Deel 3: De aanpak voor testautomatisering

In de vorige blog hebben we stil gestaan bij het feit dat testautomatisering gewoon testen is en dat dezelfde zaken belangrijk zijn. Verder hebben we de testpiramide en de verschillende lagen in de piramide bekeken.

In deze blog staan we eerst nog even stil bij de testpiramide en een potentiële valkuil van een veel gebruikte metriek voor unit en component testen. Daarna gaan we dieper in op de aanpak voor testautomatisering in de verschillende lagen van de testpiramide en wie het werk uitvoert.

De 100% code coverage illusie

Bij het implementeren van testautomatisering in het algemeen maar bij het implementeren van unittests in het bijzonder moet je oppassen dat metrieken je niet op het verkeerde been zetten.

Er zijn veel tools, zowel geïntegreerd in de software Integrated Development Environments (IDEs), als ook losse software als Sonarqube, die metrieken leveren over de testdekking van unittests. Deze tools drukken testdekking uit in een %. Vaak meten ze meerdere typen dekking als code coverage, branch coverage, line coverage etc. 100% code coverage lijkt heel mooi, het zegt immers dat alle code wordt geraakt bij het testen. Als je hier iets langer over nadenkt zegt dit echter niets over wat er daadwerkelijk wordt getest! Het is zelfs mogelijk 100% code coverage te bereiken zonder ook maar 1 echte test (assert) uit te voeren. Een test is namelijk alleen maar een echte test als er ook een vergelijking plaatsvindt met een verwacht eindresultaat. Het is dus verstandig om in detail (code review!) naar de geïmplementeerde tests kijken om vast te stellen of er een kwalitatief goede testdekking is geïmplementeerd (De vraag WAT wordt er getest moet duidelijk te beantwoorden zijn). Tools als Sonarqube houden rekening met dit risico en tellen alleen tests met minimaal één assert mee in de code coverage metriek.

De code coverage metriek is ook heel goed andersom te gebruiken. Van een stuk code met beperkt of niet aanwezige unittest coverage weet je zeker dat het niet getest wordt. Je moet je dan afvragen of dit te verantwoorden is kijkend naar het risicoprofiel van het stuk code in het totale product.

De aanpak voor testautomatisering verschilt per laag

We hebben gezien dat testautomatisering op elk testniveau kan worden toegepast. Het is belangrijk om even stil te staan bij de verschillen tussen testautomatisering op de verschillende testniveaus.

IDEs maken het mogelijk om unittests op de lokale ontwikkelomgeving te draaien. Unittests zijn ook software en hetzelfde geldt voor test doubles die nodig zijn om stukken software in isolatie te kunnen testen.

De API-tests en E2E tests worden op de gebouwde, draaiende code uitgevoerd in een testomgeving. Dit kan dezelfde testomgeving zijn als waarin de handmatige testen plaatsvinden maar het is beter om de geautomatiseerde tests in een eigen testomgeving te draaien, zie sectie “Eisen aan de testomgeving en testdata”

In de literatuur en op internet zie je vaak de term BDD test framework. BDD (Behavior-Driven-Development) is een ontwikkelmethodiek waarbij het gedrag van software wordt vastgelegd aan de hand van het opstellen van concrete voorbeelden. Deze concrete voorbeelden worden vervolgens opgeschreven in een voor iedereen begrijpelijke syntax, meestal Gherkin. Met Gherkin kun je gewenst gedrag in een eenvoudige Given-When-Then syntax vastleggen. Hiermee specificeer je WAT het product precies moet doen onder bepaalde condities. Dit sluit naadloos aan bij testautomatisering op het E2E-niveau. De Gherkin syntax is echter niet gebonden aan alleen het E2E-niveau. Op elk testniveau, dus ook voor een unittest en API-test kan het gedrag in deze syntax worden beschreven om de leesbaarheid en onderhoudbaarheid van de geautomatiseerde tests te vergroten. Er kleeft ook een nadeel aan het toepassen van Gherkin op deze testniveaus; het creëert overhead waardoor de testen trager kunnen worden.

Wie doet wat?

Testautomatisering is niet het alleen-domein van de tester. De verschillende soorten testautomatisering maken dat verschillende personen verantwoordelijk zijn.

We hebben gezien dat de unit- en componenttests uitgevoerd worden met de IDE op de code. Dit maakt dat de ontwikkelaars verantwoordelijk zijn voor het ontwikkelen en draaien van de unit- en componenttests.

Voor geautomatiseerd testen zijn op alle lagen test frameworks nodig om het testen te faciliteren. Het bouwen van deze frameworks is codeontwikkeling. Veelal gebeurt dit door een zogenaamde softwareontwikkelaar-in-test of iemand in de functie van testautomatiseerder.

De tester krijgt vaak een bredere rol wanneer er sprake is van testautomatisering. Vanuit zijn vakkennis ziet hij erop toe dat er een teststrategie is binnen het team. De teststrategie geeft richting aan de vraag welke risico’s er zijn en op welk testniveau ze het best gedekt kunnen worden. Daarbij houdt hij in de gaten dat de overlap in testdekking tussen de verschillende testniveaus niet te groot wordt maar ook dat er geen gaten in de testdekking vallen. Hij werkt nauw samen met de ontwikkelaars om ervoor te zorgen dat de juiste testdekking in de

unit- en componenttests wordt geborgd.

Daarnaast kan de tester degene zijn die de testgevallen in Gherkin syntax opstelt. Hiervoor zijn geen programmeervaardigheden vereist omdat het testframework de Gherkin syntax inleest en de tests uitvoert.

Vooruitblik volgende blog

Tot zover deze blog. In de volgende blog ga ik dieper in op verschillende soorten test frameworks.

STAY TUNED!

Wim ten Tusscher