**Որակի ապահովման հիմունքներ**Հարցազրույցի նախապատրաստման ուղեցույց

**QA. . . . .BASICS**

Արտաշես Բաղդասարյան

Խմբագիր՝ Տաթև Սուքիասյան

2022 Երևան

ISBN

**Հեղինակի կողմից**

**Ցանկացած ծրագիր ունի թերություններ ամենատարբեր մակարդակներում և դու պետք է գտնես դրանք։**

Սիրելի ընթերցող, այս գրքում ներառվել են այնպիսի նյութեր, որոնք կօգնեն ձեզ հասկանալ որակի ապահովման հիմունքները, աշխատանքի սկզբունքը, և շատ այլ հետաքրքիր ինֆորմացիաներ՝ կապված այս ոլորտի հետ։ Այս կարճ 4 տարիների ընթացքում հասցրել եմ ձեռք բերել մեծ փորձ՝ շնորհիվ իմ լավ գործընկերների։ Ցանկությունս մեծ է գիտելիքներս փոխանցելու և օգնելու ձեզ առաջին քայլը կատարելու դեպի IT ոլորտ։

Գիրք գրելու միտքը հղացավ այն պահից, երբ սոցիալական ցաներում նկատեցի բազմաթիվ մարդականց մտահոգություններ այն մասին, որ հայերենով նյութեր չկան։ Քանի որ համավարակի շրջանակներում սկսեցինք աշխատել տանից, ուստի ունեցա ժամանակ կիսվել տեսական և պրակտիկ գիտելիքներով «linkedin» սոցիալական կայքում։ Հետզհետե ստանալով շատ նամակներ, որ ներկայացված ինֆորմացիան հասկանալի և լավ է նկարագրված, որոշվեց ստեղծել այս գիրքը։

Գրքում կհանդիպեք շատ օտարալեզու տերմիններ, որոնք երբեմն թարգմանված են, երբեմն ոչ։ Դա գալիս է նրանից, որ այս ոլորտի առաջին լեզուն անգլերենն է և չենք կարող բոլորը թարգմանել այնպես, որ պարզ լինի առաջին հայացքից։ Բացի այդ մինչև թեստավորում սովորելը պետք է ունենալ անգլերենի ամենաքիչը B1 մակարդակ, որպեսզի կարողանաք մրցակցել և դառնալ շատ լավ մասնագետ։ Այս ամենն, իհարկե, չի նշանակում, որ այս գիրքը յուրացնելու համար պետք է անգլերենի որոշակի մակարդակ։ Այստեղ դիտարկված է այն փաստը, որ գիրքն ամեն դեպքում հայալեզու է և պետք է ծառայի ցանկացած հայի։ Հեշտության համար բոլոր տերմինների բացատրությունը կգտնեք «Բառարան» բաժնում։

Հարկ է նշել, որ այս ոլորտն այժմ շատ հեռանկարային է և այս գրքի նպատակն է ձեզ զինել այնպիսի գիտելիքներով, որոնք ձեզ թույլ կտան անցնել հարցազրույցը և դառնալ որևէ ընկերության աշխատակից։

Գիրքը կրում է «Ուղեցուցային» բնույթ, այն կարճ է, հստակ, հասկանալի։ Այն իր ﬔջ ներառում է բազմաթիվ տեսական հարցեր, որոնց առընչվելու եք ցանկացած հարցազրույցի ժամանակ։

Գիրքը նպատակ ունի ընդլայնվելու։ Երկրորդ մասում կներառﬖեն ավելի շատ գործնական գիտելիքներ կապված API-ների, կոդի և բազայի հետ։

Ցանկանում եմ շնորհակալություն հայտնել բոլոր ընկերներիս և գործընկերներիս, ովքեր օգնել են և նպաստել են այս գրքի ստեղծմանը։

Եթե հավանեք այս գիրքը կամ գտնեք որոշ թերություններ, ապա ուրախ կլինեմ պատասխանել ձեր հարցերին և շտկել բոլոր այն թերությունները, որոնք կմատնանշվեն։

Linkedin: https://www.linkedin.com/in/artashes-baghdasaryan/

email: [artashes.baghdasaryan97@gmail.com](mailto:artashes.baghdasaryan97@gmail.com)

Սիրով՝ Արտաշես Բաղդասարյան։

**Խմբագրի կողմից**

Հասարակության մեջ ձևավորվել է մի կարծիք, ըստ որի ՏՏ ոլորտ մտնելու ամենահեշտ ու ամենակարճ ճանապարհը «Որակի ապահովման մասնագետ» (QA) դառնալն է։ Ինչ որ մասով կարելի է քննարկել այս կարծիքը, քանի որ հնարավոր է սկսել աշխատանքը առանց ծրագրավորման խորը գիտելիքներ ունենալու, սակայն մրցունակ մասնագետ դառնալու համար միջակությունը երբեք չի կարող լինել այն հենքը, որի վրա հնարավոր կլինի կառուցել ապագան, ուստի անհրաժեշտ կլինի տիրապետել նաև ծրագրավորման ինչ որ լեզվի, բազաներին և ցանցերին։ Բացի այս, մեզնից յուրաքանչյուրը ակամա թեստավորող է։ Առօրայում մենք օգտագործում ենք բազում ծրագրեր, որոնց մասին ձևավորվում են դրական ու բացասական կարծիքներ։ Թեստավորողի աշխատանքը կայանում է նրանում, որ նա դեռևս ծրագրի ստեղծման փուլից սկսած կարող է իր ներդրումն ունենալ, հայտնել խնդիրների մասին և օգնել ընկերությանը ստեղծել ավելի որակյալ ծրագիր։

Իրականում ՏՏ ոլորտին վերաբերող ամբողջ գրականությունը հիմնականում անգլերեն է, ինչը, շատ հաճախ, կրկնակի խոչնդոտ է հանդիսանում նոր մասնագիտություն ձեռք բերելու համար։ Ճիշտ է նաև այն, որ առանց անգլերենի իմացության, շատ դժվար է հաստատվել այս ոլորտում, քանզի այնուամենայնիվ, թվային աշխարհի լեզուն անգլերենն է։ Սակայն, մի բան է այն օգտագործել որոշ տեղերում, մեկ այլ բան է սովորել այդ լեզվով։

Շնորհիվ հեղինակի՝ Արտաշես Բաղդասարյանի, այսօր մեզնից յուրաքանչյուրն ունի հնարավորություն մայրենի լեզվով սովորելու դարի մասնագիտություններից մեկը։

Իրականում այս գիրքը բանալի է, որը լավ ուսումնասիրելու դեպքում հնարավորություն կտա բացելու բազում դռներ։ Նույնիսկ նրանք, ովքեր առհասարակ հեռու են թվային աշխարհից, կարդալով այս գիրքը, կկարողանան հասկանալ, քանի որ ամեն բան գրված է շատ պարզ ու մատչելի։

Որոշ հիմնական տերմինների ու հասկացությունների կհանդիպեք գրքում անգլերենով, որոնց բացատրությունը, բնականաբար, նույնպես կգտնեք։ Սակայն հարկ ենք համարել, տվյալ բառերը օգտագործել հենց անգլերենով, որովհետև աշխատանքում դրանք հաճախակի օգտագործվող տերմիններ ու բառեր են, որոնց պետք է սովորել։

Այժմ ապրում ենք մի ժամանակահատվածում, որտեղ ուսուցումը դարձել է ուղղորդող, իսկ մարդիկ գնալով ավելի ու ավելի շատ են սկսում զբաղվել ինքնակրթությամբ։ Այս ուղեցույցը լավագույնս կարող է ուղղորդել ցանկացած մեկին, ով իրոք ցանկանում է դառնալ «Որակի ապահովման մասնագետ»։

Վստահ եմ, որ շատերն էին սպասում այս գրքին և ուրախ եմ փաստել, որ սա այլևս իրողություն է։

Հարգանքներով՝ Տաթև Սուքիասյան։

**Բովանդակություն**

[**Գլուխ** **1․ Թեստավորման սկզբունքները, փուլերը և աշխատանքի նկարագրությունը** 12](#_Toc120734757)

[**1․1 Որակի ապահովման 7 սկզբունքները․** 13](#_Toc120734758)

[**1․2 Agile-Scrum Մեթոդոլոգիա** 20](#_Toc120734759)

[**1․3 Ծրագրի զարգացում - Program Development[9]** 22](#_Toc120734760)

[**1․4 Թեստավորման գործընթաց – STLC** 24](#_Toc120734761)

[**(Software testing life cycle)** 24](#_Toc120734762)

[**1․5 Թեստավորման տեսակները կամ դասակարգումն ըստ ֆորմալիզացման աստիճանի** 31](#_Toc120734763)

[**1․6 Թեստ քեյս (դեպք) - Test Case** 32](#_Toc120734764)

[**1․7 Կիրառելի case (դեպք) - Use case․ կյանքի իրական սցենարներ** 35](#_Toc120734765)

[**1․8 Ի՞նչ է testware-ը** 37](#_Toc120734766)

[**1․9 Սխալի հաշվետվություն կամ զեկույց - Bug report** 38](#_Toc120734767)

[**1․10 Բագի տեսակավորում - Bug triage** 47](#_Toc120734768)

[**1․11 Կոնֆիգուրացիայի/թողարկման կառավարում – Configuration/Version management** 48](#_Toc120734769)

[**1․12 Ի՞նչ են bug leakage-ը և bug release-ը**[21] 50](#_Toc120734770)

[**1․13 Ստատիկ թեստավորում - Static test** 51](#_Toc120734771)

[**Գլուխ** **2․** **Դինամիկ թեստավորում Dynamic testing** 54](#_Toc120734772)

[**2․1 Black Box testing (Սև տուփի թեստավորում)** 54](#_Toc120734773)

[**2․2 White Box testing (Սպիտակ տուփի թեստավորում)** 56](#_Toc120734774)

[**2․3 Gray Box testing (Մոխրագույն տուփի թեստավորում)** 58](#_Toc120734775)

[**2․4 Թեստավորման Մակարդակներ – Testing Levels** 60](#_Toc120734776)

[**2․4.1 Unit testing** 62](#_Toc120734777)

[**2․4.2 Integration testing** 76](#_Toc120734778)

[**2․4.3 System testing** 78](#_Toc120734779)

[**2․4.4 Acceptance testing** 80](#_Toc120734780)

[**Գլուխ** **3․** **Functional testing Ֆունկցիոնալ թեստավորում** 82](#_Toc120734781)

[**3․1 Smoke testing** 83](#_Toc120734782)

[**3․2 Regression testing** 84](#_Toc120734783)

[**3․3 Sanity testing** 85](#_Toc120734784)

[**3.4 Interface testing** 87](#_Toc120734785)

[**Գլուխ 4․** **Ոչ** **ֆունկցիոնալ** **թեստավորում** - **Non** **Functional** **Testing** 89](#_Toc120734786)

[**4․1 Անվտանգության թեստավորում - Security testing** 90](#_Toc120734787)

[**4․2 Հասանելիության թեստավորում - Availabilty testing․** 93](#_Toc120734788)

[**4․2 Կատարողականի թեստավորում - Performance testing** 97](#_Toc120734789)

[**4.3 Ծանրաբեռնվածության թեստավորում - Load testing - Stress testing** 103](#_Toc120734790)

[**4.4 Ծավալային թեստավորում - Volume testing** 109](#_Toc120734791)

[**4․5 Տեղորոշման թեստավորում - Localization testing** 112](#_Toc120734792)

[**4․6 Տեղադրման թեստավորում - Install testing** 115](#_Toc120734793)

[**4.7 Վերականգնման թեստավորում - Recovery testing** 120](#_Toc120734794)

[**4․8 Կայունության թեստավորում - Endurance testing** 121](#_Toc120734795)

[**4․9 Ծայրահեղ ծանրաբեռվածության թեստավորում - Spike testing** 122](#_Toc120734796)

[**4․10 Հուսալիության թեստավորում - Reliability testing** 125](#_Toc120734797)

[**4․11 Օգտագործելիության թեստ - Usability testing** 127](#_Toc120734798)

[**4․12 Ընդարձակելիության թեստավորում - Scalability testing** 128](#_Toc120734799)

[**4․13 Համատեղելիության թեստավորում - Compatibility testing** 130](#_Toc120734800)

[**Գլուխ 5․** **Blackbox** **testing** **տեխնիկաներ** 132](#_Toc120734801)

[**5.1 Համարժեքային բաժանում - Equivalence partitioning** 132](#_Toc120734802)

[**5․2 Սահմանային թեստավորում - Boundary testing** 132](#_Toc120734803)

[**5.3 Վիճակի անցման թեստավորում - State transition tables** 133](#_Toc120734804)

[**5․4 Որոշումների աղյուսակային թեստավորում - Decision table testing** 137](#_Toc120734805)

[**5․5 Սխալների ենթադրության տեխնիկա - Error Guessing Technique** 141](#_Toc120734806)

[**5․6 Գրաֆների վրա հիմնված թեստավորում - Graph-based testing** 142](#_Toc120734807)

[**Գլուխ** **6․ Հավելում** 145](#_Toc120734808)

[**6․1 Web testing – Cross Browser Testing** 145](#_Toc120734809)

[**6․2 Combobox-ի (Dropdown menu) գաղտնիքները․․․** 148](#_Toc120734810)

[**6․3 Թեստ քեյսեր Numeric՝ թվային, դաշտերի համար** 149](#_Toc120734811)

[**Բառարան** 151](#_Toc120734812)

[**Գրականության ցանկ** 162](#_Toc120734813)

# Գլուխ 1․ Թեստավորման սկզբունքները, փուլերը և աշխատանքի նկարագրությունը

Թեստավորումն ինքնին փիլիսոփայություն է, որը ծնվել է շատ դարեր առաջ, երբ փորձում էին ստեղծել նոր մի բան։ Ընդհանրապես այն ինչ ստեղծվում է մարդու կողմից, կարիք ունի թեստավորվելու։ Օրինակ, նոր նյութեր կամ սարքեր ստեղծելիս, շատ կարևոր մաս է կազմում կենսագործունեության անվտանգությունը։ Սրանից հետևում է, որ նոր ստեղծված սարքը չպետք է սպառնա մարդու կյանքին և նրա առողջությանը։ Գոյություն ունի մեկ այլ չափանիշ ևս, որին պետք է սևեռել մեր ուշադորւթյունը, այն է սարքերի էրգոնոմիկական չափանիշները՝ մարդու ֆիզիոլոգիական և հոգեբանական հնարավորություններին հարմարեցնող դիսցիպլինան։

Բացի այն, որ ցանկացած նոր բան պետք է թեստավորվի, այն պետք է շարունակաբար վերահսկվի՝ խափանումների դեպքում դրանց առաջացման պատճառները վերլուծելու, շտկելու և նորից հաճախորդին հանձնելու համար։

Այսպիսով, թեստավորման գաղափարները հասկանալու համար միասին սկսենք ուսումնասիրել որակի ապահովման 7 սկզբունքները, որոնց միջոցով սահմանվում է, թե ինչ է թեստավորումը։

## **1․1 Որակի ապահովման 7 սկզբունքները․**

**Ի՞նչ է թեստավորումը, ինչպես և երբ այն կատարել, որտեղից սկսել և** **երբ ավարտել։**

Մինչ այս հարցերն առավել խորությամբ ուսումնասիրելն ու թեստավորման համար անհրաժեշտ գործիքակազմին ծանոթանալը, ձևակերպենք որակի ապահովման 7 սկզբունքները, որոնք թեև բավականին աբստրակտ և մակերեսային են, բայց ներառում են արժեքավոր գաղափարներ և մոտեցումներ արդյունավետ թեստավորման համար։

1. **Թեստավորումը ցույց է տալիս թերությունների առկայությունը ոչ թե բացակայությունը**

Նույնիսկամենապրոֆեսիոնալ թիմի կողմից ստեղծված ծրագրային ապահովումը[1] (ծրագիրը), չի կարող զերծ լինել թերություններից։

Թեստավորման նպատակն է գտնել այն թերությունները, որոնք բերում են ծրագրի ոչ նորմալ աշխատանքին, կամ ընդհանրապես դրա խափանմանը։

Ճիշտ կազմակերպված թեստավորման ցիկլը զգալիորեն նվազեցնում է ծրագրի վերջնական տարբերակում թերությունների հայտնաբերումը, այդպիսով վերջնական օգտատիրոջ համար ապահովելով առավել հուսալի և անխափան ծրագիր։

Սակայն, նույնիսկ այն, որ թեստավորողները չեն կարողացել հայտնաբերել թերություններ, չի երաշխավորում, որ ծրագիրը զերծ է թերություններից (այսուհետ` bug[2]):

*«Թեստավորումը ցույց է տալիս bug-ի առկայությունը, այլ ոչ դրա բացակայությունը»*

Սա նշանակում է, որ թեստավորմանը ո՛չ թե պետք է նայել որպես գործիք, որը մեզ կերաշխավորի թերություններից զերծ (bug free) ծրագիր, այլ գործիք, որը ցույց է տալիս, որ ծրագրում կան թերություններ և մատնանշում է դրանք։ Եթե երբևէ նստեք համակարգչի դիմաց և սկսեք թեստավորել որևէ ծրագիր, ապա պետք է մտածեք հենց այս կերպ: Մտածողության այս ձևը ձեզ թույլ կտա լինել ավելի ազատ և լայն մտահորիզոնով։

**2․** **Սպառիչ թեստավորումն անհնար է**

Թեստավորումը կարող է համարվել սպառիչ, եթե թեստավորվում է ծրագրի ամբողջ ֆունկցիոնալը, հաշվի առնելով բոլոր կարգավորումները և նրանց բոլոր հնարավոր կոմբինացիաները: Դա իրենից ներկայացնում է օգտագործողի կողմից մուտքագրվող տվյալների[3] բոլոր հնարավոր ձևերի ստուգում: Այսինքն, մուտքագրում ենք ոչ միայն ճիշտ, այլև սխալ տվյալներ, հասկանալու համար, թե ինչպես կարձագանքի ծրագիրը: Բացի այս, գոյություն ունի նաև թեստավորման տեխնիկական միջավայր, որի փոփոխությունները ևս պետք է ստուգել: Օրինակ՝ ծրագրի օգտագործման ընթացքում համացանցի կամ համակարգչի հետ կապված խնդիրների առկայությունը և այլն։ Այլ կերպ ասած, բոլոր ֆունկցիաների[4] թեստավորումը` օգտագործելով բոլոր վավեր (valid[5]) և անվավեր (invalid[5]) մուտքերն ու նախապայմանները, անվանում ենք սպառիչ թեստավորում:

Այս ամենից կարելի է եզրակացնել, որ սպառիչ թեստավորումը դառնում է անհնար՝ համենայնդեպս խելամիտ ժամկետներում։ Սպառիչ թեստավորման անհնարինությունն առավել ակնհայտ է դարձնում հետևյալ պարզ, բայց հաճախ հանդիպող օրինակից։

Ենթադրենք, պետք է փորձարկել մուտքային դաշտ[15], որն ընդունում է թվային արժեք՝ 1, 2 կամ 3։ Այս դեպքում ամեն ինչ շատ հեշտ է՝ բավական է ստուգել ծրագրի աշխատանքը 1, 2 և 3 մուտքերի դեպքում։ Իսկ եթե նույն մուտքային դաշտն ընդունում է 1-ից 100 արժեքները, ապա կարող են արդյունավետ լինել թեստավորման արդեն ավտոմացված գործիքները։ Սակայն պատկերացրեք, նույն դաշտը ընդունում է մի քանի միլիարդ արժեքներ: Հնարավոր չէ ստուգել բոլոր հնարավոր արժեքները` ժամանակի սղության և սահմանափակումների պատճառով: Եթե մենք շարունակենք թեստավորել բոլոր հնարավոր թեստային պայմանները, ապա ծրագրի (այսուհետ` application-ի[1]) ծրագրավորման ժամանակը և ծախսերը կշատանան:

Այս խնդրի լավագույն լուծումը դառնում է մուտքագրվող դաշտի **ռիսկային կետերի** (թվային դաշտերի համար ռիսկային կարող են լինել, օրինակ՝ զրոն, բացասակ թվերը, կոտորակային թվերը, եզրային արժեքները և այլն) և կամայական մի քանի արժեքների թեստավորումը։

Եթե որպես օրինակ վերցնենք հաշվիչի գումարման ֆունկցիան ստուգելը, ապա ինքներդ ասեք, թե քանի՞ տվյալ կարելի է ներմուծել սպառիչ թեստավորման համար – միլիոնավոր և միլիարդավոր (անվերջ)։

1. 1+2
2. 1+3
3. 1+4
4. 1+5
5. … 1+ 1800
6. … 1+ 999999999…
7. 1.1+0.1
8. 1.1+0.01
9. 1.1+0.001
10. 1.1+0.0000000001
11. ․․․

Հենց այս կետում է առաջ գալիս հետևյալ հարցը՝ «Ո՞ր պահին է ծրագիրը կամ ֆունկցիոնալը[3] համարվում թեստավորված»: Այս հարցի պատասխանը քննարկվում է հաճախորդի (այսուհետ` Customer[6]) և տեխնիկական թիմի միջև։

Բացի այդ, հենց այս հարցին պատասխանելու համար ստեղծվել են բազմաթիվ թեստավորման տեխնիկաներ, որոնց իմաստը կայանում է նրանում, որ կարողանանք ամենաքիչ թեստավորման դեպքերով ստանալ առավելագույն հուսալիություն։

3․ **Վաղ թեստավորում**

SDLC-ի (Software Development Life Cycle)–ի ցանկացած փուլում կարող են առաջանալ խնդիրներ, բացթողումներ, bug-եր, որոնք կարող են հանգեցնել ծրագրային կոդի փոփոխության, կամ, որն ավելի վատ է, համակարգի ենթակառուցվածքի վերանայման։

Նման փոփոխություններն իրենցից ենթադրում են` արդեն կատարված աշխատանքի վերափոխում, ինչը, հատկապես մոնոլիտ՝ միատարր համակարգերում կարող է հանգեցնել շղթայական փոփոխությունների ամբողջ համակարգում։

Ակնհայտ է, որ նման աշխատանքը ցանկալի չէ առաջին հերթին բիզնեսի տեսանկյունից: Հետևաբար SDLC-ի վաղ փուլերում հայտնաբերված թերությունները շտկելն առավել քիչ ծախսատար են:

*Վաղ թեստավորումների անցկացումը նվազեցնում է թերությունների շտկման ծախսերը:*

Դիտարկենք երկու սցենար․ առաջինն այն է, որ դուք ֆունկցոնալ պահանջների անհամապատսխանություն եք հայտնաբերել դրանց հավաքագրման փուլում, իսկ երկրորդը` երբ անհամապատասխանությունը հայտնաբերվել է ամբողջովին մշակված համակարգում:

Առաջին դեպքում խնդրի լուծումը շատ ավելի քիչ ռեսուրս և ժամանակ կպահանջի, քան երկրորդ դեպքում: Հետևաբար, անհրաժեշտ է թեստավորել ոչ միայն արդեն պատրաստ ծրագրային հատվածը, այլ գործընթացը սկսել ավելի վաղ՝ ֆունկցոնալ պահանջների հավաքման փուլում։

Որպես օրինակ դիտարկենք հաշվիչի ստեղծումը։ Ենթադրենք` Customer-ը ցանկացել է ունենալ նվազագույն ֆունկցիոնալով հաշվիչ՝ գումարում, հանում, բազմապատկում և բաժանում, բայց չի նշել, թե առավելագույնը ինչ թվի հետ պետք է կարողանա աշխատել այն։ Հաշվիչը ստեղծվում է, հանձնվում Customer-ին, սակայն վերջինս պահանջում է, որ այն կարողանա իրար գումարել անհավանական մեծ թվեր։ Իհարկե, խնդիրը լուծելի է, բայց մենք` որպես որակի ապահովման մասնագետներ, պետք է ավելի շուտ հասկանայինք, որ պահանջներն ամբողջությամբ չեն նկարագրված և այդ հարցը բարձրացնեինք դեռ քննարկումների ժամանակ։ Այժմ, երբ հաշվիչը պատրաստ է առանց այդ ֆունկցիոնալի, դրա կոդի մեջ փոփոխություններ կատարելը կարող է ունենալ շատ ծանր հետևանքներ։ Փորձելով շտկել այն` կա հավանականություն, որ այլ տեղերում այն կհանգեցնի այլ և ավելի մեծ bug-երի։

4. **Դեֆեկտների կլաստերներ**

Պրակտիկ փորձից ելենելով կարելի է ենթադրել, որ ծրագրային դեֆեկտները հակված են կլաստերավորման՝ այսինքն, ծրագրի կոնկրետ մեկ կամ մի քանի մասերը (այսուհետ` module[7]), ծրագրային բլոկը կամ ֆունկցիոնալը ամենաշատն են պարունակում bug-երը կամ դրանք առավել հաճախ են օպերացիոն անսարքությունների պատճառ դառնում։

Ըստ Պարետոյի[[1]](#footnote-1) սկզբունքի (80/20), խնդիրների 80%-ը գալիս են module-ների 20%-ից, իսկ մնացած 20% -ը` մնացած 80% module-ներից:

Այսպիսով, կարելի է ենթադրել, որ module-ների 20% -ից եկող թերությունները շտկելով, կարելի ազատվել ամբողջ համակարգի սխալների 80%-ից։

Այստեղ պետք է լինել մի փոքր ինտուիտիվ, հասկանալու համար, թե ծրագրի ո՞ր հատվածն է, որ կարող է պարունակել ամենաշատ bug-երը և սկսել աշխատանքը հենց այդտեղից։ Ցավոք, սա հնարավոր չէ միանգամից սովորել, այն պահանջում է ժամանակ և պրակտիկա։

Ենթադրենք, ունենք application, որը բաղկացած է 10 modul-ից, որոնցից մեկը պատասխանատու է մնացած modul-ների միջև տվյալների փոխանակության համար։ Այս modul-ի աշխատանքն այնպիսին է, որ ամբողջ application-ի 60%-ը կախված է հենց դրա աշխատանքից։ Այսինքն, եթե մենք ստուգենք այս modul-ը և bug-եր չգտնենք, ապա կարելի է ասել, որ ամբողջ application-ը աշխատում է մեծ մասամբ հուսալի։

Որպես որակի ապահովման մասնագետներ, մենք ինքներս, երբեմն, պետք է հասկանանք, թե ինչի վրա է պետք ուշադրություն դարձնել, հատկապես System testing-ի ժամանակ, որի մասին կխոսենք հետագա գլուխներում։

5. **Թունաքիմիկատների պարադոքս**

Երբ դուք հիվանդ ժամանակ միշտ խմում եք նույն դեղերը, ապա ժամանակի ընթացքում զգում եք, որ ձեր օրգանիզմն իմունիտետ է ձեռք բերել այդ դեղերի նկատմամբ և դրանք այլևս չեն օգնում ձեզ։

Նույնն էլ կատարվում է թեստավորման մեջ, եթե մենք անընդհատ կիրառում ենք նույն փորձարկման դեպքերը (այսուհետ` Test Case) առանց փոփոխության:

«Թունաքիմիկատների պարադոքսը»՝ թեստավորման մեջ, նույն Test Case-երը կրկին ու կրկին իրականացնելու գործընթացն է։ Դրանք ժամանակի ընթացքում կկորցնեն իրենց ակտուալությունը և նոր bug-երն ի հայտ բերելու հնարավորությունը:

Այսպիսով, «թունաքիմիկատների պարադոքսը» հաղթահարելու համար անհրաժեշտ է պարբերաբար վերանայել Test Case-երը և ավելացնել կամ թարմացնել դրանք՝ ավելի շատ թերություններ հայտնաբերելու համար:

6. **Թեստավորումը կախված է կոնտեքստից**

Չնայած թեստավորման համար նախատեսված բազմաթիվ ավտոմատացման գործիքների առկայությանը, կամ ընդհանուր կիրառելի մոտեցումներին, թեստավորման ենթակա ցանկացած համակարգ ունի իր առանաձնահատկությունները և թեստավորման պրոցեսի կազմակերպումը կախված է ծրագրի կոնտեքստից:

Օրինակ՝ առցանց բանկային համակարգը պահանջում է թեստավորման այլ մոտեցում՝ ի տարբերություն էլեկտրոնային գրարդարանի կայքի: Եթե երկրորդ դեպքում շեշտը կարելի է դնել կայքի հարմարավետության և ֆունկցիոնալի հասանելիության վրա, ապա բանկային համակարգի դեպքում առաջնային են համակարգի անվտագությունը, հուսալիությունը և խափանումների նկատմամբ կայունությունը։

7. **Սխալի բացակայություն - Անսարքություն**

Համակարգը ոչ միայն պետք է հնարավորինս զերծ լինի թերություններից, այլ նաև պետք է համապատասխանի ֆունկցիոնալ պահանջներին։

99%-ով թերություններից զերծ ծրագիրը կարող է դեռևս անօգուտ լինել, եթե ծրագրակազմում ներառվել են սխալ պահանջներ կամ այն չի բավարարում բիզնեսի կարիքներին: Հետևաբար, թեստավորումը պետք է ի հայտ բերի ոչ միայն տեխնիկական թերությունները, այլ նաև հետևի ծրագրի համապատասխանությանը ֆունկցիոնալ պահանջներին։

Սա հիմնականում այն կետն է, որն ավելի շատ վերաբերում է թեստավորողի մտահորիզոնին։ Որակի ապահովման մասնագետը միայն bug-եր գտնելու համար չէ․ նա պետք է ծրագրին կարողանա նայել հաճախորդի, մենեջերի[12], ծրագրավորողի (Developer[9]) աչքերով, հասկանալու համար, թե արդյո՞ք ստեղծված ֆունկցիոնալն այն է ինչին սպասում է հաճախորդը։

Այս սկզբունքը հանգեցնում է այն մտքին, որ ինչքան էլ ծրագիրը լինի առանց bug-երի, միևնույնն է, եթե այն չի կարողանալու ծառայել իր վերջնական նպատակին, ապա ծրագիրը համարվում է անսարք։

## **1․2 Agile-Scrum Մեթոդոլոգիա**

Ըստ էության ծրագրավորման թիմերի մեծամասնությունն աշխատում են Agile մեթոդոլոգիայով, որն ամենևին բարդ չէ հասկանալու համար։ Շատ կարճ հասկանանք Agile-Scrum-ի աշխատանքի սկզբունքը։

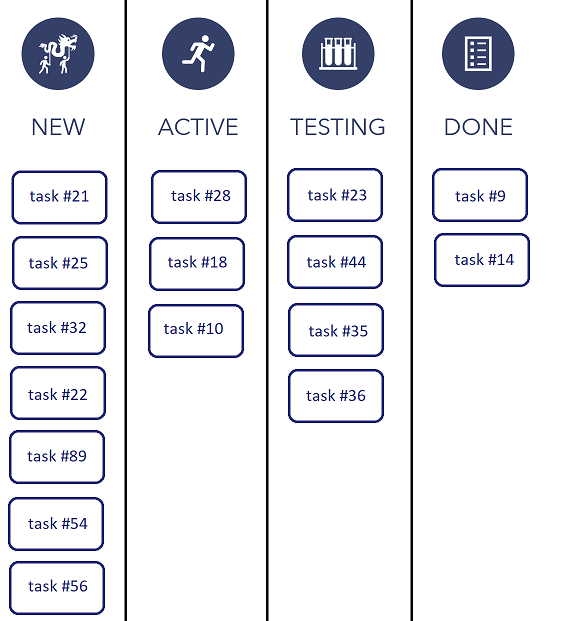
1․ Երբ կատարվում է պահանջների հավաքագրում[7], ստեղծվում են առաջադրանքներ (այսուհետ` task-եր[10]): Դրանք բոլորը գրվում են ինչ-որ մի տեղ՝ backlog-ում։ Սովորաբար օգտագործվող ամենահայտնի գործիքը Jira-ն է, բայց կան նաև այլ ծրագրեր՝ TFS, Trello, Binfire, Basecamp, Asana և այլն․․․

2․ Ամեն 1-4 շաբաթը մեկ (որպես օրինակ դիտարկենք 2 շաբաթը) կատարվում է պլանավորում, թե որ task-երը պետք է իրականացնել։ Ստեղծվում է 2 շաբաթանոց սպրինտ և այնտեղ տեղադրվում են բոլոր այն task-երը, որոնք հարկավոր է իրականացնել։

3․ Եթե ընթացքում առաջանում են կրիտիկական bug-եր, ապա դրանք շատ արագ ավելացվում են ներկա սպրինտի մեջ և ֆիքսվում (ուղղվում)։

4․ Սպրինտի վերջում, եթե որոշ task-եր դեռևս մնում են չկատարված, ապա տեղի է ունենում վերլուծություն, թե ինչո՞ւ ժամանակի մեջ չեն կարողացել տեղավորվել և ձեռնարկվում են մենեջմենթի քայլեր նման խնդիրները շտկելու համար։

Ամեն սպրինտ ունի հիմնականում 4 սյունյակ.



New – այստեղ տեղադրվում են նախատեսված task-երը, որոնք բերված են backlog-ից, և անհրաժեշտ bug-երը

Active – Այստեղ ցույց է տրվում, թե որ developer-ը որ task-ի կամ bug-ի վրա է աշխատում այս պահին

Testing – երբ developer-ը վերջացնում է task-ը/bug-ը, այն տեղափոխում է Under Test սյունյակ, որտեղ QA-ն հասկանում է, որ այն հարկավոր է թեստավորել

Done – Երբ task-ը/bug-ը թեստավորված է, ապա այն վերջնականապես տեղափոխվում է Resolved սյունյակ

Սովորաբար ամեն օր կատարվում են 15 րոպե տևողությամբ stand-up-ներ, որտեղ զեկուցվում է, թե նախորդ օրը ինչ է կատարվել, և եթե որևէ մեկը առընչվել է մեծ խնդրի, ապա օգնում են միմյանց և առաջարկում լուծման տարբերակներ։

**Agile-Kanban Մեթոդոլոգիա**

Կանբանի մեթոդաբանությունը հիմնված է աշխատանքի լիարժեք թափանցիկության և իրական ժամանակի հաղորդակցման կարողությունների վրա: Հետևաբար, կանբանը պետք է դիտարկվի որպես թիմի աշխատանքի գնահատման միակ աղբյուր:

Այս պարագայում այլևս չկան սպրինտներ, բոլոր այն task-երը, որոնք պետք է կատարվեն, տեղադրվում են կանբան board-ում և develope-ները կատարում են դրանք ըստ առաջնահերթության (Priority)։ Այս մոդելն առաջին անգամ իրականացվել է Տոյոտա ընկերության կողմից 1949 թվականին և մեծ հաջողության է հասել, քանի որ այն թույլ է տալիս լինել ավելի ազատ և ճկուն։

## **1․3 Ծրագրի զարգացում - Program Development[9]**

Սկսնակների համար անհասկանալի է, թե ինչպես են սկսում ոչնչից (0-ից) ծրագիր գրել։ Այժմ կփորձենք հասկանալ ընդհանուր պրոցեսը։

Նախ և առաջ պատվիրատուն ներկայացնում է իր պահանջները (requirements[12]) և ներկայացնում ծրագրին վերաբերող փաստաթղթեր (documentation[12]), որի հիման վրա պետք է ստեղծել application-ը։ Հետո մեկ կամ մի քանի շատ լավ ծրագրավորողներ միասին նախագծում են application-ի «Արխիտեկտուրան», այսինքն, թե ինչ սկզբունքով է այն աշխատելու կոդի տեսանկյունից։ Եթե կուզեք, ապա կարող եք մի փոքր ուսումնասիրել՝ փնտրելով “Software Architecture Patterns”։ Հիմնական աշխատանքի սկզբունքը նախագծելուց հետո (եթե նոր եք մտնում այս ոլորտ, ապա իմաստ չունի հասկանալ, թե նախագծումն ինչպես է իրականացվում) արդեն գալիս է այն հարցը, թե ինչպես են մի քանի հոգի կարողանում աշխատել նույն ծրագրի վրա։ Այս հարցի կարգավորման համար կան բազմաթիվ բիզնես լուծումներ, որոնցից ամենատարածվածը GIT-ն է։

Գոյություն ունի branch (ճյուղ) հասկացություն։ Ունենք գլխավոր մեկ գիծ, որի անունը սովորաբար լինում է developer, master կամ նման մի բան, որը համարվում է գլխավոր branch-ը՝ ճյուղը։ Ծրագրավորողը այդ հիմնական ճյուղից ստեղծում է նոր ճյուղ և սկսում է կատարել որոշակի փոփոխություններ։ Այստեղ առաջ է քաշվում ծրագրավորման թիմի կազմակերպման ամենակարևոր հարցերից մեկը, այն է 2 հոգի չպետք է աշխատեն նույն ֆունկցիաների վրա, բայց այս պահին շրջանցենք այդ հարցը։ Փոփոխությունները կատարելուց հետո developer-ը pull request է անում, այսինքն ուղարկում է հարցում, որ իր կողմից կատարված փոփոխությունները միանան հիմնական ճյուղին։ Թիմլիդն է որոշում, այդ հարցումն ընդունել, թե` ոչ։ Թիմլիդը կոդերը վերանայում է (review[13]), ուսումնասիրում, և եթե վատ գրված կոդ է նկատում, ապա ուղղորդում է, թե ինչպես ավելի ճիշտ կլինի գրել, որպեսզի հնարավոր bug-երից խուսափեն։ Որոշ շտկումներից հետո կրկին pull request է արվում և ի վորջո թիմլիդի կողմից հաստատվում ու այդ կոդերը միանում են (merge) հիմնական ճյուղին։

Երբեմն QA-ները վտանգավոր փոփոխությունների դեպքում գնում են համապատասխան branch-ի վրա և Smoke թեստ են կատարում՝ հասկանալու համար, թե արդյո՞ք գլխավոր ֆունկցիոնալը ճիշտ է աշխատում։

Իսկ ծրագրավորողներն ինչպե՞ս են իմանում իրենց task-երի կամ bug-երի մասին։ Իրենք բոլորը գրանցված են լինում ինչ-որ համակարգում՝ TFS, Jira և այլն։ Այնտեղ շատ task-եր և bug-եր են գրված լինում QA-ների[14] և BA-ների[15] կողմից։ Ամեն task ունի իր վերնագիրը, նկարագրությունը, առաջնահերթությունը և կանխատեսված ժամանակը, թե որքան կտևի այդ task-ն անելը։ Սովորաբար, ամեն 2 կամ 1 շաբաթը մեկ տեղի են ունենում ժողովներ ու որոշվում է, թե որ թասկերն են ավելի առաջնային ու հենց դրանք էլ արվում են։ Տեխնիկական մասով հարցերը քննարկվում են թիմլիդի հետ, բիզնես լոգիկայի հետ կապված հարցերը քննարկվում են BA-ների՝ Business Analystn-ների հետ, իսկ bug-երի հետ կապված հարցերը QA-ների (Տե՛ս 1․2 գլուխը)։

Եվ այսպես ամիսներ շարունակ անընդհատ գրվում է կոդ և ստուգվում՝ մինչև հասնում է ժամանակը, երբ անհրաժեշտ է հաճախորդին տրամադրել ստացված ծրագրի տարբերակի (version[16])։

Հնարավոր է, որ ներկայացված նյութը ամբողջապես պարզ չեղավ ձեզ համար, բայց դա բնական է, քանի որ միանգամից ընկալել ու հասկանալ այս ամենը այնքան էլ իրական չէ: Ժամանակն ու աշխատանքային պրոցեսը ամեն բան պարզ ու հասկանալի կդարձնեն հետագայում:

## **1․4 Թեստավորման գործընթաց – STLC**

## **(Software testing life cycle)**

Թեստավորումը գործընթաց է, այլ ոչ թե մեկ գործողություն:

Այս գործընթացը սկսվում է պահանջների վերլուծությունից, թեստի պլանավորումից, այնուհետև նախագծվում են Test Case-եր, ապա դրանք նախապատրաստվում են իրականացման և փակում են թեստը՝ գնահատելով կարգավիճակը:

Այսպիսով, թեստավորման գործընթացը կարելի է բաժանել հետևյալ հիմնական քայլերի.

1․ Պահանջների վերլուծություն

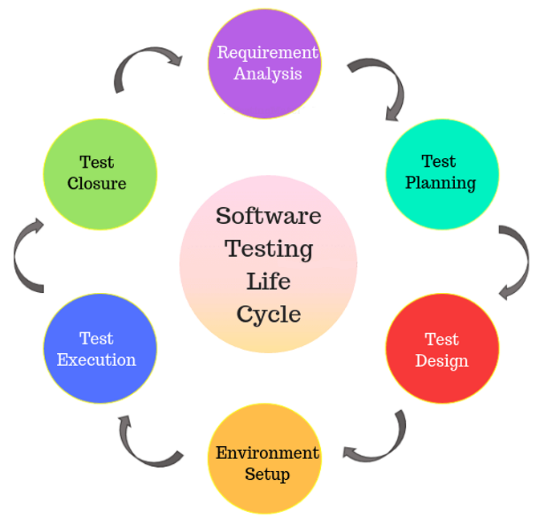
2. Պլանավորում և վերահսկում

3․ Test Case-երի ձևավորում, նախագծում (Design)

4. Միջավայրի կարգավորում

5․ Իրականացում և կատարում

6․ Ելքի չափանիշների[17] գնահատում և հաշվետվություն (Թեստի փակման գործողություններ)



**1․** *Ցանկացած ծրագիր ստեղծելիս` կատարվում է պահանջնորի վերլուծություն։*

Քանի որ QA-ներն այն մարդիկ են, ովքեր կարողանում են դիտարկել ծրագիրը որպես օգտատեր (այսուհետ` user[18]), costomer-ի պահանջները ներկայացնելուն պես կարողանում են հասկանալ, թե որոնք են այն բաց կետերը, որոնք նա` costomer-ը չի դիտարկել։

Օրինակ` հաշվիչի դեպքում, ինչպես վերևում արդեն նշեցինք, QA-ն կարող է և պետք է հարց բարձրացնի, թե ո՞րն է այն ամենամեծ թիվը, որը կարող է արտապատկերվել էկրանի վրա։

Այսինքն, ցանկացած ֆունկցիոնալի աշխատելու բոլոր տարբերակները դիտարկելուց հետո, այն անցնում է հաջորդ փուլ։

**2․** *Թեստավորման պլանավորումն ու ռազմավարությունն այն ուրվագիծն է, որը նկարագրում է ծրագրի մշակման ցիկլի թեստավորման մասը:*

Այն ստեղծվում է, որպեսզի նախագծի մենեջերը, թեստավորողներն ու ծրագրավորողները տեղյակ լինեն թեստավորման գործընթացի որոշ առանցքային հարցերից: Սա ներառում է թեստավորման մեթոդները և տեխնիկաները, աշխատանքի ընդհանուր ժամանակը և նախագծի ռեսուրսները և թեստավորման միջավայրերը: Թույլ է տալիս ընդհանուր առմամբ հասկանալ իրականացվելիք գործընթացները, և ըստ այդմ որոշել անհրաժեշտ թեստային ռեսուրսները, ինչպիսիք են` մարդիկ, թեստավորման միջավայրերը, օգտագործվելիք սարքավորումները (համակարգիչ, հեռախոս, պլանշետ և այլն)։

Այսպիսով, թեստավորման պլանավորումն ու վերահսկումն ունի հետևյալ հիմնական խնդիրները.

* Հասկանալ թեստավորման նպատակները, որոշել շրջանակը և ռիսկերը։
* Որոշել թեստավորման մոտեցումները։
* Իրականացնել թեստային քաղաքականությունը և/կամ թեստային ռազմավարությունը։
* Պլանավորել թեստերի վերլուծության ձևը և bug-ը ձևակերպելու ընդհանուր սկզբունքը, թեստերի իրականացումը և գնահատման չափորոշիչները[17]։
* Ելքի չափորոշիչները (Exit criteria[17]) որոշելու համար անհրաժեշտ է սահմանել այնպիսի չափանիշներ, ինչպիսիք են ծածկույթի չափորոշիչները (coverage criterias[17]): Ծածկույթի չափորոշիչը թեստավորման այն տոկոսային քանակն է, որը պետք է կատարվի թեստավորման ընթացքում: Այսինքն, մենք պետք է ունենանք մի կետ, որտեղ կկարողանանք ասել, որ թեստավորումն ավարտված է:

Այսպիսով, այս փուլում կատարվում են կոնցեպտուալ մշակումներ՝ հասկանում են, թե ինչ ստանդարտով պետք է գրվեն Test Case-երը, թե քանի հոգի է հարկավոր թեստավորման համար, թե ինչպիսի գործիքներ (այսուհետ` tool) պետք է օգտագործվեն և այլն։

**3․** *Test Case-երի նախագծումը փորձում է գտնել հետևյալ խնդիրների լուծումերը․*

* Վերանայել թեստի հիմքերը, այսինքն այն տեղեկատվությունը, որը մեզ անհրաժեշտ է թեստի վերլուծությունը սկսելու և մեր սեփական Test Case-երը ստեղծելու համար: Հիմնականում դա փաստաթղթավորումն է, որի հիման վրա էլ գրվում են Test Case-երը: Այդպիսի փաստաթղթերից են` պահանջները (requirement), նախագծման առանձնահատկությունները, արտադրանքի ռիսկերի վերլուծությունը, նախագծի կառուցվածքը (architecture) և արտաքին տեսքը: Փաստաթղթերի հիման վրա կարելի է հասկանալ, թե ծրագիրն ինչ ֆունկցիոնալներ պետք է ունենա և դրանք խորությամբ ուսումնասիրել:
* Պարզել թեստի նախապայմանները և պայմանները:
* Հստկակեցված նկարագրել ելքային[8] արդյունքները (Expected result):
* Նախագծել Test Case-երը:
* Պահանջների և համակարգի ստուգված լինելու չափորոշիչներ: Օրինակ` որևէ ֆունկցիոնալ ստուգելու համար, ո՞ր պահից կարող ենք ասել, որ այն հաստատ աշխատում է:

Այսպիսով, այս փուլում արդեն սկսվում են գրվել Test Case-եր, հասկացվում է ծրագրի հիմնակայն գործառույթը և նախապատրաստվում են բուն թեստավորմանը։

**4.** *Թեստավորման միջավայրի մշակում և անհրաժեշտ գործիքակազմի կոնֆիգուրացիա․*

Մինչև թեստավորմանն անցնելը, հարկավոր է հասկանալ, թե գրված ծրագիրն ինչպիսի՞ միջավայրում է աշխատելու։ Օրինակ, եթե իրական կյանքում ստացվելու է այնպես, որ server-ը[19] client-ից[20] գտնվելու է ամենաշատը 5000 կմ հեռավորության վրա և դրանից կախված առաջանալու են որոշ խնդիրներ` կապված ցանցի[30] արագության հետ, ապա միջավայրը կկարգավորվի հենց այնպես, ինչպես դա կլիներ 5000 կմ client-server հեռավորության դեպքում։

Կամ այլ օրինակ․ server-ի մասում լինելու են որոշ սահմանափակումներ, որոնք հաճախորդը ի սկզբանե զգուշացնում է։ Այս պարագայում էլ թեստավորման միջավայրում server-ը կարգավորվում է վերջնական կարգավորումներին համապատասխան ձևով։

**5.** *Իրականացում և կատարում.*

Թեստի իրականացման և կատարման ընթացքում մենք հաշվի ենք առնում թեստի մեջ ընդգրկված նախապայմանները, սցենարները, թեստավորման միջավայրը, սկրիպտները, և ցանկացած այլ թեստային ենթակառուցվածք (այսուհետ` testware):

Թեստերի իրականացումն ունի հետևյալ հիմնական խնդիրները.

* կատարել Test Case-եր և դնել առաջնահերթություններ (priority)՝ օգտագործելով որոշ տեխնիկաներ և ստեղծել այդ թեստերի համար թեստային տվյալներ (test data[3]): Application-ը ստուգելու համար անհրաժեշտ է մուտքագրել որոշ տվյալներ՝ գործառույթների մեծ մասը ստուգելու համար: Ցանկացած այդպիսի հատուկ տվյալներ, որոնք օգտագործվում են թեստերում, անվանում ենք թեստային տվյալներ՝ test data:
* Test Case-երից ստեղծել թեստերի հավաքածու (այսուհետ` test suite)` թեստերի արդյունավետ կատարման համար: Test suite-ը (կամ test run) Test Case-երի հավաքածու է, որը հաճախ պարունակում է մանրամասն հրահանգներ և տեղեկատվություն կոնֆիգուրացիաների մասին և օգտագործվում է նմանատիպ Test Case-երը միասին խմբավորելու համար:
* Իրականացնել և հաստատել միջավայրը` Environment-ը[28]:

Թեստի կատարումն ունի հետևյալ հիմնական քայլերը.

* Կատարել test suit-ները և անհատական ​​case-եր` հետևելով թեստային ընթացակարգերին:
* Վերստուգել նախկինում ձախողված թեստերը` ուղղումը հաստատելու համար:
* Մուտքագրել թեստի կատարման արդյունքը և ծրագրի version-ն[16]։ Արդյունքները գրանցվում են թեստային տեղեկամատյանում՝ (այսուհետ` test log)։ Test log-ը իր մեջ ներոռում է թեստավորման ողջ գործընթացը: Այստեղ կարող ենք գտնել մեր իրականացրած թեստերը, հասկանալ, թե ինչ կարգով ենք իրականացրել, թե ով է կատարել այդ Test Case-երը և ո՞րն է թեստավորման կարգավիճակը (անցնել/ձախողվել): Այս նկարագրությունները փաստաթղթավորված են և կոչվում են test log:
* Համեմատել ստացված արդյունքները ակնկալվող արդյունքների հետ:
* Այն դեպքերում, երբ փաստացի արդյունքը չի համընկնում ակնկալվող արդյունքի հետ, գրվում է սխալի հաշվետվություն (այսուհետ` bug report) և բացվում bug:

**6.** *Ելքի չափանիշների գնահատում և հաշվետվություն. Թեստի փակման գործողություններ.*

Ծրագրի ռիսկերի գնահատման հիման վրա սահմանվում են չափանիշներ` յուրաքանչյուր թեստային մակարդակի համար, ըստ որի մենք չափելու ենք «բավարար թեստավորումը»: Այս չափանիշները յուրաքանչյուր նախագծի համար ընտրվում են յուրովի:

Ելքի չափանիշները նկարագրվում են, երբ՝

- կատարվում են բոլոր Test Case-երը որոշակի անցման տոկոսով,

- սխալների, bug-երի քանակը ընկնում է որոշակի մակարդակից ցածր,

- Երբ վերջանում են ժամկետները:

Ելքի չափանիշների գնահատումն ունի հետևյալ հիմնական խնդիրները.

ա) Ստուգել test log-երը թեստային պլանավորման մեջ նշված ելքի չափանիշների նկատմամբ:

բ) Հասկանալ, թե արդյո՞ք անհրաժեշտ է ավելի շատ թեստավորում, կամ պե՞տք է փոխել նշված ելքի չափանիշները, թե՞ ոչ:

գ) Շահագրգիռ կողմերի (customer, management, business analyst) համար կազմել թեստային ամփոփ հաշվետվություն:

Թեստի փակման գործողություններն իրականացվում են ծրագրի ավարտի ժամանակ: Թեստավորումը կարող է փակվել նաև այլ պատճառներով, ինչպիսիք են.

-Երբ նախագիծը չեղարկվում է:

-Երբ ինչ-որ նպատակի հասած ենք լինում:

-Երբ կատարվում է նոր version-ի տրամադրում կամ թարմացում:

Թեստի փակման գործողություններն ունեն հետևյալ հիմնական խնդիրները.

ա) Ավարտել և արխիվացնել testware-ները, ինչպիսիք են գրությունները, թեստավորման միջավայրերը և այլն՝ հետագայում նորից օգտագործելու համար:

բ) Testware-ի պահպանումը կազմակերպությանը հանձնելու համար: Դրանք հետագայում սպասարկելու են ծրագրակազմին:

գ) Գնահատել, թե ինչպես է անցել թեստավորումը և դասեր քաղել հետագա ծրագրերի վրա աշխատելու համար:

## **1․5 Թեստավորման տեսակները կամ դասակարգումն ըստ ֆորմալիզացման աստիճանի**

Սրանք լինում են 3 տիպի։

1․ Test Case-երի վրա հիմնված թեսատավորում: Այս դեպքում թեստավորողն ունի Test Case-եր կամ փաստաթղավորում (documentation) և թեստը կատարում է ըստ այնտեղ գրված քայլերի հերթականության։ Սա թեստավորման ամենատարածված մեթոդն է, որը նաև հնարավորություն է տալիս հասնել application-ի հետազոտության առավելագույն ամբողջականությանը՝ գործընթացի խիստ համակարգվածության, չափումների օգտագործման հեշտության և լայն շրջանակի շնորհիվ:

2․ Հետազոտական թեստավորում: Այս ​​թեստերը, ի տարբերություն Test Case-երի վրա հիմնված թեստերի, կանխորոշված ​​չեն և չեն իրականացվում խստագույն պլանի համաձայն: Այս թեստի ընթացքում թեստավորողը իրեն թույլ է տալիս մի փոքր շեղվել հիմնական Test Case-ից և ընթացքում ավելացնել նոր կետեր։ Կամ հնարավոր է, որ մի քանի կետեր թեստի ընթացքում բաց թողնվեն՝ կախված թեստավորողի տեսակետից։ Թեստավորման այս տեսակը երբեմն շփոթում են «ad hoc» թեստավորման հետ, որը նման է այս թեստավորմանը, սակայն ավելի շատ կրում է իմպրովիզացիոն բնույթ։

3․ Ազատ, ինտուիտիվ թեստավորում (ad hoc testing): Լիովին ոչ ֆորմալ մոտեցում, որի դեպքում նախատեսված չեն օգտագործել ո’չ Test Case-եր, ո’չ check list-եր կամ սցենարներ: Թեստավորողն ամբողջովին ապավինում է իր պրոֆեսիոնալիզմին և ինտուիցիային՝ կիրառելով այնպիսի գործողություններ (հիմնված իր փորձի վրա) որոնք, ինչպես նա կարծում է, կարող են սխալ հայտնաբերել: Այսպես պատահում է այն ընկերություններում, որտեղ չկան բավարար դրամական կամ ժամանակային ռեսուրսներ՝ application-ը Test Case-երով ծածկելու (Coverage) համար։

## **1․6 Թեստ քեյս (դեպք) - Test Case**

Test Case-եր գրելն այն հիմնական գործողություններից մեկն է, որն անում է QA-ը ծրագրի թեստավորման կյանքի ցիկլում (STLC): Բայց արդյունավետ Test Case գրելը շատ լուրջ հմտություններ է պահանջում, որը կարելի է ձեռք բերել ծրագրի խորը ուսումնասիրության և, իհարկե, փորձի շնորհիվ: Լավ Test Case-եր գրելու համար պետք կլինի պարզել պահանջները, սահմանել դրանք և վերլուծել:

**Ի՞նչ է Test Case-ը**

«Test Case-ը մի շարք պայմանների կամ փոփոխականների շարք է, որոնց միջոցով թեստավորողը որոշում է, թե արդյո՞ք ստուգման ենթակա համակարգը բավարարում է նախապես նշված պահանջներին և գործում է ճիշտ, թե՞ այն ունի շտկման կարիք»:

Test Case-երը պետք է լինեն հնարավորինս կարճ գրված ու բովանդակալի, հեշտությամբ հասկանալի, և քայլերը պետք է արագ իրականացվեն: Նրանք պետք է միմյանցից անկախ լինեն և դրանց fail/pass լինելը նույնպես պետք է միմյանցից անկախ լինի: Բնականաբար, յուրաքանչյուր լավ թեստ պետք է որոշի իր ակնկալվող արդյունքը:

Դուք կարող եք դրանք գրել Excel ֆայլերի մեջ կամ կարող եք օգտագործել այնպիսի գործիքներ, ինչպիսիք են` Quality Center, Test Director, TestLink, TestRail և այլն։ Եթե բազմաթիվ Test Case-եր արդեն իսկ գրված են Excel-ով, ապա այժմյան գործիքները հնարավորություն ունեն Import (ներմուծում) անելու։

Test Case-երի հիմնական պարունակությունը։

|  |  |
| --- | --- |
| **Test Case ID** | Թեստ քեյսի չկրկնվող՝ unique, համար |
| **Title** | Այն կարող է պարունակել modulе-ի անուն և պետք է կարճ նկարագրի, թե ինչը պետք է ստուգվի |
| **Assumptions, preconditions** | Ենթադրություններ և նախապայմաններ՝ եթե կան |
| **Test Data** | Թեստային տվյալներ, որոնք հարկավոր են Test Case-ը ստուգելու համար (թվային արժեքներ, նկար, երգ․․․) |
| **Test Steps** | Թեստավորման մանրամասն քայլերի նկարագրություն |
| **Expected results** | Ինչպես պետք է ծրագիրը պահեր իրեն ըստ պահանջների |

**Ինչպե՞ս գրել արդյունավետ Test Case-եր**

Test Case-երը գրելիս` պետք է հետևեք որոշ քայլերի, որպեսզի համոզվեք, որ կգրեք լավ և արդյունավետ Test Case-եր:

1. Որոշեք թեստավորման շրջանակը և նպատակը․

Առաջին հիմնական կետը, նախքան արդյունավետ Test Case-եր գրելը այն է, որ դուք պետք է որոշեք ստուգվելիք application-ի պահանջները: Դուք պետք է հասկանաք թեստավորման նպատակը, պետք է հասկանաք առանձնահատկությունները և օգտագործողի պահանջները:

2. Սահմանեք, թե ինչպես պետք է իրականացնել թեստավորման գործողությունները.

Մինչև արդյունավետ Test Case-եր գրել սկսելը, նախ պետք է սկսեք սահմանել արդյունավետ թեստային սցենարներ, պետք է ավելի լավ պատկերացում ունենաք ծրագրի ֆունկցիոնալ պահանջների մասին: Դուք պետք է իմանաք այն մասին, թե ինչպես կարող են օգտագործվել բոլոր գործողությունները, որպեսզի թեստի սցենարները և Test Case-երը գրվեն այս կարևոր ցուցումների հիման վրա: Դուք պետք է իմանաք իրական կյանքում կատարվելիք գործողություննրը։

3․ Հասկացեք ոչ ֆունկցիոնալ պահանջները:

Ֆունկցիոնալ պահանջներին զուգահեռ, ոչ ֆունկցիոնալ պահանջները հավասարապես կարևոր են: Հիմնականում ոչ ֆունկցիոնալ պահանջների մեջ են ներառվում օպերացիոն համակարգի, hardware-ի պահանջները: Այստեղ են դիտարկվում անվտանգության առանձնահատկությունները, ինչպես նաև լրացուցիչ նախադրյալները, որոնք անհրաժեշտ են թեստային տվյալների նախապատրաստման և թեստավորման համար:

Օրինակ, պատկերացնենք մի սցենար, որտեղ բանկային տվյալների ֆորմայի մեջ լրավցում է user-ի ինֆորմացիան: Այդ դեպքում ծրագրավորողի կողմից պետք է սահմանվի համապատասխան time-out, որպեսզի համոզվենք, որ ծրագրը չի կանգնել օգտագործողի տեղեկատվությունը լրացնելու ժամանակ: Միևնույն ժամանակ, եթե օգտագործողը մի քանի անգամ դադար է տալիս որոշակի գործողություն կատարելու ընթացքում, ապա համակարգը պետք է ինքնաբերաբար դուրս գա ծրագրից, կանխորոշված ուշացումից հետո, որպեսզի ապահովի ծրագրի անվտանգությունը:

4. Սահմանեք Test Case-երի շրջանակը:

Test Case-ի շրջանակները նախագծվում են` կախված համակարգի պահանջներից: Սովորաբար, Test Case-երը պետք է իրականացնեն և ծածկեն գործառույթները, համատեղելիությունը, UI ինտերֆեյսը, սխալի տոլերանտությունը և մի քանի այլ կատեգորիաների կատարումը:

5. Ծանոթացեք ծրագրային modul-ների նշանակությանը:

Պարզ և հեշտ է հասկանալ թեստավորվող ծրագրի module-ների վերլուծության նշանակությունը: Լավ Test Case-եր գրելու համար դուք պետք է ծանոթանաք դրանց կարևորությանը և պետք է հասկանաք դրանց ինտեգրումը։

Օրինակ բերենք առցանց գնումների վեբ կայքից․ ենթադրենք փորձում եք թեստավորել պատվերը գնումների զամբյուղ տեղափոխելու ֆունկցիան: Այս դեպքում կարևոր է հաշվի առնել ո’չ միայն այն փաստը, որ գործողությունը բարեհաջող կատարվում է կամ` ո’չ, այլև պետք է հաշվի առնել գրապահոցների կառավարումը և ստուգել, թե արդյո՞ք այդ գիրքը առկա է պահեստում և կարո՞ղ է արդյոք բարեհաջող պատվեր գրանցվել և խանութից դուրս բերվել, թե՞ ոչ: Եթե գիրքը պահեստում չկա, ապա չպետք է լինի հնարավորություն այն զամբյուղում ավելացնելու։

## **1․7 Կիրառելի case (դեպք) - Use case․ կյանքի իրական սցենարներ**

Use case-երը գրավոր նկարագրություն են այն մասին, թե ինչպես են օգտատերերը (այսուհետ` user) կիրառելու ձեր ծրագիրը: Այն ուրվագծվում է user-ի տեսանկյունից: Use case-ի յուրաքանչյուր դեպք ներկայացված է որպես պարզ քայլերի հաջորդականություն՝ սկսած user-ի նպատակից և ավարտվում է այդ նպատակը կատարելով:

Օրինակ` մենք ամեն օր մտնում ենք facebook և մեծ մասամբ սեղմում ենք like կոճակը, comment գրում, զբաղվում նամակագրությամբ, սակայան համեմատաբար քիչ ենք օգտագործում օրինակ խմբեր որոնելու համար նախատեսված ֆիլտրերը և այլն: Իհարկե facebook-ի վրա դժվար է յուրօրինակ քիչ օգտագործվող ֆունկցիոնալներ գտնել, բայց դրանք իրականում կան։ Use case-երն այստեղ կնկարագրվեն որպես comment գրել, սեղմել like կոճակը, ուղարկել նամակ և այլն։ Այսինքն սրանք այն դեպքերն են, որոնք ամենաշատն են օգտագործվում վերջնական user-ի կողմից։

Use case-երը շատ արժեքավոր են, որովհետև դրանք օգնում են հասկանալ, թե ծրագիրն ինչպես պետք է իրեն պահի աշխատանքի ժամանակ, ինչպես նաև, դրանք օգնում են կազմակերպել brainstorm-եր (մտքերի փոխանակում, քննարկում) , որպեսզի հասկանալ, թե ինչպես կարելի է զարգացնել որևէ ֆունկցիոնալ։ Այստեղ նկարագրվում են բոլոր այն նպատակները, որոնք հարկավոր են վերջնական user-ին։

Use user-ները նկարագրում են քայլերի այն ամենահավանական հերթականությունը, որոնք կանխատեսվում են, որ կկատարի վերջնական օգտատերը։

|  |  |
| --- | --- |
| **Ինչ են ներառում Use Case-երը** | **Ինչեր չեն ներառում Use Case-երը** |
| 1.Ով է օգտագործում ծրագիրը  2.Ինչ է ցանկանում անել user-ը  3.User-ի վերջնական նպատակները  4.Այն քայլերը, որոնք user-ը կկատարի՝ իր նպատակին հասնելու համար  5.Ծրագիրն ինչպես պետք է արձագանքի[37] այդ հարցմանը | 1.Իրականացման հատուկ լեզու  2.User ինտերֆեյսի մանրամասն տեսքը |

## **1․8 Ի՞նչ է testware-ը**

Մենք գիտենք, որ hardware ինժեներները արտադրում են ապարատային սարքավորումներ, իսկ software ինժեներները ստեղծում են ծրագրեր` software application-ներ՝ hardware-ների համար:

Այս ամենի նման՝ QA ինժեներները պատրաստում են testware՝ software application-ների համար:

Testware բառը կազմված է «TEST» և «WARE» բառերից, որտեղ`

Test-ը այն է, ինչ մենք կատարում ենք մեր ծրագրի վրա:

Ware - նշանակում է, թե ինչ կանոնավոր լրացուցիչ պարագա մենք պետք է օգտագործենք test case գրելու և թեստավորման համար: Ծրագիրը, փաստաթղթավորումը և թեստավորման հետ կապված ամեն բան անցնում են testware-ի միջով։

Testware-ը թեստավորման պլանավորման, նախագծման և իրականացման, թեստի ընթացքում օգտագործված բոլոր հասկացությունները սահմանելու համար անհրաժեշտ տերմին է, ինչպիսիք են փաստաթղթերը, գրությունները, մուտքերը, ակնկալվող արդյունքները, կոնֆիգուրացիոն և cleanup (մաքրման) պրոցեդուրաները, ֆայլերը, տվյալների բազաները[18], միջավայրը և ցանկացած ծրագիր կամ թեստերում օգտագործվող հավելյալ գործիքակազմերի ծառայությունները:

## **1․9 Սխալի հաշվետվություն կամ զեկույց - Bug report**

Եթե սխալի՝ bug-ի, մասին ձեր զեկույցը արդյունավետ է, ապա դրա արագ ուղղվելու (ֆիքսվելու) հնարավորություններն ավելի մեծ են: Այսպիսով, bug-ի շտկումը կախված է նրանից, թե որքանով եք դա արդյունավետ հայտնում: Bug report կազմելու համար օգտվում են բազմաթիվ բիզնես լուծումներից, ինչպիսիք են, ճրինակ Jira-ն։ Bug-ի մասին գեղեցիկ տեղեկացնելը ոչ այլ ինչ է, քան հմտություն: Եթե թեստավորողը ճիշտ չի հայտնում սխալի մասին, ծրագրավորողը, ամենայն հավանականությամբ, չի անդրադառնա այդ bug-ին՝ հայտարարելով, որ այն անարդյունավետ է: Իսկ թեստավորողն այդ ժամանակ իրեն վատ կզգա և իրեն արդարացնելու համար կասի այնպիսի խոսքեր, ինչպիսիք են «Ես ճիշտ եմ զեկուցել», «կարող եմ վերարտադրել», «Դա իմ մեղքը չէ» և կստեղծվի տհաճ իրավիճակ: Հենց դրանից խուսափելու համար անհրաժեշտ է հնարավորինս ճիշտ և դիպուկ նկարագրել bug-երը:

Որո՞նք են application-ի bug-ի մասին զեկույցի լավ որակները:

Յուրաքանչյուրը կարող է գրել bug report: Բայց ոչ բոլորն են կարողանում այն գրել արդյունավետ և ճիշտ:

Դուք պետք է կարողանաք տարբերակել լավ bug report-ը վատից: Դա շատ պարզ է, կիրառեք հետևյալ բնութագրերը և տեխնիկան` bug-ի մասին հաշվետվություն տալու համար:

**Բնութագրեր և տեխնիկա**

1․ Ամեն bug պետք է լինի առանձին և ունենա իր չկրկնվող համարը։ Իհարկե, ավտոմատացված համակարգերում դա կատարվում է ինքնաբերաբար, սակայն, եթե չունեք նման հնարավորություն, ապա ամեն bug-ին տվեք հերթական չկրկնվող համար։ Իսկ հետո նշեք, թե որ bug-երն եք հենց դուք գտել։

2․ Ձեր գրածը պետք է լինի վերարտադրելի. եթե ձեր bug-ը չի կարող վերարտադրվել, ապա այն երբեք չի շտկվի: Պետք է հստակ նշել bug-ը վերարտադրելու քայլերը: Մի ենթադրեք կամ մի շրջանցեք վերարտադրողական որևէ քայլ: Այն bug, որը նկարագրված է քայլ առ քայլ, հեշտ է վերարտադրել և շտկել:

3․ Եղեք կարճ. խնդրի մասին շարադրություն մի գրեք: Եղեք հստակ և իմաստուն: Փորձեք խնդիրը նվազագույն բառերով ամփոփել արդյունավետ կերպով: Մի համատեղեք բազմաթիվ խնդիրներ, նույնիսկ, եթե դրանք կարծես նման են: Յուրաքանչյուր խնդրի համար գրեք տարբեր զեկույցներ:

**Bug-ի արդյունավետ հաշվետվություն**

1․ Սխալների մասին զեկուցումը application-ի թեստավորման կարևոր կողմն է: Սխալների արդյունավետ զեկույցը լավ կոմունիկացիայի՝ շփման կողմեր է հաստատում ծրագրավորման թիմի հետ և թույլ է տալիս խուսափել խառնաշփոթներից կամ թյուրիմացություններից:

2․ Bug-ի մասին լավ զեկույցը պետք է լինի պարզ և հակիրճ, առանց բացակայող կետերի: Ցանկացած հստակության բացակայություն հանգեցնում է թյուրիմացության և դանդաղեցնում է նաև ծրագրավորման գործընթացը: Թերությունները գրելն ու զեկուցելը թեստավորման կյանքի ցիկլի ամենակարևոր, բայց անտեսված ոլորտներից է:

3․ Լավ գրելը շատ կարևոր է վրիպակ ներկայացնելու համար: Ամենակարևոր կետը, որը թեստավորողը պետք է հիշի, զեկույցում հրամանի տարրեր պարունակող արտահայտություններ չօգտագործելն է: Սա խախտում է բարքերը և ստեղծում է անառողջ աշխատանքային հարաբերություններ: Օգտագործեք առաջարկող երանգ:

4․ Մի՛ կարծեք, որ եթե ծրագրավորողը սխալ է թույլ տվել, ապա կարող եք կոպիտ բառեր օգտագործել: Ո՛չ։

5․ Հաղորդելուց առաջ հավասարապես կարևոր է ստուգել, թե արդյո՞ք նույն bug-ի մասին այլ զեկույց հաղորդվել է, թե` ոչ, որպեսզի չստացվի, որ նույն bug-ը գրվի 2 կամ ավել անգամ: Կրկնվող bug-ը թեստավորման ցիկլի մեջ ավելորդ բեռ է: Ստուգեք հայտնի bug-երի ամբողջ ցուցակը: Ժամանակ առ ժամանակ լինում են դեպքեր, երբ ծրագրավորողները կարող են իմանալ այդ խնդրի մասին, բայց անտեսել այն՝ թողնելով հետագայի վրա: Կարող են օգտագործվել նաև Bugzilla-ի նման գործիքներ, որոնք ավտոմատ որոնում են կրկնօրինակ bug-երը: Այնուամենայնիվ, ավելի լավ է ձեռքով որոնել ցանկացած կրկնօրինակ bug:

6․ «Ինչպե՞ս» և «որտե՞ղ է» առաջացել bug-ը։ Զեկույցը պետք է հստակ պատասխան տա, թե ինչպես է իրականացվել թեստը և որտեղ է գտնվել թերությունը: Ընթերցողը պետք է հեշտությամբ վերարտադրի սխալը և գտնի, թե որտեղ է այդ սխալը:

7․ Հիշեք, որ “Bug report” գրելու նպատակը ծրագրավորողին հնարավորություն տալն է պատկերացնել խնդիրը: Նա պետք է հստակ հասկանա bug-ի զեկույցի թերությունը: Մի մոռացեք տրամադրել բոլոր համապատասխան տեղեկատվությունը՝ testware-ը, որը կփնտրի ծրագրավորողը:

8․ Նաև հիշեք, որ վրիպակների մասին զեկույցը կպահպանվի հետագա օգտագործման համար և պետք է լավ գրված լինի պահանջվող տեղեկատվությամբ: Օգտագործեք իմաստալից նախադասություններ և պարզ բառեր ձեր սխալները նկարագրելու համար: Մի օգտագործեք խառնաշփոթ մտքեր, որոնք վատնում են կարդացողի ժամանակը:

9․ Հաղորդեք յուրաքանչյուր bug որպես առանձին խնդիր: Bug-ի մեկ զեկույցում բազմաթիվ խնդիրների ներկայացնելը սխալ է, քանի որ այդ դեպքում դուք չեք կարողանա այն փակել, քանի դեռ բոլոր խնդիրները լուծված չեն:

10․ Հետևաբար ավելի լավ է հարցերը բաժանել առանձին վրիպակների: Սա ապահովում է յուրաքանչյուր bug-ի առանձին իրականացվելիությունը: Լավ գրված bug-ի մասին զեկույցը օգնում է ծրագրավորողին վերարտադրել սխալը իրենց տերմինալում:

**Ինչպե՞ս հաղորդել Bug:**

Օգտագործեք bug-ի մասին զեկույցի հետևյալ պարզ ձևանմուշը.

Այստեղ ներկայացված է bug-ի պարզ զեկույցի ձևաչափը: Այն կարող է լինել տարբեր` կախված ձեր կողմից օգտագործվող վրիպակի մասին զեկույցի գործիքից: Եթե ​​bug-ի մասին զեկույց եք գրում ձեռքով, ապա պետք է լրացվեն բոլոր այն դաշտերը, որոնք ավտոմատացված տարբերակով արվում են ինքնուրույն, օրինակ` bug-երի համարակալումը և այլն:

Զեկուցող - Ձեր անունը և էլ. Փոստի հասցեն:

Պռոդուկտ - Ո՞ր պռոդուկտի մեջ եք գտել այդ սխալը: Սովորաբար մեկ ամբողջական պռոդուկտը բաղկացած է լինում մի քանի ենթահամակարգերից։

Version - Պռոդուկտի վերսիան, եթե այդպիսիք կան:

Բաղադրիչ - Սրանք պռոդուկտի հիմնական ենթամասերն են (oրինակ` search, history):

Պլատֆորմ - Նշեք ծրագրի պլատֆորմը, որտեղ գտաք այդ bug-ը: Գոյություն ունեն տարբեր պլատֆորմներ, ինչպիսիք են «PC», «MAC», «HP», «Sun» և այլն:

Օպերացիոն համակարգ - Նշեք բոլոր գործող համակարգերը, ինչպիսիք են Windows, Linux, Unix, SunOS, Mac OS, որտեղ դուք գտել եք այդ սխալը,: Նշեք OS- ի տարբեր տարբերակները, ինչպես նաև, օրինակ` Windows NT, Windows 2000, Windows XP և այլն:

Նկարագրություն - սխալի մանրամասն նկարագրություն:

Նկարագրության դաշտի համար օգտագործեք հետևյալ դաշտերը.

* Վերարտադրեք քայլերը - Պարզ նշեք bug-ի վերարտադրման քայլերը:
* Ակնկալվող արդյունք (Expected result) - Ինչպե՞ս պետք է ծրագիրը աշխատեր վերը նշված քայլերից հետո:
* Փաստացի արդյունք (Actual result) - Ո՞րն է վերը նշված քայլերից հետո իրական արդյունքը, այսինքն՝ սխալի պահվածքը:

Սրանք bug-ի զեկույցում կարևոր քայլերն են: Կարող եք նաև ավելացնել «Հաղորդման տեսակը» որպես ևս մեկ դաշտ, որը նկարագրելու է սխալի տեսակը:

Առաջնահերթություն-Priority - Ե՞րբ պետք է շտկվի վրիպակը: Առաջնահերթությունը սովորաբար սահմանվում է P1- ից P5- ի սահմաններում: P1- ը որպես «Շտկել սխալը որքան հնարավոր է արագ», իսկ P5-ը` «Շտկել, երբ ժամանակ ունենաք»:

Խստություն – Severity - Սա նկարագրում է սխալի ազդեցությունը:

*Խստության տեսակները.*

* Արգելափակիչ-blocker. Հետագա թեստավորման աշխատանքներ չեն կարող իրականացվել:
* Կրիտիկական-Critical. Ծրագրի խափանում, տվյալների կորուստ:
* Մեծ-Major. Գործառույթի մեծ կորուստ:
* Փոքր-Minor. Գործառույթի փոքր կորուստ:
* Չնչին-Trivial. UI-ի որոշ բարելավումներ:
* Բարելավում-Enhancement. Նոր գործառույթի կամ գոյություն ունեցող ֆունկցիոնալի բարելավում:

Կարգավիճակը. Երբ bug եք մուտքագրում, ապա դրա ստատուսը կլինի “New”:

Հետագայում bug-ը անցնում է տարբեր փուլեր, ինչպիսիք են Fixed, Verified, Reopen, Won't Fix և այլն:

Հանձնարարված (Assign To) - Եթե դուք գիտեք, թե որ ծրագրավորողն է պատասխանատու այդ module-ի համար, որտեղ տեղի է ունեցել bug-ը, ապա կարող եք նշել այդ ծրագրավորողին և նրա էլ. Փոստի հասցեն: Հակառակ դեպքում, այն թող մնա դատարկ և մենեջերը հետագայում այն կկցի համապատասխան մարդուն:

Եզրակացություն - Bug-ի մի հակիրճ ամփոփում հիմնականում 60 կամ ավելի քիչ բառերով: Համոզվեք, որ ձեր ամփոփագիրը նկարագրում է, թե որն է խնդիրը և որտեղ է գտնվում:

**Զեկույցի տեսակները ներառում են.**

1) Կոդավորման սխալ

2) Դիզայնի սխալ

3) Նոր առաջարկ

4) Փաստաթղթերի վրա հիմնված ֆունկցիոնալ

**Ստորև բերված են մի քանի այլ լրացուցիչ խորհուրդներ bug-երի մասին լավ զեկույց կազմելու համար.**

# 1) Անմիջապես զեկուցեք խնդրի մասին

 Եթե ​​թեստավորման ժամանակ գտնում եք որևէ սխալ/bug, ապա պետք չէ սպասել, որ հետագայում մանրամասն սխալների մասին զեկույց գրեք: Փոխարենը, անմիջապես գրեք bug-ի մասին զեկույցը: Սա կապահովի վրիպակների լավ և վերարտադրելի զեկույց: Եթե ​​որոշեք գրել սխալների մասին զեկույցը հետագայում, ապա ձեր զեկույցում կարևոր քայլերը բաց թողնելու հավանականությունը մեծ կլինի։

# 2) Bug-ի հաշվետվություն գրելուց առաջ երեք անգամ վերարտադրեք սխալը

Ձեր bug-ը պետք է վերարտադրելի լինի: Համոզվեք, որ ձեր քայլերը բավականաչափ ճշգրիտ են, որպեսզի վերարտադրեն bug-ը, առանց որևէ երկիմաստության: Եթե ​​ձեր վրիպակը ամեն անգամ չէ, որ վերարտադրվում է, դուք կարող եք այն ամեն դեպքում գրել և նշել, որ այն ունի պարբերական բնույթ։

# 3) Փորձեք նույն սխալի առաջացումը նմանատիպ այլ module-ների վրա

Երբեմն developer-ը օգտագործում է նույն կոդը տարբեր նմանատիպ module-ների համար: Այսպիսով, մեկ module-ում հայտնաբերված սխալը մեծ հավանականություն կա նաև, որ կգտնվի նմանատիպ այլ module-ներում ևս: Դուք նույնիսկ կարող եք հանդիպել ձեր գտած bug-ի ավելի ծանր տարբերակին:

# 4) Սխալների լավ զեկույց գրեք

Սխալների ամփոփումը կօգնի developer-ներին արագորեն վերլուծել սխալի բնույթը: Անորակ զեկույցը անտեղի կվատնի developer-ի և թեստավորողի ժամանակը: Հարկավոր է հնարավորինս ճիշտ ու հակիրճ ամփոփագիր ներկայացնել:

# 5) Կարդացեք bug-ի մասին զեկույցը նախքան ներկայացնելը

Կարդացեք բոլոր նախադասությունները, ձևակերպումները և քայլերը, որոնք օգտագործվում են bug-ի մասին զեկույցում: Տեսեք, թե արդյո՞ք որևէ նախադասություն երկիմաստություն չի առաջացնում, որը կարող է բերել սխալ մեկնաբանությունների: Սխալ բառերից կամ նախադասություններից պետք է խուսափել, որպեսզի սխալի զեկույցը լինի հստակ:

# 6) Չօգտագործեք վիրավորական լեզու

Հաճելի է, որ դուք լավ աշխատանք եք կատարել և bugբագ եք գտել, բայց մի օգտագործեք այս փաստը ծրագրավորողին քննադատելու կամ որևէ անհատի վրա հարձակվելու համար: Մարդկային հարաբերություններն ու ընկերական մթնոլորտը աշխատավայրում ամենակարևորներից մեկն է:

Ահա բոլոր այն կետերը, որոնք պետք է ներառվեն bug report-ի մեջ։



**Bug report-ի օրինակ**

**Title** - The deposit\_payment\_provider\_click event is not triggered when the icon of Payment provider clicked

**Preconditions:**

* Have registered client with on IFX production and GTM configured

**Steps to reproduce:**

* Go to Client Portal of IFX and login with valid credentials
* Go to Transactions > Deposits
* Click on the Credit Card icon
* Check the events from GTM

**Actual Result:**

* The event is of deposit\_payment\_provider\_click is not triggered

**Expected Result:**

* The event of deposit\_payment\_provider\_click should be triggered

**Եզրակացություն**

Կասկած չկա, որ bug-ի մասին ձեր զեկույցը պետք է լինի որակյալ փաստաթուղթ: Կենտրոնացեք սխալի մասին լավ զեկույցների վրա և որոշ ժամանակ անցկացրեք այս առաջադրանքի վրա, քանի որ սա թեստավորողի, developer-ի և մենեջերի միջև կապի հիմնական կետն է: Bug-ի մասին լավ զեկույց գրելու ձեր ջանքերը ոչ միայն կխնայեն ընկերության ռեսուրսները, այլև կստեղծեն լավ հարաբերություններ ձեր և developer-ների միջև: Ավելի լավ արտադրողականության համար գրեք ավելի լավ Bug report: Դուք bug զեկույց գրելու մասնագետ եք:

## **1․10 Բագի տեսակավորում - Bug triage**

Bug-ի տեսակավորումը գործընթաց է, որտեղ յուրաքանչյուր վրիպակին տրվում է առաջնահերթություն (այսուհետ` priority) ելնելով դրա խստությունից (այսուհետ` severity), հաճախականությունից, ռիսկից և այլն: Bug Triage-ի նպատակը թերությունների գնահատումն է, առաջնահերթությունն ու նշանակումը: Թիմը պետք է վավերացնի թերության ծանրությունը, ըստ անհրաժեշտության փոփոխություններ կատարի, վերջնական լուծումներ տա թերությունների լուծմանը և ռեսուրսներ նշանակի: Հիմնականում օգտագործվում է agile project manager-ի մեջ:

Որոշակի bug-եր գտնելուց հետ տեղի է ունենում bug-երի տեսակավորման ժողով, որին մասնակցում են project manager-ը, թեստավորման թիմլիդը, տեխնիկական լիդերը, developer-ների թիմլիդը, ինչպես նաև developer-ները, թեստավորողները և բիզնես անալիտիկները։

Ի՞նչ է տեղի ունենում թերությունների տեսակավորման ժողովի ժամանակ․

1․ Թեստավորման թիմի ղեկավարը զեկույց է ուղարկում նոր թերությունների մասին, վերլուծվում է յուրաքանչյուր թերություն՝ պարզելու համար, թե արդյո՞ք դրանց վերագրվել է ճիշտ priority և severity:

2․ Անհրաժեշտության դեպքում առաջնահերթությունները վերադասավորվում են:

3․ Թերությունները վերլուծվում և գնահատվում են ըստ դրանց ծանրության աստիճանի:

4․ Տեղի է ունենում քննարկում թերությունների բարդության, ռիսկերի, մերժման, bug-երի վերանշանակման վերաբերյալ:

5․ Bug-երը կցվում են համապատասխան developer-ներին։

6․ QA ինժեները փոփոխություններ է կատարում յուրաքանչյուր թերության վերաբերյալ և քննարկում դրանք համապատասխան մարդու հետ:

7․ «Մեկնաբանություն» դաշտը թարմացվում է՝ նշելով ժողովի արդյունքում եզրակացված հիմնական կետերը:

Այսպիսով bug-ի տեսակավորման ժամանակ որոշվում է, թե որքան կարևոր է այն, երբ կշտկվի, ով կշտկի և որքան կտևի դրա շտկման ժամանակը։

## **1․11 Կոնֆիգուրացիայի/թողարկման կառավարում – Configuration/Version management**

Ինչպես գիտենք, շատ ծրագրեր նախատեսված են միանգամից մի քանի հաճախորդների համար։ Սակայն ամեն հաճախորդի պահանջը շատ անհատական է (որտեղից էլ և սկսվում են խնդիրները)։

Օրինակ կան պահեստների, HR, CRM ծրագրեր, որոնցից ամենահայտնին 1C ծրագիրն է, որոնք նախատեսված են ապրանքները գրանցելու, տեղաշարժերը գրանցելու և այլ ծառայությունների համար։ Այդ նույն ծրագիրը կարող է օգտագործվել ինչպես շատ հասարակ փոքրիկ սուպերմարկետում կամ խանութում, այնպես էլ շատ մեծ ցանց ունեցող ընկերությունում։ Բնականաբար փոքր և մեծ բիզնեսի պահանջներն այդ ծրագրից տարբեր են։ Փոքր բիզնեսին այդ ծրագիրը հարկավոր է ավելի քիչ ֆունկցիոնալներով, իսկ մեծ բիզնեսին` շատ նոր գործիքներով և գործառույթներով։ Օրինակ` փոքր խանութը կարող է չունենալ պահեստներում ապրանքի տեղաշարժի գրանցման հնարավորություն, քանի որ նրա պահեստը մեկն է, իսկ մեծ ընկերությանը, որն ունի մի քանի պահեստ, արդեն հարկավոր կլինի։

Իսկ ինչպե՞ս են լուծվում նման խնդիրները։

Նման խնդիրներն ունեն 2 լուծման տարբերակ․

1․ Պետք է ծրագրերը պահվեն տարբեր version-ներով։ Այսինքն ունենալ նույն ծրագրի մի քանի տարբերակ՝ առանձնահատուկ ֆունկցիոնալներով։

2․ Ունենալ մեկ ծրագիր, բայց այն ամբողջովին դարձնել կոնֆիգուրացվող (կարգավորվող)։

Կրկին խոսենք օրինակներով։ Եթե փոքր խանութը ցանկանար ունենալ ծրագիրը առանց ապրանքները պահեստների միջև տեղաշարժելու հնարավորության, ապա առաջին տարբերակի դեպքում այդ ծրագիրը կլիներ առանձին և դա կտրվեր հաճախորդին, իսկ երկրորդ տարբերակի դեպքում, պարզապես, ծրագրի այդ ֆունկցիոնալը կանջատվի (կարգավորումներով կամ պրիվիլեգիաներով) և կտրվի հաճախորդին։

Իհարկե, այժմ աշխարհը գնում է 2-րդ տարբերակի ուղղությամբ, քանի որ փորձը ցույց է տվել, որ ավելի ձեռնտու է ունենալ մեկ ծրագիր և շատ կոնֆիգուրացիաներ, քան ունենալ նույն ծրագրի հարյուրավոր տարբերակներ և խճճվել դրանց սպասարկման խնդիրների մեջ։

Սակայն այս դեպքում հարկավոր է այդ ամենը փաստաթղթավորել, քանի որ 100 հաճախորդ ունենալու դեպքում բոլորի կնֆիգուրացիաները տարբեր են լինում և դրանք հարկավոր է որևէ կերպ վերահսկել։

Այսպիսով Configuration մենեջմենթը գործընթաց է, երբ վերահսկվում և փաստաթղթավորվում են որևէ ծրագրի ընթացքում կատարված փոփոխությունները: Թողարկման վերահսկումը, փոփոխության վերահսկումը և վերանայման վերահսկումը ծրագրի կառավարման կարևոր կողմերն են: .

## **1․12 Ի՞նչ են bug leakage-ը և bug release-ը**[21]

**Bug leakage**-ը այն երևույթն է, երբ թեստավորման ընթացքում գոյություն ունեցող bug-ը չի հայտնաբերվում և ծրագրի release տալուց ենթադրվում է, որ ամեն բան աշխատում է այնպես, ինչպես սպասվում է: Սակայն release տալուց հետո վերջնական user-ը հայտնաբերում է որոշակի թերություններ, որոնք չի հայտնաբերել QA թիմը։ Այս երևույթը կոչվում է Bug leakage (սխալի արտահոսք), քանի որ bug-ը հասնել է վերջնական user-ին նախքան այն հայտնաբերվելը: Ստացվում է, որ վերջնական օգտատերերը կարողանում են հայտնաբերել bug: Իհարկե, սա կյանքում հաճախ հանդիպող դեպքերից է, քանի որ հնարավոր չէ 100%-ով խուսափել bug-երից (տե՛ս որակի ապահովման 1-ին և 2-րդ սկզբունքները)։

Նմանատիպ bug-երը կանխելու համար պետք է ծրագիրը շատ լուրջ ուսումնասիրության տակ առնել և հասկանալ ծրագրի իրապես թույլ կողմերը։ Բացի այդ, որքան էլ QA թիմը regression test (թեստավորման տեսակ է, որի մասին կխոսենք հետագայում) անի, միևնույնն է, QA թիմին անհրաժեշտ է տալ ժամանակ՝ անհատական թեստեր (ad-hoc testing) անելու համար։

**Bug release** անել, նշանակում է թողարկել ծրագիրը, նույնիսկ եթե այն ունի որոշակի bug-եր: Այս վրիպակները, սովորաբար, այնքան էլ վտանգավոր չեն: Դրանք այն սխալներն են, որոնք այդքան էլ չեն ազդում ծրագրի վրա: Bug release-ը տեղի է ունենում այն ժամանակ, երբ ընկերությունը ավելի շուտ կնախընտրի թողարկել ծրագիրը bug-երով, քան կշտկի դրանք: Developer-ները գտնում են, որ bug-ը շտկելը ավելի թանկ է և ժամանակատար: Այդ իսկ պատճառով նրանք թողարկում են ծրագիրը, նույնիսկ եթե այն ունի bug-եր:

Այսպիսով, Bug leakage-ը չի արվում դիտավորյալ՝ դրանք այն bug-երն են, որոնք վրիպել են QA-ների աչքից և դրանք գտնում է վերջնական user-ը, իսկ Bug release-ի դեպքում գիտակցում ենք, որ bug-եր կան, բայց, այնուամենայնիվ, ծրագիրը թողարկում ենք, քանի որ գիտենք, որ եղած bug-երը չեն ազդի ծրագրի հիմնական աշխատանքի վրա և վերջնական օգտատերը այն չի էլ նկատի (կամ, եթե նույնիսկ նկատի, նա դրան մեծ ուշադրություն չի դարձնի):

## **1․13 Ստատիկ թեստավորում - Static test**

Երբ ծրագիրը ստուգում են առանց այն Start անելու, անվանում ենք ստատիկ թեստավորում (այսուհետ` static test)։ Այսինքն, այն իրենից ներկայացնում է պահանջների և կոդերի review` դիտարկում և վերանայում:

Իրական կյանքից որպես օրնակ կարող ենք վերցնել թատրոնը։ Մինչ ներկայացումը կհասնի վերջնական հանդիսատեսին, նախ և առաջ աչքի տակով անցկացնում են սցենարը և գուցե կատարում փոփոխություններ, անում բազում պարապմունքներ, անում փորձնական բեմադրություններ։ Փորձնական բեմադրությունները կհամարենք դինամիկ թեստավորում, իսկ սցենարի վերանայումը՝ ստատիկ։ Սցենարի որևէ սյուժե ուսումնասիրելու ընթացքում ներկա են լինում սցենարիստը, բեմադրողը և դերասանները, որոնք ներգրավված են այդ սյուժեում։ Սցենարը կարդալիս` ամեն մեկն անում է իր առաջարկները և գուցե գտնվում են թերություններ, որոնք ունեն շտկման կարիք։ Բեմադրության փորձերի ժամանակ ամեն սյուժե նորից խորապես քննարկվում է և կատարելագործվում։

Հենց այսպիսի գործընթաց էլ տեղի է ունենում ծրագրավորման կյանքի ցիկլի ընթացքում։ Որոշակի կոդեր գրելուց հետո ուսումնասիրվում են ընդհանուր դակումենտացիան և դրան հիման վրա գրված կոդերը։ Այսինքն, երբ հաճախորդը ասում է, որ իրեն պետք է ծրագիր հետևյալ ֆունկցիոնալներով և տալիս է պահանջների դոկումենտացիա, դեռ չի նշանակում, որ այդ պահանջներն ընթացքում չեն փոփոխվելու։ Ծրագրավորման ընթացքում առաջանում են զանազան լավ և հետաքրքիր մտքեր, ինչպես նաև երբեմն առաջանում են իրականացման հետ կապված խնդիրներ: Դրանց շնորհիվ կամ պատճառով կարիք է լինում անընդհատ կապի մեջ լինել հաճախորդի հետ և փոխհամագործակցության շնորհիվ հասնել լավ արդյունքի։

Ստատիկ թեստավորումը բաժանվում է երկու մասի՝ պահանջների փաստաթղթերի վերանայում և ստատիկ անալիզ:

Վերանայումը իր հերթին լինում է չորս տեսակի.

1․ Ոչ ֆորմալ - ոչ ֆորմալ դիտարկման ընթացքում փաստաթղթերի ստեղծողը բովանդակությունը դնում է լսարանի առջև, և բոլորը տալիս են իրենց կարծիքը, և այդպիսով թերությունները հայտնաբերվում են վաղ փուլում:

2․ Քայլարշավ - հիմնականում դա իրականացվում է փորձառու անձի կամ փորձագետի կողմից` սխալները ստուգելու համար, որպեսզի հետագա ծրագրավորման կամ թեստավորման համար այլևս խնդիրներ չլինեն:

3․ Գործընկերների ակնարկ - գործընկերների վերանայումը նշանակում է միմյանց փաստաթղթերի ստուգում` թերությունները հայտնաբերելու և շտկելու համար: Դա հիմնականում արվում է թիմի շրջանակներում: Այսինքն, ամեն developer կատարում է այդ պահանջների փաստաթղթի մի որոշ հատված և այդ հատվածը քննարկվում է հենց developer-ների միջև։

4․ Տեսչություն - փաստաթղթի ստուգումն, ըստ էության, ավելի բարձր մարմինների կողմից։

**Ստատիկ վերլուծություն.**

Ստատիկ վերլուծությունը ներառում է կոդի որակի գնահատում, որը գրվում է developer-ների կողմից: Կոդի վերլուծությունը և նույնը ստանդարտի հետ համեմատելու համար օգտագործվում են տարբեր գործիքներ: Բնական է, որ կոդ չգրող թեստավորողի համար դա անհասկանալի և բարդ կլինի, բայց ոչ անհնարին։ Թեստավորման այս տեսակին մասնակցում են բացառապես տեխնիկական գիտելիքներ ունեցող մարդիկ։

Այն նաև օգնում է հետևյալ թերությունների հայտնաբերման համար.

(a) Unused variables - չօգտագործված փոփոխականներ

(b) Dead code - կոդի հատված, որը երբեք չի աշխատելու

(c) Infinite loops - անվերջ ցիկլեր

(d) Variable with undefined value - չորոշված արժեքով փոփոխական

(e) Wrong syntax - սխալ սինթաքս

**Եզրակացություն․** Static testing-ը այն ամենօրյա քննարկումներն են, որոնք տեղի են ունենում պռոեկտի մեջ ընդգրկված բոլոր մարդկանց միջև՝ կոդ գրողների, հաճախորդի, թեստավորողների և այլ մարդկանց միջև։ Static testing-ը մեր պատկերացրած թեստավորման հետ այնքան էլ կապ չունի։ Այն ավելի շուտ կարելի է կոչել «Ստատիկ քննարկումներ և դիտարկումներ»։

# Գլուխ 2․ Դինամիկ թեստավորում Dynamic testing

Դինամիկ թեստավորում նշանակում է թեստավորել ծրագիրը գործարկված վիճակում։

Որպես օրինակ դիտարկենք ավտոմեքենայի թեստավորումը: Երբ մեքենայի շարժիչը գործարկված չէ, դրա թեստավորումը կանվանեինք static testing։ Սակայն մեքենան իրավամբ ստուգելու համար պետք կլինի թեստավորել նրա արագության ու արգելակման ոտնակները, փոխանցման տուփի աշխատանքը և այլ ֆունկցիաներ, որոնք հնարավոր է ստուգել միայն շարժիչի գործարկված լինելու դեպքում՝ դինամիկ աշխատանքի ժամանակ։ Նույն սկզբունքն էլ վերաբերվում է software dynamic testing-ին։ Dynamic testing ենք անվանում ծրագրի աշխատանքի ժամանակ կատարվող թեստավորման գործողությունների շարքը։

Դինամիկ թեստավորումն ունի 3 հիմնական մոտեցում՝ Black box, White box և Gray box:

## **2․1 Black Box testing (Սև տուփի թեստավորում)**

Ինչո՞ւ է այս թեստավորումը կոչվում **Black Box**, քանի որ QA-ի աչքերով, application-ը նման է սև տուփի, որի ներսում հայտնի չէ, թե ինչ է կատարվում: Այս մեթոդով փորձում են գտնել bug-եր հետևյալ կատեգորիաներում.

1․ Սխալ կամ բացակա ֆունկցիաներ

2․ Interface error-ներ[22]

3․ Error-ներ data structure-ներում կամ external database[24] հասանելիության մեջ

4․ Behavior սխալներ[25]

5․ Initialization և termination error-ներ

Օրինակ` QA-ն առանց կայքի ներքին ստրուկտուրան իմանալու թեստավորում է web էջը՝ օգտագործելով browser։ Input-ներ տալով (տվյալներ մուտքագրելով) ստուգում է ակնկալվող արդյունքները։ Օրինակ` թվային դաշտում չի կարելի ներմուծել տառեր, փակագծեր (~!@#$%^&\*()\_+{}[];’:”,.<>?), կախված գործողության ձևից, գուցե, չպետք է կարողանա ներմուծել բացասական, ռացիոնալ թվեր, 0 թիվը և այլն։

Պետք է նաև ստուգել դաշտերի երկարությունը։ Դաշտերի արժեքներից կախված` գուցե, որոշ դաշտեր պետք է disable լինեն[26]։ Կարող է և պետք է ստուգվեն default արժեքները[27] ևս։

Օրինակները բազմազան են:

Այս բոլորը դեռ նախնական` basic թեստեր են, որոնք պետք է անցնեն միշտ, իսկ ֆունկցիոնալը ստուգելու համար ձեզ հարկավոր են փաստաթղթեր, որտեղից արդեն կարող եք ստուգել customer-ի կողմից պահանջված ֆունկցիոնալությունը։

Հենց այս սկզբունքով է իրականացվում հիմնական ֆունկցիաների ստուգումը։

Որպես սահմանում կարող ենք ասել, որ «Black box testing-ը կատարվում է ծրագիրը գործարկող ֆայլը բացելու միջոցով (Windows-ի դեպքում .exe)՝ առանց հասանելիություն ունենալու դրա կոդերին»։

Երբ մենք օգտագոծում ենք Microsoft word, skype, facebook, amazon, Autocad, Google Chrome և այլ ծրագրեր, մենք որպես user հասանելիություն չունենք այդ ծրագրերի կոդերին, քանի որ դրանք վերափոխված են այլ տեսակի ֆայլերի (Windows-ի դեպքում .dll ֆորմատի ֆայլերի), որոնք մենք չենք կարող բացել և փոփոխել։ Ստացվում է մենք աշխատում ենք այդ ծրագրով, չիմանալով, թե ինչ կոդեր են գրված դրանց համար և ծրագիրը մեզ համար դառնում է սև տուփ։

Երբ լսեք հետևյալ արտահայտությունը՝ «Կատարել black box testing», ապա այն նշանակում է պարզապես գործարկել ծրագիրը և սկսել թեստավորումը։

Դիտարկենք մի դեպք, երբ անհրաաժեշ է թեստավորել օրինակ idram հեռախոսային հավելվածը black box testing-ի սկզբունքով։ Դա նշանակում է, որ դուք պետք է ներբեռնեք idram հավելվածը՝ իրենց տված ֆայլով (հնարավոր է, որ ձեզ տան ալֆա տարբերակը), install անեք ձեր հեռախոսի մեջ, գործարկեք այն և սկսեք ձեր թեստավորումը։ Այս պարագայում դուք չունեք հասանելիություն կոդերին և թեստավորումը կատարում եք User Interface-ից։ Դա նշանակում է, որ դուք կատարում եք թեստավորում՝ black box testing-ի մոտեցումով։

## **2․2 White Box testing (Սպիտակ տուփի թեստավորում)**

Երբ ծրագիրը գործարկում ենք կոդի միջոցով և ոչ ավտոմատացված գործիքներով և սկսում ենք թեստավորել ծրագիրը, անվանում ենք white box testing:

Թեստավորման այս տեսակը պահանջում է ծանոթություն կոդի միջոցով application-ների գործարկման միջավայրերի հետ՝ IDE-ների (Visual studio, NetBeans, Eclipse, PyCharm, Xcode, RubyMine և այլն), ինչպես նաև ֆունկցիաները ստուգելու ծրագրավորման հիմունքների հետ։

Ի՞նչ է ստուգվում White Box Testing-ի միջոցով: Սպիտակ տուփի թեստավորման մոտեցումը ներառում է ծրագրային կոդի թեստավորում հետևյալ կետերի համար.

* Ներքին անվտանգության դատարկ խոռոչներ
* Վատ կառուցվածքային ուղիներ կոդավորման գործընթացներում
* Կոդի միջոցով կոնկրետ մուտքերի հոսքը (input)
* Ակնկալվող արդյունքը (output)
* Պայմանական օղակների ֆունկցիոնալությունը
* Յուրաքանչյուր օբյեկտի և գործառույթի թեստավորումն անհատական հիմունքներով

**White Box թեստավորման գործիքներից են**

* Parasoft Jtest
* EclEmma
* NUnit
* PyUnit
* HTMLUnit
* CppUnit

**White box testing-ի առավելությունները**

* Կոդի օպտիմիզացում՝ թաքնված սխալների հայտնաբերմամբ:
* Սպիտակ տուփի test case-երը կարող են հեշտությամբ ավտոմատացվել:
* Թեստավորումն ավելի մանրակրկիտ է, քանի որ ծածկագրի բոլոր ուղիները սովորաբար ծածկված են:
* Փորձարկումը կարող է սկսվել SDLC-ում վաղաժամ, նույնիսկ եթե GUI-ն հասանելի չէ:

**White Box testing-ի թերությունները**

* White Box testing-ը կարող է բավականին բարդ և թանկ լինել:
* Developer-ները, ովքեր սովորաբար կատարում են White box test case-երը, ատում են այն:
* White Box testing-ը պահանջում է մասնագիտական ռեսուրսներ՝ ծրագրավորման և իրականացման մանրամասն ըմբռնմամբ:
* White Box testing-ը ժամանակատար է, ավելի մեծ ծրագրավորման հավելվածները ժամանակ են պահանջում ամբողջությամբ թեստավորվելու համար:

Այս մոտեցմամբ թեստավորում իրականացնելու համար անհրաժեշտ է իմանալ կոդ գրել և debug անել (թերությունը շտկել), ուստի այն ավելի շատ կապ ունի test automation-ի հետ։ Ավելի խորը կդիտարկենք թեստավորման այս մոտեցումը unit test-երի մասին խոսելիս։

## **2․3 Gray Box testing (Մոխրագույն տուփի թեստավորում)**

Gray box testing-ը մոտեցում է, որը կարող է օգտագործվել ծրագրի խոցելիությունը գնահատելու համար: Այս մեթոդով թեստավորողը ունի սահմանափակ գիտելիքներ թեստավորվող բաղադրիչի աշխատանքի վերաբերյալ, ի տարբերություն Black box թեստավորմանը, որտեղ թեստավորողը չունի ներքին գիտելիքներ, և White box թեստավորմանը, որտեղ թեստավորողն ունի ամբողջական գիտելիքներ ծրագրի ներքին կառուցվածքի մասին:

Դուք կարող եք իրականացնել Gray box testing որպես ներթափանցման թեստավորման ձև (Penetration testing): Այս թեստերում թեստավորողը սովորաբար գիտի, թե որոնք են հավելվածի ներքին բաղադրիչները, բայց ոչ այն, թե ինչպես են այդ բաղադրիչները փոխազդում միմյանց: Այսինքն թեստավորողը ստուգում է ծրագիրը և՛ հավանական hacker-ի տեսանկյունից, և՛ սովորական օգտատիրոջ տեսանկյունից:

Gray box testing-ը ամենաարդյունավետն է web հավելվածների, ինտեգրման թեստավորման, բաշխված միջավայրերի, բիզնես տիրույթի թեստավորման և անվտանգության գնահատումներ կատարելու համար: Սա նշանակում է, որ չնայած այն բանի, որ այս դեպքում թեստավորողը ունի որոշակի պատկերացում ու գիտելիքներ ծրագրի ներքին կառուցվածքի ու կոդերի մասին, այնուամենային դա չպետք է խանգարի թեստավորման բնականոն ընթացքին:

**Gray box testing-ի գործընթացը**

Gray box testing-ի ժամանակ թեստավորողը պարտավոր չէ նախագծել test case-եր: Փոխարենը, test case-երը ստեղծվում են ալգորիթմների հիման վրա, որոնք գնահատում են ներքին վիճակները, ծրագրի վարքագիծը և Application-ի կազմվածքը (architecture): Այնուհետև թեստավորողը կատարում և մեկնաբանում է այս թեստերի արդյունքները:

Gray box testing կատարելիս` դուք պետք է իրականացնեք հետևյալ քայլերը.

* Բացահայտել և ընտրել մուտքերը սպիտակ և սև տուփերի թեստավորման մեթոդներից:
* Բացահայտել այս մուտքերից հավանական ելքերը:
* Բացահայտել թեստավորման փուլի հիմնական ուղիները:
* Հասհականալ ենթաֆունկցիաները խորը մակարդակի թեստավորման համար:
* Բացահայտել ենթաֆունկցիաների մուտքերը:
* Բացահայտել ենթաֆունկցիաների հավանական ելքերը:
* Կատարել ենթաֆունկցիոնալ test case-եր:
* Գնահատել և ստուգել արդյունքները:

Web կայքերի դեպքում Gray box testing անում են API հարցումների միջոցով։ Այլ կերպ ասած, Gray box testing-ի ժամանակ տվյալներն ուղարկվում են ոչ թե Graphic User Interface-ից[22], այլ API հարցումներ կատարող ծրագրերից (Օրինակ` Postman):

Այսօրինակ թեստավորում կատարելու համար հարկավոր է ունենալ մի փոքր գաղափար ծրագրավորումից, մասնավորապես ֆունկցիաների կանչից և Protocol-ների աշխատանքի սկզբունիքից։ Այլ կերպ ասած սա կես manual QA աշխատանք է, կես automation QA:

**Եզրակացություն 2․1-2․3 գլխերի վերաբերյալ**

Յուրաքանչուր մոտեցում ունի իր առանձնահատկությունները։ Այստեղ հարկ է նշել, որ ցանկացած մոտեցում հնարավոր է իրականացնել ավտոմատացնող գործիքներով։

1․ Black box ավտոմատացման գործիքներ – Ranorex, Katalon Studio, TestComplete, RAPISE by Inflectra, Abbot Java GUI Test Framework, AutoIt UI testing, CubicTest, eggPlant UI Automation Testing, FitNesse և այլն․․․

2․ Gray box ավտոմատացման գործիքներ – Postman, Appium, Chrome Dev Tools, Burp Suite, Junit, Nunit, DBUnit, RestAssured, Cucumber և այլն․․․

3․ White box ավտոմատացման գործիքներ - EMMA, NUnit, CppUnit, JUnit, JsUnit, Veracode, EclEmma, RCUNIT, cfix, Googletest, և այլն։

Նշում – Selenuim-ով և մի քանի այլ գործիքներով հնարավոր է կատարել և՛ Black box, և՛ Gray box, և՛ White box։

Սելենիում և շատ այլ ավտոմատացման գործիքները կիրառելու համար անհրաժեշտ են ծրագրավորման գիտելիքներ։

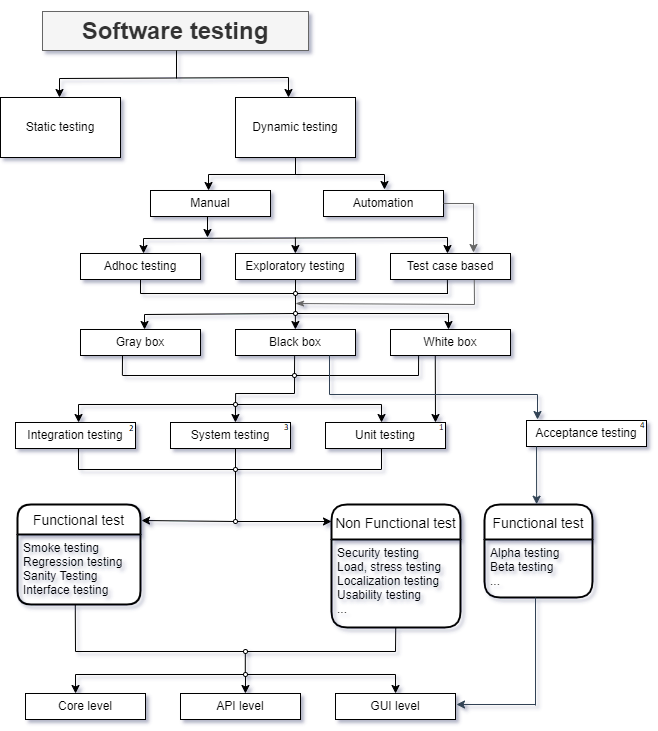
## **2․4 Թեստավորման Մակարդակներ – Testing Levels**

Թեստավորման մակարդակները չորսն են՝

1. Unit testing
2. Integration testing
3. System testing
4. Acceptance testing

Պետք է նշենք, որ յուրաքանչյուր մակարդակ ունի իր ենթամակարդակները։ Որպեսզի հասկանանք ավելի պարզ, դիտարկենք հետևյալ նկարը և ներկայացնենք թեստավորման յուրաքանչյուր մակարդակը։

Ստորև ներկայացված է թեստավորման դասակարգումը։



## **2․4.1 Unit testing**

Unit testing արվում է մի միայն white box testing-ի տարբերակով և համարվում է ցանկացած ֆունկցիոնալի թեստավորման ամենատարրական, բազային մակարդակը։ Այլ կերպ ասած` թեստավորման առաջին մակարդակն է։ Unit Testing-ը թեստավորման մակարդակ է, որտեղ թեստավորվում է ամենափոքր միավորը (բաղադրիչը/բլոկը)՝ ֆունկցիան։

Unit test-ը կարող է կատարվել Manual QA-ի կողմից Debug-ի կամ white box testing-ի այլ տեխնիկաների միջոցով միայն ու միայն այն դեպքում, եթե նա ունի ծրագրավորման գիտելիքներ և հմտություններ։ Ստացվում է, որ Manual QA-ը պե’տք է իմանա ծրագրավորում և չպե՛տք է խուսափի այն ուսումնասիրելուց։ Սակայն հանուն արդարության նշենք նաև, որ այս թեստերը հիմնականում գրվում են developer-ների կողմից և սկզբնական շրջանում կարելի է որոշ չափով անտեսել թեստավորման այս մակարդակը։ Այնուամենայնիվ, հակիրճ հասկանանք, թե ինչ է այն իրենից ներկայացնում։

Unit test-երը գրվում և իրականացվում են, ինչպես արդեն նշեցինք, հիմնականում կոդ գրողի կոմից` համոզվելու համար, որ առանձին բլոկներն աշխատում են այնպես, ինչպես սպասվում էր: Բլոկներն իրենց հերթին ունեն ավելի փոքր կառուցվածքային մասեր` ֆունկցիաներ, պրոցեդուրաներ, class-ներ[28], 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 թեստավորվեն ինչպես դրական, այնպես էլ բացասական պայմաններով դեպքերը։

Ծրագրավորման մեջ ֆունկցիաները կարող ենք բաժանել 2 մասի՝ առաջինները վերադարձնում են արժեք, երկրորդները չեն վերադարձնում, այլ input data-ի հետ կատարում են որոշակի գործողություններ։ Երկրորդ տարբերակում, երբ չենք կարող ստուգել վերադարձվող արժեքը (քանի որ այն չկա), օգտագործում են Mock տեխնիկան։ Նախապես ստեղծում են ձևական (dummy) տվյալներ, ուղարկում այն համապատասխան ֆունցկիային և վերջում ստուգում, թե արդյո՞ք Mock օբյեկտը համապատասխանորեն ձևափոխվեց։

Unit թեստի պատրաստման ցուցումներ.

**Input Data (Տվյալների մուտքագրում)**

* Պարտադիր դաշտերի փորձարկում (required field)
* Եզակի դաշտային արժեքների ստուգում
* Null[29] արժեքի ստուգում
* Դաշտը ընդունում է միայն թույլատրելի նշաններ
* Բացասական արժեքների փորձարկում
* Դաշտի երկարության ստուգում
* Անհավանական արժեքի փորձարկում
* Դաշտերի մեկը մյուսից կախվածության ստուգում
* Սահմանային պայմանների ստուգում
* Համոզվել, որ սխալ տվյալի դեպքում համապատասխան ձևով կաշխատի

**Date Validation**[30] **(Ամսաթվերի վավերացում)**

* Տարբեր ամսաթվերի ձևաչափեր
* ԱՄՆ-ի կամ Մեծ Բրիտանիայի ոճի ամսաթվերի ձևաչափ (mm/dd/yyyy)
* Դրական դեպքեր. valid ամսաթվեր
* Բացասական դեպքեր. Անվավեր ամսաթվերը ինչպիսիք են
* Ամիսը չի ընդունում 00 և 13-ը որպես արժեքներ
* Օրը չի պարունակում 00 և 32
* 28, 29, 30-ը վավերացված են ճիշտ:
* Անհրաժեշտության դեպքում ստուգել հանգստյան և բանկային արձակուրդների օրերը
* Կապը տարիների և փետրվարի 29-ի միջև

**Time Validation (Ժամանակի վավերացում)**

* Ժամանակի տարբեր ձևաչափեր, ինչպիսիք են 12/24 ժամ ձևաչափը, AM/PM
* Դրական դեպք. Ստուգել վավեր ժամեր
* Բացասական դեպք. Ստուգել անվավեր ժամեր
* Ստուգել հանգստյան օրերի և աշխատանքային արձակուրդների ազդեցությունը

**Postcode validation (Փոստային կոդի վավերացում)**

* Ստուգել Փոստային ինդեքսի մասնակի մուտք եղած դեպքը և փոստային ինդեքսի ձևաչափը
* Ավելացնել white space (пробел, բացատ)
* Ստուգել, թե արդյո՞ք կա ձեռքով մուտքագրելու տարբերակ

**System Interfaces (Համակարգի ինտերֆեյս)**

* Ստուգել, թե արդյո՞ք ինտերֆեյսի բոլոր դաշտերը/պարամետրերը պատշաճ կերպով իրականացվել են
* Բոլոր դաշտերը պետք է պատշաճ կերպով աշխատեն ըստ validation արժեքների
* Անվտանգության ստուգում ավտոմատացված ինտերֆեյսերում
* Ստուգել parent-child փոխհարաբերությունները

**Usability (Օգտագործելիություն)**

* Ստուգել, թե արդյո՞ք դաշտերի դասավորությունը համահունչ է
* Ստուգել տառատեսակները, գույները, չափերը և այլն
* Ստուգել, թե արդյո՞ք յուրաքանչյուր պատուհանի վերնագիրն ունի ճիշտ անվանում
* Ստուգել, թե արդյո՞ք էկրանը հնարավոր է մեծացնել և փոքրացնել
* Ուղղագրության ստուգում
* Անհրաժեշտության դեպքում default (լռելայն) արժեքների ստուգում[27]
* Պարտադիր դաշտերը պետք է ընդգծվեն աստղանիշով կամ համապատասխան այլ ձևաչափով

**Security (Անվտանգություն)**

* Բացասական փորձարկում. Գաղտնաբառը տեսանելի է
* Մուտքի ստուգում. Բազմաթիվ մակարդակ
* Դրական դեպք. Գաղտնաբառի փոփոխում
* Սխալ հաղորդագրությունները չպետք է բացահայտեն որևէ համակարգային տեղեկատվություն
* Ստուգել SSL-ի ճիշտ տեղադրումը
* Ստուգել, թե արդյո՞ք գործում են Logout- ի պայմանները
* Ստուգել, թե արդյո՞ք պահպանված գաղտնաբառը կոդավորված է
* Ստուգել վավեր User Id և անվավեր User Id-ով
* Ստուգել վավեր գաղտնաբառով և տարբեր անվավեր գաղտնաբառերով
* Ստուգել մուտք գործելու համար` ուղղակիորեն մուտքագրելով վավեր URL (ներ): Համակարգը պետք է հարցնի մուտքի login password-ը:
* Ապահովել, որ browser-ը չհիշի գաղտնաբառերը

**Business application logic (Բիզնեսի կիրառման տրամաբանություն)**

* Վարքը բիզնես ընթացքի դադարեցման / անջատման մասին
* Սարքավորումների ձախողման պահվածքը (Behavior) [25]
* Հաշվարկված գումարի չափը
* Ապահովել օգտագործված թեստային հաշիվների ամբողջ տեսականին, տեսակները / կարգավիճակը / պայմանները
* Ստուգել արժույթը
* Ստուգել, որ կրկնօրինակ record-ներ գոյություն չունենան
* Օգտագործել մեծ կամ շատ մեծ քանակություններ/թվեր, ստուգելու համար, թե արդյո՞ք դրանք ամբողջությամբ ցուցադրվում են էկրանի վրա

**Reporting (Հաշվետվություն)**

* Բոլոր անհրաժեշտ դաշտերը հասանելի են
* Դաշտերի համար պետք է լինի համապատասխան տարածքներ
* Միացնել scrolling and Panning-ը (dragging)
* Էջի համարակալումը հասկանալի է (N-րդ/M էջից) և պետք է հնարավորություն լինի տեղափխվել առաջին/վերջին էջ
* Հաշվետվությունները ճիշտ export են լինում Excel/Word/pdf և այլ ձևաչափերով
* Հաշվետվությունները կարող են ճիշտ տպագրվել, և բոլոր տվյալները ճիշտ են երևում

**Environment (Միջավայր)**

* Ստուգել բոլոր browser-ներով, օպերացիոն համակարգերով

**Email (Էլ. փոստ)**

* Ստուգել, արդյո՞ք էլ. Փոստի ուղարկման դեպքում տրվում է հաստատման հաղորդագրություն
* Ստուգել, որ էլեկտրոնային հասցեներում նշված հղումները գործեն ինչպես հարկն է
* Ստուգել, որ հասցեով reply-ն ճիշտ աշխատում է։

**Unit test - պարզ օրինակներ**

Unit test գրելու համար հարկավոր են ծրագրավորման հմտություններ։ Ստորև ներկայացվում է 2 պարզ օրինակ, որոնց միջոցով կարելի է հասկանալ Unit test-երի հիմքը։ Unit test-ի 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;

}

}

Ներկայացնենք, թե ինչ պետք է իրականացվի: 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 այն 2 թվերը, որոնք այն 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, որպեսզի 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”);

}

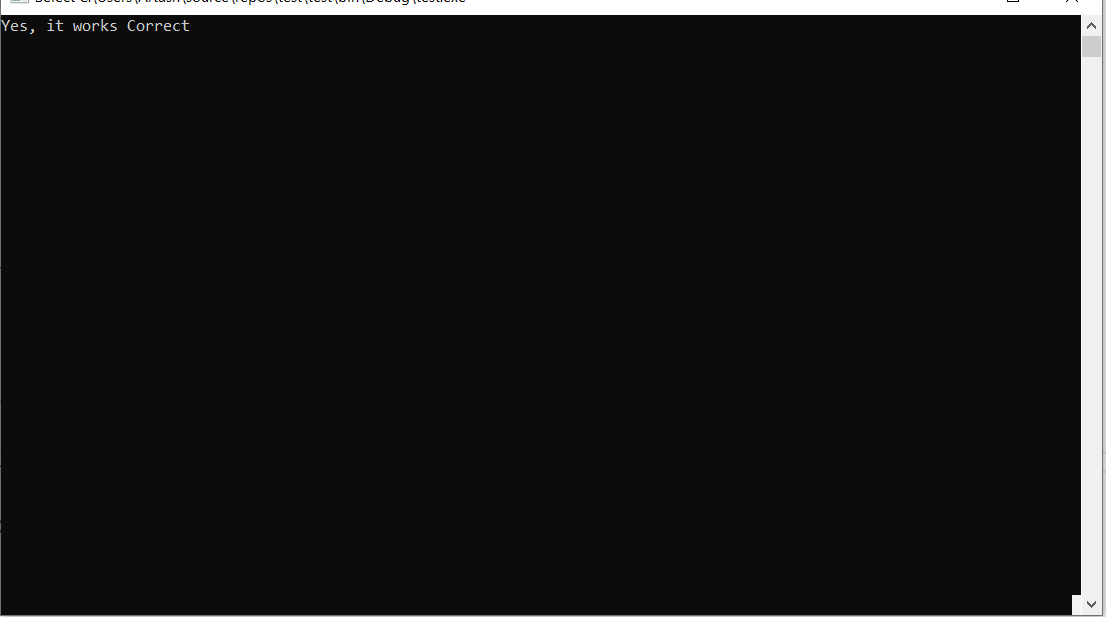
}

}

Աշխատացնելուց հետո կստանանք հետևյալ պատկերը։



Ցավոք հայկական տառերը չճանաչեց, գրենք լատիներենով և կրկին աշխատացնենք։



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

Այժմ եկեք փոխենք փոխանցվող թվերը և անյպիսի արժեքներ տանք, որ դրանց գումարը շատ մեծ լինի, օրինակ 2․000․000․000 (իսկ թե ինչու հենց 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!!!");

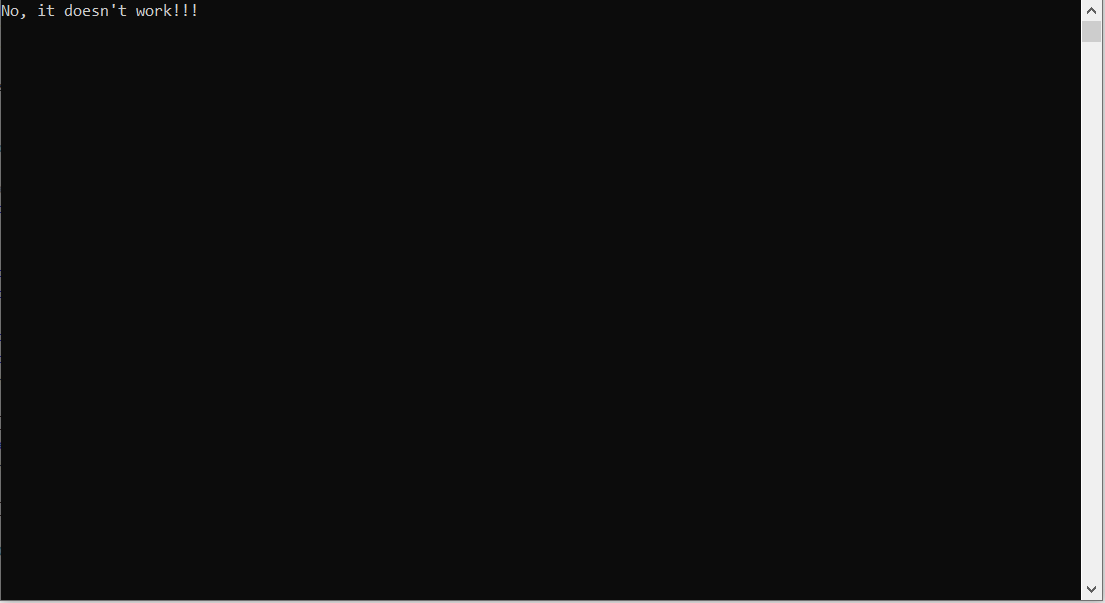
}

Console.ReadLine();

}

}

Իսկ արդյունքո՜ւմ՝ ֆեյլ։



Այդ ֆունկցիան չկարողացավ 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 Test Case, այնուհետև գտանք bug, բայց այն գտնելուց հետո հարկավոր է ինչ-որ տեղ գրել դրա մասին։ Կարելի է կիրառել հետաքրքիր տարբերակ, երբ հենց bug-ը գտնվում է, այն ավտոմատ ուղարկվում է մեյլ։ Իհարկե, երբեք ոչ ոք այդպես չի անի, սակայն հետաքրքրության համար կարող եք փորձել, միանգամից կզարգացնեք ձեր կոդ գրելու հմտությունները։ Իսկ իրականում ստեղծում են text 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 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");

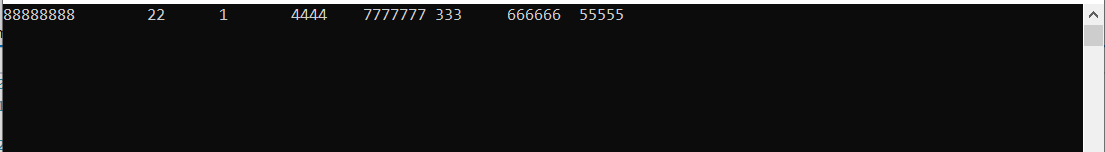
TestList.ForEach(i => Console.Write("{0}\t", i));

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։

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");

}

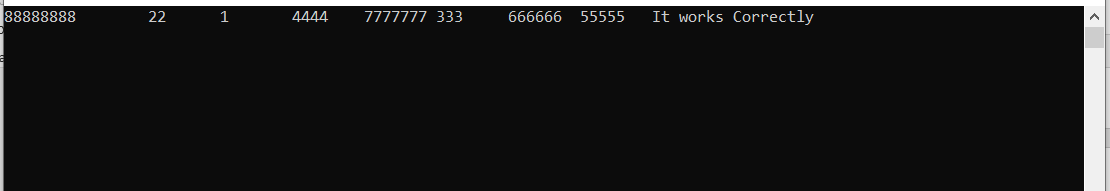
else

{

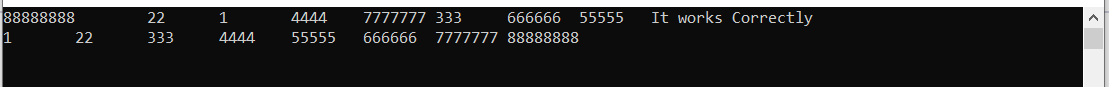
Console.WriteLine("It doesn’t work");

}

Արդյունքում կունենանք հետևյալ պատկերը



Համոզվելու համար արտածենք և ստուգենք։



Ահա ամբողջ կոդը։

    using System;

using System.Collections.Generic;

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();

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");

}

}

}

Այսքանով ավարտենք Unit test-ի մասին խոսելը և անցնենք առաջ։

Եվ մի խորհուրդ․ ամեն դեպքում ձեռք բերեք ծրագրավորան մինիմալ գիտելիքներ։ Դրանք թեստավորողին դարձնում են ավելի շրջահայաց և ուժեղ մասնագետ։

## **2․4.2 Integration testing**

Շատ application-ներ սովորաբար բաղկացած են լինում տարբեր module-ներից, մասերից և ամեն module-ի development-ը հանձնարարվում է տարբեր թիմերի կամ անձանց։ Այս module-ներն ընդհանուր կոնցեպտով, բայց անհատական կերպով իրականացվում են տարբեր թիմերի կողմից։ Երբ application-ը դեռ developmenti փուլում է և 2 կամ ավելի module-ներ համակցվում են այդ ընթացում և թեստավորվում, ապա այդ փուլը կոչվում է ինտեգրացիոն թեստ։

Integration testing-ի ժամանակ ստուգվում են՝

1․ թե արդյո՞ք մշակված մոդուլները ինտեգրացիայից հետո աշխատանքով համապատասխանում են ստանդարտներին և տալիս են ակնկալվող արդյունքները․

2․ module-ից module data փոխանցվող ֆունկցիոնալները․

3․ ինտեգրացիան երրորդ կողմի գործիքակազմի միջոցով (3rd Party)

Integration Test Case-ի օրինակ

Integration Test Case-ը տարբերվում է այլ Test Case-երից այն առումով, որ այն հիմնականում կենտրոնանում է module-ների միջև ինտերֆեյսերի և տվյալների/տեղեկատվության հոսքի վրա: Այստեղ առաջնահերթությունը պետք է տրվի ինտեգրվող հղումներին, այլ ոչ թե միավոր գործառույթներին, որոնք արդեն թեստավորված են Unit թեստի ժամանակ:

Բերենք Integration Testing-ի օրինակ հետևյալ սցենարի համար: Պատկերացնենք, որ հավելվածն ունի 3 մոդուլ՝ «Մուտքի էջ» (Login Page), «Փոստարկղ» (Mailbox) և «Ջնջել էլ. նամակները» (Delete emails), և դրանցից յուրաքանչյուրն ինտեգրված է տրամաբանորեն:

Այստեղ շատ չենք կենտրոնանում մուտքի էջի թեստավորման վրա, քանի որ դա արդեն արվել է Unit Testing-ում: Բայց ստուգենք, թե ինչպե՞ս է այն կապված փոստարկղի էջի հետ:

Ստուգենք Mail Box-ի ինտեգրումը Delete Mails մոդուլի հետ:

|  |  |  |  |
| --- | --- | --- | --- |
| *Test Case ID* | *Նպատակը* | *Նկարագրություն* | *Ակնկալվող արդյունք* |
| 1 | Ստուգել մուտքի և փոստարկղի մոդուլի միջև կապը | 1․ Մուտքագրել մուտքի տվյալները  2․ Click անել Մուտք կոճակին | Համակարգը պետք է տեղափոխվի փոստարկղի էջ |
| 2 | Ստուգել ինտերֆեյսի կապը Փոստարկղի և «Ջնջել նամակները» մոդուլի միջև | 1․ Փոստարկղից ընտրել էլփոստը  2․ Սեղմել «Ջնջել» կոճակը | Ընտրված էլ․ նամակը պետք է հայտնվի «Ջնջված նամակներ» էջում |

Մեկ այլ օրինակ է Frontend և Backend մոդուլների ինեգրացիոն թեստավորումը, որը կատարվում API call-երի միջոցով։

Integration Testing-ը լայն գաղափար է, որը կարող է կատարավել թե՛ blackbox և թե՛ graybox մոտեցումներով։ Այստեղ կարևոն այն գաղափարն է, որ աշխարհում ցանկացած սարք կամ ծրագիր բաղկացած է երկու և ավելի ֆունկցիաներից կամ module-ներից, որոնք այսպես թե սյնպես պետք է ինտեգրվեն իրար հետ և դառնան մեկ համակարգ, ուստի Integration Testing նշանակում է թեստավորել այդ երկու մոդուլների միջև կապի ճիշտ աշխատանքը։

## **2․4.3 System testing**

System testing-ը ամբողջ համակարգի թեստավորումն է, որպես մեկ միավոր: Բոլոր module-ները/բաղադրիչները ինտեգրված են ստուգելու համար արդյո՞ք համակարգը աշխատում է այնպես, ինչպես սպասվում էր, թե` ոչ:

System testing-ն իրականացվում է Integration Testing-ից հետո: Սա կարևոր դեր է խաղում բարձրորակ ծրագիր ստեղծելու գործում:

Եթե հավելվածն ունի երեք A, B և C module, ապա A և B մոդուլները, կամ B և C մոդուլները, կամ A և C մոդուլները համակցելով թեստավորումը հայտնի է որպես Integration Testing: Բոլոր երեք մոդուլների ինտեգրումը և որպես ամբողջական համակարգի թեստավորումը կոչվում է System testing:

Բերենք մի օրինակ։

Ավտոարտադրողը մեքենան որպես ամբողջական մեքենա չի արտադրում միանգամից: Մեքենայի յուրաքանչյուր բաղադրիչ արտադրվում է առանձին՝ նստատեղեր, ղեկ, հայելի, արգելակ, մալուխ, շարժիչ, մեքենայի շրջանակ, անիվներ և այլն:

Յուրաքանչյուր ապրանք արտադրվելուց հետո առանձին ստուգվում է, թե արդյո՞ք այն աշխատում է այնպես, ինչպես պետք է աշխատի, և դա կոչվում է Unit testing:

Այժմ, երբ յուրաքանչյուր մաս ինտեգրվում է մեկ այլ մասի հետ, այդ հավաքված համակցությունում ստուգվում է, թե արդյո՞ք ինտեգրումը որևէ կողմնակի ազդեցություն չի թողել յուրաքանչյուր բաղադրիչի ֆունկցիոնալության վրա, և արդյո՞ք երկու բաղադրիչներն աշխատում են միասին այնպես, ինչպես սպասվում էր․ դա կոչվում է Integration Testing:

Երբ բոլոր մասերը հավաքվում են, և մեքենան պատրաստ է, այն դեռ չի կարելի համարել վերջնական պատրաստ:

Ամբողջ մեքենան պետք է ստուգվի տարբեր ասպեկտների համար, ըստ սահմանված պահանջների։ Օրինակ, արդյո՞ք մեքենան կարող է սահուն վարել, արգելակել, շարժակները և այլ ֆունկցիոնալները ճի՞շտ են աշխատում, թե՞ ոչ, 2500կմ անընդհատ վարելուց հետո մեքենան անսարքության նշաններ ցույց չի՞ տալիս, չի՞ փչանում և այլն։ Արդյո՞ք մեքենան ընդհանուր առմամբ ընդունված է և հավանության արժանի, մեքենան կարելի՞ է վարել ցանկացած տեսակի ճանապարհներով՝ հարթ և անհարթ, շեղ, ուղիղ և այլն․․․Թեստավորման այս ամբողջ ջանքերը կոչվում են System testing և դա ոչ մի կապ չունի Integration Testing-ի հետ:

System testing-ը հիմնականում կենտրոնանում է հետևյալի վրա.

* Արտաքին ինտերֆեյսներ - External interfaces[22]
* Բազմածրագրային և բարդ գործառույթներ - Multiprogram and complex functionalities
* Անվտանգություն - Security
* Վերականգնում - Recovery
* Կատարողական - Performance
* Օպերատորի և օգտագործողի սահուն փոխգործակցությունը համակարգի հետ
* Տեղադրելիություն - Installability
* Փաստաթղթեր - Documentation
* Օգտագործելիություն - Usability

Այսպիսով, System testing-ը Ֆունկցիոնալ և ոչ ֆունկցիոնալ կոմպլեքս թեստավորումն է, որը փորձում է հասկանալ, թե որքանով է արտադրանքը ստաբիլ, ինչպես են նրա բոլոր բաղադրիչները փոխհամագործակցում ներքին և արտաքին միջավայրերի հետ։

System testing-ը կատարվում է իրական կյանքի թեստ քեյսերով և միայն ու միայն blackbox մոտեցմամբ։

## **2․4.4 Acceptance testing**

Acceptance testing-ը կամ User Acceptance testing-ը (UAT) թեստավորման այն մակարդակն է, որտեղ համակարգը ստուգվում է բիզնես պրոցեսին ընդունելի և վավեր լինելու համար։ Նման թեստերն իրականացվում են հաճախորդների կողմից, հենց իրենց հարմար միջավայրերում։ Ինչո՞ւ հաճախորդի մոտ, քանի որ այնտեղ կարող են առաջանալ network[31], path[32] և այլ զանազան խնդիրներ, որոնք քիչ հավանական է, որ կկարողանայինք ստանալ լոկալ՝ տեղային, միջավայրում։

Այս թեստը տեղի է ունենում System testing-ից հետո՝ երբ բոլոր հիմնական bug-երը ֆիքսված են լինում։ Սա նաև SDLC-ի վերջին կետն է, երբ փորձարկվում է նաև հաճախորդի կողմից՝ հասկանալու համար, թե ի վերջո ինչ արտադրանք են գնել և ինչը իրենց չի գոհացնում։ Այն կոչվում է նաև Customer Acceptance testing (CAT):

Acceptance testing-ը black box թեստ է։

Accteptance testing-ի հիմնական տեսակներն են ալֆան և բետտան։

Alpha testing-ն իրականացվում է Հաճախորդի կողմից, ծրագրավորողների օգնությամբ, որպեսզի ապահովվեն համապատասխան միջավայրային կոնֆիգուրացիաները։ Այն իրականացնում են end user-ները կամ կազմակերպության օգտատերերը` նախքան այն կյանքի կկոչվի, և զեկուցում են Ալֆա թեստավորման ընթացքում հայտնաբերված թերությունների մասին: Բոլոր զեկուցված սխալներից հետո շտկվում են դրանք և կրկին տրվում է նոր version։ Թողարկման տարբերակը (Build version[16]), որի վրա կատարվում է Alpha թեստավորումը, կոչվում է «Alpha Release»:

Betta testing-ն իրականացվում է առանց developer-ների օգնության, երբ տեղադրման միջավայրն ամբողջովին անծանոթ է և access չկա այդ միջավայրին։ Այս թեստավորումն իրականացվում է սահմանափակ օգտատերերի կողմից, և այս թեստավորման ընթացքում հայտնաբերված բոլոր խնդիրները զեկուցվում են շարունակաբար, ինչը նպաստում է համակարգի բարելավմանը: Ծրագրավորողները ձեռնարկում են գործողություններ այն բոլոր հարցերի վերաբերյալ, որոնք զեկուցվել են Betta testing մեջ՝ bug-ի զեկուցումից հետո, իսկ հետո application-ը պատրաստ է վերջնական թողարկմանը: Այս փուլում տրամադրված թողարկման տարբերակը (Build version) կոչվում է «Beta Release»:

# Գլուխ 3․ Functional testing Ֆունկցիոնալ թեստավորում

Functional testing-ը թեստավորման մեծ ենթաճյուղ է։ Ֆունկցիոնալ թեստավորման ժամանակ մենք ստուգում ենք, թե արդյո՞ք յուրաքանչյուր բաղադրիչ (component) գործում է այնպես, ինչպես սպասվում էր, թե՞ ոչ, հետևաբար այն կարելի է անվանել նաև «Component testing»:

Թեստավորման այս տեսակը պարտադիր է, անկախ նրանից, թե ինչ տեսակի application է թեստավորվում: Այս թեստը նրա համար է, որ ստուգվեն փաստաթղթում նշված հիմնական գործառույթները, նկարագրված ֆունկցիոնալը․ այլ կերպ ասած՝ ստուգվում է application-ի պահվածքը[23]` համաձայն թեստի սպեցիֆիկացիաների:

Ֆունկցիաները (կամ առանձնահատկությունները) թեստավորվում են` փոխանցելով դրանց մուտքային տվյալներ և ուսումնասիրելով արդյունքը: Functional testing-ը երաշխավորում է, որ application-ը պատշաճ կերպով բավարարում է նախապես նշված պահանջները: Թեստավորման այս տեսակը կապված չէ այն բանի հետ, թե ինչպես է տեղի ունենում implementation-ը, այլ կապված է միայն արդյունքների հետ:

Սովորաբար, Functional testing-ը ներառում է հետևյալ քայլերը.

* Հասկանալ գործառույթները, որոնք application-ը պետք է իրականացնի:
* Ստեղծել մուտքային տվյալներ` ելնելով ֆունկցիաների սպեցիֆիկացիաներից:
* Որոշել արդյունքը` ելնելով ֆունկցիաների առանձնահատկություններից:
* Իրականացնել test case-ը:
* Համեմատել սպասվող և իրական արդյունքներն իրար հետ:

Functional testing-ն առավել արդյունավետ է, երբ Test case-երը ստեղծվում են օգտագործողի/բիզնեսի պահանջներից ելնելով: Երբ համակարգի փաստաթղթերից (համակարգի պահանջներ / նախագծային փաստաթղթեր) ստեղծվում են թեստային պայմաններ, այդ փաստաթղթերի թերությունները չեն հայտնաբերվելու test-ի միջոցով, և դա կարող է հանգեցնել end-uesr-ի զայրույթին, երբ նրանք սկսեն օգտագործել application-ը։

Functional testing-ը ունի մի քանի ենթահամակարգեր, որոնք նկարագրված են այս գլխում։

## **3․1 Smoke testing**

Smoke testing-ը թեստավորման առաջին գործընթացն է, որն իրականացվում է ստուգելու համար, թե արդյո՞ք Application-ը պատրաստ է կամ կայուն` հետագա testing-ի համար: ․  
  
 Smoke Test տերմինը գալիս է սարքավորումների թեստավորումից․ սարքավորումները հիմնական թեստավորման ուղարկելուց առաջ իրականացրել են թեստ` ստուգելու համար, թե արդյո՞ք այն ծուխ չի առաջացնում, կամ արդյո՞ք առաջին անգամ միացնելիս այն չի բռնկվում:  
 Smoke testing-ն սկսելուց առաջ անհրաժեշտ է ստեղծել մի քանի test case-եր, որոնք օգտագործվելու են Smoke testing-ի համար: Այս թեստերը կատարվում են նախքան իրական թեստավորման սկսվելը և ստուգում են ծրագրի կրիտիկական ֆունկցիաները՝ համոզվելու համար, որ դրանք առհասարակ աշխատում են: Այլ կերպ ասած` սրանք մակերեսային թեստեր են և սրանց նպատակը ընդգրկուն թեստավորումներ կատարելը չէ: Թեստավորողը պետք է ստուգի ծրագրում մեկ պատուհանից մյուսը անցում կատարելու և այնտեղ որոշ պարզ բաներ ավելացնելու ֆունկցիաները: Պետք է տրվեն պարզ հարցեր. «Կարո՞ղ է user-ը մուտք գործել application», «Արդյո՞ք user-ը կարողանում է տեղափոխվել մեկ պատուհանից մյուսը» և այլն․․․

Ենթադրենք ունենք մի սոցիալական կայք, որը բաղկացած է ամենատարբեր module-ներից, որոնցից մեկը որոնման մեխանիզմն է և մեզ ասում են Smoke test անել որոնման ֆունկցիոնալի համար։ Ի՞նչ կանենք այդ դեպքում։ Մեկ անգամ կորոնենք որևէ անուն լատինատառով, հետո կփորձենք մեկ այլ տառատեսակով, կփնտրենք մարդու, ով գրանցված չէ համակարգում, կփնտրենք առանց որևէ բան լրացնելու որոնման դաշտում: Եվ եթե այս test case-երը հաջողությամբ տան իրենց սպասված արդյունքները, ապա այն կանցնի թեստավորման հետագա փուլեր։

Այսպիսով, երբ ստուգում ենք ամենապարզ, բայց կրիտիկական ֆունկցիոնալները, ապա այն անվանում ենք Smoke Test:

## **3․2 Regression testing**

Regression testing-ը հայերեն թարգմանելիս կարելի է անվանել «Հետընթացային թեստավորում»։ Regression-ը թեստավորման մի տեսակ է, որն իրականացվում է ստուգելու համար, թե արդյո՞ք նոր գրված ֆունկցիոնալները չեն ազդում նախկինում աշխատող ֆունկցիաների վրա։ Այլ կերպ ասած` նոր feature (ֆունկցիոնալություն, չխառնել future` ապագա, բառի հետ) ավելացնելուց հետո, նախկինները չպետք է տուժեն, կամ 1 bug ուղղելու փոխարեն չպետք է հայտնվեն ևս 2 կամ ավելի bug-եր․․․Դժվար է որոշել, թե որքան անգամ է անհրաժեշտ վերաթեստավորել ծրագիրը` լավագույն արդյունքի համար, հատկապես development-ի ցիկլի ավարտին մոտ:

Ավտոմատացված մեթոդները Regression testing իրականացնելու լավագույն և անվտանգ տարբերակներն են: Լավ թեստավորողները պետք է հասկանան, թե հատկապես ո՞ր հավանական ֆունկցիաների վրա կարող է ազդել նոր feature-ը։ «Regression testing-ի հիմնական նպատակն է համոզվել, որ փոփոխված կոմպոնենտը չի ազդում ֆունկցիոնալի անփոփոխ մասի վրա»: Այս հատվածում ամենակարևոր դերը test case-երինն է. պետք է ստուգվեն ֆունկցիայի բոլոր test case-երը։

Regression testing-ի ժամանակ QA-ն պետք է կատարի միայն նախկինում գրված test case-երը և արդյունքը համեմատի այդ հին test case-ի սպասվող արդյունքի հետ (եթե իհարկե նոր ֆունկցիոնալը հենց այդ արդյունքի փոփոխման նպատակը չէ): Եթե արդյունքների միջև հակասություն չկա, ապա կարող ենք ասել, որ Regression testing-ն անցել է:

Դիտարկենք մի այսպիսի օրինակ:

Ենթադրենք facebook-ում ծրագրավորողն ավելացրել է հնարավորություն Search-ի արդյունքները like-երի քանակով ֆիլտրելու։ Այս դեպքում Regression testing-ի շրջանակներում միանշանակ պետք կլինի ստուգել մնացած ֆիլտրերի աշխատանքը ևս (օրինակ Posts from, Date posted, People, Photos և այլն) որքան էլ, որ դրանց մասով կոդ չի փոխվել։ Քանի որ դա շատ երկար գործընթաց է, ապա այդ թեստավորումը հարկավոր է ավտոմատացնել:

## **3․3 Sanity testing**

Հայերեն թարգմանությամբ այն անվանենք «Սանիտարական թեստավորում»։ Application-ի build[15] ստանալուց հետո, երբ կատարվում է regression թեստ, կոդերի կամ ֆունկցիոնալության փոքր շտկումներից հետո Sanity testing-ն իրականացվում է ստուգելու համար, թե արդյո՞ք նախորդ build-ի մասին հաղորդված սխալներն ուղղված են, և ապա նորից անցնում են regression testing-ին: Sanity testing-ի հիմնական նպատակն է ստուգել պլանավորված գործառույթները: Ամբողջական regression testing-ի փոխարեն, կատարվում է Sanity testing:

Ենթադրենք ունենք ինչ-որ ծրագրի version և regression testing-ի արդյունքում հայտնաբերում ենք 10 հատ bug։ Մի քանի օր անց մեզ նորից տալիս են այդ ծրագրի նոր version և ասում են, որ 10 bug-ից 7-ն ուղղված են։ Այս դեպքում մենք ոչ թե կկատարենք regression testing՝ այսինքն կստուգենք այն մասերը, որոնք հնարավոր է փչացած լինեն այս ուղղումից հետ, այլ կկատարենք Sanity testing` համոզվելու համար, թե այդ նշված 7 bug-երն իրո՞ք ուղղվել են, թե` ոչ։

Sanity թեստերը օգնում են խուսափել ժամանակի և միջոցների վատնումից: QA-ն պետք է fail անի տվյալ build-ը, եթե անհրաժեշտ փոփոխությունները դեռ չկան։

Sanity testing-ը իր բնույթով մի փոքր նման է acceptance testing-ին․ ստանում ենք version և ստուգում մեզ անհրաժեշտ փոփոխությունները և եթե դրանք մեզ չեն բավարարում, ապա խնդրում ենք կրկին որոշ շտկումներ կատարել։

1. Sanity testing-ը regression testing-ի ենթահամակարգ է
2. Sanity testing-ը սովորաբար գրված չէ (Test Case-ի տեսքով, քայլերի տեսքով և այլն)
3. Sanity testing-ը կոնկրետ մեզ հուզող ֆունկցիոնալների շատ նեղ և խորքային թեստն է։
4. Sanity testing-ի միջոցով ստուգվում է, որ application-ը աշխատի համաձայն դակումենտացիայի սպեցիֆիկացիաներին և համապատասխանի հաճախորդի պահանջներին։
5. Sanity testing-ը ստուգում է ֆունկցիոնալները end-user-ի տեսանկյունից
6. Sanity testing-ը ֆիքսված bug-երի կամ ֆունկցիոնալության փոքր հատվածը ստուգելու համար նախատեսված թեստ է, որը ստուգում է նաև հարակից մի քանի ֆունկցիաներ։

## **3.4 Interface testing**

Յուրաքանչյուր application բաղկացած է մի քանի բաղադրիչներից՝ սերվերից, տվյալների բազայից, պլագիններից և այլն: Որպեսզի application-ը անթերի աշխատի, հարկավոր է, որ բաղադրիչները միմյանց հետ զուգահեռ ճիշտ աշխատեն: Այսինքն, այս բաղադրիչների միջև պետք է լինի անխափան հաղորդակցություն:

Սովորաբար application-ը բաղկացած է լինում client[20] և server[19] մասերից, այսինքն, client-ում կատարվում են գործողություններ և որպեսզի փոփոխված տվյալները օրինակ save լինեն, ապա հարկավոր է այդ տվյալները փոխանցել server, որտեղ և կկատարվի համպատասխան գործողությունը և հետ կուղարկվի պահանջվող/սպասվելիք data-ն՝ տվյալը։ Որպես օրինակ վերցնենք հայտնի ինտերնետային կայքէջերից մեկը` Facebook-ը: Ենթադրենք, ուզում եք գրառում անել, գրում ենք որևէ տեքստ և սեղմում post: Հիմա հասկանանք, թե ինչ է տեղի ունենում այսպես ասած հետին ֆոնին, որպեսզի մեր գրառումը պահպանվի: Մեր նյութը http հարցման միջոցով փոխանցվում է server և որևէ service, որն աշխատում է բազաների հետ, հարցում է ուղարկում այդ բազա՝ տվյալ գրառումն insert անելու համար։ Հիմա պատկերացնենք դեպք, երբ այս ողջ աշխատանքի ընթացքում առաջանում է կապի խափանում: Ի՞նչ պետք է լինի այդ դեպքում, ծրագիրն ինչպե՞ս պետք է իրեն պահի։ Կամ, ենթադրենք, կապի խափանման փոխարեն մեկ այլ server-ում ինչ-որ խնդիր է առաջանում։

Հենց այս տարբեր բաղադրիչների (մեր օրինակում client-server) միջև կապը անվանվում է ինտերֆեյս։ Interface testing-ի նպատակն է համոզվել, որ application-ի յուրաքանչյուր բաղադրիչն ապահովված է համապատասխան անվտանգությամբ, և կապի մեջ չի առաջանում որևէ խնդիր:

Interface testing-ը սովորաբար ներառում է web server-ի և application server-ի միջև թեստավորումը, ինչպես նաև թեստավորումը տվյալների բազայի server-ի և application server-ի միջև:

Interface testing-ի մի քանի ցուցումներ․

1. Server-ի reset լինելուց (վերագործարկումից) առաջացած հետևանքների ստուգումը ծրագրի ֆունկցիոնալության վրա:
2. Server-ի ճիշտ աշխատանքի ստուգում:
3. User-ի կողմից կատարված սխալների ճիշտ նույնականացման ստուգում և համապատասխան սխալի դեպքում ճիշտ հաղորդագրություններ ուղարկելը (օրինակի համար http error 429)
4. Server-ի աշխատանքի արդյունավետության ստուգում:
5. Ստուգել, թե արդյո՞ք անվտանգության ասպեկտը պահպանվում է, երբ application-ի տարբեր բաղադրիչներ աշխատում են միմյանց հետ:
6. Ստուգել, թե որքանո՞վ է ցանցի խափանումը ազդում տարբեր բաղադրիչների միջև հաղորդակցման վրա:

Եվ մի նկատառում ևս․ Interface-ը երբեք պետք չէ շփոթել User Interface-ի՝ GUI-ի հետ։ User Interface-ն ավելի հեշտ է իր բնույթով, որի բացատրությունը կգտնեք բառարանում։

# Գլուխ 4․ Ոչ ֆունկցիոնալ թեստավորում - Non Functional Testing

Non Functional Testing-ն թեստավորման այն մեթոդն է, որով ստուգվում են համակարգի հատկանիշները, ինչպիսիք են` հիշողության արտահոսքը, կատարողականը (performance) կամ համակարգի կայունությունը: Ոչ ֆունկցիոնալ թեստավորումն իրականացվում է թեստային բոլոր մակարդակներում:

Ոչ ֆունկցիոնալ թեստի լավ օրինակ կլինի ստուգել, թե քանի՞ մարդ կարող է միաժամանակ մուտք գործել համակարգ:

Թեստավորման այս մակարդակը նույնպես շատ կարևոր է client-ի համար, քանի որ դժվար թե մեկն ուզի ծրագիր, որը, օրինակ, դանդաղ է աշխատում։ Ծրագիրը գրելուց առաջ պահանջների մեջ պետք է ներառված լինեն որոշ ստանդարտներ և ծրագրավորման թիմը չպետք է դուրս գա այդ սահմաններից։

Non Functional Testing-ի նպատակները.

1. Non Functional Testing-ի միջոցով բարձրացնել արտադրանքի օգտագործելիությունը, արդյունավետությունն ու կայունությունը:
2. Օգնում է նվազեցնել արտադրության ռիսկը և ծախսերը` կապված արտադրանքի ոչ ֆունկցիոնալ կողմերի հետ:
3. Օպտիմիզացնել ապրանքի installing-ը, setup-ը, executing-ը
4. Բարելավել և կատարելագործել արտադրանքի վարքի և օգտագործման տեխնոլոգիաների իմացությունը:

Ոչ ֆունկցիոնալ թեստավորման բնութագրերը:

Non Functional Testing-ի խնդիրները.

1. Ոչ ֆունկցիոնալ թեստը պետք է լինի հստակ և չափելի, այնպես, որ սուբյեկտիվ կարծիքներ չհնչեն, ինչպիսիք են` «լավ է», «ավելի լավ է», «լավագույնն է» և այլն:
2. Պահանջների հավաքագրման գործընթացի սկզբում ճշգրիտ կրիտերիաներ դժվար թե ստացվեն պատվիրատուի կողմից, ուստի կարիք կլինի դրանք հարցնել ուղիղ պատվիրատուից:

## **4․1 Անվտանգության թեստավորում - Security testing**

Security testing-ը թեստավորման հիմնական կողմերից մեկն է, երբ խոսքը վերաբերում է բանկային համակարգին, վեբ-հոսթինգին, էլեկտրոնային առևտրի վեբ-կայքին կամ այլ ծրագրերին: Այստեղ, ֆունկցիոնալությունը ստուգելու փոխարեն, թեստավորումն ավելի շատ կենտրոնացած է նորմալ և ոչ նորմալ պայմաններում ծրագրերի աշխատանքի անվտանգությունը ստուգելու համար: Բացի այս թեստավորողները փորձում են նաև գտնել ծրագրի խոցելի կետերը, որոնց միջոցով hacker-ները կկարողանան մուտք գործել համակարգ և վնասել այն:

QA-ները պետք է ստուգեն Security testing-ի հետևյալ հատկանիշները.

1. Authentication (նույնականացում) - Այս հատկանիշում թեստավորվում և հաստատվում է օգտագործողի թվային նույնականացումը: Օրինակ, երբ մուտք եք գործում ձեր էլ.փոստի ծառայություն, ձեզ հարկավոր է մուտքագրել օգտագործողի login-ի և password-ի եզակի ու ճիշտ համադրություն, այդ դեպքում համակարգը ձեզ թույլ է տալիս մուտք գործել ձեր հաշիվ, բայց եթե դրանցից որևէ մեկը սխալ եք մուտքագրել, ապա չեք կարողանա մուտք գործել: Թույլատրել մուտք գործել ձեր էլ.փոստ՝ սա վավերացման գործընթաց է: Այն հաստատում է, որ օգտվողը վավերացված է տվյալ ծառայությունից օգտվելու համար:
2. Authorization (թույլտվություն) - այս հատկանիշը հաստատում է, որ օգտագործողը կամ ծրագիրը իրավասու է կատարել որևէ փոփոխություն կամ տեսնել որևէ content application-ում: Այստեղ օգտագործողը և նրա հասանելիության իրավունքները մանրակրկիտ թեստավորվում են: Օրինակ, եթե ձեր գրասենյակի որևէ համակարգչում, դուք ադմինիստրատոր չեք, ապա չեք կարողանա ինքնուրույն փոփոխություններ կատարել կամ ներբեռնել ծրագրեր: Դա անելու համար ձեզ հարկավոր է ադմինիստրատորի իրավունքներ (կամ ադմինիստրատորի անունը և գաղտնաբառը): Այսինքն, միևնույն ծրագրից օգտվող տարբեր անձինք կարող են ունենալ տարբեր տեսակի հասանելիության մակարդականեր, կախված իրենց աշխատանքային իրավասություններից: Այլ կերպ ասած խոսքը արտոնությունների (privileges) մասին է։
3. Availability (հասանելիություն) - ինչպես անունն է ասում, սա ստուգում է application-ի հասանելիությունը, այսինքն` արդյո՞ք այն միշտ աշխատում է, և անհրաժեշտության դեպքում տեղեկատվությունն ու ծառայություններն արդյո՞ք հասանելի են: Բացի այդ, սա ներառում է բազայի backup (կրկնօրինակում), որը հեշտությամբ գեներանում է ցանկացած ձախողման դեպքում:
4. Confidentiality (գաղտնիություն) - սա ապահովում է, որ տեղեկատվությունն ու ծառայությունները հասանելի լինեն միայն նախատեսված user-ներին և ցուցադրվեն միայն հարցման դեպքում: Այս առանձնահատկության մեջ կատարվում է ներթափանցման՝ penetration թեստ և այդ թեստավորման մեջ հայտնաբերված բոլոր թերություններն ուղղվում են։
5. Integrity (ճշգրտություն) - սրա միջոցով փորձում ենք հասկանալ, թե արդյո՞ք user-ներին տրամադրված տեղեկատվությունը ճիշտ է և թարմացված:

Անվտանգության թեստավորումները պետք է սկսվեն վաղ փուլում` թերությունները և որակի գինը նվազագույնի հասցնելու համար: Պահանջների հավաքման պահին անվտանգության թեստավորումներ սկսելը լավ փորձառություն է, սա ապահովում է, որ վերջնական արտադրանքի որակը բարձր լինի:

Անվտանգության թեստավորման տեսակները.

Այժմ ներկայացնենք անվտանգության թեստավորման տեսակները: Application-ի վրա կատարվում է անվտանգության թեստավորման վեց տեսակ.

1. Vulnerability scanning (խոցելիության սկան) - այս թեստավորման ընթացքում ստուգվում է թեստավորման ենթակա ամբողջ համակարգը՝ պարզելու համար բացերն ու խոցելի կետերը:
2. Penetration testing (ներթափանցման թեստավորում) - Hacker-ի հարձակումը մոդելավորվում է թեստավորվող համակարգի վրա: Անվտանգության թեստավորողը մեկն է, ով պետք է մտածի որպես հակեր և փորձի կոտրել համակարգը՝ հանդես գալով իբրև այդ համակարգից դուրս գտնվող մեկը:
3. Ethical hacking - սա տարբերվում է ներթափանցման թեստավորումից, քանի որ այստեղ թեստավորվող համակարգը հարձակման է ենթարկվում ներսից՝ գտնելու համար բոլոր անվտանգության թերությունները և բացթողումները: Հարց կառաջանա, թե ի՞նչ իմաստ ունի հարձակման ենթարկել application-ը ներսից, եթե դուք արդեն իսկ այնտեղ եք․ դրա մասին կխոսենք թեստավորման գրքի հետագա ձեռնարկներում:
4. Risk assessment (ռեիսկերի գնահատում) - Ստուգման ենթակա համակարգի անվտանգության հետ կապված կատարվում է ռիսկերի գնահատում, իսկ որոշակի ֆակտորների հիման վրա ռիսկերը դասակարգվում են բարձր, միջին և ցածր:
5. Security scanning (անվտանգության սկան) - սա սկանավորում է թեստավորման ենթակա ամբողջ համակարգը և պարզում ցանցի թույլ կողմերը: Այնուհետև դրանք մանրամասն ուսումնասիրվում են, վերլուծվում և ուղղվում:
6. Security review (անվտանգության ակնարկ) - Ինչպես անունն է հուշում, սա անվտանգության ստանդարտների վերանայման գործընթացն է:

Սրանք տարբեր կազմակերպություններում անվտանգության ստուգման որոշ տեսակներ էին, որոնք իրականացվում են այդ իսկ ընկերություններում: Application-ի անվտանգությունը ստուգելու ստանդարտ միջոց չկա, դա զուտ կախված է այն նպատակից, որին ուզում եք ծառայեցնել application-ը և պետք է հասկանալ, թե որո՞նք են դրանում ներգրավված անվտանգության ռիսկերը: Միշտ սկսեք անվտանգության թեստավորումը Interface թեստավորումից, սա մուտքի առաջին կետն է: Այնուհետև մտածեք, թե ինչ է անելու հակերը, հավանաբար, եթե նա մուտք գործի համակարգ, արդյո՞ք նա պատրաստվում է գողանալ վարկային քարտի տվյալները կամ մուտքագրել որոշ գաղտնի ֆայլեր և այլն: Ուշադրություն դարձրեք ստացված սխալի հաղորդագրություններին: Այս բոլոր տեսակի թեստավորումների ավարտից հետո փորձեք խորը փորփրել և տեսնել, թե որոնք կարող են լինել այն բացերը, երբ խոսքը վերաբերում է անվտանգության կոդերի մակարդակին: Որոշ փորձառությամբ դուք կկարողանաք ավելի լավ իրականացնել անվտանգության թեստավորումները:

Թեստավորման այս տեսակը այնքան էլ սկսնակների համար չէ։

## **4․2 Հասանելիության թեստավորում - Availabilty testing․**

Այստեղ խոսքը գնում է ոչ այնքան թեստավորման, որքան նախագծի արդյունավետ կառավարման մասին։

Ժամանակակից աշխարհում կիրառական ծրագրերը ապավինում են և կախված են բազմաթիվ սերվերներից, ներքին և արտաքին ցանցային հաղորդակցություններից, տվյալների բազայից, օպերացիոն գործընթացներից և այլ ենթակառուցվածքային ծառայություններից, որոնք բոլորը պետք է աշխատեն միասին: Այստեղ բիզնեսի իդեալը տեղեկատվության շարունակական հոսքն է, որտեղ ցանկացած ընդհատում արժե գումար։ Հետևաբար, մեծ հասանելիություն ու հասարակության լայն շերտերի կողմից տարածում ունեցող ծրագրեր (ինչպիսիք են օրինակ՝ facebook-ը, instagram-ը, tik-tok-ը և այլն) ստեղծելը դառնում է բիզնեսի կարևորագույն ռազմավարությունը:

Ընկերություններին, որոնք ավելի ու ավելի են ձգտում web-ի կամ mobile-ի վրա հիմնված application-ներին` կարևոր բիզնես գործունեության համար, անհրաժեշտ են ինժեներական մատչելիության տարբերակներ` ծախսերի արդյունավետությամբ ծառայությունների պահանջները բավարարելու համար: Իրականում, ոչ բոլոր ծրագրերն են պահանջում ակնթարթային 24/7 արձագանքով վերագործարկում: Որոշ ծրագրեր կարող են ձախողվել, այսինքն ինչ-որ ժամանակահատված չաշխատել՝ առանց լուրջ հետևանքներ առաջացնելու։ Կան նաև ծրագրեր, որոնք կարող են մի կերպ համակերպվել չնախատեսված ձախողման հետ, սակայն այս դեպքում անհրաժեշտ է անպայման ունենալ արագ վերականգնման տարբեր ռազմավարություններ։ Մինչդեռ ծրագրեր էլ կան, որոնք պետք է ապահովեն շատ բարձր հասանելիություն՝ օգտագործելով սպասարկման վերարտադրման ռազմավարություններ ՝ երաշխավորելու ակնթարթային, թափանցիկ վերականգնում, գրեթե, առանց տեսանելի ձախողման:

Որպես ընդհանուր գաղափար, հասանելիությունը չափումն է այն բանի, թե որքան հաճախ է application-ը մատչելի օգտագործման համար: Ավելի կոնկրետ․ հասանելիությունը տոկոսային հաշվարկ է` հիմնվելով այն բանի վրա, թե որքան հաճախ է application-ը իրականում մատչելի` սպասարկման հարցումները կարգավորելու համար, համեմատած ընդհանուր, պլանավորված և մատչելի գործողության հետ: Հասանելիության պաշտոնական հաշվարկը ներառում է ուղղելու համար նախատեսված ժամանակը, քանի որ ուղղվող ծրագիրը դեռևս մատչելի չէ օգտագործման համար:

Խոսենք օրինակների լեզվով։ Պատկերացնենք, որ մուտք ենք գործում facebook.com և տեսնում ենք, որ այն այդ պահին անհասանելի է (այո, երբեմն նման բաներ լինում են)։ Ըստ ձեզ, որքա՞ն ժամանակ անց այն կվերականգնվի։ Պատասխան՝ առավելագույնը 1 ժամում։ Երբեմն նման հայտնի application-ները փչանում են, բայց դրանք կարգավորվում են այնքան արագ, որ չենք էլ հասցնում նկատել այդ խնդիրները։ Այսինքն, նմանատիպ կայքերը ունեն ահռելի մեծ հասանելիություն։ Իսկ, եթե օրինակ, ձեր պորտֆոլիո էջը 1 օր չաշխատի, որտեղ միջինում կարող է մտնել օրական մեկ կամ երկու հոգի, ապա աշխարհում ոչինչ դրանից չի տուժի։

Հասանելիության հաշվարկում օգտագործվում են մի քանի չափումներ.

1. Ձախողումների միջև միջին ժամանակը (ՁՄՄԺ –[ժամ/ձախողումների քանակ])
2. Վերականգնման միջին ժամանակը (ՎՄԺ - [վերականգնման ժամանակ/ֆեյլի քանակ])

Հասանելիության բանաձև՝ (ՁՄՄԺ / (ՁՄՄԺ + ՎՄԺ)) X 100

Օրինակ, պատկերացրեք մի ծրագիր, որը նախատեսված է խիստ անընդհատ գործելու համար: Եթե ենթադրենք, որ մեր կատարած հաշվարկները նախատեսված են 1000 շարունակական ժամի համար և այդ ընթացքում տեղի են ունենում երկու 1 ժամանոց fail-երը, ապա հասանելիությունը կլինի ((1000/2) / ((1000/2) + 1)) X 100 = (500/501) X 100= 99.8%։

Հասանելիությունը նկարագրելու հանրաճանաչ ձևերից մեկը «9-երն» են, ինչպիսիք են երեք հատ ինը՝ 99.9% մատչելիության համար: Այս դեպքում մենք պետք է թվաբանություն անենք, որպեսզի պարզենք, թե որ երեք ինն են (99.9% առկայություն) ներկայացնում մեկ տարվա ընթացքում մոտ 8,5 ժամ ծառայության ընդհատում: Հաջորդ մակարդակի բարձրացումը՝ չորս հատ ինը (99,99%), ներկայացնում է մեկ տարվա ընթացքում ծառայության ընդհատման շուրջ 1 ժամը: Հինգ հատ ինը (99,999%) տարեկան կազմում է ընդամենը 5 րոպե դադար:

Երբ հասնում եք ավելի բարձր մակարդակի հուսալիության, տեղի են ունենում մի քանի հետաքրքիր բաներ.

1. Application-ի համար նախատեսված սարքավորումների ծախսերն աճում են սերվերի, ցանցի և սկավառակի ավելորդության պատճառով:
2. Բարդ bug-երի հայտնաբերումը և վերացումը դառնում է ավելի բարդ, ինչը պահանջում է ավելի բարձր պատրաստվածություն և հետևաբար ավելի հմուտ software ինժեներներ:
3. Բարձր հասանելիությունը պահանջում է յուրաքանչյուր ավտոմատ և մարդկանց վրա հիմնված թեստավորման համապարփակ թեստավորում, որը կարող է ազդել ձեր ծրագրի վրա, քանի դեռ այն գործարկման մեջ է:

Լավ նորությունն այն է, որ բիզնես ծրագրերում շատերը կարող են արդյունավետ գործել 99.9% ճշտությամբ: Հաշվի առնելով մարդկանց, ինժեներական գործընթացների և տեխնոլոգիաների ճիշտ համադրությունը, երեք հատ ինը շատ նվաճելի, մատչելի և սովորական է սպասարկման մակարդակի բնույթի համաձայնագրերում:

Պատկերացրեք հետևյալը. Եթե դուք լիարժեք չեք թեստավորել տեղակայման և կոնֆիգուրացիոն գործընթացը, ապա ինչպե՞ս կիմանաք, որ այն կաշխատի կեսգիշերին, երբ Application-ը կթարմացվի: Նաև մի անտեսեք ի վերջո ձեր application-ը վերականգնելու երբեմն դժվար խնդիրը` առանց այլ ծրագրերի կիրառման:

## **4․2 Կատարողականի թեստավորում - Performance testing**

Performance testing-ըստուգում է ծրագրի արագությունը, արձագանքման ժամանակը, հուսալիությունը, ռեսուրսների օգտագործումը իրենց սպասվող ծանրաբեռնվածության ներքո: Կատարողականության թեստավորման նպատակը ոչ թե ֆունկցիոնալ թերություններ գտնելն է, այլ համակարգչային ծրագրում կամ սարքում կատարողականի խափանումների վերացումը:

Performance testing-ի ուշադրության կենտրոնում է ծրագրի`

1. Արագությունը (speed) - Որոշում է, թե արդյո՞ք ծրագիրը արագ է արձագանքում։
2. Մասշտաբայնությունը (scalability) - որոշում է այն առավելագույն ծավալը, որը կարող է կառավարել ծրագիրը։
3. Կայունությունը (Stability) - որոշում է, թե արդյո՞ք ծրագիրը կայուն է տարբեր ծանրաբեռնվածությունների տակ։

**Performance testing-ի տեսակները**

Load testing (Ծանրաբեռնվածության թեստավորոում) - ստուգում է ծրագրի հնարավորությունները ակնկալվող օգտագործողների ծանրաբեռնվածությունների տակ: Նպատակը՝ ծրագրի կյանքի կոչվելուց առաջ վերացնել կատարողականի խոչընդոտները:

Stress testing (Սթրես թեստավորում) - ենթադրում է թեստավորել ծրագիրը ծայրահեղ ծանրաբեռնվածության ներքո՝ տեսնելու համար, թե ինչպես է այն կարգավորում բարձր traffic-ը կամ տվյալների մշակումը: Նպատակը` ծրագրի խափանման կետը պարզելն է:

Endurance testing (Կայունության թեստավորում) - կատարվում է համոզվելու համար, որ ծրագիրը կարող է երկար ժամանակահատվածում կարգավորել սպասվող ծանրաբեռնվածությունը:

Spike testing (Պիկի թեստավորում) - թեստավորվում է ծրագրաշարի արձագանքը՝ օգտագործողների կողմից առաջացած ծանրաբեռնվածության հանկարծակի մեծ թռիչքների հետևանքով:

Volume testing (Ծավալային թեստավորում) - ծավալային թեստավորման ժամանակ մեծ թվով տվյալներ են ավելացվում տվյալների բազայում և ստուգվում է համակարգչային ծրագրի ընդհանուր վարքը: Նպատակը՝ ծրագրի աշխատանքների ստուգումն է տվյալների բազայի տարբեր ծավալների ներքո:

Scalability testing (Մասշտաբելիության թեստավորում) – նպատակը՝ որոշել ծրագրի արդյունավետությունը «մասշտաբի բարձրացման գործում»՝ աջակցելու համար օգտագործողի ծանրաբռենվածության բարձրացմանը: Այն օգնում է պլանավորել ձեր ծրագրի կարողությունների ավելացումը:

**Կատարողականի ընդհանուր խնդիրներ**

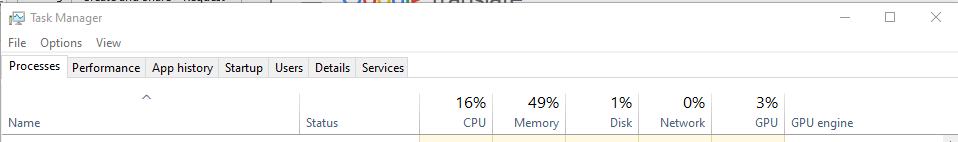
Կատարողականի հետ կապված խնդիրների մեծ մասը պտտվում են արագության, արձագանքման ժամանակի, ծանրաբեռնվածության ժամանակի շուրջ: Արագությունը հաճախ ծրագրի ամենակարևոր հատկանիշներից մեկն է: Դանդաղ աշխատող ծրագիրը կկորցնի հավանական օգտատերերին: Performance testing-ն արվում է համոզվելու համար, որ ծրագիրը բավականաչափ արագ է աշխատում՝ օգտատիրոջ ուշադրությունն ու հետաքրքրությունը պահելու համար:

Հայացք գցենք ընդհանուր կատարողական խնդիրների հետևյալ ցուցակին և կնկատենք, որ արագությունը դրանցից շատերի մեջ ընդհանուր գործոն է:

1. Long Load time - բեռնման ժամանակը սովորաբար այն ժամանակն է, որը պահանջվում է ծրագիրը գործարկելու համար: Դա, ընդհանուր առմամբ, պետք է հասցվի նվազագույնի: Չնայած, որոշ ծրագրերում անհնար է ծավալը load անել մեկ րոպեի ընթացքում, բայց հնարավորության դեպքում բեռնման ժամանակը պետք է պահվի մի քանի վայրկյան:
2. Արձագանքման վատ ժամանակ - արձագանքման ժամանակն այն ժամանակն է, որը տևում է այն պահից, երբ օգտատերը տվյալներ է մուտքագրում ծրագրի մեջ և սպասում ծրագրի արձագանքմանը: Ընդհանրապես, դա պետք է լինի շատ արագ: Կրկին, եթե օգտատերը ստիպված է շատ երկար սպասել, սկսում է կորցնել իր հետաքրքրությունը:
3. Վատ մասշտաբայնություն – սա տեղի է ունենում այն ժամանակ, երբ ծրագիրը չի կարողանում handle անել (բեռնել, ապահովել) օգտատերերի սպասվող քանակը կամ, երբ չի կարողանում սպասարկել օգտատերերի լայն շրջանակ։ Իսկ դա հասկանալու համար հարկավոր է Load testing անել։
4. Պրոցեսորի ավելորդ չափով օգտագործումը (օրինակ՝ ծրագիրը չպետք է օգտագործի CPU-ի 100%-ը)։
5. Հիշողության օգտագործում (ծրագիրը չպետք է օգտագործի RAM-ի 100%-ը)։
6. Ցանցի ավելորդ չափով օգտագործումը (չպետք է օգտագործի ավելորդ MB-եր)։
7. Օպերացիոն համակարգի սահմանափակումներ։
8. Սկավառակի ավելորդ չափով օգտագործումը։

**Կատարողականի թեստավորման չափորոշիչներ**[16]

Ահա թե ինչերին է ուշադրություն դարձնում Performance testing-ը:



* Պրոցեսորի օգտագործմանը (Processor Usage) – պրոցեսորի ժամանակի ծախսին ոչ ակտիվ thread-ների վրա: (Ուսումնասիրել ծրագրավորման thread-ների բաժինը)։
* Հիշողության օգտագործմանը (Memory usage) - համակարգչային գործընթացների համար մատչելի RAM: Օրինակ՝ շատ հայտնի է, որ Google Chrome browser-ը շա՜տ մեծ քանակի հիշողություն է պահանջում, ուստի խորհուրդ չի տրվում 2GB-անոց RAM-ով աշխատել Google Chrome-ի հետ։ Նույնը կարելի է ասել Photoshop, AutoCad և այլ ծրագրերի մասին։
* Սկավառակի ժամանակին (Disk time) - թե որքան ժամանակ է հարկավոր սկավառակից read և write անելու համար։ Օրինակ՝ միշտ ասում են, որ SSD-ին ավելի արագ է, քան HDD-ն: Ինչու՞, քանի որ տվյալների փոխանցումը կատարվում է տարբեր եղանակներով՝ առաջինը՝ էլեկտրոնային, իսկ երկրորդը մեխանիկական եղանակով։
* Թողունակությանը (Bandwidth) - ցույց է տալիս բիթերը ամեն վայրկյանում, որոնք օգտագործվում են ցանցային ինտերֆեյսի միջոցով:
* Մասնավոր բայթերին (Private bytes) – սա պրոցեսորի կողմից հատկացված բայթերի քանակն է, որը չի կարելի օգտագործել այլ գործընթացների մեջ: Սրանք օգտագործվում են հիշողության արտահոսքն ու օգտագործումը չափելու համար:
* Հատկացված հիշողությանը (Committed memory) - վիրտուալ հիշողության օգտագործված քանակին:
* Հիշողության էջերին մեկ վայրկյանում (Memory pages/second) - Սա սկավառակի վրա գրված կամ կարդացված էջերի քանակն է ծրագրային ծանր սխալները լուծելու համար: Ծրագրային էջի անսարքություններ են լինում այն ​​դեպքում, երբ կոդը չի կանչվում ընթացիկ աշխատանքային set-ից, այլ վերցվում է սկավառակի վրայից: (Ուսումնասիրել Memory pages/second)
* Էջի սխալներին մեկ վայրկյանում (Page faults/second) - ընդհանուր արագությանը, որով անսարքությունները մշակվում են պրոցեսորի կողմից: Սա կրկին տեղի է ունենում, երբ գործընթացը պահանջում է կոդ իր աշխատանքային կազմից դուրս: (Ուսումնասիրել Page faults/second)
* CPU-ի ընդհատմանը մեկ վայրկյանում (CPU interrupts per second) - սա ապարատային ընդհատումների միջին քանակն է, որոնք պրոցեսորը յուրաքանչյուր վայրկյան ընդունում է և վերամշակում:
* Սկավառակի հերթի երկարությանը (Disk queue length) – request-ներ կարդալու և գրելու միջին քանակին, որոնք դրված են հերթի մեջ:
* Ցանցի ելքային տվյալների հերթի երկարությանը (Network output queue length) - փաթեթներում ելքային փաթեթի հերթի երկարությանը: Երկուսից ավել փաթեթի առկայությունը նշանակում է, որ հարկավոր է դադարեցնել հետաձգումը: (Ուսումնասիրել Network[31] output queue length)
* Ցանցի տրամադրած բայթերի ընդհանուր քանակին մեկ վայրկյանում (Network bytes total per second) - ինտերֆեյսի միջոցով (հիշել ինտերֆեյս թեստավորումը) բայթերի ուղարկման և ստացման արագությանը՝ ներառյալ framing նիշերը:
* Արձագանքման ժամանակին (Response time) [33] – օգտատիրոջ կողմից ծրագրին հարցում ուղարկելու և պատասխան ստանալու միջև ընկած ժամանակահատվածին:
* Թողունակությանը (Throughput) - համակարգչի կամ ցանցի կողմից ստացված հարցումների քանակին մեկ վայրկյանում։
* Միացված pool-երի քանակին (Amount of connection pooling) - օգտատերերի հարցումների քանակին, որոնք բավարարվում են համակցված միացումների միջոցով: Որքան շատ հարցումներ բավարարվեն pool միացումների միջոցով, այնքան ավելի լավ կլինի կատարողականը:
* Առավելագույն ակտիվ սեսսիաներին (Maximum active sessions) - առավելագույն թվով սեսսիաներին, որոնք կարող են միաժամանակ ակտիվ լինել: Սա վերաբերում է DB-ին։
* Համընկնման գործակիցներին (Hit ratios) - սա կապ ունի SQL statement-ների[34] քանակի հետ, որոնք պահվում են cash տվյալների միջոցով՝ I/O (input/output) գործառնությունների փոխարեն: Սա լավ տեղ է սկսելու bottlenecking-ը լուծելու համար:
* Վայրկյանում կատարված մուտքերին (Hits per second) - Load testing-ի ժամանակ յուրաքանչյուր վայրկյանում սերվերում կատարված այցելությունների քանակին:
* Հետադարձ հատված (Rollback segment) - տվյալների քանակը, որը կարող է ճիշտ ժամանակին հետ վերադառնալ ցանկացած պահի:
* Տվյալների բազայի արգելափակումներին (Database locks) – table-ների և տվյալների բազաների lock-ը պետք է դիտարկվի և զգուշությամբ կարգավորվի:
* Թոփ սպասումներին (Top waits) - սրանք դիտարկվում են, որոշելու համար, թե ի՞նչ սպասման ժամեր կարող են կրճատվել, երբ փորձում ենք արագ տվյալներ վերցնել հիշողությունից:
* Թեմաների քանակին (Thread count) - ծրագրի արագագործությունը կարելի է չափել ըստ thread-ների քանակի, որոնք ակտիվ են և գործում են:
* Աղբահանություն (Garbage collection) - Դա կապ ունի չօգտագործված հիշողությունը համակարգ վերադարձնելու հետ: Արդյունավետության համար Garbage collection-ը պետք է վերահսկվի: (Ուսումնասիրել Garbage Collector)

**Performance testing-ի Test Case-երի օրինակներ․**

1. Ստուգել, որ response-ի (հարցման պատասխանը) ժամանակը 4 վրկ-ից ավել չլինի, երբ 1000 օգտատեր միաժամանակ մուտք են գործում կայք/ծրագիր:
2. Ստուգել, թե արդյո՞ք ծրագրի ծանրաբեռնվածության ժամանակ response-ի ժամանակը ընդունելի սահմաններում է, երբ ցանցի միացումը դանդաղ է։
3. Ստուգել օգտատերերի այն առավելագույն քանակը, որոնք ծրագիրը կարող է կարգավորել նախքան այն crash կլինի, այսինքն կխափանվի:
4. Ստուգել տվյալների բազայի execution ժամանակը, երբ 500 տվյալ միաժամանակ կարդում կամ գրվում են:
5. Ստուգել ծրագրի պրոցեսորը, հիշողության օգտագործումը և DB սերվերը առավելագույն ծանրաբեռնվածության պայմաններում։
6. Ստուգել ծրագրի արձագանքման[33] ժամանակը ցածր, նորմալ, միջին և ծանրաբեռնվածության պայմաններում:

## **4.3 Ծանրաբեռնվածության թեստավորում - Load testing - Stress testing**

Load testing-ը որոշում է համակարգի աշխատանքը իրական կյանքի ծանրաբեռնվածության պայմաններում: Այն հիմնականում օգտագործվում է հասկանալու համար, թե ծրագիրը բավարար կերպով աշխատո՞ւմ է այն դեպքում, երբ շատ օգտատերեր փորձում են միաժամանակ մուտք գործել կամ օգտագործել այն: Այն որոշում է համակարգի աշխատանքը` ակնկալվող հատուկ ծանրաբեռնվածության ներքո։

Օրինակներ՝

* Համացանցից մի շարք խոշոր ֆայլերի ներբեռնում։
* Միաժամանակ բազմաթիվ գործարկում մի համակարգչի կամ սերվերի վրա։
* Միևնույն պրինտերի վրա տարբեր համակարգիչներից փաստաթղթերի տպում։
* Սերվերի համար մեծ թրաֆիքի ապահովում։
* Կոշտ սկավառակի վրա անընդհատ տվյալների ավելացում և ընթերցում։

Ի՞նչ է Load testing-ը

1. Load testing-ն օգտագործվում է ծրագրի տեղադրման համար օգտագործվող ենթակառուցվածքի բավարար կրիտերիաները ստուգելու համար։
2. Այն օգտագործվում է պարզելու համար, թե արդյո՞ք ծրագրի կատարողականը կայուն է, երբ այն գտնվում է իր ծանրաբեռնվածության գագաթնակետին։
3. Այն մեզ ասում է, թե քանի՞ user-ներ կարող են զուգահեռաբար գործածել ծրագիրը և ծրագրի մասշտաբը, որը պահանջվում է ապարատային, ցանցային հզորության և այլնի առումով, որպեսզի ավելի շատ օգտատերեր կարողանան մուտք գործել ծրագիր։
4. Այն օգնում է բացահայտել ծրագրի առավելագույն գործունակությունը, ինչպես նաև որոշել, թե որ տարրն է դեգրադացիայի պատճառ հանդիսանում: Օրինակ, եթե օգտատերերի թիվը ավելանա, ապա որքա՞ն պրոցեսոր և հիշողություն կօգտագործվի հավելյալ, որքա՞ն կլինի ցանցի արձագանքի ժամանակը:
5. Load testing-ը կարող է իրականացվել վերահսկվող պայմաններում` տարբեր համակարգերի հնարավորությունները համեմատելու կամ մեկ համակարգի հնարավորությունները ճշգրիտ չափելու համար:
6. Load testing-ը ներառում է ծրագրի համար իրական կյանքի ծանրաբեռնվածությունը: Այն օգնում է ձեզ որոշել, թե ինչպես է ձեր ծրագիրը պահում իրեն, երբ մի քանի օգտատեր միաժամանակ օգտագործում են այն:
7. Load testing-ը տարբերվում է Stres testing-ից, որը գնահատում է այն աստիճանը, որով համակարգը շարունակում է աշխատել, երբ ենթարկվում է ծայրահեղ աշխատանքային ծանրաբեռնվածության, կամ երբ դրա ապարատային կամ ծրագրային ապահովման որոշ մասը վնասված է:
8. Load testing-ի հիմնական նպատակն է սահմանել առավելագույն թիվ, որը համակարգը կարող է իրականացնել առանց կատարողականի զգալի վատթարացման:
9. Շատ կազմակերպություններում Load testing-ն իրականացվում է ծրագրաշարի մշակման կյանքի ցիկլի վերջում, մինչդեռ որոշ կազմակերպություններ ընդհանրապես չեն կատարում Load testing: Ծրագրի իրականացման հետ կապված խնդիրներ ունենալու դեպքում դա կարող է հանգեցնել հաճախորդի եկամտի կորստի։

Load testing-ի գործիքների տեսակները

Load testing-ը կարող է իրականացվել ինչպես ձեռքով, այնպես էլ ավտոմատացման թեստավորման գործիքների միջոցով.

1․ Երբ Load testing-ը կատարվում է ձեռքով

Load testing-ի այս տեսակը կատարվում է ձեռքով, ուստի այն չի կարող բավարար սթրես ապահովել ծրագրի վրա: Նաև կարող է շատ թանկ լինել Load testing-ը ձեռքով, քանի որ այն պահանջում է շատ աշխատուժ:

2. Ընկերության կազմում մշակված թեստավորման գործիքներ

Կազմակերպությունը կարող է կառուցել իր սեփական գործիքները` իր ծրագրերի վրա Load testing կատարելու համար:

3. Load testing-ի գործիքներ (open source)

Այս գործիքները մատչելի են և անվճար, բայց դրանք նույնքան լավը չեն, որքան լիցենզավորված թեստավորման գործիքները: Open source Load testing-ի ամենատարածված գործիքը Jmeter-ն է:

4. Լիցենզավորված Load testing-ի գործիքներ (Enterprice)

Լիցենզավորված Load testing-ի գործիքները սպասարկում են բազմաթիվ protocol-ների, այնպես որ Load testing-ը հնարավոր է իրականացնել բազմաթիվ տարբեր տեսակի ծրագրերի վրա՝ ERP/CRM, Streaming Media և այլն: Ամենատարածված օգտագործվող լիցենզավորված Load testing-ի գործիքը Loadrunner-ն է:

|  |  |
| --- | --- |
| Load testing | Stress testing |
| Load testing-ը պլանավորված թեստ է` համակարգին ուղղված որոշակի թվով հարցումներ կատարելու համար` համակարգի կատարողականը ստուգելու համար: | Stres testing-ը նախատեսված է գտնելու համար ծրագրի խափանման կրիտիկական կետը․ միաժամանակյա հարցումների քանակը ավելանում է՝ անգամ, եթե կատարողականը սկսում է դանդաղ և վատ աշխատել։ |
| Load testing-ի նպատակն է ապացուցել, որ համակարգը կարող է կարգավորել զուգահեռ օգտատերերի ակնկալվող քանակը` նվազագույն և ընդունելի կատարողականի վատթարացմամբ: | Stres testing-ի նպատակն է է գտնել համակարգի վատթարացման այն կետը որտեղ ծրագիրը վերջնականապես crush է լինում։ |
| Load testing-ի ժամանակ համակարգում միաժամանակյա օգտատերերի քանակը խելամիտ կերպով ավելանում է։ | Stres testing-ի գործընթացում մտադրությունն է շարունակել համակարգում մեծացնել ծանրաբեռնվածությունը, մինչև ռեսուրսները չգերծանրաբեռնվեն։ |
| Load testing-ը և Stress testing-ը, անշուշտ, միմյանցից լիովին անկախ չեն | Load testing-ի վերին սահմաններին հասնելուց հետո, կարող եք անցնել Stres testing-ի, որտեղ դուք սեղմում եք համակարգը՝ անցնելով առկա ռեսուրսների սահմանները: |

Ինչպե՞ս իրականացնել Load testing

1․ Թեստավորման միջավայրի ստեղծում

Այս քայլում տեղադրվում է load testing-ի միջավայրը: Թեստային միջավայրը պետք է հնարավորինս մոտ լինի իրական միջավայրին` ապարատային, ցանցային, ծրագրային ապահովման առանձնահատկություններին և այլն։

2․ Կատարողականի չափանիշների հստակեցում

Load testing-ի համար կատարողականության չափանիշները սահմանվում են այս փուլում, և հաջողվածության չափորոշիչները նույնպես սահմանվում են: Սա կարող է ներառել ելունակության ընդունելի սահմանափակումներ, արձագանքման ժամանակ, տրանզակցիաներ և այլն։

3. Թեստի պլանավորում

Այն ներառում է հստակ սահմանված load testing-ի պլանի մշակում և թեստավորման միջավայրի կարգավորում:

4. Վիրտուալ օգտատերերի ստեղծում (vuser)

Ներառում է Vuser սկրիպտների ստեղծում, որոնք պարունակում են տարբեր vuser-ներ տարբեր առաջադրանքների համար:

5. Սցենարների ստեղծում

Սցենարը` մեքենաների (ագենտների), vuser-ների և սկրիպտների համադրություն է, որոնք գործարկում են թեստավորման ժամանակ: Loadrunner-ում սցենարները ստեղծվում են Loadrunner վերահսկիչի միջոցով: Սցենարները կարող են լինել երկու տեսակի՝ manual և նպատակային ուղղվածություն:

6. Սցենարի գործարկում և մոնիթորինգ

Սերվերի վրա ծանրաբեռնվածությունը զուգակցվում է մի քանի vusers գործարկելու միջոցով՝ միաժամանակ առաջադրանքները կատարելու համար: Սցենարը գործարկելուց առաջ անհրաժեշտ է ստեղծել սցենարի կազմաձևում և ժամանակացույց: Մենք կարող ենք գործարկել ամբողջ սցենարը, vuser խմբերը կամ անհատական vuser-ներ: Այս փուլում հավաքագրվում են թեստի տարբեր չափորոշիչներ։

7. Թեստի արդյունքների վերլուծություն

Սա load testing-ի գործընթացում ամենակարևոր քայլն է: Այս փուլում թեստավորողը վերլուծում է կատարման խոչընդոտները՝ սցենարի կատարման ընթացքում առաջացած գրաֆիկների և report-ների միջոցով: Load testing-ը կարող է անհրաժեշտ լինել կրկնել հայտնաբերված հարցերը շտկելուց հետո:

## **4.4 Ծավալային թեստավորում - Volume testing**

Սա թեստավորման այն տեսակն է, երբ բազայում շարունակաբար լցվում են շատ մեծ թվով տվյալներ և ստուգում են ծրագրի կատարողականը։ Այն կոչվում է նաև հեղեղային թեստավորում։ Օրինակ, ենթադրենք դուք փորձում եք ստեղծել սոցիալական նոր կայք, որտեղ մարդիկ գրանցվում են: Այս դեպքում դուք պետք է ստեղծեք միլիոնավոր user-ներ, այսինքն գերծանրաբեռնեք բազան, և տեսնեք, թե ինչ կկատարվի ծրագրի հետ։

Volume testing-ի միջոցով կարող ենք՝

1․ Ստուգել, թե արդյո՞ք կան տվյալների կորուստներ։

2․ Ստուգել համակարգի արձագանքման ժամանակը։

3․ Ստուգել, թե արդյո՞ք տվյալները ճիշտ են պահվում, թե՝ ոչ։

4․ Ստուգել` արդյո՞ք տվյալները վերագրվում են առանց որևէ notification-ի։

5․ Ստուգել warning և error հաղորդագրությունները՝ արդյո՞ք դրանք մեծ ծավալների տվյալների խնդրից են բխում։

6․ Ստուգել, թե արդյո՞ք մեծ ծավալով տվյալները ազդում են պրոցեսների արագության վրա։

7․ Հասկանալ, թե արդյո՞ք համակարգն ունի անհրաժեշտ հիշողության աղբյուրներ։

8․ Հասկանալ, թե արդյո՞ք ծավալային թեստը կատարվում է ամբողջ համակարգի վրա։

9․ Հասկանալ, թե արդյո՞ք կա վտանգ, եթե տվյալների ծավալը նշվածից ավելին է։

10․ Հասկանալ, թե կա՞ որևէ երաշխիք, որ ամսվա ընթացքում ավելի մեծ ծավալով տվյալներ չեն լինի, քան սահմանված է։

Օրինակ 1:

Նախագծերից մեկը մեծ համակարգ էր, որը բաղկացած էր և՛ վեբ հավելվածից, և՛ բջջային հավելվածից: Բայց վեբ հավելվածն ինքնին ուներ 3 module, որոնք մշակվում էին 3 տարբեր թիմերի կողմից:

Երբեմն, նույնիսկ տվյալների բազան դանդաղ էր աշխատում, երբ մենք բոլորս «միասին» ավելացնում էինք տվյալներ մեր թեստավորման համար: Դա նյարդայնացնող էր, և աշխատանքը դժվարանում էր տվյալների հսկայական ծավալի պատճառով, ուստի աշխատանքը հեշտացնելու համար մենք ստիպված էինք հաճախակի մաքրել DB-ն:

Տվյալները, որոնցով աշխատում էր «live» [21] համակարգը, մոտ մեկ 30ԳԲ էր, հետևաբար, բջջային հավելվածի համեմատությամբ, վեբ հավելվածը շատ հաճախ թեստավորվում էր տվյալների ծավալի համար: Վեբ հավելվածի QA թիմերն ունեին իրենց ավտոմատացման սկրիպտները, որոնք աշխատում էին գիշերը և կատարում այս թեստավորումը:

Օրինակ 2:

Մեկ այլ օրինակ էր environment[35], որն ուներ ոչ միայն վեբ հավելված, այլև SharePoint հավելված և նույնիսկ installer: Այս բոլոր համակարգերը հաղորդակցվում էին նույն տվյալների բազայի հետ տվյալների փոխանցման համար: Այդ համակարգի կողմից մշակված տվյալները նույնպես շատ հսկայական էին, և եթե որևէ պատճառով DB-ն դանդաղի, նույնիսկ install անողը կդադարի աշխատել:

Հետևաբար, ծավալային թեստավորումն իրականացվել է կանոնավոր հիմունքներով, և DB-ի կատարողականը մանրակրկիտ դիտարկվել է ցանկացած խնդրի համար:

Իսկ ի՞նչ է հարկավոր Volume test կատարելու համար․ իհարկե համապատասխան backend-ի սկրիպտներ՝ SQL, MongoDB և այլն․․․Դուք պետք է կարողանաք գեներացնել շատ դատաներ և սկսել թեստավորումը։ Ստորև ներկայացված են այն գործիքները, որոնցով հնարավոր է իրականացնել ծավալային թեստավորում։

1) DbFit:

Սա բաց կոդով (open source) գործիք է: DbFit թեստավորման framework-ը (շրջանակը) գրված է Fitness-ով, թեստերը գրվում են աղյուսակների միջոցով և կարող են իրականացվել Java IDE կամ CI ցանկացած գործիքի միջոցով:

2) HammerDb:

HammerDb-ն նաև բաց կոդով գործիք է, որը կարող է լինել ավտոմատացված, բազմաշերտ և նույնիսկ թույլ է տալիս գործարկման ժամանակի սկրիպտավորում: Այն կարող է աշխատել SQL, Oracle, MYSQL և այլ բազաների հետ:

3) JdbcSlim:

JdbcSlim հրամանները կարող են հեշտությամբ ինտեգրվել Slim Fitness-ում և այն աջակցում է բոլոր տվյալների բազաներին, որոնք ունեն JDBC դրայվեր: Ուշադրության կենտրոնում է կոնֆիգուրացիան, թեստային տվյալները և SQL հարցումները առանձին պահելը:

4) NoSQLMap:

Սա բաց կոդով Python գործիք է, որը նախատեսված է ավտոմատ կերպով հարձակումներ իրականացնելու և DB-ի կոնֆիգուրացիաները խափանելու համար՝ սպառնալիքը վերլուծելու համար: Այն աշխատում է միայն MongoDB-ի համար:

## **4․5 Տեղորոշման թեստավորում - Localization testing**

Localization testing-ը մեթոդ է, որի միջոցով ստուգում են, որ ծրագիրը աշխատի ըստ տեղական մշակույթի կամ պարամետրերի: Localization testing կատարելու նպատակը որոշակի տեղանքի համար լեզվական և մշակութային համապատասխան կողմերի ստուգումն է: Այլ կերպ ասած, դա ծրագրերը կիրառելու գործընթաց է՝ ուղղված նպատակային լեզվին և երկրին:

Տեղորոշման թեստավորման ժամանակ ստուգվում են բովանդակությունը և UI-ը:

Այն գլոբալիզացված ծրագրի թեստավորման գործընթաց է, որի UI-ը, default լեզուն, արժույթը, ամսաթիվը, ժամանակի ձևաչափը և փաստաթղթերը մշակվում են ըստ նպատակային երկրի կամ տարածաշրջանի: Այն ապահովում է, որ ծրագիրը բավականաչափ ընդունելի լինի տվյալ երկրում օգտագործելու համար:

Օրինակ․

1. Եթե ծրագիրը նախատեսված է Հնդկաստանում գտնվող Թամիլ Նադուի նահանգի համար, ապա նախագծված ծրագիրը պետք է լինի թամիլերեն լեզվով, թամիլական վիրտուալ ստեղնաշար պետք է լինի և այլն։

2. Եթե ծրագիրը նախատեսված է ԱՄՆ-ի համար, ապա ժամի ձևաչափը պետք է փոխվի ԱՄՆ ստանդարտ ժամի համաձայն (mm/dd/yyyy): Բացի այդ, լեզուն և փողի արժույթը պետք է համապատասխանեն ԱՄՆ ստանդարտներին:

3․ Դե իսկ արաբական երկրներում ամբողջ UI-ը պետք է շրջվի 180 աստիճանով։

*Ինչպե՞ս կատարել տեղորոշման թեստավորում*

Տեղորոշման բնորոշ թեստավորման համար մենք ստեղծում ենք build-ի վերիֆիկացիայի ստուգում, ֆունկցիոնալ ստուգում, ռեգրեսիոն թեստավորում և վերջնական sign-off:

1. Build-ի հաստատման ստուգումը ֆունկցիոնալ թեստավորման փոքր ենթաբազմություն է, որն իրականացվում է նախքան QA-ի ցանկացած մանրամասն թեստավորումը:

2. Լավ թեստավորումն այն է, երբ Test case-երի գործարկման և կատարման ընթացքում log-երում[36] հայտնաբերվում են թերությունները:

Գործողություններ, որոնք պետք է կատարել Localization testing իրականացնելու համար։

1․ Վարձել տեղորոշման ընկերություն` մասնագիտացված i18n ճարտարագիտության ոլորտում:

2․ Համոզվել, որ ձեր տեղայնացման թեստավորման ռազմավարությունը հնարավորություն է տալիս ավելի շատ ժամանակ տրամադրել կրկնակի բայթ լեզուներին (UTF16, UTF32):

3․ Համոզվել, որ պատշաճ կերպով միջազգայնացրել եք ձեր ծածկագիրը DBCS-ի համար, նախքան թարգմանության ուղարկելը։

*Test case-եր*

1. Բառարանները մատչելի՞ են հղման և ստուգման համար:

2․ Ժամը և ամսաթիվը պատշաճ հարմարեցվա՞ծ են նպատակային շրջանի համար:

3․ Հեռախոսահամարի ձևաչափերը ճի՞շտ են նպատակային շրջանի համար:

4․ Արժույթը ճի՞շտ է նպատակային տարածաշրջանի համար:

5․ Արդյո՞ք լիցենզիան և կանոնները համապատասխանում են գործող կայքին (տարածաշրջանին):

6․ Տեքստի բովանդակության դասավորությունը էջերում, տառատեսակների համադրությունը և գծի հավասարեցումը զե՞րծ են սխալներից:

7․ Հատուկ նիշերի, հիպերհղումների և թեժ բանալիների գործառույթների ստուգում:

8․ Վավերացման հաղորդագրություն մուտքային դաշտերի համար:

9․ Ստեղծված նոր build-ը ներառո՞ւմ են բոլոր անհրաժեշտ ֆայլերը:

10. Ապահովել ծրագրի կամ վեբ հավելվածների տեղայնացված ինտերֆեյսի համադրությունը աղբյուրի օգտատիրոջ ինտերֆեյսի հետ` նպատակային գործողության մեջ։

Այսպիսով տեղորոշման թեստավորումը նվիրված է ճիշտ կոնտենտին, տառասխալներին, ճիշտ default արժեքներին և այլ․․․

## **4․6 Տեղադրման թեստավորում - Install testing**

Ծրագիրը ինչ-որ կետի հասցնելուց հետո այն տեղադրում են հաճախորդի մոտ․ build են անում, սարքում են .exe ֆայլ («Ինստալյացիոննիկ») և այն տեղադրում են հաճախորդի մոտ։ Ցավոք, դա այդքան էլ հեշտ գործընթաց չէ։ Երբեմն install ֆայլով կարող ենք մի քանի հավելումներ ևս տեղադրել։ Բացի դրանից, կան շատ ծրագրեր, որոնց տեղադրման համար միայն .exe ֆայլը բավարար չէ և հավելյալ անհրաժեշտ են մի քանի .dll ֆայլեր, ճիշտ կոնֆիգուրացիաներ, հավելյալ driver-ներ՝ որոնց հետ աշխատում է ծրագիրը, և այլ հավելումներ․․․

Install testing սկսելու համար նախ պետք է որոշել, թե քանի տարբեր համակարգային կոնֆիգուրացիաներ ենք ցանկանում (ցանկացած OS իր հերթին ունի վերսիաներ և կոնֆիգուրացիաներ): Պատրաստենք մեկ հիմնական կոշտ սկավառակ: Ձևափոխենք այդ HDD-ն ամենատարածված կամ լռելյայն ֆայլային համակարգով, տեղադրենք այդ HDD-ում ամենատարածված OS համակարգը (Windows): Տեղադրենք որոշ հիմնական պահանջվող բաղադրիչներ: Այդ բազային սկավառակում կարող ենք ստեղծել այլ կազմաձևեր: Կազմել յուրաքանչյուր կոնֆիգուրացիաների մեկ հավաքածու, ինչպես օպերացիոն համակարգը և ֆայլի ձևաչափը, որոնք կօգտագործվեն հետագա թեստավորման համար: Սակայն սրանք ձեռքով անելը կտևի ավելի քան 1 ժամ։

OS-եր տեղադրելու ավտոմատացման ձևերից են OS Image-ներ ստեղծելը (կարող է օգտագործվել, օրնակ, Norton Ghost ծրագիրը)։ Image ստեղծել նշանակում է պահել կոնֆիգուրացված OS-ը ֆայլի տեսքով և հետագայում այն ուղղակի past անել։ Դա տևում է մոտ 5-10 րոպե։

OS համակարգը տեղադրելուց հետո արդեն պետք է անցնել ծրագրի տեղադրման աշխատանքներին։

1․ Install testing կատարելու համար օգտագործեք բլոկ-սխեմաներ։ Դրանք հեշտացնում են ձեր կյանքը։ Այդ հիմնական սխեմայի վրա ավելացրեք ևս մի քանի Test case-եր, օրինակ, եթե ծրագիրը առաջին անգամ չէ, որ թողարկվում է, ապա փորձեք ավելացնել տրամաբանական տեղադրման տարբեր ուղիներ:

2․ Եթե նախկինում արդեն տեղադրվել է ծրագրի կոմպակտ, հիմնական տարբերակ, ապա հաջորդ թեստավորման դեպքում տեղադրեք ծրագրի ամբողջական տարբերակը նույն path-ի[37] վրա, ինչն օգտագործվում է կոմպակտ տարբերակի համար:

3․ Եթե դուք օգտագործում եք flow դիագրամ՝ սկավառակի վրա տարբեր ֆայլեր գրելու, ստուգելու համար, ապա հակառակ դեպքում օգտագործեք նույն flow դիագրամը հակառակ՝ սկավառակի վրա տեղադրված բոլոր ֆայլերը ջնջելու թեստավորելու համար։

4․ Թեստավորումն ավտոմատացնելու համար օգտագործեք flow դիագրամներ: Շատ հեշտ կլինի դիագրամները ավտոմատ սկրիպտների վերածել:

5․ Թեստավորեք install սկրիպտները, որոնք օգտագործվում են անհրաժեշտ սկավառակի ազատ տեղը ստուգելու համար: Եթե installation-ը պահանջում է անհրաժեշտ 1ՄԲ սկավառակի տեղ, ապա համոզվեք, որ հենց 1ՄԲ է օգտագործվում: Եթե օգտագործում է ավելի շատ տեղ՝ գրեք դա որպես bug:

6․ Թեստային սկավառակի տարածության պահանջը թեստավորել տարբեր ֆայլային համակարգերի ձևաչափերով: Օրինակ` FAT16-ը ավելի շատ հիշողություն կպահանջի, քան NTFS կամ FAT32 ֆայլային համակարգերը:

7․ Հնարավորության դեպքում ստեղծել հատուկ համակարգ միայն սկավառակի image-ներ ստեղծելու համար: Ինչպես ասվեց վերևում, դա կփրկի ձեր թեստավորման ժամանակը:

8․ Install testing իրականացնելու համար օգտագործեք բաշխված թեստավորման միջավայր: Բաշխված միջավայրը պարզապես խնայում է ձեր ժամանակը և կարող եք արդյունավետորեն կառավարել բոլոր տարբեր Test case-երը մեկ մեքենայից:

Դրա համար լավ մոտեցումը master մեքենա ստեղծելն է, որը ցանցում կառաջարկի տարբեր child մեքենաներ: Դուք կարող եք սկսել տեղադրումը միաժամանակ տարբեր մեքենաների վրա` master համակարգից:

9. Փորձեք ավտոմատացնել համակարգը` սկավառակի վրա ֆայլերի քանակը գրելու և ստուգելու համար: Դուք կարող եք պահպանել այդ ֆայլերի ցուցակը, որը պետք է գրվի սկավառակի վրա, excel ֆայլի մեջ և կարող է տալ այդ ցուցակը որպես մուտք դեպի ավտոմատացված համակարգ, որը ստուգելու է յուրաքանչյուր ուղին` ճիշտ տեղադրումը հաստատելու համար:

10․ Դադարեցրեք տեղադրման գործընթացը պրոցեսի մեջտեղում: Տեսեք համակարգի պահվածքը և արդյո՞ք համակարգը վերադառնում է իր սկզբնական վիճակին առանց որևէ խնդրի: Դուք կարող եք թեստավորել այս «տեղադրման խափանումը» տեղադրման յուրաքանչյուր փուլում:

11․ Սկավառակի հիշողության ստուգում. Install testing-ի սցենարում սա կարևոր ստուգումներից է: Այս ստուգումը կատարելու համար կարող եք ընտրել տարբեր մեխանիկական և ավտոմատացված մեթոդներ:

12. Տեղադրումը ստուգելիս՝ կարող եք թեստավորել նաև unistall-ը․ համոզվեք, որ սկավառակի վրա գրված բոլոր ֆայլերը հեռացվելուց հետո վերացել են: Երբեմն unistall-ը հեռացնում է միայն վերջին update-ի ֆայլերը` հին տարբերակի ֆայլերը պահելով:

|  |  |  |  |
| --- | --- | --- | --- |
| ***Ձախողման պատճառը*** | ***Հնարավոր ազդեցություն*** | ***Severity (կրիտիկական / բարձր / միջին / ցածր)*** | ***Ինչպե՞ս մոդելավորել և ստանալ այդ խնդիրը*** |
| Արտաքին սերվերն անհասանելի է | Սերվերում I/O ձախողում | Կրիտիկական | Տարբեր կետերում անջատել սերվերը: |
| Սերվերը հասանելի է, բայց չի պատասխանում այնպես, ինչպես սպասվում էր: | Հնարավոր է սխալ, ծայրահեղ դեպքերում crash | Բարձր (կախված խնդրի մեծությունից) | Սերվերի կողմից սիմուլացնել սխալ պատասխաններ։ |
| էլեկտրամատակարարման խափանում | Սխալ՝ օժանդակ էներգիայի աղբյուրի դադարի դեպքում | Կրիտիկական | Անջատել էներգիայի աղբյուրը, փոխել հզորությունը հանկարծակիորեն |
| Անլար ցանցի ազդանշանի կորուստ | Ցանցի I/O-ի դադար։ Շատ դեպքերում առաջանում են bug-եր։ | Բարձր | Փոխել ցանցային պարամետրերը OS համակարգում կամ հնարավորության դեպքում անջատել ցանցը։ |
| Արտաքին սարքը չի արձագանքում | Սարքի I/O-ի դադար։ Շատ դեպքերում առաջանում են bug-եր։ | Բարձր | Անջատել/ միացնել սարքը կամ փոխել համապատասխան պարամետրը, եթե կա, որը կդադարեցնի I/O-ը։ |
| Արտաքին սարքը պատասխանում է անսպասելի ձևով | Հավանական են խնդիրներ, որոշ դեպքերում crash | Բարձր | Սարքի կողմից սիմուլացնել սխալ պատասխաններ։ |
| Ֆիզիկական պայմաններ, ինչպիսիք են ջերմաստիճանը, խոնավությունը և այլն: | Ավելի դանդաղ պատասխան, ընդհանուր անջատում։ | Կրիտիկական | Ամբողջ միջավայրը մտցնել հնարավորինս անբարենպաստ ֆիզիկական պայմանների մեջ և անցկացնել դրա բոլոր թեստավորումները |
| Մոտակա սարքերի էլեկտրական խանգարումներ | Error, դանդաղ աշխատանք | Միջին | Թեստերի անցկացման ընթացքում միմյանց ազդանշաններ փոխանցող սարքերի հեռավորությունը վերահսկելը/փոխելը: |
| Սերվիսի դադար | Error | Ցածր, միջին | Աշխատանքի ընթացքում անջատել սերվիսը |
| Բացակա ռեսուրսներ (օրինակ .dll ֆայլ) | crash | բարձր | Աշխատանքի ընթացքում ջնջել բոլոր ռեսուրսները |
| Միաժամանակյա գործարկում (օրինակ, ինչպես է աշխատում Chrome-ը, եթե դուք ունեք տեղադրված այլ բրաուզերներ) | Ծրագրի սխալ վարք | միջին | Գործարկել 2 նույն կամ նմանատիպ այլ ծրագրեր միաժամանակ |
| Բազայի գերբեռնում | Դանդաղ աշխատանք, error | բարձր | Կատարել Load testing |
| Ցանցից անջատում | Ցանցի I/O-ի դադար։ Շատ դեպքերում առաջանում են bug-եր։ | բարձր | Փոխել ցանցային պարամետրերը OS համակարգում կամ հնարավորության դեպքում անջատել ցանցը։ |
| Ցանցի գերբեռնում | Դանդաղ աշխատանք։ Որոշ դեպքերում առաջանում են bug-եր։ | բարձր | Ցանցի վրա գեներացնել ծանրաբեռնվածություն, մասնավորապես այն բաղադրիչների վրա, որոնք պետք է առաջացնեն I/O AUT-ի վրա։ |
| Սերվիսի դադար | Error | Ցածր, միջին | Աշխատանքի ընթացքում անջատել սերվիսը |

## **4.7 Վերականգնման թեստավորում - Recovery testing**

Recovery testing կատարվում է ստուգելու համար, թե որքա՞ն արագ և ավելի լավ կարող է վերականգնվել ծրագիրը crash-ից և ապարատային ձախողումից հետո։ Recovery testing-ի ժամանակ ծրագիրը տարբեր եղանակներով հարկադրված այնպես են անում, որ ձախողվի, որպեսզի ստուգեն, թե արդյո՞ք վերականգնումը իրականացվում է պատշաճ կերպով:

Օրինակ, երբ ծրագիրը համացանցից տվյալներ է ներբեռնում, ապա պարզապես անջատվեք համացանցից: Որոշ ժամանակ անց միացեք այն և վերլուծեք ծրագրի պահվածքը, արդյո՞ք այն կշարունակի տվյալների ներբեռնումն այն կետից, որտեղից անջատվել էր:

Մեկ այլ օրինակ google chrome-ի վրա․ երբ մի քանի tab-եր բացված են լինում և համակարգիչը անջատվում է, ապա միացնելուց և google chrome-ը գործարկելուց հետո կտեսնեք, որ վերջինս առաջարկում է վերականգնել վերջին անգամ բացված կայքերը։

## **4․8 Կայունության թեստավորում - Endurance testing**

Endurance testing կատարվում է համոզվելու համար, որ ծրագիրը կարող է երկար ժամանակահատվածում չխափանվել ստաբիլ օգտագործման դեպքում: Հիմնականում այն օգտագործվում է հիշողության արտահոսքերը ստուգելու համար:

Օրինակ` ծրագրային ապահովման թեստավորման դեպքում համակարգը կարող է պահել իրեն այնպես, ինչպես սպասվում էր, երբ թեստավորվում է 1 ժամ, բայց երբ նույն համակարգը թեստավորվում է 3 ժամ անընդմեջ, ապա առաջանում են այնպիսի խնդիրներ, ինչպիսիք են հիշողության արտահոսքը, որը ձախողում է համակարգը:

Թեստի արդյունքում կարելի է հայտնաբերել հետևյալ խնդիրները.

1․ Հիշողության լուրջ արտահոսք, որը, ի վերջո, կհանգեցնի ծրագրի կամ Օպերացիոն համակարգի խափանման:

2․ Համակարգի շերտերի միջև connection[38] close-ի fail-երը կարող է կասեցնել համակարգի այս կամ այն մոդուլը:

3․ Որոշ պայմաններում տվյալների բազայի connection close-ը կարող է հանգեցնել համակարգի ամբողջական crash-ի:

4․ Ժամանակի ընթացքում համակարգի արձագանքման աստիճանական քայքայումը, քանի որ երկարաձգված թեստավորման արդյունքում ծրագիրը դառնում է ավելի քիչ արդյունավետ:

Հիշողության արտահոսքեր (Memory Leak) լինում են, երբ, օրինակ, object-ները ստեղծվում են, բայց չեն ջնջվում, կամ երբ չափից շատ ռեսուրսներ են օգտագործվում և այլն։ Դրանք սովորաբար լինում են ծրագրավորման սխալ արխիտեկտուրայից։ Դրանք կարելի է նաև գտնել այնպիսի tool-երով, ինչպիսիք են Electric Fence, Valgrind memcheck, Sar և շատ այլ ծրագրեր։

## **4․9 Ծայրահեղ ծանրաբեռվածության թեստավորում - Spike testing**

Օգտագործվում է ծանրաբեռնվածության ծայրահեղ աճով և նվազեցումով թեստավորելու համար:

Spike testing-ի հիմնական նպատակն է գնահատել ծրագրի վարքագիծը՝ օգտատիրոջ կողմից ծանրաբեռնվածության կտրուկ աճի կամ նվազման պայմաններում և որոշել օգտատիրոջ կողմից ծանրաբեռնվածության բարձրացումից հետո ծրագրի վերականգնման ժամանակը: Այն իրականացվում է ծրագրերի թույլ կողմերը գնահատելու համար:

Նույնն է, թե մեքենայի վրա միանգամից ամբողջովին սեղմես գազի ոտնակը և որոշ ժամանակ անց միանգամից բաց թողնես, այնուհետև տեսնես, թե արդյո՞ք այն բարեհաջող եկավ և կանգնեց իր նորմալ վիճակի վրա։

**Spike testing-ի սցենարների օրինակ.**

1. Երբ էլեկտրոնային առևտրի խանութը սկսում է հատուկ գործարքներ մեծ զեղչերով, ինչպիսին է «Սև ուրբաթ»-ը:

2. Երբ վեբ հավելվածը ուղիղ կապով հեռարձակում է նախընտրած հեռուստածրագիրը:

3. Երբ ամենօրյա գործարքի կայքում գնում է ֆլեշ վաճառք:

4. Երբ կայքի որոշակի բովանդակությունը ինտերնետի միջոցով դառնում է վիրուսային:

5․ Արտադրության համար թողարկվում է նոր համակարգ, և բազմաթիվ օգտատերեր ցանկանում են մուտք գործել համակարգ:

Նշված և շատ այլ դեպքերում բազմաթիվ օգտատերեր միանգամից մուտք են գործում կայք և վեբ ծրագիրը պետք է պատրաստ լինի դրան։

Spike testing-ի իրականացման 6 քայլեր

1) Որոշել օգտատերերի կողմից ծրագրի առավելագույն ծանրաբեռնվածության հզորությունը։

2) Պատրաստել թեստավորման միջավայրը և կազմաձևել այն՝ կատարողականի պարամետրերը գրանցելու համար:

3) Ծրագրային հավելվածի վրա կիրառել սպասվող առավելագույն ծանրաբեռնվածությունը՝ օգտագործելով ձեր նախընտրած կատարողական գործիքը։

4) Արագորեն ծանրաբեռնվածության ավելացում որոշակի ժամանակահատվածում:

5) Աստիճանաբար իջեցնել ծանրաբեռնվածությունը իր սկզբնական մակարդակին:

6) Վերլուծել կատարման գծապատկերները: Հաշվի առնվող չափանիշերն են՝ ձախողումները, ժամանակի խլումը, վիրտուալ օգտատերերը և այլն:

**Spike Loads-ի վերականգնման սցենարները**

Օգտագործեք cloud հարթակներ, ինչպիսիք են AWS, Azure, օգտատիրոջ կողմից ծրագիրը ծանրաբեռնելուն զուգահեռ սերվերի հզորությունը դինամիկ բարձրացնելու համար

Թույլ մի տվեք, որ որոշ user-ներ ծրագիր մուտք գործելու հնարավորություն ունենան, որպեսզի համակարգը չբախվի մեծ ծանրաբեռնվածության: Սա կանգնեցնում է նոր մարդկանց մուտքը համակարգ, երբ այն գտնվում է նախագծված ծանրաբեռնվածության առավելագույն մակարդակում։ Այսպիսով համակարգը պաշտպանվում է ավելորդ ծանրաբեռնվածության սպառնալիքից:

Հնարավոր է նաև, որ կայքի ադմինիստրատորը թույլ տա օգտատերերին միանալ համակարգին, սակայն նախազգուշացնելով, որ մեծ ծանրաբեռնվածության պատճառով ծրագիրը կարող է դանդաղ արձագանքել: Դա կարող է անբարենպաստ ազդեցություն ունենալ համակարգի գործունեության վրա, սակայան, այնուամենայնիվ, օգտատերը կկարողանա աշխատել համակարգի հետ:

## **4․10 Հուսալիության թեստավորում - Reliability testing**

Reliability testing-ն ուղղված է համակարգի deploy անելուց առաջ գտնելու այն բոլոր bug-երը, որոնք կարող են խանգարել ծրագրի աշխատանքին: Reliability testing-ի նպատակը ծրագրի հուսալիության որոշումն է և պարզելը, թե արդյո՞ք ծրագիրը համապատասխանում է հաճախորդի հուսալիության պահանջներին: Այն կատարվում է ալֆա և բետտա թեստավորումների ժամանակ կամ դրանցից առաջ։

Եթե մենք ի վիճակի ենք կրկնել test case-երը, և եթե անընդհատ ստանում ենք նույն ճիշտ պատկերը, ապա ծրագիրը կարելի է համարել «հուսալի»:

Պատկերացնենք «IQ թեստ» հանձնող ինչ-որ մարդու, ով վաստակել է 144 միավոր: 6 ամիս անց, երբ նա անցնում է նույն «IQ թեստը», ապա վաստակում է 68 միավոր: Նման դեպքում մենք չենք կարող նրան համարել «հուսալի»: Կամ մեկ այլ օրինակ․ ենթադրենք, որ որևէ մեկը մասնակցում է երգի մրցույթի և 3 ժյուրիներից վաստակում է 9,8,9 միավորներ: Այս գնահատականը կարելի է համարել «հուսալի», քանի որ դրանք գրեթե չեն տարբերվում միմյանցից: Բայց եթե նա վաստակեր 9,3,7, ապա դա չէր կարող համարվել «հուսալի»:

Reliability testing-ն ընդգրկում է ինչպես ֆունկցիոնալ, այնպես էլ ոչ ֆունկցիոնալ թեստավորումների արդյունքները․ սթրես թեստավորումը, անվտանգության թեստավորումը, ֆունկցիոնալության ստուգումը, ալֆա և բետտա թեստավորումները և մնացած ամենը, ինչը թիմին հնարավորություն է տալիս որոշելու ծրագրի տարբեր խնդիրներ:

**Reliability testing-ի նպատակները․**

* Հայտնաբերել անընդհատ խոցելի կետեր ունեցող տեղերը։
* Գտնել Fail-երի քանակը, որոնք տեղի են ունենում ժամանակի նախանշված միջակայքում:
* Գտնել ծրագրի միջին աշխատելաժամանակը:
* Գտնել հիմնական fail-երի պատճառները։

Հիմնական պարամետրերը, որոնք չափվում են որպես հուսալիության մաս, տրված են ստորև.

1. MTTF: Mean Time To Failure (Fail-ի միջին ժամանակ)

2․ MTTR: Mean Time To Repair (Ուղղելու միջին ժամանակ)

3․ MTBF: Mean Time Between Failures (= MTTF + MTTR) (Fail-երի միջև միջին ժամանակ)

*Ցանկացած ֆունկցիոնալ և ոչ ֆունկցիոնալ թեստավորման ժամանակ որքան ուշ-ուշ Fail լինի ծրագիրը, այնքան այն ավելի հուսալի է։ Հուսալիության թեստավորումն իրականում թեստավորում չէ, այլ վերլուծություն, անալիզ՝ հիմնված ընդհանուր թեստավորման արդյունքների վրա։*

Գոյություն ունի նաև HALT թեստավորում։ Բարձր արագացված կյանքի թեստավորումը (highly accelerated life test-HALT), այն Stres testing-ի տեսակ է, որը նախատեսված է ծրագրի հուսալիության բարձրացման համար: HALT թեստավորումը օգտագործվում է արտադրական, հետազոտական և զարգացման խոշոր կազմակերպությունների կողմից` բարելավելու համար արտադրանքի հուսալիությունը մի շարք ոլորտներում, ինչպիսիք են էլեկտրոնիկան, համակարգիչները, բժշկությունը և ռազմական ոլորտը:

HALT-ը թեստավորման տեխնիկա է, որը կոչվում է թեստավորում մինչև fail, որտեղ ապրանքը թեստավորվում է մինչև fail-ը: HALT-ը չի օգնում որոշել կամ ցույց տալ ոլորտում հուսալիության արժեքը կամ fail-ի հավանականությունը: Շատ HALT-եր օգտագործվում են ապրանքի կյանքը կամ հուսալիությունը ցույց տալու համար:

## **4․11 Օգտագործելիության թեստ - Usability testing**

Usability testing-ը միջոց է հասկանալու, թե որքա՞ն հեշտ է ինչ-որ բան օգտագործել՝ թեստավորելով այն իրական օգտատերերի հետ:

Օգտատերերին խնդրում են կատարել առաջադրանքներ, (սովորաբար, դրանց հետևում են հետազոտողները), տեսնելու համար, թե որտե՞ղ են նրանք բախվում խնդիրների և շփոթության մեջ ընկնում: Եթե ավելի շատ մարդիկ բախվեն նմանատիպ խնդիրների, առաջարկություններ կներկայացվեն օգտագործման այդ խնդիրները լուծելու համար:

Եղել են չէ՞ դեպքեր, երբ բացել եք որևէ ծրագիր և չեք հասկացել, թե ձեր ուզած գործողության համար ինչից պետք է սկսել։ Որպեսզի նման խնդիրներ չառաջանային, հարկավոր էր հասկանալ, թե օգտատերերը սովորաբար ո՞ր պահին են շփոթության մեջ ընկնում և փորձեին փոփոխել կա՛մ դիզայնը, կա՛մ էլ աշխատանքի ձևը՝ WorkFlow-ն[39]։

Շատ հայտնի մի օրինակ․ ցանկացած կայքում, երբ դուք խճճվել եք և ցանկանում եք վերադառնալ գլխավոր էջ, ապա ո՞րտեղ կփնտրեք “Home” կոճակը, դե իհարկե վերևի ձախ անկյունում՝ լոգոյի վրա։ Իհարկե սա շատ պարզ օրինակ էր վեբ դիզայնից, բայց աշխատանքի ընթացքում էլ ծրագիրը պետք է լինի ինտուիտիվ։

Կամ ենթադրենք, թե ունենք դոկումենտացիա, որտեղ նկարագրվում է, թե ինչպես կարելի է կատարել ինչ-որ գործողություն։ Տարբեր մարդկանց տալով դոկումնետացիայի որոշակի մաս, կարելի է հասկանալ, թե որքան մատչելի է այնտեղ ներակայացվում ծրագրի աշխատանքը։

Ընդհանուր առմամբ սա նման է Acceptance testing-ին, սակայն օգտագործելիության թեստավորումը կատարվում է ոչ թե release-ից առաջ, այլ ծրագրի ծրագրավորման ընթացքում և միտում ունի ծրագրի կատարելագործման։

**Առավելությունները**

1․ Հետադարձ կապը ուղղակիորեն թիրախային լսարանից։

2․ Ներքին բանավեճերը կարող են լուծվել կախված նրանից, թե ինչպես են օգտատերերն արձագանքում քննարկվող տարբերակներին։

3․ Խնդիրներն ու հնարավոր խնդիրները ընդգծվում են նախքան Release-ը։

**Usability testing-ի բիզնես առավելությունները․**

1․ Դա մեծացնում է օգտագործման և կրկնակի օգտագործման հավանականությունը։

2․ Դա նվազագույնի է հասցնում արտադրանքի ձախողման ռիսկը։

3․ Օգտատերերն ավելի լավ են կարողանում հասնել իրենց նպատակներին, ինչը հանգեցնում է այն բանի, որ բիզնեսը համապատասխանում է իր նպատակներին։

## **4․12 Ընդարձակելիության թեստավորում - Scalability testing**

Թեստավորման մեթոդաբանություն է, որտեղ ծրագրի կատարողականը ստուգվում է օգտատիրոջ կողմից արվող հարցումները բարձրացնելու կամ իջեցնելու ունակություններով, կամ նմանատիպ այլ կատարողական չափանիշներով։

Այս թեստավորման համար օգտագործված պարամետրերը տարբերվում են՝ կախված ծրագրի տեսակից. վեբ էջի համար դա կարող է լինել օգտատերերի քանակը, պրոցեսորի օգտագործումը, ցանցի օգտագործումը, մինչդեռ վեբ սերվերի համար դա կլինի մշակված հարցումների քանակը:

Առաջին հայացքից սա շատ նման է Load testing-ին․ թվարկենք դրանց տարբերությունները։

Load Testing-ը կատարվում է այն առավելագույն ծանրաբեռնվածության տակ, որի դեպքում համակարգը խափանվում է: Load testing-ի հիմնական նպատակը մաքսիմում օգտատերերի քանակի հայտնաբերումն է, որից հետո օգտվողները չեն կարողանա օգտագործել համակարգը:

Scalability testing-ը տարբերվում է Load testing-ից այն փաստով, որ Scalability testing-ը համակարգը չափում է նվազագույն և առավելագույն ծանրաբեռնվածությունների բոլոր մակարդակներում, ներառյալ տվյալների բազայի մակարդակները: Առավելագույն ծանրաբեռնվածությունը պարզելուց հետո ծրագրավորողները պետք է պատշաճ կերպով արձագանքեն որոշակի ծանրաբեռնվածությունից հետո համակարգը ընդարձակելի դարձնելու համար:

Օրինակ, եթե Scalability testing-ի համար որոշվում է, որ առավելագույն ծանրաբեռնվածությունը պետք է լինի 10,000 օգտատեր, ապա համակարգի ընդարձակելի լինելու համար ծրագրավորողները պետք է միջոցներ ձեռնարկեն կատարելագործելով այնպիսի գործոններ, ինչպիսիք են` արձագանքման ժամանակի նվազեցումը 10,000 օգտատիրոջ սահմանը հասնելուց հետո, կամ RAM-ի չափի ավելացումը` շատ ավելի օգտատերեր 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։

Load Testing-ի ժամանակ ծրագրիրը դրվում է առավելագույն ծանրաբեռնվածության միանգամից, մինչդեռ Scalability testing-ը ընդգրկում է ծանրաբեռնվածության աստիճանական ավելացումը որոշակի ժամանակահատվածում։

Load Testing-ը որոշում է այն կետը, երբ ծրագիրը խափանվում է, մինչդեռ մասշտաբայնությունը փորձում է պարզել ծրագրի խափանման պատճառը քայլ առ քայլ և օգնում քայլեր ձեռնարկել խնդրի լուծման համար:

Այսպիսով, Load Testing-ը օգնում է բացահայտել կատարողականի խնդիրները, մինչդեռ Scalability testing-ը օգնում է պարզել, թե արդյո՞ք համակարգը կարող է ընդարձակվել օգտատերերի աճող թվին համեմատ:

## **4․13 Համատեղելիության թեստավորում - Compatibility testing**

Թեստավորման այս տեսակը նախատեսված է հասկանալու համար, թե արդյո՞ք ծրագիրը կամ արտադրանքը բավականաչափ լավ կարողանում է աշխատել տարբեր բրաուզերներում, տվյալների բազայում, ապարատային համակարգում, օպերացիոն համակարգում, շարժական սարքերում և ցանցերում:

Ծրագիրը կարող է տարբեր կերպ աշխատել տարբեր version-ների, համացանցի արագության և կոնֆիգուրացիաների դեպքում: Ուստի կարևոր է ծրագիրը թեստավորել բոլոր հնարավոր ձևերով` bug-երը նվազեցնելու և դրանք հաղթահարելու համար: Ինչպես ոչ ֆունկցիոնալ թեստերը, համատեղելիության թեստավորումը նշանակում է, որ ծրագիրը հաջողությամբ գործում է տարբեր բրաուզերներում, վերսիաներում, OS-երում և ցանցերում:

Compatibility testing-ի տեսակները

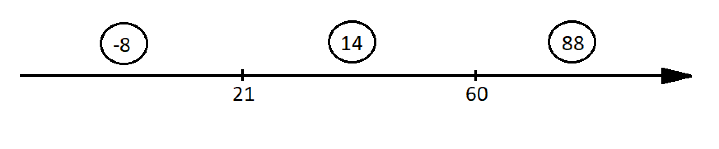
* Բրաուզերների Compatibility testing
* Սարքավորումների Compatibility testing
* Ցանցերի Compatibility testing
* Բջջային սարքերի Compatibility testing
* Օպերացիոն համակարգերի Compatibility testing
* Version-ների Compatibility testing

# Գլուխ 5․ Blackbox testing տեխնիկաներ

## **5.1 Համարժեքային բաժանում - Equivalence partitioning**

Այս մեթոդով մուտքային տիրույթի տվյալները բաժանվում են տարբեր համարժեք տվյալների դասերի՝ class-ների, և ծրագիրը սկսում է թեստավորվել յուրաքանչյուր դասից 1 արժեք ընտրելով: Այս մեթոդը օգտագործվում է Test case-երի ընդհանուր քանակը նվազեցնելու համար։

Օրինակ` ունենք դաշտ, որն ընդունում է 21-ից 60 բնական թվերը։ Այն կբաժանենք 3 դասի՝ invalid ներքևից (-∞;21), valid [21;60], invalid վերևից (60;∞) և թեստավորման ընթացքում, եթե ընտրել ենք թեստավորման այս մեթոդը, ապա այդ դաշտին կփոխանցեք հետևյալ կամայական արժեքները յուրաքանչյուր դասից՝ -8, 14, 88։



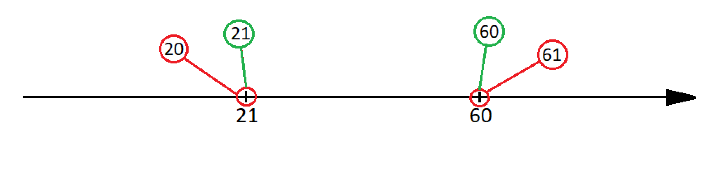
Թեստավորման այս տեսակը իրականացվում է շատ արագ թեստավորելու համար, սակայն ավելի ճիշտ կլինի ժամանակ հատկացնել և ստուգել սահմանային արժեքներով, որը նկարագրված է հաջորդ ենթագլխում։

## **5․2 Սահմանային թեստավորում - Boundary testing**

Սահմանային արժեքներով ստուգումն այն պարզ թեստավորման տեսակներից մեկն է, որն օգտագործում են Smoke թեստի ժամանակ։ Նորից ենթադրենք, թե ունենք դաշտ, որն ընդունում է 21-ից 60 բնական թվերը, ինչպես նախորդ օրինակում։ Թեստավորման այս մեթոդի դեպքում օգտագործում ենք 4 տարբեր տվյալ՝

* Min value – 1
* Min value
* Max value
* Max value +1

Մեր օրինակում Min value=21, Max value=60, հետևաբար` ֆունկցիային կփոխանցենք հետևյալ արժեքները՝ 20, 21, 60, 61։

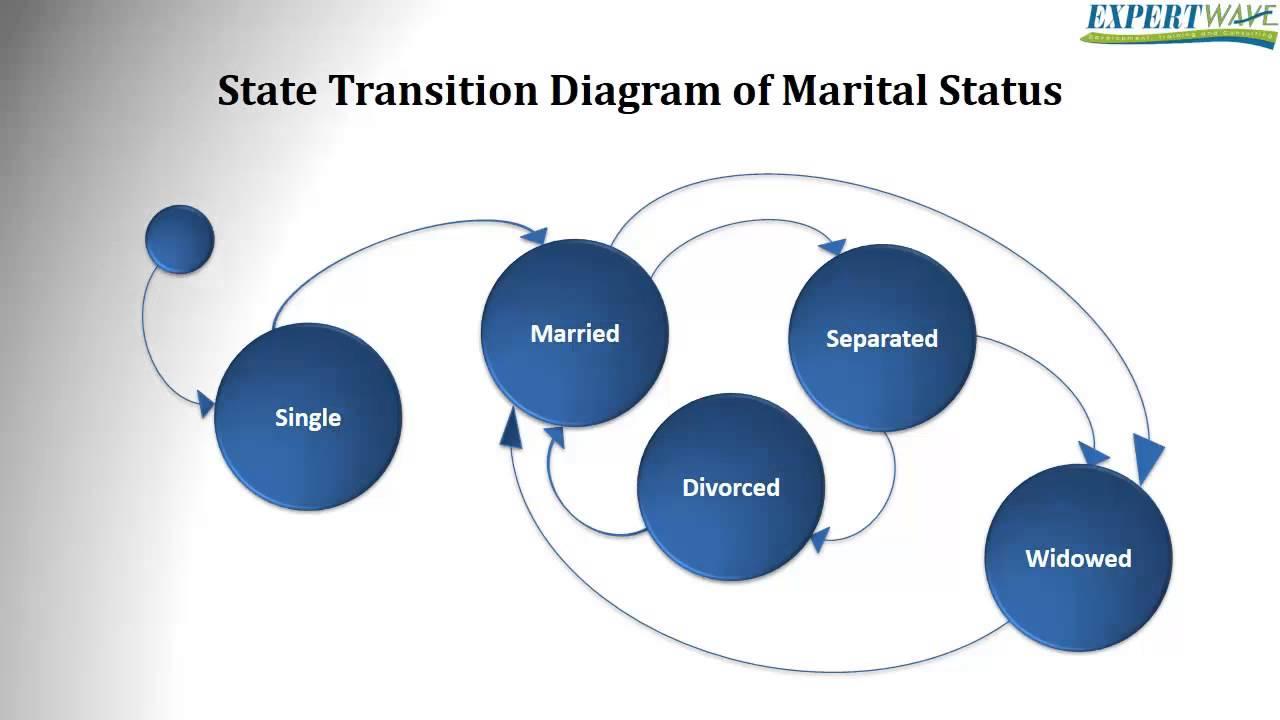


Այս դեպքում կարող ենք ասել, որ, ստուգելով 4 դեպք, կարողանում ենք ունանալ ավելի մեծ հավանականություն, որ մնացած բոլոր թվերով այն ճիշտ կաշխատի (մոտ 90%): Իսկ որքա՞ն հավանականություն կա առաջին դեպքում (համարժեքային բաժանում), փորձեք վերլուծել ինքներդ և պատասխանել դրան։

## **5.3 Վիճակի անցման թեստավորում - State transition tables**

Շատ-շատ ծրագրեր, որոնք նախատեսված են փաստաթղթաշրջանառության, տրանսպորտային համակարգերի օպտիմալացման, պահեստների և այլ ոլորտների համար, ունեն աշխատանքի որոշակի քայլերի հերթականություն, որոնք անցնում են որոշակի state-երի՝ վիճակների, միջով։ Թեստավորման այս տեսակը նախատեսված է A կետից հասնել Z կետ ամենակարճ, բայց բոլոր վիճակներով, այսպիսով ստուգելով, որ բոլոր տարբերակներով անցումներն աշխատում են ճիշտ։

Ենթադրենք, ունենք ամուսնությունների գրանցման պետական ծրագիր, որը նախատեսված է յուրաքանչյուր նոր կարգավիճակի հետ տալ վկայագրեր՝ գրանցել ամուսնական զույգերին, տալ բաժանման թուղթ, տալ վերաամուսնական թուղթ և այլն։ Ստորև ներկայացված է այն սխեման, թե ինչ քայլերի հերթականությամբ է հնարավոր աշխատեցնել ծրագիր։



Նկարից կտեսնենք, որ՝

1․ Մարդու սկզբնական կարգավիճակը «Միայնակ» է և նա կարող է անցնել միայն «Ամուսնացած» կարգավիճակին։

2․ «Ամուսնացած» կարգավիճակից հնարավոր է անցնել 2 այլ վիճակների՝ «Առանձնացած» և «Այրի»։

3․ «Առանձնացած» կարգավիճակից հնարավոր է անցնել «Բաժանված» և «Այրի» կարգավիճակներին։

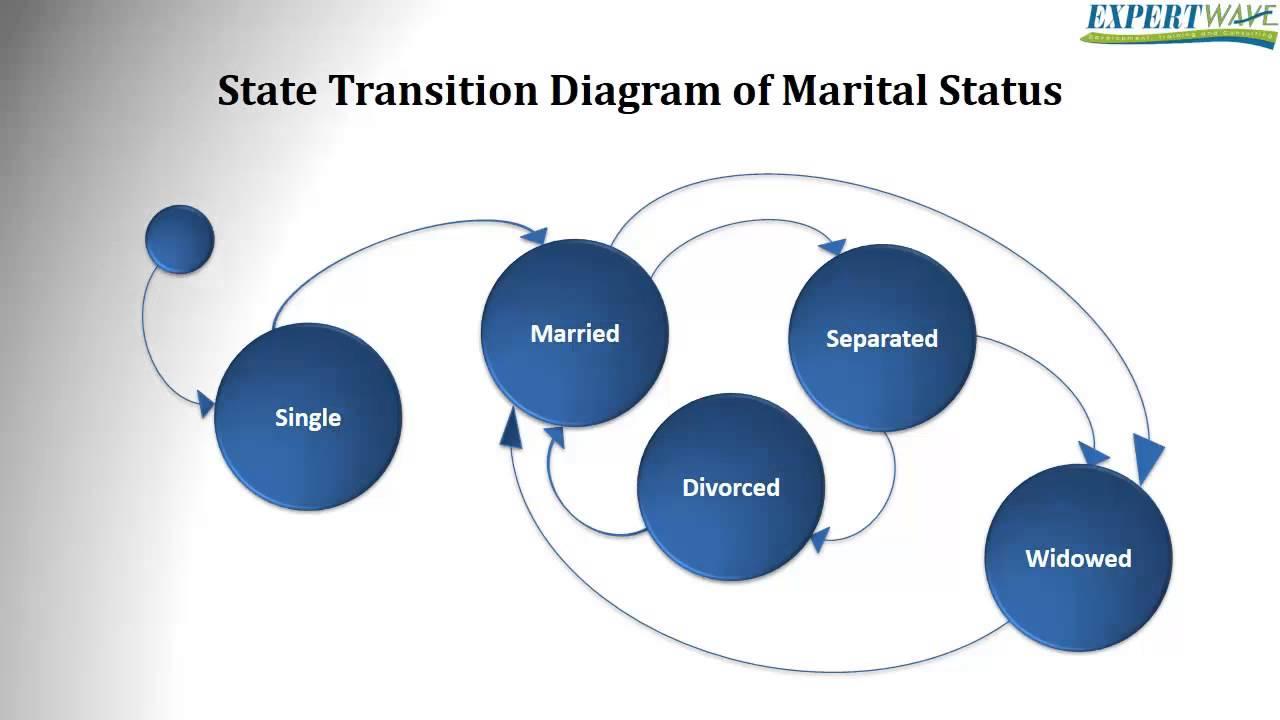
4․ «Այրի» վիճակից հնարավոր է վերադառնալ միայն «Ամուսնացած» կարգավիճակին։

Եթե կազմենք աղյուսակ, ապա կունենանք հետևյալ պատկերը․

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Ներկա կարգավիճակ** | **Ամուսնության վկայագիր** | **Համաձայնագիր** | **Մահվան վկայագիր** | **Բաժանման թուղթ** |
| Միայնակ | Ամուսնացած |  |  |  |
| Ամուսնացած |  | Առանձնացած | Այրի |  |
| Առանձնացած |  |  | Այրի | Բաժանված |
| Բաժանված | Ամուսնացած |  |  |  |
| Այրի | Ամուսնացած |  |  |  |

Այժմ, հարկավոր է գտնել այն ամենակարճ ճանապարհը, որի ժամանակ մենք կկարողանանք անցնել այս բոլոր կարգավիճակներով։

Եթե «Ամուսնացած» վիճակից անցնենք «Առանձնացած» վիճակին, ապա կունենանք ընտրության 2 տարբերակ, դրա համար հարկավոր է ընտրել այնպիսի ճանապարհ, որպեսզի հետագա տարբերակները լինեն մաքսիմալ քիչ։ Այդ իսկ պատճառով մենք սկզբից կգնանք «Այրի» վիճակով, կվերադառնանք «Ամուսնացած» վիճակին, այնուհետև կգնանք «Առանձնացած», ապա «Բաժանված» վիճակներին և այսպիսով ամենակարճ ճանապարհով մենք կկարողանանք անցնել տվյալ օբյեկտի բոլոր վիճակներով։ Այսինքն, մենք կթեստավորենք ծրագրի բոլոր հնարավոր վիճակների անցման ֆունկցիաները ամենակարճ ճանապարհով։



Իհարկե, այս տարբերակը մի փոքր կարող է և վիճելի լինել, քանի որ այն բավականին փոքր է, սակայն ավելի մեծ դիագրամների դեպքում հստակ կան ճանապարհներ, որոնց միջոցով կարելի է ամենակարճ և օպտիմալ ճանապարհով անցնել այդ Workflow-ի բոլոր վիճակներով։ Ստորև բերված օրինակում արդեն իսկապես բարդ է հասկանալ, թե որն է այն օպտիմալ ճանապարհը, որով կարելի է անցնել այս workflow-ի բոլոր վիճակներով։

Մեկ այլ օրինակը հետևյալն է։ Պատկերացնենք, որ թեստավորում ենք երաժշտական ծրագիր, որը կարողանում է աշխատել վեց տեսակի աուդիո ֆայլերի հետ՝ mp3, flac, dff, aac, ogg, wav: Այս պարագայում չկա օպտիմալ լուծում, քանի որ ցանկացած աուդիո ֆայլի տեսակից կարող ենք անցում կատարել մեկ այլ ֆայլի և չկա ոչ մի սահմանափակում։ Ստացվում է, որ ամեն հաջորդ քայլին մենք ունենում ենք ընտրության 5 տարբերակ։ Հետևաբար այստեղ, ցավոք, չկա ամենակարճ ճանապարհ։ Բայց, ամեն դեպքում, երբ սկսում ենք թեստավորել ծրագիրը, փորձելով անցնել մի երաժշտական ֆայլից մյուսը, անվանում ենք վիճակի անցման թեստավորում։

Եթե մի փոքր խորը դիտարկենք, ապա հենց Playing վիճակից անցումը Paused վիճակ և հակառակը ևս հանգեցնում է ծրագրի վիճակի փոփոխման, հետևաբար այս դեպքը ևս կկոչվի վիճակի անցման դեպք՝ state transition case:

Այսպիսով, ցանկացած ծրագիր ունի որոշակի վիճակներ և շատ կարևոր է, որ մի վիճակից մյուսին անցնելիս համակարգը չխափանվի։ Այդ իսկ պատճառով, պետք է փորձել աղյուսակի վերածել վիճակների անցումների տարբերակները և գտնել թեստավորման ամենակարճ ճանապարհը՝ բոլորի ժամանակը խնայելու համար։

## **5․4 Որոշումների աղյուսակային թեստավորում - Decision table testing**

Որոշմների աղյուսակը Blackbox testing-ի տեխնիկա է, որը ներկայացնում է մուտքերի և ելքերի համադրություններ, որտեղ մուտքերը (input) պայմաններ կամ դեպքեր են, իսկ արդյունքները (output) գործողություններ կամ էֆեկտներ՝ հետևանքներ: Decision table testing-ը պարունակում է բոլոր պայմանների և գործողությունների համադրությունները: Դրանք կարող են կիրառվել նաև այն դեպքում, երբ ծրագրի գործողությունը կախված է շատ տրամաբանական որոշումներից: Այս մեթոդները կարող են օգտագործվել թեստավորման ցանկացած մակարդակում:

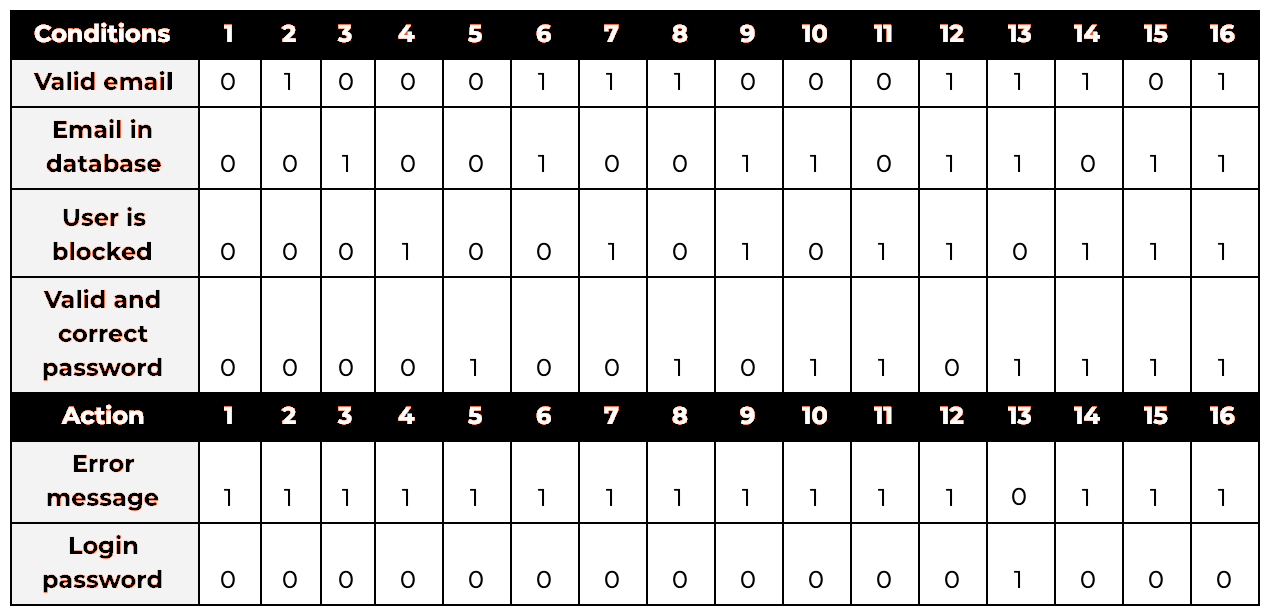
Ինչպե՞ս ստեղծել Decision table:

Բացատրելու համար, թե ինչպես է ստեղծվում Decision table-ը կբերենք օրիկնակ: Պատկերացնենք, թե ունենք մի էջ, որտեղ օգտագործողը կարող է լրացնել իր email-ը և password-ը՝ համակարգ մուտք գործելու համար։ Բացի այն, որ համակարգը ստուգում է, թե արդյո՞ք նման login password-ի համընկնում կա տվյալների բազայում, այն նաև կարող է թուլ չտալ օգտատիրոջը մուտք գործել համակարգ, քանի որ նրա հաշիվը արգելափակված է։

Այսպիսով, տվյալները լրացնելու և համակարգի վարվելակերպի ի՞նչ տարբերակներ ունենք․

1. Օգտատերը կարող է մուտքագրել valid կամ invalid email-եր։
2. Օգտատիրոջ email-ը DB-ում է։
3. Օգտատերը կարող է լինել արգելափակված։
4. Օգտատերը մուտքագրում է ճիշտ կամ սխալ գաղտնաբառ։

Ստորև ներկայացված են բոլոր հնարավոր և անհանար դեպքերը, որոնք պետք է վերլուծել և հասկանալ, թե որո՞նք են non-sense դեպքեր և կրճատել դրանք։ Իհարկե, որոշումների աղյուսակներում մենք կարող ենք օգտագործել Այո/Ոչ կամ True/False, որոնք անալոգ են 0/1 արժեքներին:



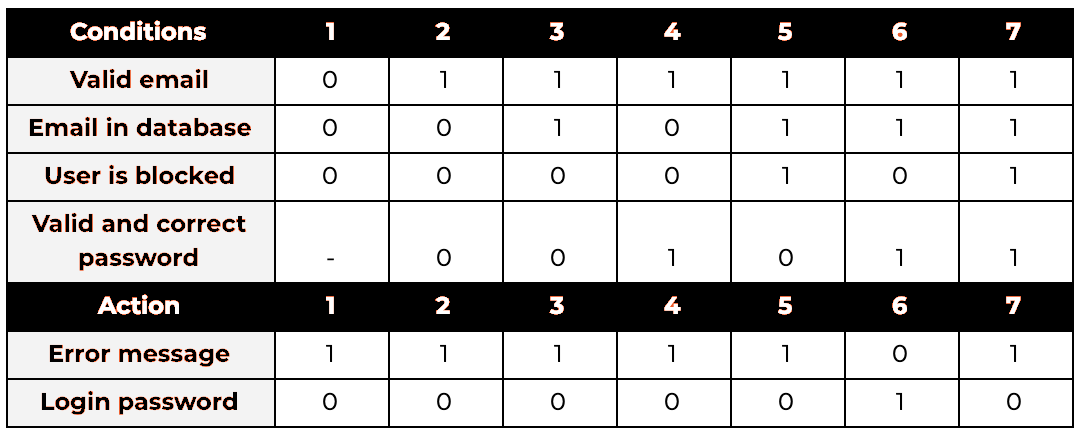
Աղյուսակը թվում է չափազանց բարդ և ունի օպտիմալացման կարիք։ Հարկավոր է հանել անհնարին դեպքերը և այն համատեղելի դեպքերը, որոնց համար պայմաններից մեկի փոփոխությունը էական ազդեցություն չունի համակարգի գործողությունների վրա:

Մենք կարող ենք տեսնել, որ որոշ դեպքեր իմաստ չունեն՝ 3-րդ, 9-րդ, 10-րդ և 15-րդ դեպքերը չեն կարող տեղի ունենալ, քանի որ, երբ email-ն անվավեր է, այն չի կարող պահվել տվյալների բազայում, ուստի մենք կարող ենք ջնջել այդ դեպքերը: Այսինքն, չի կարող լինել այնպես, որ կարողանանք մուտքագրել բազայում գոյություն չունեցող email, որը կա բազայում։ Դա նույնն է, որ փորձենք սառնարանից ջրի շիշ վերցնել՝ իմանալով, որ այնտեղ ջրի շիշ չկա։

Հեռացման հաջորդ դեպքերը 4-րդ, 7-րդ, 11-րդ և 14-րդն են։ Եթե email-ը չի պահվում տվյալների բազայում, այն չի կարող արգելափակված լինել: Վերը նշված զուգահեռ օրինակով՝ ջրի շիշը չի կարող փակված (արգելափակված) լինել, եթե գիտենք, որ սառնարանում չկա ջրի շիշ։

Չեն կարող լինել նաև դեպքեր, երբ email-ը լինի valid, բայց այն չլինի բազայում (2-րդ, 7-րդ, 8-րդ, 14-րդ)։ Դա նույնն է, որ մենք սառնարանից վերցնենք ջուր, բայց սառնարանում նման բան չունենանք։

Կարող ենք նաև միավորել 1-ին և 5-րդ դեպքերը, երբ email-ը invalid է (գաղտնաբառի արժեքը նշանակություն չունի): Մի քանի դեպք պարզեցնելուց և վերացնելուց հետո տեսնենք, թե աղյուսակն ինչպիսի տեսք է ունենում:



Ահա և բոլոր այն հնարավոր դեպքերը, որոնք կարողացանք դուրս բերել աղյուսակից և դրանցից յուրաքանչյուրի համար, արդեն, կգրվեն Test case-եր։

Ներկայացնենք մեկ այլ առաջադրանք, որպեսզի հասկանանք, թե ինչպես կարելի է գտնել ծրագրի խոցելի տեղերը իրական կյանքում, այս մեթոդի միջոցով։ Ստորև ներկայացված են պահանջները։

1. Ցիստեռնի մեջ ջուրը պետք է ունենա 30-60°C ջերմաստիճան:
2. Ջեռուցիչը պետք է անջատվի, երբ ցիստեռնի ջուրը իջնի նվազագույն մակարդակից ցածր:  
   Այսինքն՝
3. Ջեռուցիչը պետք է միանա, երբ ջերմաստիճանը իջնում է 30°C-ից:
4. Ջեռուցիչը պետք է անջատվի, երբ ջերմաստիճանը հասնում է 60°C-ի:

Ելնելով վերը նշվածից՝ փորձեք ստեղծել որոշումների աղյուսակ ու, դեպքերը բացառելով, ստանալ վերջնական բոլոր հնարավոր տարբերակներով աղյուսակ։

Ինչպե՞ս կարող է Decision table-ը օգնել ծրագրի թեստավորման մեջ:

Լավ ստեղծված Decision table-ը կարող է օգնել կարգավորել համակարգի ճիշտ արձագանքը՝ կախված մուտքային տվյալներից, քանի որ այն պետք է ներառի բոլոր պայմանները: Այն հեշտացնում է տրամաբանության ձևավորումը և այդպիսով բարելավում մեր արտադրանքի development-ն ու testing-ը: Դիզայնավորված աղյուսակներով տեղեկատվությունը ներկայացվում է ավելի պարզ, հասկանալի ձևով, ուստի դրանք գտնելն ավելի հեշտ է, քան համակարգի տրամաբանությունը նկարագրող տեքստում: Եվ, վերջապես, իհարկե, այս տեխնիկայի միջոցով թեստավորելն օգնում է գտնել եզրային դեպքեր և հայտնաբերել համակարգում բացակայող ֆունկցիաները։

## **5․5 Սխալների ենթադրության տեխնիկա - Error Guessing Technique**

Թեստավորման այս տեխնիկան չունի ոչ մի տեխնիկա։ Այն բացառապես հիմնված է թեստավորողի փորձի վրա։

Error Guessing Technique-ի համար դժվար է ընթացակարգ սահմանել, քանի որ այն հիմնականում ինտուիտիվ և ժամանակավոր գործընթաց է: Հիմնական գաղափարը հնարավոր սխալների կամ սխալների հակված իրավիճակների ցանկը թվարկելն է, այնուհետև ցուցակի հիման վրա գրել Test case-երը: Օրինակ` ցանկացած ծրագրի աշխատանքի մեջ 0 արժեքի առկայությունը bug-ի հակված իրավիճակ է, քանի որ փորձը ցույց է տվել, որ միշտ 0-ի հետ կապված առաջանում են խնդիրներ: Հետևաբար, պետք է գրել Test case-եր, որոնց համար որոշակի մուտքային արժեքներ ունեն 0-ն և որոնց համար որոշակի ելքային սպասվող արժեքներ կարող են լինել 0: Ինչպես նաև որտեղ կարող է առկա լինել մուտքերի կամ ելքերի փոփոխական քանակ (օրինակ՝ ցուցակի մուտքերի քանակը պետք է ցույց տրվի), «false»-ի և «մեկ»-ի դեպքերը (օրինակ՝ դատարկ ցուցակ, ընդամենը մեկ մուտք պարունակող ցուցակ) bug-երի հակված իրավիճակներ են:

Կախված թեստավորողի ինտուիցիայից և փորձից՝ բոլոր թերությունները չեն կարող շտկվել։ Կան որոշ գործոններ, որոնք պետք է հաշվի առնվեն․

* Թեստավորողի ինտուիցիան
* Պատմական ուսուցում
* Ckecklist-ի վերանայում
* Ծրագրային ապահովման ռիսկերի վերլուծություն
* Հավելվածի UI
* Ընդհանուր թեստավորման կանոններ
* Նախորդ թեստի արդյունքներ
* Անցյալում թերությունների առկայություն
* Տվյալների բազմազանություն, որոնք օգտագործվում են թեստավորման համար
* Application-ի շատ խորը իմացություն

Որպեսզի ավելի լավ կարողանանք ենթադրել խնդիրները, մեզ պետք կգան գիտելիքներ նաև exception-ների մասին։

* Arithmetic exceptions
* NullPointer Exception
* StringIndexOutOfBound Exception
* FileNotFound Exception
* NumberFormat Exception
* ArrayIndexOutOfBounds Exception
* IOException
* և այլն

Նմանատիպ exception-ներն արդեն կքննարկվեն այս գրքի հաջորդ ձեռնարկում։

Թեստավորման այս տեխնիկան ամենատարածվածն է և ինչպես արդեն հասցրեցիք հասկանալ, այն պահանջում է ոչ միայն գիտելիքներ, այլ նաև ինտուիցիա և վերլուծության շատ լուրջ հմտություններ (Analytical thinking)։ Հենց այս պատճառով է, որ թեստավորողի առաջնային հմտություններից մեկը պետք է լինի «Լավ և ճիշտ փորփրելը»։

Թեստավորման այս տեխնիկան չունի սահմանումներ, չի կարող ապահովել full coverage, սակայն ամենակիրառելին է։

## **5․6 Գրաֆների վրա հիմնված թեստավորում - Graph-based testing**

Թեստավորման այս տեխնիկան կիրառվում է այն ծրագրերի մեջ, որոնք աշխատում են գրաֆների տեսության հիման վրա։ Որպեսզի հասկանանք գրաֆների թեստավորման տեխնիկան, նախ պետք է հասկանալ, թե ինչ է հենց ինքը՝ գրաֆը։

Ժամանակակակից աշխարհը պարունակում է շատ մեծ տվյալներ ամենատարբեր ոլորտների մասին։ Կան ծրագրեր, որոնք կատարում են որոնողական աշխատանքներ՝ այդ ահռելի տվյալների բազայից համապատասխան տվյալ գտնելու և մատուցելու համար (Google, Yahoo, Bing): Համաձայնեք, որ այնքան էլ հեշտ չէ աշխատել մեծ տվյալների հետ և այստեղ պետք է կիրառվեն մաթեմատիկական ալգորիթմներ՝ այդ ամենը օպտիմալացնլու համար։ Եթե մենք փնտրենք “Fugue”, ապա այն մեզ ոչ միայն կբերի ֆուգայի մասին տեղեկություններ, այլև կբերի այլ տվյալներ, ինչպիսիք են Բախը և երաշժտությունը։ Ինչո՞ւ։ Քանի որ դրանք տվյալների բազայում որոշակի կապերով սերտ միացված են իրար և առաջինի փնտրման դեպքում այդ կապերի միջոցով բերվում են այլ տվյալներ ևս։ Եթե փնտրման ալգորիթմը լիներ գծային, ապա այդ տվյալների որոնումը և մատուցումը գուցե կտևեր 10 վայրկյանից ավել։ Ի հեճուկս նման խնդիրների, մաթեմատիկոսների կողմից ստեղծվել է գրաֆների տեսությունը, որի նպատակն է լուծել A կետից B կետ հասնելու ճանապարհի խնդիրը օպտիմալ ձևով՝ ամենակարճ ճանապարհով։

Այսպիսով, գրաֆներով աշխատող ծրագրի թեստավորումը կատարվում է ևս գրաֆներով, հասկանալու համար, թե որոնք են այն հիմնական ճանապարհները, որոնցով ծրագիրը ներկայացնում է վերջնական արդյունք։

Գրաֆների վրա հիմնված թեստավորման համար նախ կառուցվում է գրաֆի մոդելը, այնուհետև փորձում է ծածկել գրաֆի մոդելի որոշ տարրեր:

* Քայլ 1. Կառուցել գրաֆի մոդել

– Ի՞նչ տեղեկատվություն պետք է հավաքել և ինչպե՞ս ներկայացնել այդ տեղեկատվությունը:

• Քայլ 2. Որոշել թեստի պահանջները (TR)

– Թեստավորման պահանջը գրաֆիկական մոդելի կառուցվածքային միավոր է, որը պետք է ծածկվի թեստավորման ընթացքում։

• Քայլ 3. Ընտրել թեստավորման ուղիներ՝ այդ պահանջները բավարարելու համար։

• Քայլ 4. Ստանալ թեստի տվյալներ, որպեսզի այդ թեստավորման ուղիները հնարավոր լինի կատարել։

**Գրաֆ թեստավորման տեսակները**

• Կառավարման հոսքի գրաֆիկ - ֆիքսում է տեղեկատվությունը այն մասին, թե ինչպես է կառավարումը փոխանցվում/կատարվում ծրագրում:

• Տվյալների հոսքի գրաֆիկ - մեծացնում է CFG-ն տվյալների հոսքի մասին տեղեկություններով։

• Կախվածության գրաֆիկ - ընդգրկում է տվյալների/վերահսկման կախվածությունը ծրագրի մոդուլների միջև։

• Պատճառահետևանքային գրաֆիկ - ծրագրի մուտքային պայմանների միջև հարաբերությունների մոդելավորում (որոնք հայտնի են որպես պատճառներ) և ելքային պայմաններ (որոնք հայտնի են որպես հետևանքներ)։

Ավելի խորը անդրադարձ կկատարվի թեստավորման այս տեխնիկային հաջորդ գրքի մեջ, քանի որ այն պարունակում է շատ տեխնիկական և մաթեմատիկական գիտելիքներ։

# Գլուխ 6․ Հավելում

## **6․1 Web testing – Cross Browser Testing**

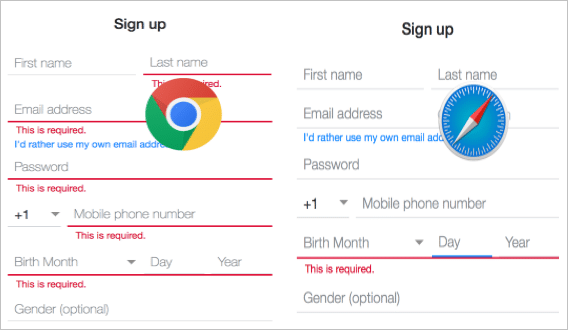
Cross Browser Testing-ը թեստավորման այն տեսակն է, որը պետք է ստուգի, թե արդյո՞ք ծրագիրն աշխատում է տարբեր browser-ներում, այնպես, ինչպես և սպասվում է: Դա հավելվածի համատեղելիությունը տարբեր browser-ների հետ հաստատելու գործընթացն է:

Բազմաթիվ անգամներ, երբ ծրագրում խնդիրներ են առաջացել որևէ browser-ով աշխատելիս, նույն խնդիրը չի հայտնաբերվել մեկ այլ browser-ի դեպքում։ Իսկ մենք գիտենք, որ browser-ները բավականին շատ են։

Բոլորս էլ նկատած կլինենք, որ որոշ կայքեր պատշաճ կերպով չեն աշխատում որոշ browser-ներում, և մենք պարզապես կարծում ենք, որ կայքը լավ չի աշխատում: Սակայն, եթե փորձենք այն բացել այլ browser-ի վրա, կայքը կբացվի բավականին լավ: Այսպիսով, այս պահվածքը բացատրում է կայքի համատեղելիությունը տարբեր browser-ների հետ:

Յուրաքանչյուր browser-ում կայք-էջի տեղեկատվությունն այլ կերպ է ցուցադրում: Այսպիսով, որոշ browser-ներում կարող են պակասել այն հնարավորությունները, որոնք վեբ կայքը փորձում է ցույց տալ և այն կոտրված տեսք կունենա այդ browser-ի վրա:

Օրինակ, ինչպես ցույց է տրված ներքևում, գրանցման ձևերի սխալները նույնը չեն տարբեր browser-ների դեպքում: Նաև տարբեր են տեքստի գույնը, տառատեսակն ու հատկանիշները։



Տեխնոլոգիայի առաջխաղացման պայմաններում browser-ների համար կան մի քանի տարբերակներ, և, բնականաբար, ցանկալի է, որ կայքը աշխատի բոլոր browser-ների վրա։ Այսպիսով, անհրաժեշտ է դառնում թեստավորել ձեր կայքի համատեղելիությունը տարբեր browser-ների հետ: Սովորաբար օգտագործվող browser-ներից են Chrome, Safari, Firefox, Internet Explorer և այլն:

Ի՞նչ է Cross Browser testing-ը

1․ Cross Browser թեստավորումն ուղղակի նշանակում է թեստավորել ձեր վեբ կայքը կամ ծրագիրը տարբեր browser-ներում՝ ստուգելու և համոզվելու համար, որ այն աշխատում է պատշաճորեն և չունի կախվածություն որևէ browser-ից:

2․ Սա կիրառվում է և՛ վեբ, և՛ մոբայլ application-ների համար (եթե browser-ները մի քանիսն են, ապա հեռախոսները 100-ապատիկ ավելի շատ են):

Հարց է առաջանում, թե արդյո՞ք բոլոր browser-ների բոլոր վերսիաներով հարկավոր է թեստավորել։ Պատասխանն է` իհարկե ո’չ։ Հաճախորդը, բիզնես անալիստների թիմը կամ մարկետինգի թիմը կտա ձեզ այն browser-ների և դրանց վերսիաների ցանկը, որոնց վրա հարկավոր կլինի թեստավորել։

**Ինչպե՞ս իրականացնել Cross Browser testing**

Առաջին և ոչ էֆեկտիվ տարբերակը manual testing-ն է։ Թեստավորողը պետք է միևնույն Test case-ը ստուգի բոլոր browser-ներով։ Սակայն առանց շատ մանրանալու էլ արդեն իսկ հասկանալի է, որ այս տարբերկը շատ անիմաստ է ժամանակատարության առումով, հետևաբար միանգամից անցնենք ավտոմատացված համակարգի մասին խոսելուն։

Cross Browser testing**-**ը հիմնականում կատարում է նույն Test case-երի սեթը տարբեր Browser-ների վրա։ Կրկնվող առաջադրանքի այս տեսակը լավագույնս հարմար է ավտոմատացման համար: Այսպիսով, ավելի թանկ, բայց ժամանակի արդյունավետությամբ կարելի է իրականացնել այս թեստավորումը՝ օգտագործելով գործիքներ: Շուկայում կան բազմաթիվ գործիքներ, որպեսզի դա հեշտացվի: Ահա, թե ինչեր են դրանք ապահովում․

1․ Դրանք տրամադրում են VPN[40] (Վիրտուալ մասնավոր մեքենա), որի միջոցով դուք կարող եք միանալ remote համակարգիչներին և ստուգել ձեր JAVA, AJAX, HTML, Flash և այլ էջերը: Դրանց մեծ մասը անվտանգ են, բայց քանի որ դուք ինֆորմացիա եք տրամադրում այդ երրորդ կողմին, ապա ամեն դեպքում հարկավոր կլինի կատարել որոշակի անվտանգության վերլուծություններ:

2․ Ստատիկ կերպով պահպանում են screenshot-եր և հնարավորություն են տալիս համեմատելու, թե որ browser-ում ինչպես է երևում ծրագիրը։

3․ Աշխատում են բազմաթիվ browser-ների բազմաթիվ վերսիաների հետ և սիխրոնիզացնում են աշխատանքը։

4․ Կարողանում են միաժամանակ ցույց տալ browser-ի տարբեր չափերի դեպքում արտապատկերված տեսքը։

5․ Եթե առաջանում է խնդիր, ապա դա կարողանում է ներկայացնել video recording-ի ձևով։

6․ Աշխատում են և՛ վեբ, և՛ մոբայլ app-երի համար։

7․ Հնարավոր է թեստավորել private էջեր, որոնք պահանջում են authentication։

8. Կարող են թեստավորվել նաև private network/firewall էջեր։

Նմանատիպ թեստերի համար կան հետևյալ գործիքները՝ “LambdaTest”, CrossBrowserTesting, Selenium, BrowserStack, Browserling.

Ի՞նչ թեստավորել․

1․ Բազային գործառույթներ - Հղումներ, դիալոգներ, մենյուներ և այլն։

2․ Գրաֆիկական user ինտերֆեյս - app-ի տեսքը:

3․ Պատասխան՝ response - ինչպե՞ս է application-ը արձագանքում օգտագործողի գործողություններին:

4․ Կատարողական՝ performance - Էջերի load-ը թույլատրելի ժամանակահատվածներում:

## **6․2 Combobox-ի (Dropdown menu) գաղտնիքները․․․**

Բոլորիս էլ քաջ հայտնի են combobox-երը և դրանք կան գրեթե ամենուր։ Ստորև կներկայացնեմ 12 կետ, որոնք համարվում են ստանդարտներ։ Այս օրինակները ավելի շատ բերված են այն կոմբոների համար, որտեղ կարելի է արժեքն ընտրելու փոխարեն այն նաև ավելացնել։

* Գրել որևէ տառ և համոզվել, որ այդ տառին համապատասխան գալիս են ճիշտ արժեքներ:
* Գրել որևէ գոյություն ունեցող բառ, ապա սկսել ջնջել տառերը․ ըստ էության բոլոր արժեքները պետք է կրկին հայտնվեն կոմբոյում։
* Գրել գոյություն չունեցող արժեք և համոզվել, որ ոչինչ չգտնելու դեպքում combobox-ը փակվում է։
* Բացել կոմբոն, ընտրել որևէ արժեք և համոզվել, որ կոմբոն փակվում է և արժեքն էլ ճիշտ է վերագրվում։
* Սեղմել կոմբոյի եռանկյունի վրա – կոմբոն բացվում է, կրկին սեղմելու դեպքում պետք է փակվի։
* Երբ կոմբոն բաց է, ապա Tab սեղմելիս այն պետք է փակվի (transition state)։
* Բացել կոմբոն և սեղմել կոմբոյից դուրս ինչ-որ մի տեղ․ կոմբոն պետք է փակվի։
* Կոմբոյի փակ վիճակում down սեղմելիս այն պետք է բերի կոմբոյի առաջին արժեքը։
* Սեղմել down, ապա Enter կամ Space. արժեքրը պետք է ճիշտ set լինի։
* Կոմբոյի վրայից սեղմել այլ ստեղներ (F12, ctrl-c և ալյն)․ ոչինչ չպետք է տեղի ունենա։
* Գրել ինչ-որ համապատասխանող բան և սեղմել Escape (Esc). Կոմբոն պետք է փակվի և ոչ մի արժեք չպետք է վերագրվի։ Եվս մեկ անգամ Escape սեղմելիս՝ կոմբոյում լրացրած արժեքն էլ պետք է վերանա։
* Եթե կա scroll, ապա պետք է ստուգել դրա ֆունկցիոնալությունը։

## **6․3 Թեստ քեյսեր Numeric՝ թվային, դաշտերի համար**

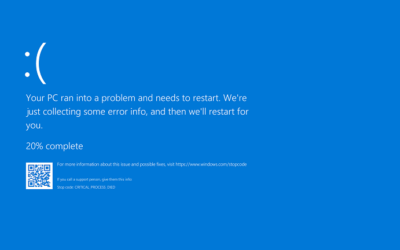
Ենթադրեք, որ ունենք դաշտ, որը կարող է ընդունել միայն թվային արժեքներ․ դե օրինակ գումարը մուտքագրելու դաշտ։ Ի՞նչ կարելի է ստուգել նման դաշտերի համար։

* Մուտքագրել իրական նորմալ արժեք,
* թողնել դատարկ՝ Null,
* մուտքագրել շատ մեծ թիվ՝ 99999999, 1000000 և այլն,
* ամբողջ թվեր՝ 1․2, 0․5, 103․95 և այլն,
* սահմանային արժեքների ստուգում․ եթե ենթադրենք, որ այդ դաշտը կարող է ընդունել միայն 3-7 բնական թվերը, ապա՝
  + սահմանային նվազագույն արժեք՝ 3,
  + սահմանային նվազագույն արժեք «- 1» ՝ 2,
  + սահմանային առավելագույն արժեք՝ 7,
  + սահմանային առավելագույն արժեք «+ 1» ՝7,
  + շատ փոքր արժեք՝ -100,
  + շատ մեծ արժեք՝ 100,
  + միջին արժեք՝ 5
* մուտքագրել 0,
* մուտքագրել 0․0,
* դատարկ տարածք (white space, пробел)՝ “ ”,
* տառ՝ ա, բ և այլն,
* մուտքագրել թիվ, ապա տառ և հակառակը՝ $5.00, 4mm,
* բացասական թվեր,
* դրական թիվ, բայց դիմացը + նշանով՝ +5,
* մուտքագրել 10-ի որևէ աստիճան այլ ձևով՝ 4e-10, -8E5,
* մեկ е ավել՝ 4ee-10,
* մուտքագրել համակարգչի համար շատ մեծ թվեր՝ օրինակ, եթե int տիպի է, ապա ներմուծել 5 միլիարդից մեծ թիվ,
* եթե ամբողջ թիվը բաժանվում է կետիկով, ապա փորձել անել դա ստորակետով,
* մուտքագրել 0-ներ՝ 00000
* Ցանկացած անվավեր արժեք ներմուծել copy/paste հրամանով։

# Բառարան

**1․ Application (App), System - ծրագիր** – համակարգեր են, որոնք նախատեսված են որոշակի գործառույթներ իրականացնելու համար։ Օրինակ` Windows-ը Application է, որը նախատեսված է մարդու կողմից կատարվող հրահանգները պրոցեսորին հասկանալի դարձնելու համար։ Կամ Microsoft Word-ը համակարգ է, որը նախատեսված է տեքստերի խմբագրման համար։

**2. Bug, crash - վրիպակ,** **exception, failure** – Այն ինչ փնտրում են QA-ները։ Ձեզ ամենայն հավանականությամբ կհետաքրքրի, թե ինչպիսի տեսք ունի այն։ Ձեզ ծանոթ կլինեն հետևյալ պատկերները, եթե օգտագործում եք Windows։



Ծրագրի աշխատանքի ընթացքում կտեսնեք չմշակված error-ներ, որոնք ակնհայտորեն կհասկանաք, որ bug-եր են։ Crush ասում են այն դեպքում, երբ ծրագիրը պարզապես փակվում է, կամ, այսպես ասած, «Դուրս է գցում»․ իսկ այդ ամենը տեղի է ունենում, քանի որ ինչ-որ մի տեղ, ինչ-որ պահի, ինչ-որ պատճառով EXCEPTION է ընկնում։

**3․ Data** – տվյալ։ Մենք օրվա մեջ շատ ենք տվյալներ լրացնում ինչպիսիք են անուն, ազգանուն, բնակության հասցե և այլն։

**4․ Function - Ֆունկցիա, գործառույթ** – ցանկացած բան, որը մարդը ստեղծում է, նախատեսված է ինչ-որ նպատակի ծառայելու համար։ Երբ ասում ենք «Իսկ ո՞րն է դրա ֆունկցիան», ապա կարելի է հասկանալ «Ինչի՞ համար է այն նախատեսված», «Ի՞նչ է այն անում», «Ո՞րն է դրա իմաստը»։ Օրինակ` տեսախցիկի ֆունկցիան նկարելն է, Google translate-ինը՝ թարգմանելը և այլն։

Երկորորդ իմաստ՝ **Ֆունկցիա (ծրագրավորում)** – Ծրագրավորումը միլիոնավոր ֆունկցիաների արդյունք է։ Սրանք տարրագույն մասնիկներ են որոնց վրա հիմնվում է ծրագրավորումը։ Դրանք կատարում են փոքրիկ գործառույթներ։

**5․ Valid, Invalid - վավեր, անվավեր** – Ցանկացած դաշտ, որ պարունակում է որևէ վալիդացիա, ապա այդ դաշտը պետք է ունենա valid և invalid արժեքներ։ Օրինակ` ծննդյան տարեթվի համար ցանկացած օր նշելը հաստատ invalid է, եթե այն մեծ է այդ պահին եղած տարեթվից։ Օրինակը ներկայացվում է ծննդյան տարեթվի վրա, բայց ասպեկտները շատ ավելի շատ են։ Օրինակ, երբ ֆիզիկայի դասընթացի ժամանակ ջերմադինամիկայի խնդիր եք լուծում և ինչ-որ նյութի ջերմաստիճանը ստանում եք -471օC, ապա դա միանշանակ և ակնհայտորեն invalid է/անհամապատասխան, քանի որ բացարձակ 0-ն -273,15 օC-ն է և դրանից ավելի ցածր ջերմաստիճան չկա։

**6․ Customer - հաճախորդ** – Սրանք այն մարդիկ են կամ կազմակերպությունները, որոնք դիմում են ծրագրավորման ընկերությանը, որպեսզի վերջիններս ստեղծեն այն ծրագիրը, որը հետագայում Customer-ը կվաճառի, կօգտագործի և այլն։ Օրինակ՝ բեռնափոխադրող ընկերությունը կարող է դիմել ծրագրավորման ընկերությանը, որպեսզի նրանք ստեղծեն մի ծրագիր՝ բեռնատարների ժամանակացույցը, տեղափոխվող ապրանքի տեսակը, քանակը գրանցելու և շատ ու շատ այլ ֆունցկիոնալներ իրականացնելու համար։ Այս պարագայում բեռնափոխադրող ընկերությանը կանվանենք Customer:

**7․ Module - մաս** – Գրեթե բոլոր ծրագրերը բաղկացած են տարբեր մոդուլներից՝ մասերից։ Ամեն մոդուլ պատասխանատու է որոշակի ամբողջական ֆունկցիոնալ ապահովելու համար։ Օրինակ՝ Facebook սոցիալական ցանցի առանձին մոդուլներ կարող ենք համարել՝

1. Անձնական պրոֆիլ (friend, unfriend)
2. Էջ (like, unlike)
3. Խմբեր (join group, leave group)
4. Որոնում
5. Խաղերի բաժին
6. և այլն․․․

**8․ Input եւ Output - Մուտքային եւ ելքային արժեքներ,** – Ինչպես նշվեց, ֆունկցիաները կատարում են որոշակի գործառույթներ, սակայն ինչ-որ բան անելուց առաջ դրանց պետք է տալ նախնական տվյալներ, որոնց հետ այն կկատարի որոշակի գործողություններ և կվերադարձնի ինչ-որ նոր արժեք՝ ելքային տվյալ։ Օրինակը նախ բերենք իրական կյանքից․ երբ ձեզ խնդրում են ջուր բերել, ապա դուք պետք է նախապես իմանաք «Տա՞ք, թե՞ սառը», «250մլ, թե՞ 5 լիտր», «շշո՞վ, թե՞ բաժակով», «ծորակի՞ց, թե ավազանից» և այլն․ այնուհետև դուք գնում եք և կատարում համապատասխան գործողությունը, արդյունքում ձեզ խնդրողին վերադարձնում եք ելքային արժեք՝ «Տաք, 5 լիտր, շշով, ծորակից ջուր»։ Այս նույն սկզբունքը՝ մուտքային և ելքային արժեքների անալիզը, կարող եք պռոեկտել ցանկացած ոլորտի վրա, որովհետև մենք ստեղծվել ենք ինչ-որ ֆունկցիաներ կատարելու համար և անընդհատ տալիս ենք հարցեր ու փորձում գտնել դրանց պատասխանները։ Ծրագարվորման մեջ էլ, երբ կանչենք «գումարել» ֆունկցիան՝ Sum(), ապա պետք է փոխանցենք, թե ի՞նչ գումարի (Ծրագրավորման մեջ տառերն էլ կարելի է գումարել իրար) հետևաբար պետք է փոխանցենք արգումենտնտեր՝ Sum(x1, x2, x3 …., xn), և ակնկալենք արդյունք, պատասխան։ Ֆունկցիան ունի հետևյալ տեսքը

DoSomeFunction(*argument-1,* *argument-2,* …, *argument-n*)

{Do something with arguments: Return *Result*}

**9. PM** – Պռոեկտի մենեջերը, մարդ է, ով կապի մեջ է Customer-ի, Developer-ների, BA-ների, QA-ների և առհասարակ բոլորի հետ։ Նա է հաշվետվություն ներկայացնում, թե ինչ փուլում է գտնվում ծրագրի development-ը, տեղյակ է Customer-ի requirement-ներից, որոշում է, թե որ թասկերն են ավելի կարևոր, կառավարում է ֆինանսական մասը և առհասարակ, ունիկալ մարդ, որը միշտ ամեն ինչից տեղյակ է և կառավարում է գործընթացները։

**10. Developer - ծրագրավորող** – մարդ, ում տալիս են առաջադրանք և նա կատարում է այն։ Հենց նրանց գրած ծրագիրն ենք մենք թեստավորում և փնտրում բազմազան խնդիրներ։

**Development** – Ծրագրը զրոյից ստեղծելու և վերջնանական արդյունքին հասցնելու գործընթացն է։ Ծրագրի զարգացում։

**11. Task, Feature - Առաջադրանք, առանձնահատկություն** – Սրանք չի կարելի շփոթել bug-երի հետ։ Սրանք Customer-ի պահանջները նկարագրող փոքրիկ առաջադրանքներ են, որոնք տալիս են ծրագրավորողներին։ Երբ ծրարգրավորողն ասում է, որ Task-ը վերջացած է, այդ ժամանակ QA-ները թեստավորում են այն և գտնում bug-եր։

**12. Documentation, Requirement** – Երբ Customer-ը դիմում է ծրագրավորման ընկերությանը, ապա իր հետ ունենում է պահանջների ցուցակ, փաստաթղթեր, դոկումենտացիա, որոնք ներկայացնում է ծրագրավորման ընկերությանը և վերջիններս այդ փաստաթղթի հիման վրա սկսում են նախագծել ծրագիրը։ Օրինակ, երբ ինչ-որ մեկը դիմի ձեզ հաշվիչ ստեղծելու համար, ապա դուք բնականաբար կհարցնեք «Իսկ ի՞նչ պետք է այն կարողանա անել»։ Իհարկե պարզ է, որ գումարում և հանում պետք է անի, բայց գուցե հարկավոր է հաշվել ինտեգրալներ, երկրաչափական մակերեսներ և ա՞յլն․․․Այսինքն Customer-ը պետք է հստակ գրի և ներկայացնի, թե ինչ է ցանկանում։ Պետք է ներկայացնի իր պահանջները՝ requirement-ները։

**13․ Review** – բառացի թարգմանած՝ «աչքի տակով անցակցնել»։

**14. QA** – Որակի պատասխանատու (թեստավորող)։ Մարդ, ով, եթե ասաց ծրագիրը լավ վիճակում է, ապա պետք է կարողանա կրել այդ պատասխանատվությունը։

**15. BA** – Բիզնես անալիտիկ։ Մարդ, ով տեղյակ է բիզնեսի ընդհանուր լոգիկայից և մշտական կապի/քննարկումների մեջ է customer-ի հետ, որոշում է, թե ծրագիրն ինչպես ավելի համապատասխան կլինի բիզնես լոգիկային։

**16. Build, Version** – Ամեն անգամ, երբ նոր ֆունկցիոնալ է ավելացվում ծրագրում, ապա ստեղծում են հերթական build, version, որպեսզի այն թեստավորեն։ Սա գրեթե նույնն է ինչ release վերսիա, պարզապես սա ոչ թե վերջնական տարբերակն է, այլ հերթական՝ թեստավորման կամ դեմոնստրացիայի համար նախատեսված։

**17․ Criteria - չափորոշիչ, չափանիշ** – Երբ փնտրում ենք «հակաբիոտիկ» բառը, ապա այդ բառը անվանում ենք կրիտերիա՝ չափորոշիչ։ Կամ երբ ընդունվում եք աշխատանքի, ապա նախօրոք իմանում եք, թե ինչ չափանիշների պետք է համախատասխանեք։

**18. User (յուզեր), օգտատեր** – յուրաքանչյուր ծրագիր ստեղծվում է ինչ-որ կարիքներ հոգալու համար և որպեսզի օգտվեն այդ համակարգից, ապա կարիք կա այնտեղ գրանցվելու։ Օրինակ՝ շատերս user ենք facebook-ում, Amazon-ում, Skype-ում, Viber, Google…Անգամ երբ օգտագործում ենք word, autocad, 1c և այլ desktop application-ներ և գրանցված չենք (քանի որ դրանք երբեմն գրանցում չեն պահանջում), ապա միևնուն է մենք համարվում ենք user։ Մենք բոլորս համարվում ենք բազմաթիվ ծրագրերի End User-ներ։

**19․ Server** – Ծրագրի այս հատվածը ստանում է հրահանգ Client-ի կողմից, կատարում որոշակի գործողություններ և հետ ուղարկում Client-ին։

**20․ Client** – Ժամանակակից ծրագրերն ունեն որոշակի աշխատանքի սկզբունք, որը հիմնված է Client-Server գաղափարի վրա: Client-ը ծրագրի այն հատվածն է, որը չի պարունակում բիզնես լոգիկա, այն միայն մշակում է ստացված հրահանգը և ուղարկում Server: Օրինակ՝ ցանկացած բրաուզեր համարվում է Client և երբ դուք, օրինակ, Search եք անում ինչ-որ բան, ապա բրաուզերը մշակում է ձեր Search-ի տվյալները և ուղարկում Server, որտեղ էլ հենց իրականանում է Search-ը և ստացված արդյունքներն ուղարկվում են Client, որն էլ իր հերթին կրկին մշակում և ձեզ է ներկայացնում ստացված ինֆորմացիան։

**21․ Release, Live, production** – Ձեզ հետ պատահել է չէ՞, որ ձեր հեռախոսը ավտոմատ թարմացում է պահանջել։ Դա նշանակում է, որ օպերացիոն համակարգում որոշակի փոփոխություններ են տեղի ունեցել և այժմ ձեզ տրամադրում են ավելի լավ տարբերակ։ Իրենց տեսանկյունից, ամեն անգամ end user-ին վերջնական թարմացում տալու պրոցեսը անվանում ենք Release, Live, կյանքի կոչվել։

**22․ Interface** – Մի փոքր դժվար բացատրվող երևույթ։ Միանգամից օրինակի վրա փորձենք հասկանալ։ Ենթադրենք, դուք ուզում եք բանկային քարտ բացել ձեր անունով․ բնականաբար պետք է գնաք բանկ, սպասեք անտանելի հերթի մեջ և վերջապես ձեր հերթի հասնելուն պես մոտենաք աշխատողին։ Այդ աշխատողը ձեզ համար ինտերֆեյս է․ դուք ասում եք նրան ձեր անունը, ազգանունը, անձնագրի տվյալները և այլն, և ի վերջո ձեզ տալիս են քարտ։ Դուք չգիտեք, թե նա ինչ է անում այնտեղ, դուք միայն փոխանցում եք տվյալներ և ստանում ձեր ուզած բանկային քարտը։ Այսպիսով, բանկի աշխատողը ձեզ համար դառնում է ինտերֆեյս, որը հնարավորություն է տալիս օգտվել բանկի ֆունկցիաներից՝ ի՛ր միջոցով։ Նունն էլ ծրագրավորման մեջ է․ մենք դիմում ենք տարբեր ծրագրերի, պռոեկտների, սերվիսների և նրանցից ստանում արդունք, անգամ չիմանալով, թե ինչպես է այն աշխատում։

**23․ Graphic User Interface** – Ցանկացած բան, որ ծրագրավորվում է, այն «փաթեթավորվում է» գեղեցիկ դիզայնի մեջ։ Սա այն Ինտերֆեյսն է, որը վերջում տեսնում է օգտատերը։ Օրինակ, երբ տորթ եք գնում, ապա այն դնում են գեղեցիկ փաթեթավորման մեջ։ Կամ, երբ օգտվում եք Microsoft Word ծրագրից, ապա այն ձեզ տալիս է այնպիսի ինտերֆեյս, որ դրանից հարմար լինի օգտվել։

**24․ Database, DB - տվյալների բազա** – Երբ մենք օգտվում ենք հեռախոսից, ապա մեր էկրանին գոնե ինչ-որ նկար ենք տեսնում։ Որպեսզի հեռախոսի օպերացիոն համակարգը (iOS, Android և այլն) կարողանա այդ նկարը ցույց տալ ձեր գլխավոր էկրանին, ապա այն պետք է գոյութուն ունենա ձեր տվյալների բազայում։ Տվյալների բազան այն միջավայրն է, որտեղ պահվում է անհրաժեշտ ինֆորմացիան։ Դա իրականացվում է SQL, OleDB, MongoDB և նմանատիպ այլ ծրագրերի միջոցով։

**25. Behavior - վարքագիծ, պահվածք** – Կան ծրագրեր, որոնք նախատեսված են Ա կետից հասնելու Բ կետ։ Այսինքն մինչ ծրագիրը Ա կետից կհասնի Բ կետ, այն պետք է կատարի որոշակի կանխորոշված քայլեր և պետք է ունենա որոշակի վարքագիծ։ Օրինակ, երբ սեղմենք Save կոճակին և ծրագիրն ինքնիրեն փակվի, ապա մենք կասենք, որ ծրագիրը սխալ վարքագիծ ցուցաբերեց, մինչդեռ այն պետք է բացեր Save-ի համար նախատեսված պատուհանը և սպասեր, որ ֆայլի անունը լրացնեն, ընտրեն, թե որտեղ են այն Save անելու և սեղմեն Save կոճակը՝ արդեն այս վերջին պատուհանից։

**26․ Enable, disable** – տարբեր դաշտեր աշխատանքի ընթացքում ակտիվանում և ապաակտիվանում են՝ կախված ծրագրի աշխատանքի սկզբունքից։ Ստորև ներկայացնում ենք նույն button-ի (կոճակի) enable և disable վիճակները:

|  |  |
| --- | --- |
|  | - enabled |
|  | - disabled |

**27․ Default value** – երբ գրանցում ենք որևէ տվյալ, ապա լինում են դաշտեր, որոնք ունենում են նախնական տվյալներ։ Օրինակ՝ Viber-ում գրանցվելիս, այն ավտոմատ հասկանում է ձեր գեոլոկացիան և միանգամից հեռախոսահամարի դաշտում բերում է +374 լռելայն արժեքը։

**28․ Class** – Ծրագրավորման վոլորտում շատ մեծ հայտնիություն ունեն OOP լեզուները, որոնք աշխատում են class-ների միջոցով։ Այս միջավայրերում կարելի է հայտարարել ֆունկցիա, փոփոխական և այլն․․․

**29․ Null** – Շատ կարևոր է տարբերակել Null արժեքը Դատարկ արժեքից: Ստորև ներկայացված ոչ այնքան գեղեցիկ նկարով շատ գեղեցիկ ներկայացված է Null-ի և տադարկ տողի տարբերությունը․



**30․ Validation** – Վստահաբար կարող եմ ասել, որ վալիդացիաների շատ եք հանդիպել։ Օրինակ կարո՞ղ եք ինչ-որ տեղ գրանցվելիս լրացնել, թե իբր ծնվել եք 3256 թվականին․ իհարկե՝ չե՛ք կարող։ Ծրագրերում սահմանափակումներ են դնում որոշակի դաշտերի վրա, որոնք չեն համապատասխանում բիզնես լոգիկային (կամ էլ առհասարակ լոգիկային, ինչպես այս օրինակում էր բերված)։

**31․ Network - ցանց** - Մենք բոլորս գտնվում ենք ինչ-որ ցանցի՝ միջավայրի, մեջ, որտեղ կարողանում ենք մտքեր փոխանակել՝ խոսելու միջոցով։ Մեր ընտանիքը մի փոքր ցանց է, որին հասանելիություն չունեն այլ ընտանիքներ՝ ցանցեր։ Նույնն էլ համակարգիչների դեպքում է․ դրանք միջավայրեր են, որոնց մեջ համակարգիչները տվյալներ են փոխանցում իրար։ Ամենամեծ ցանցն, իհարկե, World Wide Web- ն է (www), որը մեր հասարակության մեջ կարելի է ձևակերպել որպես սոցիում։

**32․ Path - ճանապարհ** – Համակարգիչները իրար հետ կապվում են որոշակի հասցեներով՝ path-երով։ Օրինակ, երբ ցանկանում ենք հարցնել, թե ո՞ր բազային ենք միանում գործ անելու համար, փոխարենը հարցնում ենք, թե «Ո՞ր path-ով ենք կպնում»։ Ամենապարզ օրինակը, մեր համակարգչով, եթե բացենք պանակ, ապա վերևում կնկատենք Path, կամ էլ ցանկացած վեբ կայք բացելուց դուք URL դաշտում գրում եք path, օրինակ <https://www.javatpoint.com/software-testing-tutorial:>



**33․ Response - արձագանք** – Սա նույնն է, ինչ ասենք «որքա՞ն ժամանակից տվյալները բերեց»։ Օրինակ՝ տարբեր կայքեր մտնելիս արձագանքների արագությունները տարբեր են՝ մեկը արագ է բացում, մյուսը՝ դանդաղ։

**34․ SQL Script, Query** – Եթե ուսումնասիրեք, թե ինչպես է SQL-ն աշխատում, ապա այն հասկանալի կդառնա ձեզ։ Սկրիպտները գրություններ են ինչպես «ֆունկցիաները» և կատարում են գործողություններ բազայի տվյալների հետ։

**35․ Environment - միջավայր** – Բավականին շփոթեցնող տերմին, որ պարզ չի դառնա, մինչև չհայտնվեք աշխատանքային միջավայրում։ Հաճախ ենք ասում «Development Environment», «Testing Environment», «Customer Environment»…Պատկերացրեք այսպես․ ծրագրավորողները աշխատում են անկախ միջավայրում, այնինք իրենց ցանցն ուրիշ է, բազան ուրիշ է, այլ սերվիսներն ուրիշ են, իսկ թեստավորողներինն ընդհանրապես ուրիշ բազա է և այլն․․․

**36․ Log** – Գործողությունների գրանցամատյան։ Սովորոբար ծրագրերը ֆիքսում են ձեր բոլոր կատարած քայլերը և դրանք գրում ինչ-որ տեքստային խմբագրիչի մեջ։ Ծրագրերը պահում են ձեր քայլերը և դրա շնորհիվ է, որ երբ Ctrl+z եք անում այն մեկ քայլ հետ է գցում։

**37․ Path - ճանապարհ** – Համակարգիչները իրար հետ կապվում են որոշակի հասցեներով՝ path-երով։ Օրինակ, երբ ցանկանում ենք հարցնել, թե ո՞ր բազային ենք միանում գործ անելու համար, փոխարենը հարցնում ենք, թե «Ո՞ր path-ով ենք կպնում»։ Ամենապարզ օրինակը, մեր համակարգչով, եթե բացենք պանակ, ապա վերևում կնկատենք Path, կամ էլ ցանկացած վեբ կայք բացելուց դուք URL դաշտում գրում եք path, օրինակ <https://www.javatpoint.com/software-testing-tutorial:>



**38․ Connection** – Կապ։ Երբ օրինակ ասում ենք «Ինտերնետը գնաց», փոխարենը կարող ենք ասել «Կապը խափանվեց» կամ «Connection-ը կորավ»։

**39․ Workflow, diagram** – Ինչպես նշվեց վարքագծի բացատրության մեջ, շատ ծրագրեր ունեն որոշակի քայլերի հերթականություն, որոնք կանխորոշված ենք․ դրանք հենց workflow-ներն են։ Այսինքն, երբ Customer-ը դիմում է ծրագրավորման ընկերությանը, բացատրում է, թե ինչ պահանջներ ունեն, ինչի համար է այն նախատեսվում և ինչպես պետք է աշխատի։ Այ հենց, թե ինչպես պետք է աշխատի, անվանում ենք workflow: Դրանք ավելի հեշտ է ներկայացնել դիագրամների և բլոկ սխեմաների միջոցով։

**40․ VPN** – Virtual Privte Network: Սա գործիք է, որը ստանում է ազդանշան ձեր համակարգչից, սակայն մինչ վերջնական սերվերին call անելը այն փոփոխում է ձեր համակարգչի կամ ցանցի հասցեն՝ call անելով այլ միջանկյալ սերվերի։

# Գրականության ցանկ

1. ***The Art of Software Testing*** *- Glenford J. Myers, Corey Sandler, Tom Badgett*
2. ***Introduction to Software Testing*** *- Paul Ammann, Jeff Offutt*
3. ***Fuzzing for Software Security Testing and Quality Assurance*** *- Ari Takanen, Jared D. DeMott, Charles Miller*
4. ***Software Testing: Concepts and Operations*** *- Ali Mili, Fairouz Tchier*
5. ***Software Testing Techniques*** *- Boris Beizer*
6. ***The Art of Software Testing*** *- Glenford J. Myers, Tom Badgett, Todd M. Thomas, Corey Sandler*
7. ***Effective Methods for Software Testing*** *- William E. Perry*
8. ***Software Testing as a Service*** *- Ashfaque Ahmed*
9. ***Testing and quality assurance for component*** *- based software - Jerry Zeyu Gao, H.-S. Jacob Tsao, Ye Wu*
10. ***Managing the Testing Process: Practical Tools and Techniques for Managing Hardware and Software Testing*** *- Rex Black*
11. ***Advanced Software Testing: Guide to the ISTQB Advanced Certification as an Advanced Test Analyst*** *- Rex Black*
12. ***Analytic methods in systems and software testing*** *- Faltin, Frederick W., Kenett, Ron, Ruggeri, Fabrizio*
13. <https://www.guru99.com/>
14. <https://www.softwaretestingmaterial.com/>
15. <https://www.javatpoint.com/>
16. <https://www.softwaretestinghelp.com/>
17. <https://www.tutorialspoint.com/>
18. <https://philosophy-question.com/>

1. Վիլֆրեդո Ֆեդերիկո Դամասո Պարետո (1848 - 1923)՝ իտալացի ինժեներ, տնտեսագետ և սոցիոլոգ։ Էլիտայի տեսության հիմնադիրներից մեկն է։ [↑](#footnote-ref-1)