Het managen van testautomatisering

Doel

Dit document is bedoeld voor iedereen die vanuit een coördinerende of managende rol of functie te maken krijgt met testautomatisering. Het biedt de benodigde diepgang en samenhang op onderwerpen die belangrijk zijn rond testautomatisering en die helpen om testautomatisering te plaatsen in de context van een overkoepelende teststrategie. Hiermee geeft het handvatten om testautomatisering succesvol te managen.

Inleiding

Over testautomatisering wordt veel geschreven. Meestal gaat het over de implementatie van tools en de technieken die nodig zijn voor het opzetten van geautomatiseerde tests. Veel minder belicht zijn de onderwerpen die van belang zijn voor het managen van testautomatisering met als doel het in te zetten als onderdeel van een gezonde teststrategie en, op een niveau hoger, de IT-development strategie. Hierbij zijn andere zaken belangrijk dan de tools en technieken voor het maken van geautomatiseerde tests.

Wat kun je verwachten van testautomatisering?

Voordat we dieper stilstaan bij testautomatisering is het belangrijk om te weten wat wel en wat niet kan met testautomatisering. Het is namelijk geen silver bullet (al willen leveranciers van tooling t.b.v. testautomatisering je dat wel graag doen geloven).

Als we ons realiseren dat:

- Specificaties nooit volledig zijn.
- Je alleen datgene kunt specificeren dat je weet. Bij complexe software is er ook gedrag waarvan je niet weet dat je het niet weet.
- Dat zaken als look, feel moeilijk scherp te specificeren zijn.

Dan wordt al snel duidelijk dat niet alle tests te automatiseren zijn. Een goede teststrategie bestaat dus uit een combinatie van geautomatiseerd checks van gespecificeerd gedrag en handmatige, exploratory tests om gedrag waarvan je niet weet dat je het niet weet te onderzoeken.

Wat niet (test) automatiseren

Tests die niet deterministisch zijn

Automatisering heeft veel voordelen, maar er is een hele belangrijke randvoorwaarde voor succesvol toepassen van geautomatiseerde tests: alleen deterministische tests kunnen geautomatiseerd worden. Dit betekent dat je de volgende zaken nodig hebt om een geautomatiseerde test te kunnen maken:

- uitgangssituatie;
- benodigde testdata;
- teststappen;
- te verwachten uitkomst (de specificatie).

Dan en alleen dan is een test voldoende scherp om hem herhaalbaar succesvol geautomatiseerd uit te voeren. Als op één of meer van de genoemde zaken 100% herhaalbaarheid ontbreekt, is er een grote kans op een falende test. Wanneer de test onterecht faalt wordt dit een “false negative” genoemd en wanneer deze onterecht slaagt een “false positive”. Er zit geen fout in de te testen code, de test is fout. Foutieve tests door andere oorzaken dan foutieve code schaden het vertrouwen van geautomatiseerd testen en moeten daarom zoveel mogelijk voorkomen worden (zie sectie: QA van geautomatiseerd testen voor meer details over dit onderwerp).

Eenmalige tests

Het is van belang om je te realiseren dat een geautomatiseerde test code is die geschreven en beheerd moet worden. Met andere woorden; een geautomatiseerde test is niet gratis! Het is daarom onverstandig om eenmalige tests te automatiseren. Voor dit soort tests is de business case negatief. Beter is het om eenmalige tests met de hand uit te voeren. Als uitzondering op deze regel gelden testen die simpelweg niet handmatig uitgevoerd kunnen worden zoals een performance test of tests die sneller te automatiseren zijn dan handmatig uit te voeren.

Wat wel automatiseren
Tests die:

- vaak herhaald moeten worden (regressietest);
- niet handmatig uitvoerbaar zijn (b.v. performance test);
- een korte doorlooptijd vereisen in een CI/CD-pipeline.

Taken die ondersteunend zijn aan het ontwikkelingsproces (voor één of meerdere rollen zoals dev, test en ops) zoals:

- testdata opvoeren in een system. De opgevoerde testdata kan vervolgens gebruikt worden in zowel handmatige als geautomatiseerde tests;
- (test)omgevingen opstarten;
- testdata kunnen resetten.

Testautomatisering is testen

Testautomatisering is testen, maar dan op een geautomatiseerde manier. Doordat geautomatiseerd testen in essentie niet verschilt van handmatig testen (controleren op verwacht resultaat) gaat veel van de bestaande theorie rond testen ook op voor testautomatisering.

We zijn gewend om handmatig testen uit te voeren op verschillende testniveaus (bv. systeemtest, integratietest, ketentest). Het principe van testniveaus geldt ook voor geautomatiseerd testen. Je kunt geautomatiseerd testen toepassen op allerlei testniveaus (zie sectie “De testautomatiseringspiramide” voor meer details)

Met testautomatisering zijn we in staat om in korte tijd veel tests uit te voeren zonder dat het tijd kost van mensen. Voor het uitvoeren van de tests is dit waar, maar het maken en beheren van tests kost wel tijd en inspanning van mensen. Ook bij geautomatiseerd testen moeten we er dus voor zorgen dat we met zo min mogelijk tests een zo groot mogelijke testdekking verkrijgen. We moeten dus goed nadenken over wat de efficiënte tests zijn voordat we tests gaan bouwen. Voor het ontwikkelen van efficiënte tests zijn twee zaken belangrijk:

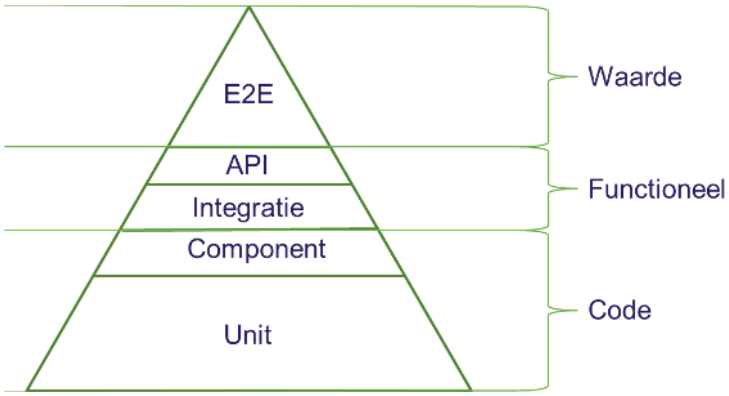
- No risk-no test. Een test mag alleen bestaan als hij een onderkend risico afdekt. Hierbij krijgen grote risico’s meer testdekking en lage risico’s minder testdekking.
- Door gebruik te maken van testtechnieken als equivalentieklassen, grenswaarden en slimme datacombinaties kunnen we de hoeveel tests beperken terwijl we toch een goede testdekking creëren.

Tests, dus ook geautomatiseerde tests hebben als doel informatie te verschaffen over het geteste (sub)systeem. Dit doen we door het opstellen van inzichtelijke testrapportage. We zullen zien dat testautomatisering op verschillende testniveaus kan worden toegepast, ook op sterk technische niveaus (whitebox tests). Het is belangrijk om de geautomatiseerde tests zoveel als mogelijk te beschrijven in herkenbare termen die aansluiten bij de belanghebbende van het testniveau. Dan slaagt testautomatisering erin waardevolle informatie te leveren waar de afnemers (ontwikkelaar, tester, business) ook echt iets aan hebben.

De testautomatiseringspiramide

Een *good practice* voor testautomatisering is de test(automatiserings)piramide. Dit is een heuristiek voor het correct opbouwen van de testdekking van geautomatiseerd testen. Een groot voordeel van testautomatisering ten opzichte van handmatig testen is dat je het product/systeem ook whitebox kunt testen. Er ontstaan nieuwe testniveaus zoals unit test en component tests, op diepere lagen in de code.

De testpiramide ziet er als volgt uit:



De piramide is onderverdeeld in drie soorten testdekking: codekwaliteit, functionele kwaliteit en waarde voor de eindgebruiker.

Unittests zijn kleine compacte tests op code units (een klasse of een methode). Ze hebben als doel aan te tonen dat code correct werkt. Hiertoe worden de afhankelijkheden tussen verschillende units verwijderd. Elke unit wordt in isolatie van de andere units getest, dit wil zeggen dat eventuele afhankelijkheden tussen units worden vervangen door test doubles (fakes/mocks/stubs/dummies). Dit heeft een aantal voordelen:

- Code units zijn klein, ze bezitten maar een beperkte functionaliteit. De tests zijn dus klein en daardoor snel.
- Als een test faalt, is snel te achterhalen in welk stukje code de fout zit; de code unit is immers geïsoleerd. Debuggen is dus snel en eenvoudig.

De kracht van unittests is ook haar beperking. Correct functionerende, losse onderdelen van de code maken nog niet dat het samengestelde geheel van code correct werkt. Door het loskoppelen van afhankelijkheden geven unit tests geen garantie voor een goed werkend geheel.

Het volgende testniveau is component test. In dit testniveau worden de echte afhankelijkheden tussen units gebruikt en worden samenhangende units (een softwarecomponent) integraal getest. Afhankelijkheden met andere componenten worden op dit niveau niet meegenomen.

Op het integratietestniveau testen we componenten in samenhang. Samenhangende componenten realiseren (deel)functionaliteit, dus op dit niveau is het mogelijk om (deel)functies en/of business logica tegen de functionele requirements te testen.

Op eventueel aanwezige API's kunnen we vaststellen of een systeem op de technische koppelvlakken correct functioneert. API's zijn relatief eenvoudig te benaderen met testtools en geven de mogelijkheid functioneel te testen conform functionele ontwerpen. Veel business logica is dan vaak al testbaar.

Als laatste rest het E2E-testniveau. Dit is het traditionele testniveau waarop we een systeem of een keten van systemen als een black-box, via de GUI, handmatig testen. Hier testen we het complete systeem op correct geïmplementeerde user journeys, op waarde voor de eindgebruiker. Op dit niveau geeft een geslaagde test veel vertrouwen in de kwaliteit van het product, aan de andere kant raakt de test veel samengestelde code waardoor debuggen tijdsintensief is en de test veel tijd kost om uit te voeren. Daarnaast kan de test door het grote aantal externe factoren eerder ten onrechte falen, waardoor de beheerinspanning toeneemt en de waarde van de test afneemt. Daarom is het van belang dat de onderliggende testniveaus voldoende dekking bieden.

Betekenis piramidevorm
Het vaststellen van geschikte testniveaus binnen de context van het product is onderdeel van het bepalen van de teststrategie. De lagen in de testpiramide kunnen per context verschillen omdat ze geënt zijn op de architectuur van het systeem. Het principe blijft echter steeds overeind: hoe lager in de piramide hoe sneller de test maar hoe meer je een geïsoleerd stuk code test waardoor het minder toevoegt aan het vertrouwen dat het systeem het juiste doet. Hoe hoger in de testpiramide hoe trager de test, maar op dit niveau geeft elke geslaagde test wel veel vertrouwen in correct gedrag voor de eindgebruiker. Hierin moet een goede verdeling worden gezocht. Dit komt tot uiting in de

piramidevorm. Die geeft aan dat de testdekking op de aanwezige risico's in het product (testen is risicogebaseerd) zoveel mogelijk naar beneden in de testpiramide moet worden gedrukt. Onderin de piramide borgen we testdekking op deelfunctionaliteit, edge cases, equivalentieklassen, foutafhandeling, etc. Deze testdekking vul je aan met een kleinere testdekking op integratie en API-niveau. De testdekking wordt gecompleteerd met een zeer beperkt aantal E2E gebaseerde tests. Hier zie je dus een duidelijk verschil met handmatig testen dat zwaar leunt op E2E gebaseerd testen.

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.

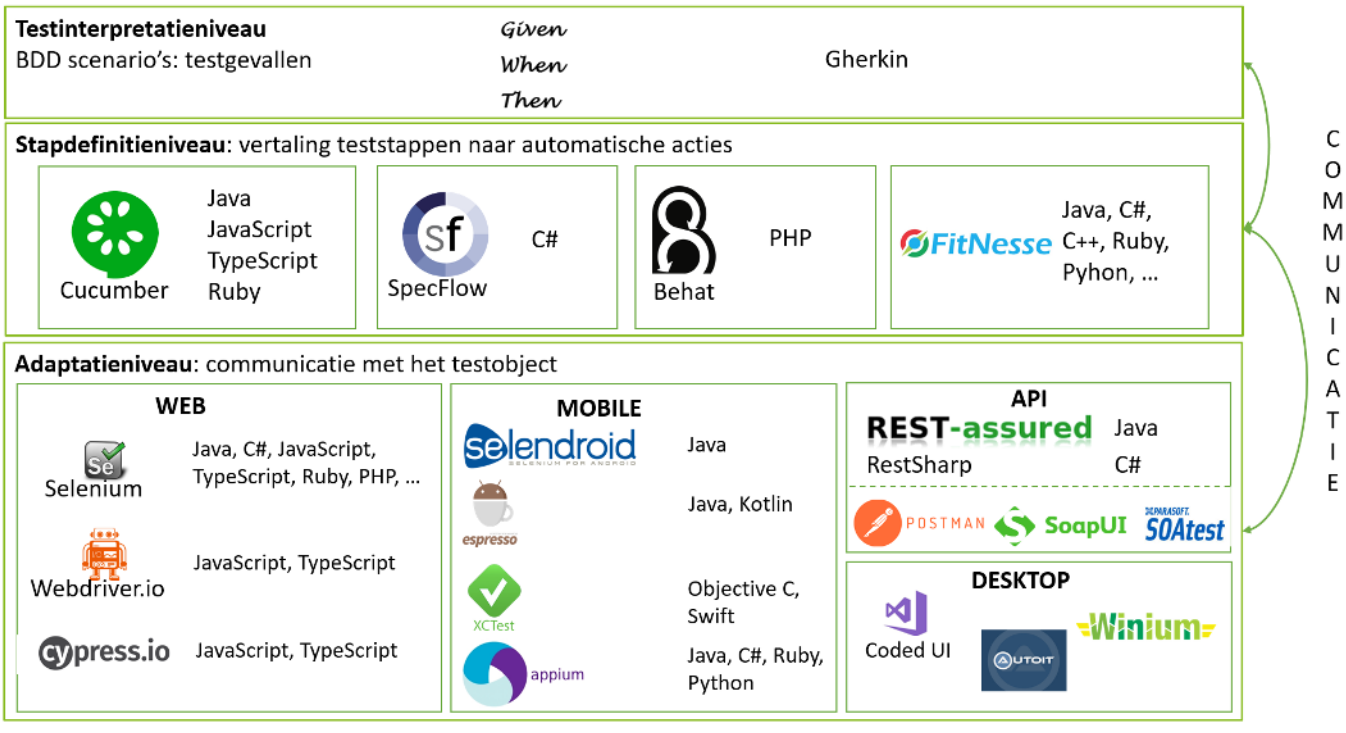
Voor geautomatiseerd testen zijn op alle lagen test frameworks nodig om het testen te faciliteren. Het bouwen van deze frameworks is codeontwikkeling. Veelal gebeurt dit door een zogenaamde softwareontwikkelaar-in-test of iemand in de functie van testautomatiseerder.

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.

Test frameworks

Testframeworks hebben tot doel het testen te vergemakkelijken. Ze regelen de randvoorwaarden en integreren met de ondersteunende tools waarmee tests eenvoudig kunnen worden opgesteld en uitgevoerd zodat de gebruiker van het framework alleen nog maar over de te implementeren tests hoeft na te denken. Testframeworks kom je tegen op elk testniveau, er zijn unittest frameworks, API-test frameworks en E2E-test frameworks.

Vaak vind je in deze frameworks drie niveaus van onder naar boven bestaande uit een adaptatieniveau een stapdefinitieniveau en een testinterpretatieniveau. Elke laag biedt een verdere vorm van abstractie van de code. Onderstaande figuur laat de drie lagen zien met de veel voorkomende frameworks per laag.



Adaptatieniveau
Het onderste niveau, het adaptatieniveau, is de laag die in geval van het API framework de API-calls doet en de responses ontvangt. Bij E2E frameworks bevat het adaptatieniveau het page-object-model van de te testen pagina. Het niveau bevat ook de elementaire handelingen die nodig zijn om de tests uit te voeren. In geval van een E2E testframework vind je er handelingen als: klik-op-een-knop, lees-een-schermelement-uit, scroll-door-een-lijst etc.

Stapdefinitieniveau
Boven op het adaptatieniveau vind je in een goed framework een laag met stapdefinities (wordt ook wel glue-code genoemd). Deze laag vertaalt de zinnen uit de (Gherkin) feature file naar functionele, gebruikers gefocuste zaken.

Deze geeft hij vervolgens door aan het adaptatieniveau. Voorbeelden van stapdefinities zijn: Selecteer product in de webshop, Ga naar het winkelmandje, Check out, Voer de betaling uit etc.

Testinterpretatieniveau
Het testinterpretatieniveau bevat in geval van een BDD testframework de Gherkin syntax zodat elke test die geschreven is volgens deze syntax uitgevoerd kan worden door het framework.

Voordelen bij goede invoering

Goed ingevoerde testautomatisering levert grote voordelen op. Goed geïmplementeerde geautomatiseerde tests

- kunnen zelfstandig draaien, ook buiten kantooruren.
- zijn 100% reproduceerbaar en herhaalbaar.
- voeren tests uit met hoge doorloopsnelheid.
- realiseren extra testdekking die met de hand moeilijk of niet haalbaar is.

Risico's bij foute invoering

Testautomatisering kan op vele manieren misgaan waardoor de genoemde voordelen gedeeltelijk of zelfs volledig te niet gedaan worden. Risico's bij het opzetten van testautomatisering zijn:

- Testautomatisering beschouwen als het geautomatiseerde equivalent van handmatige testen en daardoor wordt de heuristiek van de testpiramide niet gevolgd. Hierdoor ontstaan te veel E2E gebaseerde tests die traag verlopen en veel beheerinspanning vragen.
- Niet stilstaan bij het feit dat testautomatisering softwareontwikkeling is. Als bij testautomatisering onvoldoende wordt gelet op codekwaliteit in de vorm van onderhoudbaarheid, schaalbaarheid en beheerbaarheid dan zal het initiatief na een succesvolle, snelle start op de lange duur krakend tot stilstand komen.
- Testautomatisering coderen in een taal anders dan de ontwikkelaars gebruiken. Dit maakt dat testautomatisering op een eiland komt te staan, alleen de maker (Software Developer in Test) kan dan de code onderhouden. Door dezelfde taal te gebruiken als waarin de ontwikkelaars programmeren borg je dat het hele team de testautomatisering kan onderhouden.
- Alles automatiseren. Dit is niet slim. Testautomatisering is niet gratis dus er moet een positieve business case zijn. Alleen testen die vaak herhaald moeten worden (bv de regressietesten) hebben een positieve business case. Verder is het goed om stil te staan bij het doel van elke test; welk risico of welke requirement dek ik af met de test?

QA van geautomatiseerd testen

Bij geautomatiseerd testen is het belangrijk om stil te staan bij de vraag; hoe borg ik de kwaliteit van mijn tests?

Voor geautomatiseerde tests geldt namelijk, net als voor alles wat door een computer gedaan wordt, dat mensen er 100% op gaan vertrouwen. Ze zijn niet meer kritisch of de test wel juist is. Het is dus belangrijk om tijdens het maken van een geautomatiseerde test, elke test minimaal 1 keer geconditioneerd te laten falen. Alleen wanneer een test faalt als hij ook zou moeten falen weet je zeker dat hij het juiste test en kun je op de test vertrouwen.

Er zijn drie typen incorrect tests:

- False positive.
- False negative.
- Flaky.

Een False Positive test geeft, ongeacht de code, altijd een PASS als resultaat. False Positives zijn gevaarlijke tests omdat ze een schijnzekerheid geven, de test faalt namelijk nooit, ook niet bij een fout in het product! Je mist dus een stuk testdekking.
Een False Negative test geeft altijd een FAIL als resultaat, ook al is de code goed. False Negatives ondermijnen het vertrouwen in de testautomatisering en versterken de neiging om tests "uit" te zetten. Dit laatste is gevaarlijk omdat voor geautomatiseerde tests het principe geldt: als 1 schaap over de dam is volgen er meer. Met het tijdelijk uitzetten van een test zeg je eigenlijk dat de waarde van de test niet erg groot is, je durft namelijk ook wel zonder de test door te gaan. Daarmee stel je het bestaansrecht van de test ter discussie!

Een flaky (breekbare) test geeft niet altijd hetzelfde resultaat door condities waar geen rekening mee gehouden is. Vaak ligt hier een probleem bij de testomgeving welke resulteert in andere timing en dus een ander resultaat. De onbetrouwbaarheid van een flaky test ondermijnt ook het vertrouwen in testautomatisering.

False positive, false negative en flaky tests zijn alle drie even erg, ze ondermijnen de kwaliteit van de informatievoorziening door testen. Daarom is het van belang om kwalitatief goede tests te schrijven met onderhoudbaarheid als belangrijke vereiste. Hiervoor kunnen dezelfde methodes gebruikt worden als bij de ontwikkeling van normale productcode: code review, gebruik van linters en refactoren. In het geval dat een test faalt is het verstandig om met het team af te spreken dat een falende test nooit uitgezet wordt. Een falende test moet zo snel als mogelijk worden geanalyseerd en gerepareerd of worden verwijderd als mocht blijken dat de test irrelevant is geworden.

Mutatietesten is een interessante methode om de kwaliteit van de testautomatisering te testen. Mutatietesten is het bewust muteren van de code volgens vooraf ingestelde mutatieregels. Hiervoor zijn in veel programmeertalen tools beschikbaar. De tools planten heel specifieke fouten in de code (fault seeding). Goed geïmplementeerde testautomatisering zal de geplante fouten vinden. Het rapport met niet gevonden mutaties is het startpunt voor verbetering van de testautomatisering. Het belang (het productrisico) van de code geeft hierbij richting aan de prioritering van het werk.

Eisen aan de testomgeving en testdata

Een stabiele testomgeving met voldoende en bruikbare testdata is voor handmatig testen een vereiste. Er is niets vervelender dan een test die faalt omdat de testdata incorrect of niet bruikbaar is.

Testautomatisering stelt aanvullende eisen aan een testomgeving. Vanwege het extreem deterministische karakter van geautomatiseerde tests moet de testomgeving en de testdata 100% deterministisch zijn. Het is daarom goed om geautomatiseerde tests op een eigen testomgeving te draaien die niet voor andere doelen (b.v. handmatig testen) of door meerdere teams wordt gebruikt. Veelal zie je dat bij invoering van testautomatisering er een vraag ontstaat naar meer testomgevingen. Technieken als virtualisatie (b.v. Docker, Kubernetes) en Clouddiensten (AWS, Azure, Oracle Cloud, Google Cloud) bieden functionaliteiten die het, on-demand, opspinnen en afbreken van reproduceerbare testomgevingen met reproduceerbare testdata mogelijk maken. Een voorwaarde voor testautomatisering is dat de gebruikers volledige beheerrechten op de testomgevingen hebben.

Stabiele testdata
In transactionele systemen (b.v. een online shopping systeem) wordt data verwerkt. Een order wordt ingevoerd, betaald en verwerkt tot aan verzending. Door het uitvoeren van een test verandert dus de testdata en is deze niet opnieuw te gebruiken door dezelfde test. Omdat geautomatiseerde tests bij herhaling worden uitgevoerd, moet het dus mogelijk zijn om de testdata voor aanvang van elke test terug te zetten in de begintoestand. Dit kan op twee manieren:

- De testdatabase leeggooien en een kopie van de initiële testdata terugzetten.
- De testdatabase leeggooien en de testdata met behulp van de testautomatisering invoeren voordat de tests beginnen.

Om testen deterministisch te maken moeten externe afhankelijkheden op de testomgeving die niet onder controle zijn zoveel als mogelijk geëlimineerd worden. Dit kan door de echte instanties (bv een API) te vervangen door een stub of een mock. Een zelf beheerde stub of mock geeft je namelijk de mogelijkheid elke gewenste response te geven die nodig is voor het kunnen uitvoeren van de test. Denk hierbij aan edge cases en foutsituaties (HTTP 40x en 50x responses.

Wanneer gebruik wordt gemaakt van stubs en/of mocks is het belangrijk dat het gedrag ervan gelijk is aan het echte product. Bij invoering van stubs en mocks is het dus van belang om het gedrag eenduidig vast te leggen en te beheren (b.v. in een OpenAPI/Swagger contract).

Testdata en gebruikte stubs en mocks zijn een essentieel onderdeel van testautomatisering en het is daarom belangrijk om ze, net als de code van de geautomatiseerde tests onder versiebeheer te brengen.

CI/CD

Testautomatisering is een voorwaarde voor continuous integration (CI), continuous deployment (CD) en continuous delivery (CD). Bij CI en CD is continu inzicht vereist in de kwaliteit van de code en het product terwijl deze continu aan verandering onderhevig zijn. Dit is onmogelijk zonder het gebruik van efficiënte en effectieve testautomatisering.

Continuous integration

Bij continuous integration is het belangrijk om de kwaliteit van nieuwe en/of aangepaste code op een code branch zoveel mogelijk te borgen voordat de nieuwe en/of aangepaste code samengevoegd wordt met de bestaande code. Omdat het verstandig is vaak te integreren, moeten de geautomatiseerde tests op de branch snel uitgevoerd kunnen worden. In de testpiramide hebben we gezien dat de tests onder in de piramide snel zijn. Unit- en componenttests zijn dus de aangewezen tests om toe te passen op een ontwikkelbranch. Het al dan niet slagen van de unit- en componenttests wordt bij CI gebruikt als quality gate voor het mogen mergen van een branch naar de master.

Continuous deployment

Bij continuous deployment wordt de code automatisch uitgerold naar één of meerdere testomgevingen voor het uitvoeren van functionele testen, ketentesten, acceptatietesten of het testen van een ander kwaliteitsattribuut. In de testpiramide hebben we gezien dat deze tests wel te automatiseren zijn maar dat deze in het algemeen trager zijn in de uitvoering. Naarmate de omvang van het product groeit zal ook de hoeveelheid uit te voeren tests en daarmee de testdoorlooptijd groter worden. Als de doorlooptijd van het uitvoeren van alle tests te groot wordt voor de gewenste doorlooptijd van de pipeline, is het verstandig om de tests te gaan managen. Een manier hiervoor is het maken van subsets. Door tests onder te brengen in subsets naar bijvoorbeeld functionaliteit of subketen, kan risicogebaseerd een subset van de totale tests worden uitgevoerd om gedurende de dag de pipeline snel te houden. Gedurende de nacht kan dan de totale testset draaien om de kwaliteit van het product met voldoende diepgang te blijven testen.

Ook is het verstandig om bij continuous deployment subsets te definiëren die gebruikt kunnen worden als smoketest voor het controleren van de automatisch gegenereerde testomgevingen voordat de test op die omgeving wordt afgetrapt.

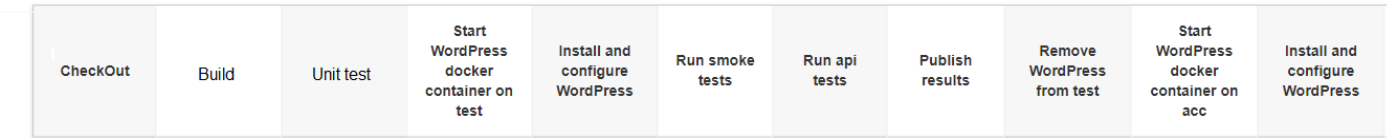
Continuous delivery

Bij continuous delivery is ook de laatste stap, het naar productie brengen van code, geautomatiseerd. Waar bij continuous deployment bepaalde tests in de pipeline nog met de hand uitgevoerd worden, zijn bij continuous delivery alle testen in de pipeline geautomatiseerd, inclusief de acceptatietesten en een geautomatiseerde smoketest op productie om te beoordelen of de uitrol goed is verlopen.
Omdat in continuous delivery geen plaats meer is voor handmatig, exploratory testen in de pipeline vraagt het een hoge volwassenheid van alle fasen in het proces. Productrisco's die we in een niet continuous delivery setting met exploratory tests afdekken moeten we nu op een andere manier ondervangen. Continuous delivery gaat dus samen met risico mitigerende maatregelen als:

- Canary releases (geschaald naar productie gaan).
- A/B tests in productie.
- Intensieve productiemonitoring op alle levels (server, applicatie en business monitoring).

Genoemde mechanismen bieden de mogelijkheid om productrisico's in productie te mitigeren. Met canary releases en A/B tests kan het exploratory testen in productie plaatsvinden. Eventuele problemen in productie worden snel gevonden (monitoring), de impact van de problemen blijft beperkt (canary releases, A/B tests) en kunnen snel opgelost worden (snelle pipeline).

Onderstaande figuur laat een gedeelte van een CI/CD-pipeline zien met de daarin aanwezige stappen:



Groeipad invoering testautomatisering

Voor groei is het goed je te realiseren dat verandering beter gaat als er pijn/noodzaak is. Dit mechanisme kun je gebruiken door, met het model en gedachtengoed van de testpiramide in het achterhoofd, tests te implementeren met oog voor beheertijd en doorlooptijd.

Als ondersteunende maatregel is het verstandig om root-cause analyse uit te voeren op elke door een geautomatiseerde test gevonden bug. Stel bij elke bug de vraag of deze met een ander type test (lager in de testpiramide) gevonden had kunnen worden. Op deze manier bouw je de business case voor de investering in de volgende verbeterstap.

Daarnaast help het om tijdens de sprint refinement expliciet stil te staan bij het opstellen van een teststrategie per sprint en per user story. Hierdoor ontstaat een grotere bewustwording voor het test(automatiserings)vraagstuk bij het team. Naarmate de bewustwording groeit en er meer ervaring is opgedaan met de geïmplementeerde geautomatiseerde tests, kunnen meer testniveaus van de piramide opgenomen worden in de teststrategie.

Afhankelijk van de context zijn er twee aanpakken voor groei.

Green field: er is nog geen testautomatisering.
Start dan op alle lagen van testpiramide tegelijk. De slagingskans van deze aanpak hangt sterk af van zowel het team (kennis van testautomatisering en kwaliteitsbewustzijn) als het product (testbaarheid) en de kwaliteit en beschikbaarheid van testomgevingen. Als je met een nieuw team aan een nieuw product begint, heeft de integrale aanpak de voorkeur.

Als er al een begin is gemaakt met testautomatisering
Vaak hebben teams al een begin gemaakt met testautomatisering vanuit de problematiek dat handmatig testen te veel resources (tijd en menskracht) kosten. In een agile scrum setting betreft het dan vaak de regressietest. Meestal zijn in deze situatie de handmatige tests 1:1 geautomatiseerd met een tool of framework. Vaak ontstaan met deze aanpak problemen rond beheerbaarheid en doorlooptijd, we zitten namelijk bovenaan in de testpiramide. In deze context is het “omlaag duwen” van tests in de testpiramide een goede aanpak.

In de eerste stap kijken we met een scherp oog naar de gebruikte tool of framework. Als een tool of framework quick en dirty is geïmplementeerd zal eerst een refactorslag uitgevoerd moeten worden om het schaalbaar, stabiel en beheerbaar te maken. Als er een tool is gebruikt (bv een record en replay tool) zal gekeken moeten worden of deze oplossing voldoende toekomstvast is. Vaak ontstaan met deze aanpak problemen rond schaalbaarheid, beheerbaarheid of stabiliteit. Indien nodig zal de bestaande automatisering omgebouwd moeten worden naar een nieuw framework.
Pas in deze stap goed op voor het bouwen van te veel testdekking! Bouw vooral de tests waarmee je de waarde van het product op user journeys aantoont. Houd een scherp oog voor doorlooptijd en de uitkomsten van de root-cause analyse op gevonden bugs. Op een gegeven moment kom je op een punt dat blijkt dat er winst te behalen is op een ander, lager testniveau (b.v. API). Dit is het moment om een niveau dieper te gaan in de testpiramide.

In stap twee is het belangrijk kennis op te doen van de productarchitectuur. Welke interfaces en API’s heeft het product? Met deze kennis kan vervolgens testdekking op losse modules, op de integratie tussen modules en op interfaces en API’s worden gecreëerd. Met het bouwen van testdekking op deze onderliggende lagen is het belangrijk om te kijken of al geautomatiseerde GUI tests overbodig worden. Een GUI edge case op een invoerveld kan in deze fase vrijwel altijd vervangen worden door een edge case op de invoermodule. Daarmee kan de GUI edge case test komen te vervallen.

In stap drie zakken we verder af in de testpiramide. Om dat te kunnen moeten we, in nauwe samenwerking met de developers, de productarchitectuurkennis verdiepen tot het niveau van componenten en code units. Met deze kennis kun je als team kijken welke testdekking verplaatst kan worden naar unit testdekking of component testdekking. De edge case test op een invoer module kan hierbij vrijwel altijd verplaatst worden naar een unittest op de invoer methode of invoer klasse.

Met het volgen van deze drie stappen ontstaat een beheersbaar groeipad naar steeds effectievere en efficiënte testautomatisering.