Why developers should love their tests

Ontketen de volledige kracht van je tests

Wie al een tijdje in het software development vak zit, zal het wel herkennen. Om de huidige feature af te ronden, moet je nog een aantal tests schrijven. Het blijkt verrassend lastig te zijn om het geheel automatisch getest te krijgen en we weten per slot van rekening al dat het werkt! Waarschijnlijk neig je er naar het maar te laten zitten, zoveel kan dat niet kwaad, toch?

We weten dat tests belangrijk zijn, maar toch hebben we er vaak niet alleen positieve gevoelens bij. Tests zouden ons eigenlijk alleen maar goeds moeten brengen. De realiteit is vaak minder rooskleurig. Wij zijn veel ontwikkelaars tegengekomen die negatieve associaties bij tests hebben. Logisch ook, we kennen allemaal de instabiele, trage tests die het om de haverklap niet meer doen. Wij zijn van mening dat die slechte voorbeelden niet zouden moeten bepalen hoe we over tests in het algemeen denken.

Samen pogen wij al jaren om erachter te komen wat het nou maakt dat tests in het ene geval die waarde toevoegen en in het andere geval je tegen lijken te houden. Met inspiratie uit boeken als Extreme Programming (XP), Clean Code en Refactoring gaan we in dit artikel dieper in op wat de principes zijn die leiden tot goede tests. Eerst herhalen we kort wat testen precies inhoud. We kijken daarna naar de oorzaken waardoor tests ons negatieve associaties opleveren. Vervolgens tonen we aan dat goede tests vereisen dat we net zo kritisch worden op onze tests als op onze productiecode. Als we onze tests op de juiste manier inzetten, geeft dit ons een middel om onze productiecode sneller aan te passen. Het leidt zelfs tot een betere kwaliteit van onze codebase!

Testen en de soorten tests

Laten we bij het begin beginnen; wat is testen eigenlijk? In dit artikel maken we het onderscheid tussen de activiteit testen en de individuele tests die we uitvoeren. Er zijn veel verschillende interpretaties van wat testen precies is, maar we zullen de hier volgende definitie hanteren: een proces dat gebruikt wordt om het correcte gedrag van een (deel) systeem aan te tonen.

Alhoewel er veel soorten tests zijn, onderkennen we voor nu drie niveaus: Unit, Integratie en End-to-End (geautomatiseerd of handmatig). Elk van deze soort tests heeft bepaalde eigenschappen. We zullen de testsoorten aan de hand van deze eigenschappen uiteenzetten.

Het laagste niveau is **Unit Testen**. Hiermee testen we de kleinst mogelijke eenheden van de code, het liefst in isolatie. De eigenschappen van Unit Tests zijn dat ze makkelijk te maken en aan te passen zijn. Daarnaast zijn ze snel uit te voeren en als er een test faalt weet je bijna zeker waar de fout zich bevindt.

Met **Integratie Testen** worden de losse eenheden als groep getest en niet meer in isolatie. Ten opzichte van Unit Tests tonen deze tests dus minder specifiek aan waar iets fout gaat als een test faalt. Daarnaast zijn ze vaak lastiger te maken en is de doorlooptijd langer.
Tot slot hebben we End-To-End (E2E) Testen. Hiermee tonen we de correcte werking van het complete systeem aan. Dit gebeurt vaak via de user interface. Aangezien een UI aan veel wijzigingen onderhevig is en er veel verschillende componenten zijn, zijn deze relatief duur om te onderhouden en instabieler dan de andere testniveaus.

Roy Straub is Software Engineer bij CodeSquad. Momenteel houdt hij zich vooral bezig met het toepassen van DevOps / Extreme programming practices bij zijn projecten. Daarnaast is hij enthousiast Kotlin gebruiker en altijd bezig zich in nieuwe talen en methodologieën te verdiepen.

Matthijs Thoolen is als Software Engineer werkzaam bij Codesquad.

34-38 TESTING.indd 34 14-10-20 14:11

Waarom onze tests frustreren

Onbalans tussen de soorten tests

Een reden waarom testen zo frustrerend is, is een onbalans in het niveau van de tests in onze codebase. Dit is een oorzaak voor de welbekende testsuites die bijna niet worden gedraaid omdat ze te lang duren en instabiel zijn.

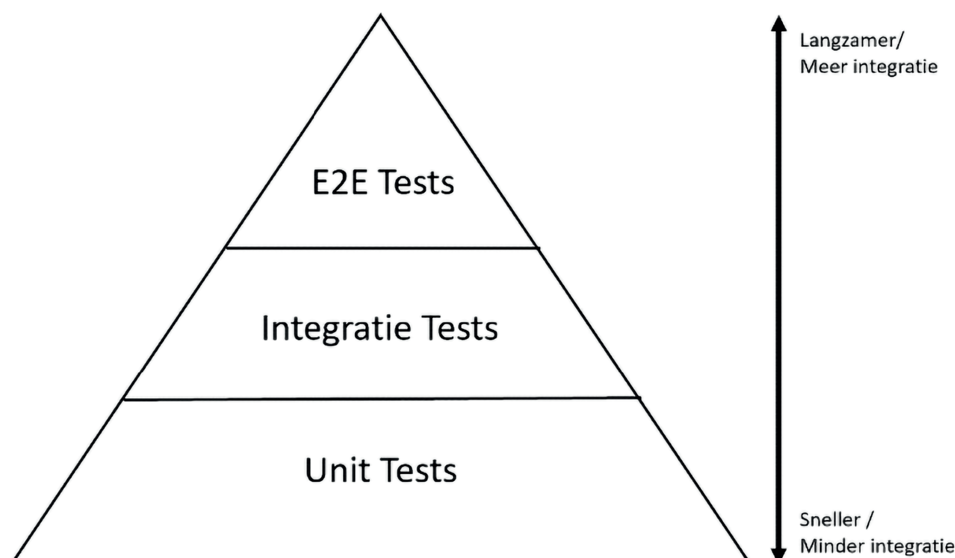
De praktijk leert dat het aantonen van correcte werking van een systeem vaak wordt gedaan vanuit het gebruikersperspectief. Daarom wordt er te vaak gekozen voor E2E of handmatige tests met alle gevolgen van dien. Het duurt langer om ze uit te voeren en tonen ze erg globaal de plek aan waar iets fout gaat.

Die instabiliteit of "flakiness" wordt vaak veroorzaakt doordat er op dit niveau veel meer dingen zijn die fout kunnen gaan, zoals een update van een browser of een instabiele testomgeving. Het gevolg is dat we de E2E tests minder gaan vertrouwen en daarmee minder vaak willen draaien. Daarmee neemt de waarde van die tests af.

Er is voor dit probleem een oplossing. De zogenaamde Test Piramide (1). Die stelt dat we naar verhouding veel unit tests, minder integratie tests en nog minder E2E tests moeten hebben. De focus van testen moet dus niet worden gelegd op het bewijzen van alle mogelijke testgevallen vanuit een gebruikersperspectief. Als dat wel gedaan wordt, merk je dat de piramide omkeert en er in verhouding dan te veel E2E getest wordt.

ALS WE ONZE TESTS OP DE JUISTE MANIER INZETTEN, GEEFT DIT ONS EEN MIDDEL OM ONZE PRODUCTIE-CODE SNELLER AAN TE PASSEN.

34-38 TESTING.indd 35 13-10-20 16:16



Figuur 1. Test Piramide

Dat betekent ook dat we dat zo veel mogelijk met Unit Testen willen doen. Zoals eerder benoemd, wijzen deze namelijk heel nauwkeurig aan waar iets fout gaat. Als er nauwkeurig wordt aangetoond waar het probleem zit kan er ook snel een oplossing worden gemaakt. Dan voegt een test dus echt waarde toe. Dat resulteert dan ook meer in een verdeling van de testgevallen zoals de bovenstaande piramide.
Handel je in lijn met de Test Piramide, dan merk je dat een falende test sneller tot de oplossing van (potentiële) bugs zal leiden. Je tests worden simpeler, makkelijker om bij te houden en sneller om uit te voeren. Daarnaast zullen ze betrouwbaarder zijn en daarmee zal er meer vertrouwen ontstaan in de tests.

Tests zonder toegevoegde waarde

We weten nu hoe we onze tests beter in balans kunnen brengen. Wanneer we dit doen, eindigen we met meer integratie- en unit tests ten opzichte van e2e tests. Op zich is dit een goed ding, maar het brengt een gevaar met zich mee: koppeling. Onze tests hebben namelijk een harde koppeling met onze productiecode. Als software engineers hebben we geleerd dat te veel koppeling een slechte eigenschap is. Dat is hier ook het geval. Die koppeling maakt het lastiger om te refactoren en brengen dus kosten met zich mee.
Als we elke test zien als extra kosten vanwege de koppeling dan moet er wel iets positiefs tegenover staan. Elke test moet waarde toevoegen. Daarmee bedoelen we dat een test ons meer vertrouwen moet geven in de productiecode. Tests die ons vertrouwen niet verhogen introduceren extra koppeling zonder dat ze voordeel brengen en zijn dus verspilling.
Dat kunnen we voorkomen door kritisch te zijn op welke testen we in onze codebase hebben. Een test die een reeds geteste condities checkt is volgens dit idee dus verspilling. Andere goede voorbeelden van tests zonder meerwaarde zijn tests voor triviale getters en setters of bestaan enkel omwille van de code coverage. Als laatste noemen we nog tests die niet eigen code maar stiekem die van een library of framework testen. We moeten dus een balans zoeken in welke tests we laten staan en welke niet. Kijk kritisch naar je testgevallen en wees niet bang om overbodige weg te gooien. Elke test die blijft staan moet verantwoord kunnen worden.

Testcode van slechte kwaliteit

Je bent het vast wel tegengekomen; een test waar je te vaak moet lezen voordat je snapt wat er precies gebeurt. Die test durf je waarschijnlijk niet aan te raken en een nieuwe testcase schrijven is ook te moeilijk. Als de test faalt, duurt het ook te lang om te zien wat er fout ging. Dit zijn natuurlijk klassieke code smells.
Waar we in productiecode proberen zo veel mogelijk vast proberen te houden aan het 'Don't Repeat Yourself (DRY) (2), principe, is het een goed idee om bij testen niet het doel uit het oog te verliezen. Als we onze testcases te dynamisch of abstract maken wordt het

34-38 TESTING.indd 36 13-10-20 16:16

moeilijker om te zien wat er precies in een test gebeurt. Daarmee verliezen we waarde van de tests, namelijk makkelijk zien hoe de code die getest wordt echt werkt. Schrijf je testen ook niet te WET (Write Everything Twice), dan zijn ze lastig te onderhouden. Het is daarom belangrijk om hierin een balans te zoeken tussen onderhoudbaarheid en expressiviteit. Probeer de tests zo te schrijven dat elke testcase afzonderlijk te begrijpen is.
Afgezien hiervan zijn ook in tests de principes van Clean Code (3) van belang. Houd de testcode duidelijk door goede namen te geven aan variabelen en individuele tests. Ook deze code moet je blijven refactoren als er iets te verbeteren valt. Samenvattend is expressiviteit van je tests een van de belangrijkste doelen, ze moeten duidelijk aangeven wat er precies wordt getest. We hebben enkele principes onderkend die moeten worden toegepast om een goede, robuuste en effectieve testsuite te krijgen. Deze principes zijn:

- Houd een gezonde balans van de soorten tests
- Elke test moet verantwoord kunnen worden
- Kwaliteit van testcode is even belangrijk als productiecode

Voordelen van goede tests

Vertrouwen

Als we dan de moeite hebben genomen om de testsuite aan de voorgaande principes van goede tests te laten voldoen, stelt dat ons in staat om eindelijk de ware schoonheid van onze tests te kunnen zien. Allereerst hebben we het vertrouwen dat een goede testsuite geeft. We kunnen door het draaien van de tests snel feedback krijgen of alle usecases nog correct werken nadat we refactoren. Daarmee wordt de drempel om te refactoren lager, wat ervoor zorgt dat de kwaliteit van onze codebase hoog blijft.

Levende documentatie

Documentatie, welke ontwikkelaar houdt er niet van? Het enige vervelende eraan is dat het geschreven moet worden. Wat als je dit niet zou hoeven doen? Sterker nog, wat als het maken er van integraal deel van je werkproces zou zijn?
Tests zijn levende documentatie voor onze productiecode. We kunnen door naar de tests te kijken precies zien wat de code hoort te doen en hoe je het hoort te gebruiken. In die zin is het dus een vorm van technische en functionele documentatie. 'Gewone' documentatie schiet tekort doordat het een momentopname is. Het is dan ook altijd de vraag of wat er staat nog klopt, terwijl het bij je tests moet kloppen omdat anders de build faalt.

Als we ervoor kiezen om Behaviour Driven Development (4) (BDD) toe te passen met een technologie zoals Cucumber (5), kunnen we onze tests gebruiken als een middel om de functionele documentatie te genereren. Als je deze specificaties dan ook nog samen opstelt met alle rollen in het team, is iedereen ten alle tijde op de hoogte van de functionele staat van het systeem. Kort gezegd is het idee, om één bron van waarheid te hebben. Dit principe uit XP heet code & tests (6) en houdt in dat de enige waarheid onze productie- en testcode zou moeten zijn.

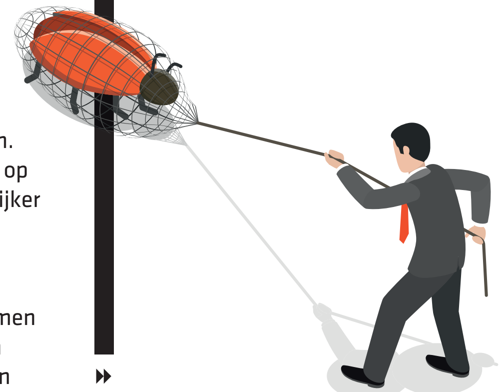
Feedback

Feedback is één van de kernwaardes van XP. Het is een onmisbare voorwaarde om goede software te kunnen maken. Wat heeft dit met tests te maken? Eigenlijk heel veel. Tests zijn een bijzonder efficiënte manier om snelle feedback te krijgen over de staat van het systeem. Eén practice uit XP brengt tests concreet samen met feedback, namelijk Test Driven Development (TDD) (6). Hoewel wij groot fan zijn van TDD is het gebruik hiervan niet voorwaardelijk om een korte feedbackloop te krijgen.
Stel je voor dat je voor elke wijziging die je maakt in een stuk code weer een heleboel data met de hand moet invoeren en controleren of de output klopt. Dat kan minuten duren. Met unit tests en de hardware van tegenwoordig kunnen we deze tijd terugbrengen tot enkele seconden. Dit zorgt ervoor dat je snel weet of een wijziging wel of niet doet wat je verwacht. Deze korte 'cycles' zorgen ervoor dat de drempel om te experimenteren zo goed als verdwijnt en dat je makkelijker grote wijzigingen in kleinere stappen kan delen. Dat verlaagt op zijn beurt weer het risico op fouten doordat kleinere stappen makkelijker zijn om te maken.

Snelheid

Ja dat las je goed, snelheid! Meestal als men het over tests heeft en de vele voordelen die het geeft, is snelheid niet waar ze aan

HOUD EEN GEZONDE BALANS VAN DE SOORTEN TESTS



34-38 TESTING.indd 37 13-10-20 16:16

denken. Wie heeft niet een keer meegemaakt dat je het maken van tests maar moet laten zitten omdat het te veel tijd kost? Wij in ieder geval vaker dan ons lief is.

Grappig genoeg is het tegenovergestelde waar. Laten we een heel concreet voorbeeld geven. Stel je hebt een formulier gemaakt waar nogal wat verschillende regels in zitten. Velden die soms wel worden getoond, soms niet. Op het moment dat we het maken proberen we het een paar keer uit en zien dat het werkt. Prima toch? Als we dat in tests moeten vastleggen kost ons dat meer tijd.

Wij zien het maken van geautomatiseerde tests als een investering. Initieel kost het inderdaad meer tijd om tests te maken dan één keer vluchtig met de hand controleren. Maar hoe zit dat als je het twee keer controleert? En vijf keer? Stel dat het enkele minuten kost om dit met hand uit te voeren en dat het met geautomatiseerde tests enkele seconden duurt. Dan is het niet moeilijk om je voor te stellen dat dit op den duur juist veel tijd kan opleveren. Ervaring leert ons dat het punt waarop deze investering tijdswinst oplevert vaak veel eerder is dan we denken. Na het opstellen van die geautomatiseerde tests heb je er de hele levensduur van het systeem profijt van!

Betere code

Goede code laat zich makkelijk testen. Dit is geen nieuwe constatering, maar wel relevant. Als de productiecode continu wordt getest merk je wrijvingspunten in je codebase. Je voelt eigenlijk al snel dat er iets niet klopt aan je code als het moeilijk te testen is. Oorzaken hiervan zijn bijvoorbeeld dat de code te veel doet (schending Single Responsibility Principle (7)), dat het onduidelijk is waarom je argumenten moet meegeven (schending Interface Segregation Principle(7)) of dat je side effects niet onder controle hebt (schending Dependency Inversion Principle (7)). Als we de code testbaar maken door deze problemen op te lossen, komt dat de kwaliteit dus ten goede.

Dit is een technisch perspectief op het argument dat tests onze code verbeteren. Een ander wellicht even belangrijk perspectief is dat we met tests de eerste afnemer zijn van onze eigen code. Wanneer je de test schrijft om de code uit te proberen word je geconfronteerd met alle goede en slechte aspecten ervan. Op deze manier komen we sneller slechte ontwerpbeslissingen op het spoor en zijn we dus ook sneller in staat dit te verbeteren.
Met TDD gaat dit nog een stap verder. Als je vanuit de test begint word je geforceerd na te denken over de publieke interface die je gaat bouwen, nog voordat de productiecode er is. Dat zorgt ervoor dat we eerst bezig zijn met wat iets moet doen, in plaats van het hoe het dit gaat doen. Wanneer we redeneren vanuit wat iets moet doen raken we niet verzonken in technische details. Dit leidt tot betere abstracties in onze code.

Conclusie

Veel mensen zien hun testen als iets vervelends. Oorzaken daarvoor zijn onder andere dat men te veel van een bepaald soort test heeft, dat men vergeet dat tests ook koppeling introduceren, dat men niet de kwaliteit van hun testen op orde houdt of juist omdat men geconfronteerd wordt met de kwaliteit van de productiecode.
Als er moeite en doordachtheid in de tests wordt gestoken krijg je daar veel voor terug. Zo krijg je een werkende, levende specificatie van het product. Daarnaast krijg je vertrouwen dat wat je gemaakt hebt ook doet wat het zou moeten. Tests zijn een effectieve vorm van snelle feedback over onze code en kunnen ons versnellen. Tot slot, als allerbelangrijkste punt, dwingen tests je na te denken over je code en word je met de code geconfronteerd. Goede tests leiden tot goede code!■

ALS ER MOEITE EN DOORDACHTHEID IN DE TESTS WORDT GESTOKEN KRIJG JE DAAR VEEL VOOR TERUG.

BRONNEN:

1. **https://testing.googleblog.com/2015/04/just-say-no-to-more-end-to-end-tests.html**
2. **Hunt, Andrew; Thomas, David (1999). The Pragmatic Programmer : From Journeyman to Master**
3. **Martin, Robert C. (2007). Clean Code: A Handbook of Agile Software Craftsmanship**
4. **https://cucumber.io/docs/bdd/**
5. **https://cucumber.io/**
6. **Beck, Kent; Andres, Cynthia (2004). Extreme Programming Explained: Embrace Change, Second Edition**
7. **Martin, Robert C. (2000). Design Principles and Design Patterns**

34-38 TESTING.indd 38 13-10-20 16:16