] <https://docs.telerik.com/devtools/justmock/basic-usage/arrange-act-assert>, accesat în data de 02.01.2024
- [9] <https://www.turing.com/kb/stub-vs-mock> , accesat în data de 03.01.2024
- [10] <https://www.parasoft.com/solutions/code-coverage/>, accesat în data de 03.01.2024