**Unit test**

Unit test գրելու համար հարկավոր են դեվելոփմենթ հմտություններ։ Ստորև փորձել եմ ներկայացնել 2 պարզ օրինակներ, որոնց միջոցով կարելի է հասկանալ Unit test-երի հիմքը։ Unit test-ի դեպքերը գրվում և իրականացվում են կոդ գրողի կոմից (ոչ թե թեստավորողի)` համոզվելու համար, որ առանձին բլոկներն աշխատում են այնպես, ինչպես սպասվում էր: բլոկներն իրենց հերթին ունեն ավելի փոքր կառուցվածքային մասեր` ֆունկցիաներ, պրոցեդուրաներ, class-ներ, ինտերֆեյսեր և այլն:

Եթե մենք վերցնում ենք ֆունկցիա, տալիս ենք մուտքային պարամետրեր, ապա պետք է ստուգել , որ ֆունկցիան վերադարձնի ակնկալվող արժեքը: Իմաստը` ստուգել, որ բլոկներն աշխատեն ըստ նախատեսվածի և ավելի կայուն լինեն exception-ների նկատմամբ: Պետք է փորձարկվեն ինչպես դրական, այնպես էլ բացասական պայմաններով դեպքերը։

Օրինակ 1․

Ստորև ներկայացված է ֆունկցիա, որը գումարում է 2 ամբողջ թվեր և այն հարկավոր է թեստավորել unit test-ի միջոցով։

    static class ClassForSum

    {

        public static int Summ(int a1, int a2)

        {

            return a1 + a2;

        }

    }

Ներկայացնեմ, թե ինչ ենք անելու․ հայտարարելու ենք 2 ամբողջ թվեր, դրանք փոխանցենք պահանջվող ֆունկցիային, և ապա ստուգենք, թե արդյո՞ք ճիշտ պատասխան ստացանք։

Ահա այսպիսի տեսք ունի մեր նոր ստեղծված ծրագիրը։

    class Program

    {

        static void Main(string[] args)

        {

        }

    }

Հենց այստեղ էլ կհայտարարենք 2 ամբողջ թվերը՝ a=14, b=9։

    class Program

    {

        static void Main(string[] args)

        {

            int a = 14;

            int b = 9;

        }

    }

Հայտարարենք նոր z փոփոխական, որին կվերագրենք թեստավորվող ֆունկցիան ու կփոխանցենք, թե որ 2 թվերը նա պետք է գումարի։

class Program

    {

        static void Main(string[] args)

        {

            int a = 14;

            int b = 9;

            int z = ClassForSum.Sum(a, b);

// կարող ենք գրել նաև int z=ClassForSum.Sum(14, 9);

//բայց միշտ փոփոխականներով աշխատելն ավելի ճիշտ է

        }

    }

Այժմ պետք է ստուգենք և հասկանանք, թե արդյո՞ք z փոփոխականը դարձավ 23, թե՞ ոչ։ Դրա համար կոնսոլ պատուհանում կարտածենք «այո՜ ճիշտ է», իսկ հակառակ դեպքում մեկ այլ բան։ Վերջում չենք մոռանում գրել Readline հրամանը, որխեսզի կոնսոլի պատուհանը չփակվի։

class Program

    {

        static void Main(string[] args)

        {

            int a = 14;

            int b = 9;

            int z = ClassForSum.Sum(a, b);

            if(z == 23)

            {

              Console.WriteLine(“Այո՜, ճիշտ է”);

            }

           Else

  {

                Console.Writeline(“Թեստը ֆեյլ եղավ”);

            }

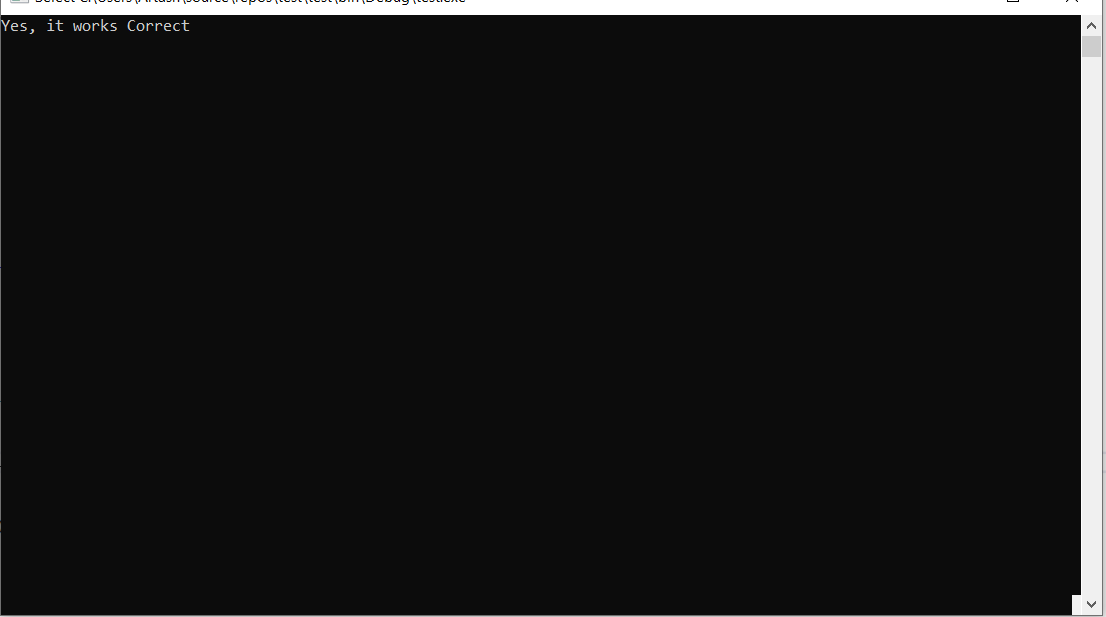
        }

    }

Աշխատացնելուց հետո կստանանք հետևյալ պատկերը։



Ցավոք հայկական տառերը չճանաչեց, գրենք լատիներենով և կրկին աշխատացնենք։



Ահա, փաստորեն մենք կանչեցինք ֆունկցիան, փոխանցեցինք թվերը և համոզվեցինք, որ ֆունկցիա ճիշտ է աշխատում, չնայած սա այնքան պարզ ֆունկցիա էր, որ անգամ կարիք չկար թետավորելու։

Այժմ եկեք փոխենք փոխանցվող թվերը և անյպիսի արժեքներ տանք, որ դրանց գումարը շատ մեծ լինի, օրինակ 2 000 000 000 (իսկ թե ինչու հենց այս թիվը կամ նման կարգի թվեր փերձեք հասկանալ) և կրկին 2 000 000 000։ Չմոռանանք if-ի մեջ փոխել արժեքը 4 000 000 000։

    class Program

    {

        static void Main(string[] args)

        {

            int a = 2000000000;

            int b = 2000000000;

            int z = ClassForSum.Sum(a, b);

            if(z == 4000000000)

            {

                Console.WriteLine("Yes, it works Correct");

            }

            else

            {

                Console.WriteLine("No, it doesn't work!!!");

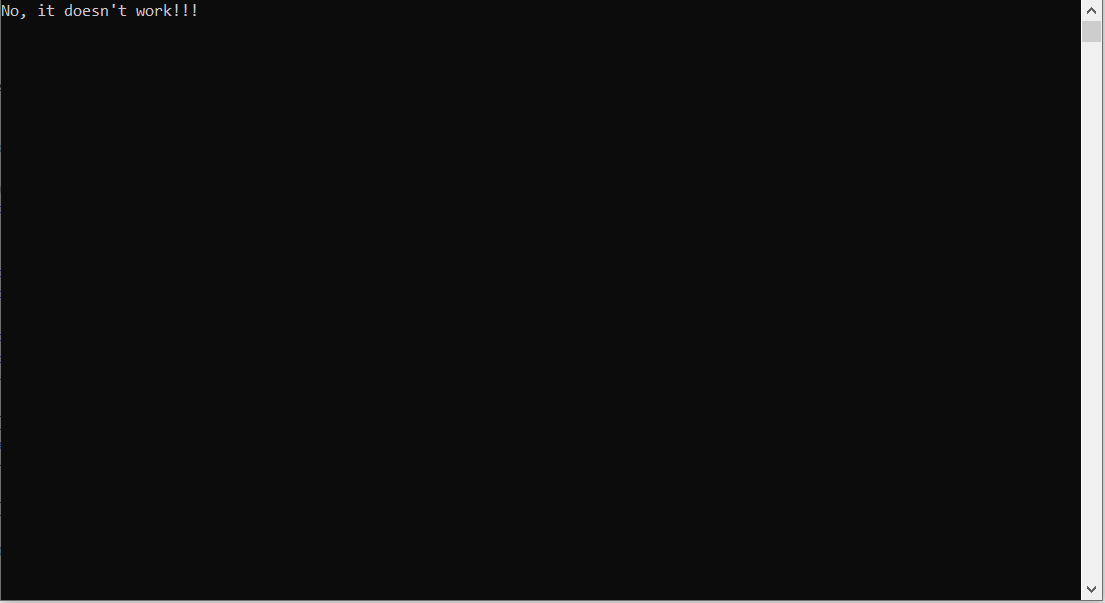
            }

            Console.ReadLine();

        }

    }

Իսկ արդյունքո՜ւմ՝ ֆեյլ։



Այդ ֆունկցիան չկարողացավ իրար գումարել 2 ամբողջ թվեր և արդեն պարզ էր, որ թեստավորվող ֆունկցիան նման թվերով արդեն չէր արդարացնելու իրեն։

Այս ֆունկցիայի համար կարելի է ասել ստեղծեցինք 2 թեստ քեյս այնուհետև բագը գտանք, բայց այն գտնելուց հետո հարկավոր է ինչ-որ տեղ գրել դրա մասին։ Կարելի է կիրառել հետաքրքիր տարբերակ, երբ հենց բագը գտնվում այն ավտոմատ ուղարկվում է ինչ-որ մեյլին։ Իհարկե երբեք ոչ ոք այդպես չի անի, սակայն հետաքրքրության համար կարող եք փորձել, միանգամից կզարգացնեք ձեր կոդ գրելու հմտությունները։ Իսկ իրականում ստեղծում են տեքստավոյ ֆայլ և Console.WrilteLine-ի փոխարեն ինֆորմացիան հավաքում են այնտեղ։

2. Բերենք ավելի հետաքրքիր օրինակ։

Ենթադրենք ունենք List կլասս, որն ունի Sort ֆունկցիա․ պետք է հասկանանք, թե արդյո՞ք այդ ֆունկցիան ճիշտ է աշխատում (թող ներեն ինձ Microsoft-ի աշխատակիցները)։

Main ֆունկցիայում ստեղծենք ինչ որ լիստ և արտածենք այն Console միջավայրում։

    class Program

    {

        static void Main(string[] args)

        {

            List<string> TestList = new List<string>();

            TestList.Add("88888888");

            TestList.Add("22");

            TestList.Add("1");

            TestList.Add("4444");

            TestList.Add("7777777");

            TestList.Add("333");

            TestList.Add("666666");

            TestList.Add("55555");

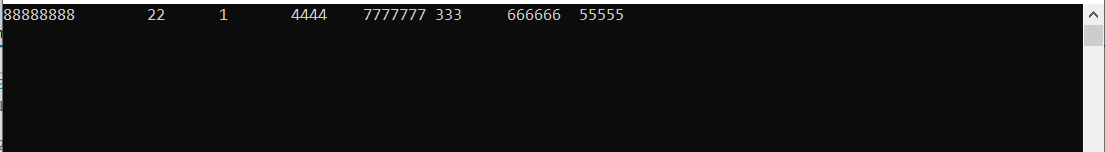
            TestList.ForEach(i => Console.Write("{0}\t", i));

            Console.ReadLine();

        }

    }

Արդյունքում ունենք հետևյալ պատկերը։



Այժմ մեզնից պահանջվում է ստուգել այս լիստի սորտավորումն ըստ բառերի երկարության և մենք գիտենք (մեզ ասում են, կամ մենք ինքներս էլ կարող ենք կոդ կարդալով հասկանալ) որ այդ ֆունկցիան կանչվում է հետևյալ կերպ։

DinoComparer dc = new DinoComparer();

TestList.Sort(0, 8, dc);

Այս ֆունկցիան կանչելուց հետո ակնկալում ենք, որ մեր TestList-ը պետք ունենա հետևյալ տեսքը՝ 1, 22, 333, 4444, 55555, 666666, 7777777, 88888888։

Դեհ փորձենք ստուգել հետևյալ կերպ։

Քանի որ մենք գիտենք ակնկալվող հերթականությունը, ապա հերթով կանցնենք և կստուգենք հետևյալ կերպ։

if(TestList[0] == "1" && TestList[1] == "22" && TestList[2] == "333" && TestList[3] == "4444" && TestList[4] == "55555" && TestList[5] == "666666" && TestList[6] == "7777777" && TestList[7] == "88888888")

{

  Console.WriteLine("It works Correctly");

}

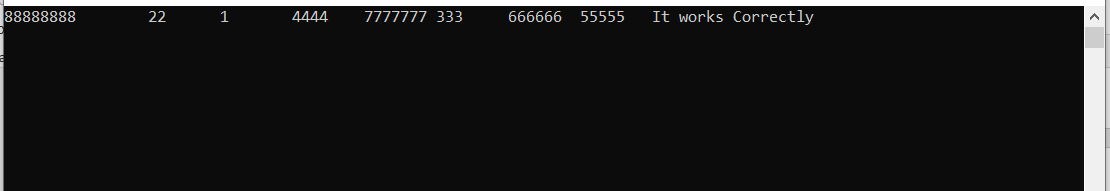
Else

{

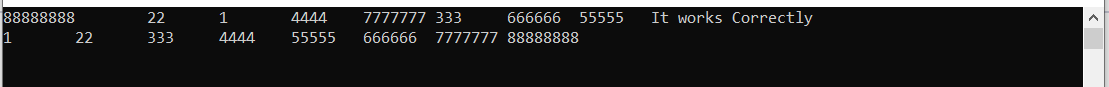
Console.WriteLine("It doesn’t work");

}

Արդյունքում կունենանք հետևյալ պատկերը



Համոզվելու համար արտածենք և ստուգենք։



Ահա ամբողջ կոդը։

    class Program

    {

        static void Main(string[] args)

        {

            List<string> TestList = new List<string>();

            TestList.Add("88888888");

            TestList.Add("22");

            TestList.Add("1");

            TestList.Add("4444");

            TestList.Add("7777777");

            TestList.Add("333");

            TestList.Add("666666");

            TestList.Add("55555");

            TestList.ForEach(i => Console.Write("{0}\t", i));

            DinoComparer dc = new DinoComparer();

            TestList.Sort(0, 8, dc);

            if(TestList[0] == "1" && TestList[1] == "22" && TestList[2] == "333" && TestList[3] ==                                              "4444" && TestList[4] == "55555" && TestList[5] == "666666" && TestList[6] ==                                                  "7777777" && TestList[7] == "88888888")

            {

                Console.WriteLine("It works Correctly");

            }

            TestList.ForEach(i => Console.Write("{0}\t", i));

            Console.ReadLine();

        }

    }

Դե ինչ, հուսով եմ կարղացա նկարագրել Unit test-ի էությունը։ Եթե լինեն հարցեր կարող եք գրել և հարցնել։

*Թեստավորեք մատիտը* - ստեղծեք test case-եր։  
  
1․ Նախ պետք է հասկանալ, թե ո՞ւմ համար է նախատեսված մատիտը և ավելի շատ ինֆորմացիա հավաքել պրոդուկտի մասին։  
- Ո՞վ է օգտագործելու - դպրոցականներ, երեխաներ, նկարիչներ։ Ո՞ր տարիքի համար է նախատեսված։ Ո՞ր երկրում է օգտագործվելու (չէ՞ որ, օրինակ Բրազիլիայում, դրանով կարող են անգամ մարդ սպանել🤭😉)  
  
2․ Ճանաչեք պրոդուկտը և հասկացեք, թե ինչպես է այն նախատեսված օգտագործել։  
- Ռետին ունի՞, որքա՞ն է մատիտ երկարությունը, ի՞նչ գույնի է, որքա՞ն կարող եմ գրել մինչև կարիք լինի կրկին սրել։  
  
Նմանատիպ այլ հարցեր տալուց և պատասխան ստանալուց հետո կարող ենք անցնել թեստ թեյսեր գեներացնելուն։  
  
1. Functional - գրել թղթի վրա, ջնջել ռետինով։ Եթե սա չաշխատեց, ապա անիմաստ է շարունակել։  
2․ Errors - գրել մաշկի վրա, ջրի տակ, մետաղի, պատի վրա (իհարկե կախված requirement-ներից)  
3․ Boundary - գրել շատ սուր ծայրով, գրել գրեթե վերջացած գլխիկով (նույնը ռետինի համար)  
4․ Performance - Որքա՞ն կկարողանամ գրել մինչև հաջորդ սրելը, միջավայրի ջերմաստիճանի և թղթի որակի ազդեցությունը գրելու վրա, փորձել ջնջել գրածը 1 ամիս անց և այլն․․․  
5․ Security - Ի՞նչ կպատահի, եթե գրաֆտը միամիտ ուտեմ կամ մատիտը ջարդելուց որևէ փոքր կտոր կթռնի՞ աչքիս  
6․ Accessibility - թեստավորել ըստ ձեռքի տարբեր չափերի, գրել աջ և ձախ ձեռքերով (գուցե բերանով)  
  
Եղեք կրեատիվ, գեներացրեք ավելի շատ test case-եր։

Integration: ինչպես է գրում տարբեր որակի թղթերի վրա, արդյոք հնարավոր է ջնջել այլ (ոչ իր վրայի) ռետիններով կամ սրել տարբեր սրիչներով։ Ձեր նշած functional case-երը կհամարեմ smoke caseեր եւ անպայման կավտոմատացնեմ։ Քանի որ այդ թեսթերը պետք է ամեն փոփոխությունից հետո կրկնեմ։ Կստուգեմ նաեւ, որ նրա փափկությունը (B, 2B ...) համապատասխանում է իր վրա գրվածին (documentation-ին)։ Կթեսթավորեմ նաեւ չ-դոկումենտացված case-երը, օրինակ աղջիկները հաճախ օգտագործում են մազերը հավաքելու համար, плотник-ները դնում են ականջի հետեւում ...😁 ու իհարկե մի թեստավորած մատիտը պետք է հարմար լինի նաեւ այդ շուկայի համար։ Stress test: ամբողջ ուժով կհենվեմ մատիտի վրա ու կտեսնեմ որքան կդիմանա։ Չմոռանանք նաեւ UI/UX case-երը, քանի որ գնորդին առաջին հերթին հետաքրքրում է տեսքն ու հարմարավետությունը (իսկ որակն անհրաժեշտ է, որպեսզի նա դառնա մեր մշտական օգտատերը)։ :)

*API*

Ահա՛, թե որտեղից կարելի է սկսել հասկանալ Web API թեստինգը։  
  
Ցանկացած click-ի ժամանակ գեներանում են request-ներ՝ HTTP հարցումներ (request-ներ), որոնք կարող եք տեսնել Inspect Element → Network բաժնում, ինչպես ցույց է տրված կարճ հոլովակում։  
  
Եթե խոսենք ոչ ծրագրավորման լեզվով, ապա կանչվում է ֆունկցիա, որը բազայից մեզ վերադարձնում է որոշակի ինֆորմացիա՝ նկարներ, լինկեր, տեքստեր և այլն։ Կարող ենք տեսնել, թե ինչ սերվերով է աշխատում այդ կայքը (մեր օրինակում gws` google web server, տրամաբանական էր չէ՞), թե http ո՞ր մեթոդն է կանչվել, ո՞ր ip-ով և այլն։  
  
Եվ հենց այս ինֆորմացիայի հիման վրա էլ, եթե ունենանք դոկումենտացիա, ապա կարելի է գեներացնել http request-ներ և տարբեր թեստ քեյսեր ստեղծել։  
  
Այլ կերպ ասած UI-ից հրամաններ կանչելու փոխարեն կարող ենք ստեղծել տարբեր թեստ քեյսերին համապատասխան http request-ներ, որպեսզի ամեն անգամ մենք մեր ձեռքով երկար ժամանակ չծախսենք այնպիսի ֆունկցինալներ ստուգելու համար, ինչպիսիք են Get (search), Post (Add data), Delete, Update գործողությունները։  
  
Հետագայում մի փոքրիկ հոլովակով կներկայացնեմ API թեստինգի ողջ էությունը

*Scalability testing*-ը տարբերվում է Load թեստավորումից այն փաստով, որ Scalability testing-ը համակարգը չափում է նվազագույն և առավելագույն ծանրաբեռնվածությունների բոլոր մակարդակներում, ներառյալ տվյալների բազայի մակարդակները: Առավելագույն ծանրաբեռնվածությունը պարզելուց հետո դեվելոփերները պետք է պատշաճ կերպով արձագանքեն որոշակի բեռից հետո համակարգը ընդարձակելի դարձնելու համար:  
  
Օրինակ. Եթե ընդարձակելիության թեստավորման համար որոշվում է, որ առավելագույն ծանրաբեռնվածությունը պետք է լինի 10,000 օգտագործող, ապա համակարգի ընդարձակելի լինելու համար դեվելոփերները պետք է միջոցներ ձեռնարկեն կատարելագործելու այնպիսի գործոններ, ինչպիսիք են` արձագանքման ժամանակի նվազեցումը 10,000 օգտագործողի սահմանը հասնելուց հետո, կամ RAM-ի չափի ավելացումը` շատ ավելի օգտագործողներին տեղավորելու համար:  
  
Այսինքն, եթե մենք ունենք առավելագույն սանդղակ, դա դեռ չի նշանակում, որ դրանից այն կողմ ծրագիրը չպետք է վատ աշխատի կամ առհասարակ չաշխատի։ Հակառակը, հարկավոր է ունենալ միջոցներ, որոնք հետագայում հնարավորություն կտան ծրագրին ընդարձակվելու։  
  
Ահա, թե ինչու է այն կոչվում ընդարձակելիության, մասշտաբելիության թեստավորում