Testing your code

Taakgroep: Testing

Context

Elk project valt of staat met voldoende qualiteit. Niemand staat te wachten op software met vervelende bugs of zelfs crashes. Tijdens het eerste schooljaar heb je tijdens de KBS `functioneel getest`, of je bent erop gewezen dat je dit te weinig hebt gedaan. In deze level ga je aan de slag met `niet-functionele testen` door middel van `unit testing`.

De kwaliteit van je software gaat sterk omhoog wanneer je vooraf of direct achteraf je unit test schrijft. Dit zorgt ervoor dat je tijdens de ontwikkeling al moet nadenken over de testbaarheid van je werk, bijvoorbeeld door het gebruiken van SOLID principes.

Om je te helpen bij het testen heeft Visual Studio standaard een aantal hulpmiddelen. Testen doe je eigenlijk al tijdens het programmeren door gebruik te maken van `debugging`. Je kan hiermee "live" meekijken wat je code precies doet en welke resultaten je krijgt.

Nadat je klaar bent met programmeren helpt Visual Studio met het schrijven van `unit tests`.

Tijdens deze level ga je ook veel gebruik maken van de `debugger` en andere hulpmiddelen om de kwaliteit van je werk te verhogen.

In de voorgaande levels heb je veelal gewerkt aan je model. Er is dus nog weinig functionaliteit gemaakt waardoor er nog niet veel te testen is. Je gaat in dit level dan ook afwisselend werken aan je programma om vervolgens de unit tests uit te breiden.

Taak: Eerste test schrijven

Aanpak

In het vorige level heb je een methode geschreven `NextTrack`. Hiervoor ga je een aantal unit tests schrijven. Let op de naamgeving van je tests!

- In je test project verwijder `UnitTest1.cs`
- Voeg aan het test project een nieuwe klasse toe met de naam `Model_Competition_NextTrackShould`

  Zoals je kan aflezen aan de naam van de klasse gaat het om de methode `NextTrack` uit de klasse `Competition` uit het `Model` project. Dit maakt het veel eenvoudiger om een falende test te vinden en op te lossen.

- Voeg direct boven de `public class Model_Competition_NextTrackShould` het volgende `attribute` toe: `[TestFixture]`. Voeg de juiste `using` toe wanneer deze `attribute` rood onderstreept is.

  De eerste stap van elke test is het opzetten ervan.

- Geef de klasse `Model_Competition_NextTrackShould` een private variable `_competition` van het type `Competition`.
- Maak een public methode `SetUp` aan en geef deze de attribute `[SetUp]`.
- Initializeer `_competition` in de methode `SetUp`

  De volgende stap is het schrijven van de daadwerkelijke test. Doorloop hiervoor de onderstaande stappen.

- Maak de public methode `NextTrack_EmptyQueue_ReturnNull`.
- Geef de methode de attribute `[Test]`
- Roep de methode `_competition.NextTrack()` aan en plaats het resultaat in de variable `result`
- Test of de variable `result null` is, zoals de bedoeling is volgens de 1e functionaliteit. Voeg hiervoor `Assert.IsNull(result)` toe aan de methode.
- Run nu alle tests door bovenin het menu te kiezen voor `Test -> Run All Tests`
- Open nu de `Text Explorer` om te kijken of je test geslaagd is.

  Als je alles goed hebt gedaan slaagt je test. Zo niet los eerst het probleem op voordat je doorgaat naar de volgende taak.

Ondersteunende informatie

Testen kent vele vormen en strategieën om zo veel aspecten van de software te kunnen testen. Wikipedia heeft een overzicht over een aantal hiervan.

Handige informatie over `debugging` kan je hier vinden.

Wil je meer weten over `unit tests` lees dan dit.

Uitleg over het gebruik van `NUnit` kan je hier lezen.

`Attributes` worden veel gebruikt. Je kan zelfs je eigen `attributes` schrijven. Wil je meer weten hierover kijk dan hier

Wil je meer weten over hoe je testen kan uitvoeren en het resultaat kan bekijken open dan deze pagina

Taak: Eerste test schrijven

Bij de vorige taak heb je code geschreven om ene test te initializeren en uit te voeren. Maar wat heb je eigenlijk getest? Heb je nu al je werk afgedekt met je test? Om hierin inzicht te krijgen biedt Visual Studio ondersteuning. In deze taak gaan we gebruik maken van de tooling in Visual Studio om meer inzicht te krijgen over de `code coverage` van je test.

Aanvulling: Het kan zijn dat jouw versie van Visual Studio een aantal tools niet heeft. Gelukkig zijn hier prima alternatieven voor. In het geval van de `code coverage` tools heeft JetBrains hier dotCover voor gemaakt. Hiervan kan je, als student, gratis gebruik van maken. Hier kan je een gratis student JetBrains account aanmaken. Gebruik voor het aanmelden je emailadres van Windesheim!

Aanpak

- Bovenin het Visual Studio menu kies `Test -> Analyze Code Coverage for All Tests`. In het geval dat je gekozen hebt voor JetBrains ga je naar `Extentions -> ReSharper -> Unit Tests -> Unit Test Coverage`

  Tip: Het kan zijn dat Visual Studio blijft hangen op de melding `Waiting for IntelliSense to finish initializing...`. Mocht dit bij jou ook het geval zijn dan kan je hier de oplossing vinden.

- Bekijk het resultaat in het `Code Coverage Results` scherm.

  Wanneer je het resultaat bekijkt bij `Not Covered (% Blocks)` dan zie je een schrikbarend hoog percentage van meer dan 90%! Dit betekent dat bijna al jouw werk ongetest is.

- Dubbelklik nu op de eerste (en waarschijnlijk enige) item. Hiermee krijg je meer inzicht over de `code coverage` van je test.

  Zoals je waarschijnlijk wel door hebt is het een soort `file explorer` achtige weergave.

- Klik de onderstaande items open `model.dll -> Model -> Competition`

  Als je aan het UML diagram hebt gehouden krijg je nu een lijst te zien waar ook de methode `NextTrack` staat. Zoals je kan zien is de `code coverage` van deze methode een stuk beter, waarschijnlijk rond de 30% (afhankelijk hoe jij de functie hebt geschreven).

- Dubbelklik nu op de methode `NextTrack` in dat lijstje

  Visual Studio opent nu `Competition.cs` met de focus op `NextTrack`. Wat meteen opvalt zijn de stukken code die rood gekleurd zijn. Dit betekent dat dit stukken code zijn die je nu nog niet test. In het geval dat je gebruik maakt van dotCover van JetBrains kan je aan de kantlijn zien welke code nog niet getest worden.

Ondersteunende informatie

Informatie over `code coverage` kan je hier lezen.

Taak: Meer testen

Je hebt je eerst test geschreven om vervolgens inzicht te krijgen wat je eigenlijk nog allemaal moet testen. Uiteraard sta je te popelen om alle code te testen om zo een 100% code coverage te bereiken. De valkuil is dat 100% eigenlijk moeilijk te bereiken is, zeker wanneer het project groter en complexer wordt. Voorafgaand aan een project spreek je met de opdrachtgever vaak een minimale percentage `code coverage` af dat je minimaal moet bereiken met unit tests. Na elke oplevering rapporteer je, in een test rapport, hierover eventueel aangevuld met andere (handmatige) tests om zo toch (bijna) tot die 100% code coverage te komen.

In deze taak gaan we de methode `NextTrack` verder testen. In de vorige test heb je getest of de methode `null` retourneerd wanneer de queue leeg is. Nu ga je testen dat de methode een `Track` retourneerd wanneer de queue niet leeg is.

Aanpak

- In de test klasse `Model_Competition_NextTrackShould` maak een test methode aan met de naam `NextTrack_OneInQueue_ReturnTrack`
- Maak in de methode een `Track` en voeg deze toe aan de queue van `_competition`
- Roep de methode `_competition.NextTrack()` aan en plaats het resultaat in de variable `result`
- Gebruik `Assert.AreEqual` om de `Track` die je op de queue hebt geplaatst te vergelijken met het resultaat in `result`.
- Run alle tests en bekijk of alle testen slagen. Zo niet, los het probleem op.
- Voer weer een `code coverage` analyse uit en bekijk hoeveel `% Not Covered` de methode `NextTrack` heeft.

  Als het goed is heb je nu (bijna) 100% code coverage behaald voor de methode `NextTrack`.

Ondersteunende informatie

Een overzicht van alle opties van `Assert` is hier te vinden.

Taak: Eerste test schrijven

Je hebt 100% code coverage behaald bij de methode `NextTrack`. Wat moet dat een goed gevoel geven, een foutloze methode! Toch vertelt het percentage code coverage maar een deel van het verhaal. Functioneel moet de methode `NextTrack` het volgende kunnen doen:

1. Retourneer `null` wanneer de queue leeg is
2. Retourneer de eerst volgende `Track` van de queue
3. Haal de `Track` van de queue wanneer deze geretourneerd wordt

In de 2 testen die je hebt geschreven heb je de bovenste twee functionaliteiten (deels) getest. Bij punt 2 staat `de eerst volgende Track`, dit heb je nog niet getest met de huidige 2 testen. De functionaliteit bij punt 3 heb je al helemaal niet nog niet getest.

In deze taak gaan we de testen verder uitbreiden om zo alle functionaliteiten te testen.

Aanpak

Je gaat een test methode schrijven waarbij de je een `Track` op de queue plaatst en tweemaal de methode `NextTrack` uitvoerd. Wanneer de methode `NextTrack` bij de tweede aanroep een `null` retourneerd kan je vaststellen dat de `Track` van de queue wordt gehaald.

- Schrijf een test methode `NextTrack_OneInQueue_RemoveTrackFromQueue`
- Voeg een `Track` toe aan de queue
- Roep de methode `_competition.NextTrack()` aan en plaats het resultaat in de variable `result`
- Roep de methode `_competition.NextTrack()` nogmaals aan en plaats het resultaat in de variable `result`
- Gebruik `Assert` om te testen of de variable `result` de waarde `null` heeft
- Voer alle tests uit. Zorg ervoor dat alle testen slagen

Nu heb je de 3e functionaliteit getest. Wat nog resteerd is vaststellen of de `Tracks` op de juiste volgorde geroutneerd door de methode `NextTrack`

- Schrijf een test methode `NextTrack_TwoInQueue_ReturnNextTrack`
- Plaats twee verschillende tracks op de queue
- Implementeer nu zelf de rest van de test waarmee je kan vaststellen dat de methode `NextTrack` in de juiste volgorde de tracks retourneerd
- Voer nu alle tests uit, zorg ervoor dat alle tests slagen

Tijdens deze level heb je stap voor stap geleerd hoe je moet vaststellen wat je moet testen en welke hulpmiddelen je hiervoor kunt gebruiken. Deze tests heb je vervolgens geïmplementeerd.

Vanaf nu is het aan jou om zelf tests te bedenken en te schrijven wanneer je klaar bent met een taak. Let op de naamgeving van je test klasse en methoden! Spreek voor jezelf een acceptabele code coverage percentage af en communiceer deze percentage deze met je docent tijdens het aftekenen.