ივანე ჯავახიშვიილს სახელობის თბილისის სახელმწიფო

უნივერსიტეტი ზუსტ და საბუნებისმეტყველო

მეცნიერებათა ფაკულტეტი



„Unit Testing“

ხელმძღვანეილ : მანანა ხაჩიძე

სტუდენტი : მანველ კლოიანი

2018

თბილისი

**რეზიუმე**

ნებისმიერი პროგრამული უზრუნველყოფა საჭიროებს გამართულად მუშობაზე ტესტირებას. ჩვენ ვტესტავთ პროგრამულ უზრუნველყოფას რისკების მართვისთვის. რისკი არის პოტენციური პრობლემა, რომელიც შეიძლება წარმოიქმნას პროგრამის არასწორად მუშობის შემთხვევაში. რაც უფრო დაბალია რისკი, მით უფრო ნაკლებია პოტენციური პრობლემების რაოდენობა. სხვა სიტყვებით რომ ვთქვათ, ნაკლები რაოდენობის ბაგი პროგრამულ უზრუნველყოფაში ნიშნავს ნაკლებ პრობლემას. პროგრამული უზრუნველყოფის ტესტირების პროცესი შეიძლება იყოს ხანგრძლივი, რთული, არაეფექტური და ძვირადღირებული. სისტემაში შესრულებული პატარა ცვლილების გატესტვისთვის ტესტერს შეიძლება დასჭირდეს ბევრი დრო და მაინც გამორჩეს რაღაც ქეისები. კოდის ერთი მინიმაულური ერთეულის შემოწმებისათვის გამოიყენება Unit Test. Unit ტესტირების მიზანია API - ის ყველაზე მცირე ტესტირებადი ნაწილების იზოლირება და მათი კორექტულად მუშაობის შემოწმება იზოლაციაში.1 პროგრამული უზრინველყოფის unit test - ის მიზანია რისკის მართვა კოდის ყველაზე პატარა ნაწილებში. ყველაზე პატარა ნაწილებში იგულისხმება ობიექტი, ობიექტის public მეთოდები და გლობალური მეთოდები.

**Abstract**

Any software needs to be properly tested. We test software to manage risk. Risk is the potential for a problem to be realized. A lower level of risk implies fewer problems. In other words less number of bugs means less potential problems. Software testing may be often a long, difficult, inefficient, and expensive process. A small change in a system can afford a QA tester to spend a long time for testing and yet some cases can be missed. A unit test is used to verify a single minimal unit of source code. The purpose of unit testing is to isolate the smallest testable parts of an API and verify that they function properly in isolation.1 The goal of a software unit test is to manage risk at the smallest unit possible. The target sizes to consider for a unit of code are single objects, their public member functions and global functions.

**გასაღები სიტვვები -** კოდის მინიმალური ერთეულების ტესტირება, პროგრამული უზრუნველყოფის ტესტირება

**Keywords** —. unittesting, software testing

2017-2018 სასწ.წელი თსუ კომპიუტერულ მეცნიერებათა დეპარტამენტი

—————————— ◆ ——————————

**1 შესავალი**

Unit Test - ი არ არის ახალი კონცეფცია პროგრამირებაში. ის არსებოს 1970-იანებიდან და წლების განმავლობაში ითვლება კოდის ხარისხის გაუმჯობესების ერთერთ საუკეთესო გზად. Unit Test არის კოდის ნაწილი, რომელიც იძახებს კოდის სხვა ნაწილს და ამოწმებს მის სწორად ფუნქციონირებას.

სტილისტურად unit ტესტები შეიძლება დაიწეროს სხვადასხვანაირად, მაგრამ ყველა unit ტესტს ახასიათებს შემდეგი თვისებები:

* უნდა იყოს ავტომატური და განმეორებადი
* უნდა იყოს მარტივად შესრულებადი
* უნდა იყოს რელევანტური ხვალაც
* ნებიმსიერს უნდა შეეძლოს მისი გაშვება ღილაკზე დაწკაპუნებით
* უნდა იყოს სწრაფი
* უნდა იყოს პროგნოზირებადი, უნდა აბრუნებდეს ერთიდაიგივე შედეგს თუ არაფერი შეცვლილა გაშვევებს შორის
* უნდა ჰქიონდეს კოდის ერთეულის სრული კონტრლი ტესტის შესრულებისას
* უნდა იყოს სრულად იზოლირებული (ანუ სხვა ტესტებისგან დამოუკიდებელი)
* ტესტის ვერ გასვლის შემთხვევაში უნდა იყოს შესაძლებელი მარტივად დადგინდეს მოლოდინი და პრობლემა

Test unit - ბის უმეტესობის შექმნის მარტივი შაბლონი:

* ტესტის set up
* ტესტის შესრულება
* შედეგის შემოწმება
* საჭიროების შემთხვევაში გარემოს რესეტი

მანუალური ტესტირება არის შრომატევადი პროცესი და ხშირია შეცდომების დაშვების ალბათობა. გარდა მაგისა, პროგრამისტები ვერ იტანენ მაგ პროცესს. სხვადასხვა Tools - ების გამოყენება უადვილებს პროგრამისტებს ტესტირების პროცესს. Unit testing ფრეიმვორკები საშუალებას აძლევენ პროგრამისტებს დაწერონ ტესტები უფრო სწრაფად, შეასრულონ ტესტები ავტომატურად და ნახონ ტესტირების შედეგები.

სტატია შედგება შემდეგი ნაწილებისგან:

* არსებული მდგომარეობა
* სისტენური მოდელი
* ამოცანის დასმა
* პრობლემის გადაწყვეტა
* ანალიზი
* დასკვნა

# 2 განხორციელებული სამუშო (არსებული მდგომარეობა)

კარგად დაწერილი კოდი ცდილობს შეამციროს დამოკიდებულებები პროგრამული უუზრუნველყოფის კომპონენტებს შორის. მაგრამ შეუძლებელია კოდის ისე დაწერა, რომ ყველა კომპონენტი იყოს ერთმანეთისგან დამოუკიდებელი. კოდში ხშირად გვიწევს ბაზის, ვებ სერვისების, ფაილური სისტემის და სხვა რესურსების გამოყენება. ასეთ შემთხვევებში შეგვიძლია გამოვიყენოთ Mock**,** ანუ იმიტირებული ობიექტები.

არსებობს სხვადასხვა ტიპის იმიტირებული ობიექტები:

* Dummy ობიექტები წარმოადგენენ მარტივ mock - ებს, რომლებიც გამოიყენებიან გარე რესურსის ნაცვლად. ასეთი ობიექტები როგორც წესი აბრუნებენ წინასწარგანსაზღვრულ შედეგს შესაბამისი მეთოდის გამოძახებისას და მეთოდში გადაცემული პარამეტრები შედეგზე არავითარ გავლენას არ ახდენენ .
* Fake - ები და stub - ები Dummy - სგან განსხვავებით პასუხს აბრუნებენ შემავალი პარამეტრების მიხედვით. Stub - ს ძირითადად არ შეუძლია განსაზღვროს თუ რამდენჯერ იყო გამოძახებული მეთოდი ან რა მიმდევრობით იყო გამოძახებული მეთოდები.
* Mock - ები უზრუნველყოფენ იგივე ფუნქციონალს, რაც fake -ები და stub - ები, მაგრამ უფრო კომპლექსურს.

გარე რესურსების არსებობა საჭიროა აპლიკაციებში, მაგრამ ასევე საჭიროა გვქონდეს საშუალება დავტესტოთ კომპონენტები ამ რესურსებისგან იზოლირებულად. დეველოპერები აღწევენ მაგ მიზანს გარე რესურსების იმიტირებით, ანუ *mock - ებით*.

Mock - ების გამოყენებისთვის და იმისთვის, რომ ნამვილად მოვახდინოთ ინდივიდუალური კომპონენტების იზოლირება ტესტებში, საჭიროა ალტერნატივა ინდივუდუალური კომპონენტების სტატიკურ binding - ებს. Dependency injection პატერნი შეიძლება გამოყენებული იქნას ამ კომპონენტების კონკრეტული იმპლემენტაციების inject -ისთვის runtime - ის დროს ან ტესტირების დროს. Dependency injection ფრეიმვორკები გვეხმარებიან ვმართოთ დამოკიდებულებების ქსელს და უზრუნველყოფენ, რომ სწორი კონკრეტული იმპლემენტაცია იყოს უზრუნველყოფილი გამოყენებული კომონენტის კონტექსტის საფუძველზე.

პროგრამული უზრუნველყოფის ტესტირებისთვის არსებობს სხვადასხვა tool - ები:

* Unit - testing ფრეიმვორკები
* Mocking ფრეიმვორკები
* Dependency injection ფრეიმვორკები

Unit ტესტირებისთვის არსებობს რამდენიმე ფრეიმვორკი. ერთობლიობაში, ამ ფრეიმვორკებს xUnit ფრეიმვორკებს ეძახიან, იმიტომ რომ მათი დასახელება ხშირად იწყება იმ პროგრამირების ენის პირველი ასოთი, რომლისთვისაც შეიქმნა ფრეიმვორკი, მაგ. CppUnit C++ - ისთვის, Junit Java - სთვის, NUnit .NET - ისთვის და HUnit Haskell - სთვის.

პროექტის ფარგლებში გამოყენებული იქნა NUnit ფრეიმვორკი. NUnit არის ერთ-ერთი ყველაზე პოპულარული .NET unit test ფრეიმვორკი. NUnit დაფუძნებულია JUnit - ზე, Java-ს unit testing ინსტრუმენტზე. .NET unit testing ფრეიმვორკი აადვილებს ტესტის დაწერას, მათ გაშვებასა და შედეგების მიღებას.

NUnit - ის ატრიბუტები:

* [TestFixture] – ატრიბუტი მიუთითებს, რომ კლასი არის ტესტ კლასი
* [Test] – ატრიბუტი მიუთითებს იმაზე, რომ მეთოდი არის ტესტ მეთოდი
* [SetUp] - ატრიბუტი გამოიყენება იმ ქმედებების შესასრულებლად, რომლებიც ყველა სხვა მეთოდამდე უნდა გაეშვას
* [TearDown] - ზოგჯერ საჭიროა რესურსების რესეტი ტესტის დასრულების შემდეგ. ეს ატრიბუტი განკუთვნილია ამ მიზნისთვის.

Assert - ები:

Assert – test runner აპლიკაციას „ეუბნება“ ტესტის საბოლოო შედეგს. NUnit ბიბლიოთეკაში არსებული ზოგი assert - ი:

* Assert.AreEqual(expected, actual)
* Assert.AreNotEqual(expected, actual)
* Assert.AreSame(expected, actual)
* Assert.AreNotSame(expected, actual)
* Assert.IsTrue(bool) / Assert.IsFalse(bool)
* Assert.IsNull(object) / Assert.IsNotNull(object)

Assert.Greater(x,y) / Assert.GreaterOrEqual(x,y)

# 3 სისტემური მოდელი

არსებობს პროგრამული უზრუნველყოფის ტესტირების სხვადასხვა დონეები:

* Unit ტესტირება — ცალკეული კლასის ან ფუნციის ტესტირება. ხშირად ასეთ ტესტირებას ასრულებენ ტესტერები.
* Integration ტესტირება — ამ ტიპის ტესტირების მიზანია შემოწმდეს სხვადასხვა კომპონენტების ერთობლიობაში მუშობის კორექტულობა
* სისტემური ტესტირება — ინტეგრირებული სისტების ტესტირება პროგრამული უზრუნველყოფის სისტემური მოთხოვნილებებზე
  + ალფა-ტესტირება — სისტემასთან რეალური მუშაობის იმიტაცია. უმეტესად ალფა-ტესტირება ტარდება პროგრამის შექმნის ადრეულ სტადიებში, თუმცა ზოგჯერ შეიძლება გამოიყენებოდეს დასრულებულ პროდუქტთან.
  + ბეტა-ტესტირება — ზოგიერთ შემთხვევაში ხდება პროდუქტის შეზღუდული/დაუმთავრებელი ვერსიის (ფუნქციურად) გავრცელება/გაყიდვა, იმის დასარწმუნებლად, რომ პროდუქტი შეიცავს ცოტა დეფექტს/შეცდომას. ზოგჯერ კი მომავალი მომხმარებლებისგან შეფასების მიღება.

# 4 ამოცანის დასმა

პროგრამული უზრუნველყოფის მუშაობის კორექტულობაზე შემოწმება წარმოადგენს პროგრამული უზრუნველყოფის დამუშავების მნიშვნელოვან ასპექტს. პროგრამისტს უნდა შეამოწმოს რა დაწერა, ასრულებს თუ არა კოდის ნაწილი იმას, რაც უნდა ასრულებდეს. კომპიუტერი ყოველთვის ასრულებს იმას, რაც იყო დაპროგრამებული, მაგრამ დაპროგრამებული კოდი ნამდვილად იყო ის, რაც უნდოდა პროგრამისტს? პროგრამისტს უნდა დარწმუნდეს, რომ პროგრამული უზრუნველოფა მუშობს კორექტულად.

# 5 პრობლემის გადაწყვეტა

როგორც უკვე ითქვა ჩვენ ვტესტავთ პროგრამულ უზრუნველყოფას პოტენციური რისკების შესამცირებლად. პროგრამული უზრუნველყოფის გატესტვის პროცესი შეიძლება იყოს ხანგრძლივი, ძვირადღირებული და არაეფექტური. მნიშვნელოვანია, რომ პროგრამული უზრუნველყოფის გატესტვა მოხდეს მისი შემუშვების პროცესში. ანუ პროგრამისტი უნდა იყოს საკუთარი პროგრამული უზრუნველყოფის პირველი ტესტერი. ეს საგრძნობლად შეამცირებს პოტენციური პრობლემების რაოდენობას.

არსებობს ტესტირების სხვადასხვა ფორმა. Unit ტესტირების ფაზა ძალიან გავს ტელევიზიის ხარისხის კონტროლის შემოწმებას. ობიექტები და ფუნქციები, რომლებიც შექმნილია პროგრამისტის მიერ მოწმდება იზოლაციაში იმაზე, რომ ისინი ასრულებენ იმას, რაც მათ მოეთხოვება. Unit ტესტები უნდა ამოწმებდნენ ტესტირებადი კოდის ყველაზე პატარა შესაძლო ნაწილებს.

Unit ტესტები პროგრამისტისთვისაა. Unit ტესტები ამოწმებენ პროგრამული უზრუნველყოფის ბლოკებს, რომლებსაც პროგრამისტები გამოიყენებენ იმისთვის, რომ შექმნან უფრო კომპლექსური კომპონენტები, და შემდგომში ეს კომპონენტები “შეფუთონ” საბოლოო სისტემაში ან აპლიკაციაში.

Unit ტესტები შეიძლება ორგანიზებული იყოს როგორც დამუშვების პროცესის ავტომატური ნაწილები. პროგრამული უზრუნველყოფის დამუშავების ნებისმიერ ეტაპზე სხვა რეგრეს და შემოწმებით ტესტებთან ერთად გაიშვება unit ტესტებიც. Unit ტესტები უფრო მომგებიანი ნაწილია პროგრამისტებისთვის, იმიტომ რომ ისინი ამოწმებენ ლოგიკის ნაწილებს იზოლირებაში ინტეგრაციამდე.

# 6 ანალიზი

Unit ტესტების გამოყენება მომგებიანია არა მარტო იმ კუთხით, რომ მოწმდება კოდის ცალკეული ნაწილები. მათი გამოყენება განაპირობებს სუფთა კოდის დაწერას და პროგრამირებაში მიღებული SOLID პრინციპების დაცვასაც.

SOLID პრინციპები დეველოპერებისთვის გახდა რუკასავით ახალი აპლიკაციების დაწერის დროს და არსებული კოდის რეფაქტორინგისას. ეს პრინციპებია:

* The Single Responsibility Principle - ყოველი ობიექტი უნდა იყოს პასუხისმგებელი მარტო ერთი რაღაცის შესრულებაზე
* The Open/Close Principle - ობიექტი უნდა იყოს ღია გაფართოებისათვის და დახურული მოდიფიცირებისთვის
* The Liskov Substitution Principle – ეს პრინციპი ასევე ცნობილია design by contract სახელით.
* The Interface Segregation Principle - კლიენტებისთვის შექმნილი ბევრი ინტერფეისი უკეთესია, ვიდრე ზოგადი მოხმარებისთვის შექმნილი ერთი ინტერფეისი
* The Dependency Inversion Principle - დამოკიდებულება აბსტრაქციებზე. არ არსებობს დამოკიდებულება კონკრეტულ ობიექტზე.

Unit ტესტი არის ძალიან ღირებული აქტივი, თითმის იგივე ღირებულობის როგორიცაა კოდი, რომელსაც ის ამოწმებს. აუცილებელია მისი მართვა ისე, როგორც კოდი, რომელიც საჭიროა პროდუქტის კომპილაციისთვის. Unit ტესტი ხდება კოდის ავტომატური ნაწილი და მისმა წაშლამ შეიძლება გამოიწვიოს წინა ბაგების „დაბრუნება“.

# დასკვნა

Unit testing არის ფართო თემა. ხარისხის კონტროლი უნდა არსებობდების პროგრამული უზრუნველყოფის დამუშავების მრავალ დონეზე. ამასთან ერთად მნიშვნელოვანია, რომ ყველა ვინც არის ჩართული სისტემის შექმნაში ხვდებოდეს, რომ unit ტესტები არასაკმარისია საბოლოო პროდუქტის კორექტულად მუშობაზე შემოწმებისათვის. Unit ტესტების საკმარისმა რაოდენობამ შეიძლება შეამციროს საბოლოო პროდუქტის ტესტირებისთვის საჭირო დრო.

პროგრამული უზრუნველყოფის unit ტესტები უზრუნველყოფენ მყარ საფუძველს პროდუქტის დანარჩენი ნაწილების შემუშავებისთვის. Unit ტესტი არის პატარა, ამოწმებს კონკრეტულ პატარა ლოგიკას იზოლაციაში და დაწერილია პროგრამისტის მიერ. Unit ტესტი შეიძლება იყოს დამუშავების პროცესის ავტომატური ნაწილი და პროდუქტის რეგრეს ტესტების პირველი სიმრავლე.

**ლიტერატურა**

1. API Design for C++, p295; Martin Reddy

[2] Art Of Unit Testing, 2nd edition, Osherove

[3] Professional Test-Driven Development with C#, James Bender and Jeff McWherter

[4] <https://www.codeproject.com/Articles/727053/The-Purpose-of-a-Unit-Test>

[5] <https://en.wikipedia.org/wiki/Unit_testing>

[6] <https://en.wikipedia.org/wiki/Software_testing>