Een onderzoek naar het testen van webapplicaties

van

Stefan de Bes (0758091)

CMI-Opleiding *Technische Informatica* – Hogeschool Rotterdam

9 januari 2014

Eerste docent *Dhr. W. Oele*
Tweede docent *Dhr. J.P. Manni*

Samenvatting

Dit afstudeerproject gaat over het ontwikkelen en implementeren van een teststrategie voor webapplicaties. Deze strategie moet voor het afstudeerbedrijf als leidraad kunnen worden gebruikt bij het opzetten van geautomatiseerd testen. Voor dit afstudeerproject is onderzoek gedaan naar de mogelijkheden van geautomatiseerd testen. Hiervoor is gekeken naar mogelijke technieken en methodes, bestaande producten, en zelf te maken producten en generieke oplossingen voor testen, die in elk project opnieuw te gebruiken zijn. Er wordt beschreven wat deze oplossingen inhouden, en op basis van de eisen en beperkingen die het afstudeerbedrijf oplegt, worden er keuzes gemaakt welke oplossingen ingezet gaan worden. Vervolgens wordt voor de gekozen oplossingen uitgewerkt hoe deze geimplementeerd kunnen worden binnen het huidige ontwikkelproces en de huidige projectopzet.

Dankbetuiging

Mijn dank gaat uit naar iedereen bij W3S, het opdrachtgevend bedrijf, voor het faciliteren van deze afstudeeropdracht en het vertrouwen in mijn conclusies. In het bijzonder wil ik Edwin van Koppen en Rutger van Schoonhoven bedanken voor de goede en geduldige begeleiding en nuttige opbouwende kritiek bij deze opdracht.

Inhoudsopgave

Inleiding

In de wereld van softwareontwikkeling is het testen een belangrijk deel van het proces. Zij het handmatig, automatisch, of met een combinatie van die twee, elke toevoeging en elke aanpassing wordt gecontroleerd op correctheid. Webdevelopment is een veld wat hier behoorlijk in achter loopt, ten opzichte van andere takken van programmeren. Webprojecten worden in het algemeen veel goedkoper verkocht dan reguliere applicaties, terwijl de complexiteit niet per definitie lager ligt.

Zo ook bij de opdrachtgever van deze afstudeeropdracht. Het testen van een oplevering betrof hier vaak een korte periode aan het einde van een project, waar een projectmanager door het resultaat klikt. De grondigheid van het testen was ook afhankelijk van hoeveel tijd er aan het einde van een project nog over was.

In de laatste jaren zijn er bij de opdrachtgever stappen genomen om het testen van functionaliteit te verbeteren, maar een concrete strategie daarachter ontbreekt. Dit komt vooral omdat er in het bedrijf niet voldoende kennis is over de mogelijkheden van testen, en de kosten en baten hiervan. Het gevolg is dat er door ontwikkelaars en projectmanagers wel tijd besteed wordt aan testen, maar dat dit vaak niet gericht genoeg gebeurt. Daardoor is het testen vrij inefficiënt en niet echt effectief. Er worden bijvoorbeeld nog te veel fouten in software gevonden na oplevering aan de klant.

Door een complete teststrategie te ontwikkelen en te implementeren, wil de opdrachtgever in de toekomst zo efficiënt mogelijk kunnen testen. De strategie zal een complete integratie in het ontwikkelproces behelzen, en de verschillende taken en verantwoordelijkheden bij het testen in kaart brengen.

Doel
Het doel van dit afstudeerproject is om in kaart te brengen wat de mogelijkheden en beperkingen zijn van geautomatiseerd testen. Er moet uitgezocht worden welke werkwijzes er voor handen zijn, en welke hulpmiddelen daarbij gebruikt kunnen worden. Daarna moet er gekeken worden welke oplossingen het beste aansluiten bij de wensen van de opdrachtgever. Vervolgens moet aangetoond worden hoe deze werkwijzen en hulpmiddelen geïmplementeerd kunnen worden in het ontwikkeltraject.

Voor dit afstudeerproject ligt de focus op het testen ter verificatie, dat wil zeggen om vast te stellen of een product werkt of niet. Security testing (het nagaan of een product afdoende beveiligd is) valt niet binnen de scope van dit project. Performance testing (het nagaan of een product snel genoeg werkt bij een bepaalde belasting) wordt in dit verslag wel meegenomen, maar

slechts als uitweiding.

Hoewel het opdracht gevende bedrijf zowel in C# als PHP ontwikkelt, wordt in dit afstudeerverslag gericht op de mogelijkheden van het testen van de C#-applicaties. Waar mogelijk zal wel gekeken worden naar een mogelijke gelijksoortige oplossing voor PHP, maar dit is puur ter vergelijking.

Projectopzet

Omdat de opdrachtgever geen duidelijk beeld heeft van de mogelijkheden van geautomatiseerd testen, zal er eerst een breed, verkennend onderzoek gedaan moeten worden naar die mogelijkheden. Vervolgens zullen de verschillende aanpakken gecategoriseerd moeten worden, met elk een overzicht van zijn sterke en zwakke punten.

Daarna moet uitgezocht worden wat er voor de opdrachtgever belangrijk is aan de te gebruiken testmethode. De wensen van de opdrachtgever moeten in kaart gebracht worden, en vergeleken worden met de eerder verzamelde informatie over de verschillende mogelijkheden. Er moet gekeken worden welke kenmerken belangrijk zijn voor de opdrachtgever, en er moeten keuzes gemaakt worden over de te gebruiken methodes en oplossingsrichtingen.

Ten slotte zullen de gekozen methodes en oplossingsrichtingen samengevoegd moeten worden tot een complete strategie, die vervolgens geimplementeerd wordt. Het is de bedoeling dat de implementatie zo goed mogelijk geintegreerd wordt in het huidige ontwikkelproces van de opdrachtgever, zowel organisatorisch als technisch.

Onderzoek

Deze fase van het project omvat het uitzoekwerk van het project. Er wordt onderzoek gedaan naar de mogelijkheden en onmogelijkheden van testen, de waarde die het toe kan voegen, en de afwegingen die gemaakt moeten worden.

Het onderzoek wordt verdeeld in drie delen: technieken en methodes, bestaande producten (applicaties en frameworks), en zelf te maken producten. De applicaties en eventuele frameworks zullen per project niet of nauwelijks aangepast worden, terwijl de technieken en methodes daarentegen in ieder project opnieuw toegepast moeten worden.

Technieken en methodes

Er zijn diverse manieren om een software-applicatie te testen. Het verschil in deze manieren zit onder andere in de mate van gebruikerstussenkomst, dekking, en effort die er voor nodig is. In dit deel van het onderzoek kijken we naar enkele veel gebruikte methodes en technieken voor het testen van webapplicaties.

De hierna volgende lijst is geen overzicht van alle methodes, maar enkel een selectie van hedendaagse methodes die voor de opdrachtgever interessant kunnen zijn. Voor een uitgewerkte lijst met voor- en nadelen van deze manieren van werken, zie bijlage B: Vergelijking methodes en technieken.

Handmatig testen
Handmatig testen is eigenlijk de simpelste variant van testen, en wordt ook wel UI testing genoemd. Bij handmatig testen gebruikt een persoon de software zoals een eindgebruiker dat zou doen. Een voordeel van handmatig testen is dat er vaak ook fouten gevonden worden waar niet specifiek op getest wordt. Nadelen zijn de benodigde mankracht en de tijdsintensiviteit.

Deze vorm van testen is al in gebruik bij de opdrachtgever. Hoewel dit afstudeerproject vooral gericht is op geautomatiseerd testen, zal er in de teststrategie ook gekeken worden naar het structureren van het handmatig testen. In de praktijk is het niet mogelijk om handmatig testen helemaal te vervangen door geautomatiseerd testen.

Automated UI testing
Automated UI testing houdt in dat een gebruiker programmatisch gesimuleerd en aangestuurd wordt. Voor een webapplicatie houdt dit in dat automatisch een browserinstantie geopend wordt, waar vervolgens diverse gebruikersinteracties door een script in uitgevoerd worden, bijvoorbeeld op een link klikken of een tekst typen.

Automated UI tests worden gebruikt om workflows te testen, zoals een registratieproces. Er kan op deze manier gecontroleerd worden of de invoer van een voorgedefiniëerde waarde leidt tot verwachte uitvoer, zoals een tekst op het scherm.

Functional testing
Functional testing is het controleren van een specifieke functionaliteit in een webapplicatie. Een extern aan te roepen functie wordt geverifieerd tegen een van tevoren opgestelde specificatie. Hiervoor is het niet altijd nodig om een browser te gebruiken, omdat de presentatie hierbij niet van belang is.

Functional testing is bijvoorbeeld handig om extern beschikbare mehodes in een API of het resultaat van een JSON-feed te controleren. Deze geven van nature output zonder opmaak.

Unit testing
Unit testing is het testen van de kleinst mogelijke eenheid van code. Als kleinste eenheid wordt vaak een enkele methode in de code gebruikt, maar indien nodig kan ook de werking van een complete klasse getest worden. Een unit test is geschreven om de te testen eenheid te isoleren, en te verifiëren op gebruikte logica.

Een unit test mag per definitie geen afhankelijkheden testen. Om dit te voorkomen, worden objecten die door de code gebruikt worden vaak vervangen door nepversies, die de werking van het echte object slechts simuleren. Om dit mogelijk te maken moet er bij het schrijven van code ook rekening mee gehouden worden dat het te testen is door middel van een unit test.

Integration testing
Integration testing kijkt naar de samenwerking tussen verschillende units, door deze te koppelen zoals dat ook in de logica van het product zelf gaat. Integration testing gaat er over het algemeen vanuit dat er ook al Unit Testing gedaan is.

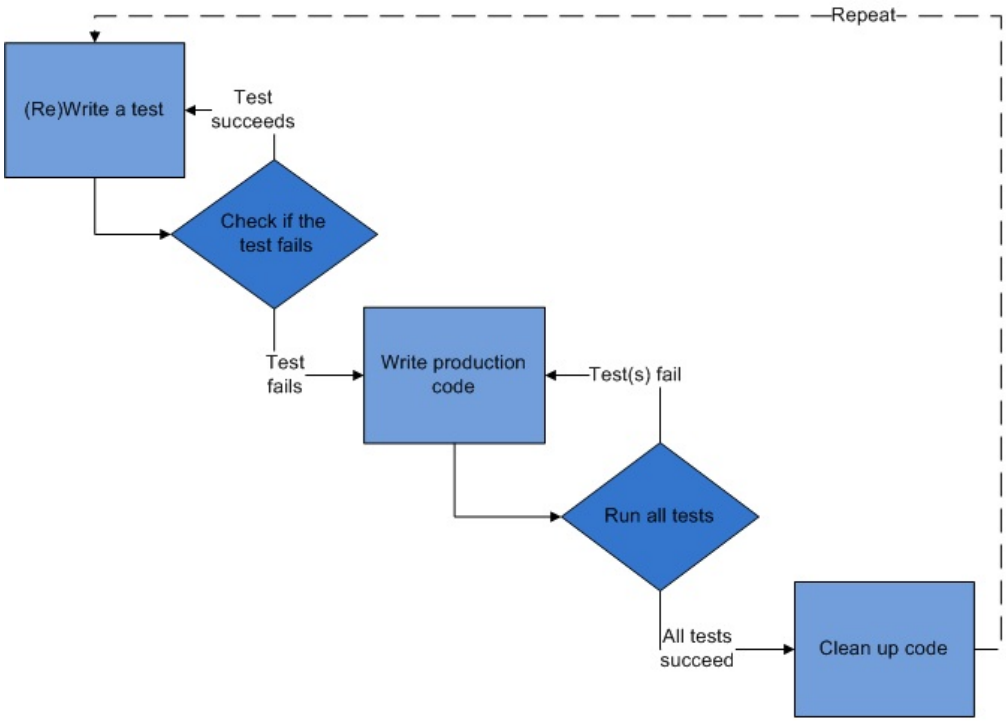
Het verschil tussen integration testing en functional testing is dat een integration test gebruik moet kunnen maken van de interne werking van een product, bijvoorbeeld om direct instanties van objecten te maken, terwijl een functionele test de applicatie benadert via de buitenkant, bijvoorbeeld door het aanroepen van een URL binnen de webapplicatie.

TDD
Test-driven Development is een ontwikkelproces waarbij eerst geautomatiseerde tests geschreven worden, en daarna pas de applicatiecode. Door de test telkens iets specifieker te maken, moet de code steeds dichter bij het gewenste eindresultaat uit komen.

De cyclus van TDD begint met het schrijven of aanpassen van een test. In TDD wordt nooit code geschreven waar nog geen test voor is; pas als een test faalt, wordt er extra code geschreven om te zorgen dat de code voldoet aan de nieuwe eisen van de test. Zodra dit het geval is, en de test slaagt, begint de cyclus opnieuw, net zolang tot de test alle voorwaarden controleert en de code hieraan voldoet.

BDD
Behavior Driven Development is een uitbreiding op Test-driven Development, waarbij de specificaties van het eindproduct één op één vertaald worden naar tests. Er kan dus heel concreet gecontroleerd worden of een applicatie aan de opgestelde specificaties voldoet.

BDD leunt meestal zwaar op een framework binnen de ontwikkelomgeving, wat er voor zorgt



Figuur 1: Een schematische weergave van het proces van Test Driven Development. De cyclus gaat door tot het product af is.

dat specificaties direct kunnen worden vertaald naar tests. Het voordeel van BDD is dat ook niet-technische betrokkenen zonder al te veel technische kennis de specificaties (en dus de tests) kunnen lezen, en met wat oefening deze zelfs op kunnen stellen.

Bestaande producten

Er is een grote verscheidenheid aan softwareproducten voor automatisch testen te vinden, variërend van frameworks waarbij de programmeur alles zelf doet tot testsuites die geen tussenkomst van een programmeur vereisen. De best bekende producten hebben we onderzocht op diverse criteria, om te kijken of deze voor de opdrachtgever bruikbaar zijn.

De criteria voor de beoordeling van producten zijn integratiemogelijkheid (de mogelijkheid tot koppelen aan andere software en inzet binnen reeds lopende processen), leercurve, ondersteuning, kosten, en overige van toepassing zijnde plus- en minpunten. Sommige producten zijn niet op alle criteria beoordeeld, omdat er een specifiek minpunt aan zat of omdat er bij een eerder criterium al bleek dat het product niet geschikt was voor gebruik binnen het opdracht gevende bedrijf. Hieronder volgt een beschrijving van de producten die voor de opdrachtgever daadwerkelijk interessant zijn. Zie voor een complete lijst van alle onderzochte producten bijlage C: productvergelijking.

xUnit
De xUnit-frameworks zijn een familie van strak opgezette API's voor diverse programmeertalen,

waaronder C# (nUnit) en PHP (PhpUnit), de talen waar de opdrachtgever in werkt. Met een dergelijk framework heeft de programmeur simpele handvatten om geautomatiseerde tests te schrijven in de programmeertaal die hij kent, in de IDE die hij gewend is, en binnen het project waar de test op van toepassing is. Het framework biedt een keur aan functies om te controleren of aan voorwaardes binnen de code voldaan wordt, en zorgt ervoor dat tests altijd in een vaste vorm geschreven worden.

WatiN
WatiN is een tool en framework voor automated UI testing. Dat houdt in dat er geautomatiseerd een browserscherm geopend wordt, gebruiksinstructies worden verwerkt (klik op een knop, typ een tekst, ga met de muis over een element) en gecontroleerd kan worden of de pagina aan bepaalde voorwaarden voldoet (bijvoorbeeld of er een specifieke tekst zichtbaar is).

WatiN bestaat zoals gezegd uit twee delen: een C#(.NET)-API en achterliggende programmatuur, wat het mogelijk maakt om programmatisch een browserscherm te openen en aan te sturen, en een macrotool, wat het mogelijk maakt om gebruikersgedrag op te nemen en om te zetten naar code die deze API aanroept. WatiN heeft goede omzetting van gebruikersgedrag naar code, maar alleen ondersteuning voor C#, en kan slechts de browsers Internet Explorer en Firefox aansturen.

Selenium
Selenium is enigzins vergelijkbaar met WatiN, eveneens een combinatie van een API en een macrotool. Bij Selenium zijn de onderdelen minder geïntegreerd, diverse onderdelen zijn apart te installeren. Voor Selenium is het ook mogelijk om door middel van Selenium Grid de UI tests gedistribueerd uit te laten voeren, bijvoorbeeld om meerdere browserinstanties tegelijk te laten werken.

De API is beschikbaar voor diverse programmeertalen, waaronder PHP en C#. De syntax is makkelijk aan te leren, en er is brede ondersteuning voor. De achterliggende programmatuur die de browser aanstuurt, Selenium RC genaamd, kan de recentste versies van de vijf populairste browsers aansturen.

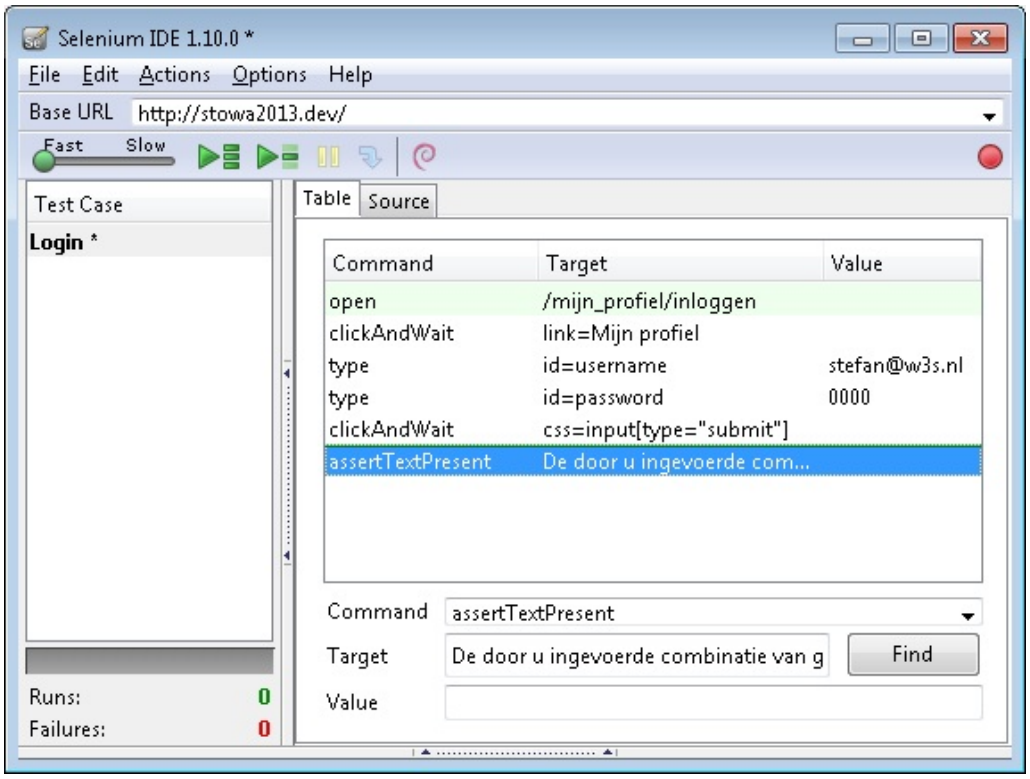
De macrotool, Selenium IDE, werkt als een Firefox-plugin. Deze kan code exporteren naar een verscheidenheid aan programmeertalen, waaronder PHP en C#, maar in de praktijk blijkt dat niet elke gebruikershandeling correct omgezet wordt naar code.

JMeter
JMeter is een website benchmarking suite die geautomatiseerd webrequests op een webpaginakan doen, en vervolgens statistieken over de response terug geeft. Dit geeft een beeld van de reactietijd van een website, en bij grotere hoeveelheden gelijktijdige requests kan dit voor stresstesting gebruikt worden.

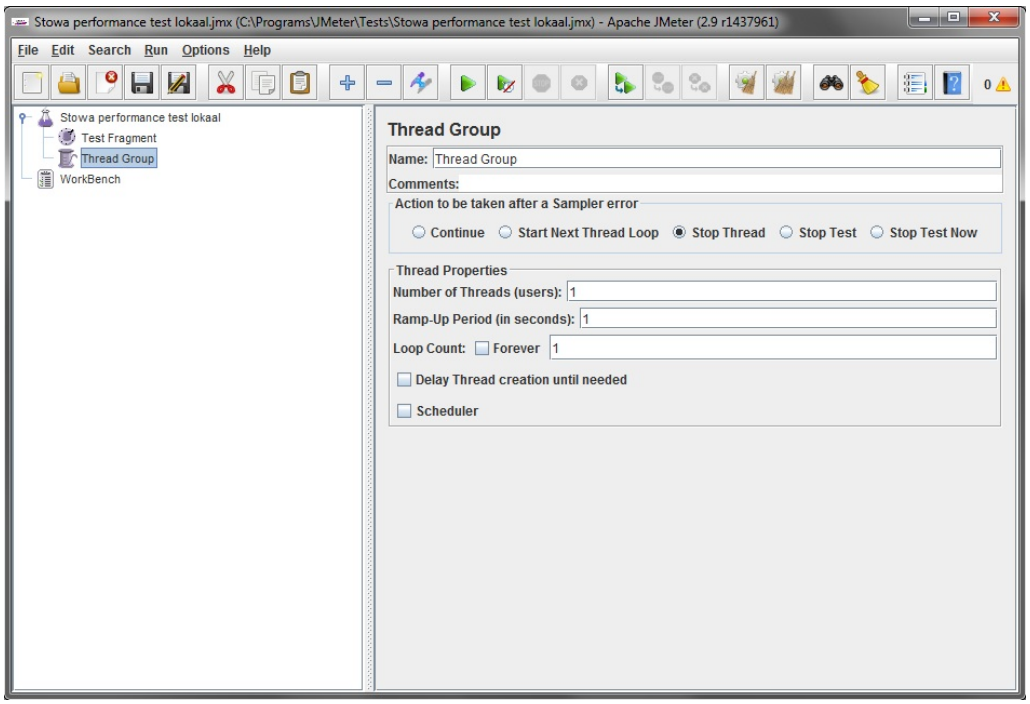
JMeter is de bekendste en best ondersteunde stress testing tool, maar heeft een erg uitgebreide interface en veel configuratie-opties, waardoor het lastig is om er goed gebruik van te maken.

Zelf te maken producten

Naast de reeds beschikbare producten zijn er ook mogelijkheden om zelf iets te maken wat het testen makkelijker maakt. Bestaande producten bieden niet altijd een passende oplossing voor de projectstructuur en manier van werken van de opdrachtgever. Alles wat herbruikbaar is, hoeft bovendien niet per project opnieuw gedaan te worden.



Figuur 2: Selenium IDE wordt gebruikt voor het opnemen van gebruikersinteractie als een macro, welke vervolgens geëxporteerd kan worden naar code.



Figuur 3: JMeter is een zeer uitgebreide benchmarktool, maar heeft een steile leercurve door de uitgebreide interface.

De volgende producten zouden een handige uitbreiding zijn op de bestaande, en zijn binnen een redelijke termijn te realiseren. Sommige van deze ideeën omvatten ook oplossingen voor problemen die buiten dit afstudeerproject vallen.

Test runner
Om de automatische tests ook buiten een ontwikkelomgeving te draaien (dus bijvoorbeeld bij het deployen van een nieuwe versie) zal een test runner gemaakt moeten worden. Deze kan aangeroepen worden door bestaande scripts, en kan bijvoorbeeld geautomatiseerd alle tests die in een project aanwezig zijn, uitvoeren. De resultaten hiervan zullen inzichtelijk gemaakt moeten worden voor klant of projectmanager, bijvoorbeeld via email.

Om geautomatiseerd te achterhalen welke tests bij een project horen, zal een programma geschreven moeten worden dat het gecompileerde project kan doorzoeken op tests. Zie voor een voorbeeld hiervan in C# bijlage 4: proof of concept: test runner.

Test dashboard
Om een klant de resultaten van geautomatiseerde tests aan een klant te presenteren, kan een dashboard ontwikkeld worden. Hier worden dan in een afgeschermde omgeving de resultaten van de laatst gedraaide verzameling tests getoond. Ook kan hier aanvullende data over een project getoond worden, zoals uptimestatistieken en deploymentinformatie.

Crawler
Om zonder al te veel moeite te achterhalen of een website dode links bevat, of pagina's die fouten geven, kan een site automatisch doorgelopen worden. Er wordt begonnen met de homepage, en als deze een geldig resultaat geeft wordt gekeken naar welke andere pagina's er gelinkt wordt. Voor die pagina's gebeurt hetzelfde, net zolang tot alle links binnen de website bezocht zijn. Een dergelijk systeem wordt een crawler genoemd.

Voor een voorbeeld van hoe deze crawler intern zou moeten werken, zie bijlage 5: proof of concept: crawler.

Test framework
Om het voor ontwikkelaars makkelijker te maken om tests te schrijven bij hun code, is het verstandig om ze hier enkele handvatten voor te geven. Een test framework is dan de aangewezen oplossing. Dit kan uiteenlopen van enkele hulpfuncties tot een compleet uitbreidbare structuur waar ze in een vast format hun tests kunnen maken.

Bestaande testframeworks bieden al veel mogelijkheden om tests goed op te zetten en te controleren of aan specifieke voorwaardes voldaan wordt. Het eigen framework zou hier op door kunnen bouwen, door een standaard testformat aan te bieden, of functies die veel gebruikte constructies doorrekenen.

Keuzes

In deze fase gaan we de eisen en beperkingen, die de opdrachtgever aan de teststrategie stelt, in kaart brengen. Zodra hier een duidelijk beeld van is, gaan we deze voorwaarden tegen de mogelijke oplossingen aan houden, zodat duidelijk wordt welke oplossingen voldoen aan de eisen van de opdrachtgever.

Zodra duidelijk is wat de opdrachtgever hoopt te bereiken met het testen, en met welke beperkingen er hierbij rekening moet worden gehouden, wordt een keuze gemaakt uit de methodes en producten die uit het onderzoek naar voren zijn gekomen. Voor deze oplossingen zullen de mogelijkheden en vereisten verder uitgewerkt worden.

Doelen van het testen

Tijdens het onderzoek is vaak overleg geweest met het hoofd Operations, over de doelen van het testen vast te stellen. Naarmate de kennis over het onderwerp vorderde, konden de doelen steeds duidelijker gespecificeerd worden.

Het voornaamste doel van testen is voor de opdrachtgever het achterhalen of eerder gebouwde functionaliteiten na doorontwikkeling nog steeds werken. Een dergelijke test heet een regressietest; er wordt in feite gekeken of een ontwikkelaar bij het maken van iets nieuws geen bestaande functionaliteit stuk gemaakt heeft.

De opdrachtgever wil met name weten of workflows in een webapplicatie, zoals een formulier invullen of een gebruiker aanmaken, nog steeds naar behoren werken. Ook als een programmeur een dergelijke functionaliteit niet bewust aangepast heeft, kan het zijn dat de werking of presentatie hiervan toch veranderd is. Dit wordt vaak veroorzaakt door een gezamenlijke basis voor functionaliteiten, of gedeelde bronnen zoals scripts en style sheets.

Omdat zowel werking als presentatie getest moeten worden, zal dit via een browser moeten gebeuren. Daar het volledig handmatig doorlopen van alle workflows en formulieren in een webapplicatie te veel werk is om bij elke aanpassing uit te voeren, is automated UI testing hier de gepaste oplossing.

Een ander doel van het testen is het achterhalen van performance in productieomgeving. Een ontwikkelaar bouwt een deel van een applicatie op een afgeschermd systeem, waar er slechts één gebruiker tegelijk gebruik van maakt, en meestal met een kleine hoeveelheid testdata. In de productieomgeving is het echter gebruikelijk dat een groot aantal gebruikers tegelijkertijd de applicatie gebruikt, en er relatief grote datasets verwerkt worden.

Om na te gaan of een applicatie in een productieomgeving nog steeds goed presteert, is het wenselijk om te testen op prestaties en schaalbaarheid van functionaliteiten. De wens is om te zorgen dat niet pas in een productieomgeving blijkt dat een applicatie of functionaliteit te langzaam is, maar dit eerder te signaleren, en indien nodig te optimaliseren of versimpelen.

Ten slotte komt de wens van de opdrachtgever om te testen simpelweg ook voort uit de vraag hiernaar van klanten. Steeds vaker vragen klanten wat de opdrachtgever doet om vast te stellen dat een functionaliteit daadwerkelijk werkt zoals gespecificeerd. De klant wil inzicht krijgen in de criteria waarop een functionaliteit beoordeeld wordt, en of de functionaliteit hier ook aan voldoet.

Zodoende is het wenselijk om op een vast punt binnen het ontwikkelproces een resultaat van gedane tests aan een klant te presenteren. Een logisch moment hiervoor is een deployment van de webapplicatie op de acceptatieomgeving.

Randvoorwaarden bij het testen

De belangrijkste voorwaarde bij het uitbreiden van het testen is dat de tijd die hiervoor nodig is, in verhouding staat tot de tijd die voor ontwikkeling van het product nodig is. Het betreft hier dan vooral de tijd binnen een project; de voor meerdere projecten herbruikbare oplossingen vallen buiten deze beperking.

De opdrachtgever wil maximaal 20 procent van de tijd die programmeurs in een project steken, besteden aan het ontwikkelen van geautomatiseerde tests. Het is voor de opdrachtgever belangrijk dat de tijd die beschikbaar is voor het schrijven van tests efficiënt gebruikt wordt.

Uit analyse van worklogs van de opdrachtgever blijkt, dat gemiddeld 10 procent van de benodigde ontwikkeltijd wordt besteed aan het oplossen van bugs die zich voor doen in een productieomgeving. Door dubbel deze tijd te gebruiken voor het implementeren van tests, hoopt de opdrachtgever dit tot een minimum te beperken.

Tevens stelt de opdrachtgever de eis dat een testoplossing in grote lijnen werkt voor zowel de PHP- als de C#-projecten. Op beide platforms moet een redelijk gelijke kwaliteit van testen mogelijk gemaakt worden. Zodoende moet er gekozen worden voor oplossingen die op beide platforms op een gelijke implementatie kunnen rekenen.

Hierbij wordt wel de kanttekening geplaatst, dat specifieke implementaties zeer goed eerst op het ene platform uitgeprobeerd kunnen worden, alvorens deze ook op het andere platform door te voeren. Ook wordt er rekening gehouden met de specifieke kenmerken en beperkingen van beide platformen.

De gekozen oplossingen

Om tot een keuze te komen voor de uiteindelijk te implementeren oplossingen, zijn deze door de afstudeerstudent gepresenteerd aan een taakgroep. Deze taakgroep bestaat uit het hoofd Operations en Quality Assurance, de Lead Developer, en een afgevaardigde van het PHP-team.

Na alle mogelijkheden uitgelegd te hebben, zijn deze één voor één bekeken en beoordeeld aan de hand van de eisen en voorwaarden die de opdrachtgever stelt. Hierna heeft de afstudeerstudent

aanbevelingen gedaan over welke oplossingen wel en niet bij de doelen van de opdrachtgever passen. Na overlegbinnen de taakgroep is er vervolgens een keuze gemaakt voor de volgende oplossingen.

Omdat geautomatiseerde tests nooit een volledige dekking kunnen bieden, is en blijft handmatig testen een noodzaak voor het vaststellen van de kwaliteit van een webapplicatie. De wens is wel dat dit gestructureerder gaat gebeuren. Er moeten afspraken gemaakt worden over verantwoordelijkheden binnen teams bij het handmatig testen.

De opdrachtgever wil automated UI testing in gaan zetten voor het controleren van workflows en formulieren. Er is hierbij gekozen voor Selenium als framework, omdat de ondersteuning hiervoor het beste is. Bij elk formulier wat ontwikkeld wordt, zal ook een macro opgenomen worden waarin dit formulier gebruikt wordt. Deze macro wordt omgezet naar een Selenium-script, welke vervolgens aan het project toegevoegd kan worden.

Voor het testen van feeds en API's wil de opdrachtgever functionele tests schrijven, gebaseerd op nUnit of phpUnit. Het is wenselijk om standaard testfuncties en helpers toe te voegen aan de basisprojecten, om het programmeurs makkelijker te maken. Uiteindelijk kan dit uitgroeien tot een eigen testframework.

Om de geautomatiseerde tests uit te voeren op een vast punt in het ontwikkelproces, moet een testrunner ontwikkeld worden die geïntegreerd kan worden in de bestaande build- en deploymentscripts. Deze dient de resultaten van de uitgevoerde tests te verwerken en op te slaan, en eventueel direct via email versturen naar een projectmanager of klant.

De resultaten van de geautomatiseerde tests moeten overzichtelijk gepresenteerd kunnen worden aan de klant. De keuze voor presentatie in een webomgeving is voor de hand liggend, gezien de ervaring hiermee, en de beschikbaarheid van een dergelijke oplossing. Er moet een testdashboard ontwikkeld worden, waar klanten een duidelijk overzicht van de testresultaten van hun projecten krijgen. Eventueel moet dit geïntegreerd worden in een bestaande omgeving, zoals de klantenportal of het CMS.

Unit tests zullen niet structureel gebruikt worden, maar incidenteel ingezet worden om de meest complexe functionaliteiten te testen. De tijdsinvestering voor het structureel inzetten wordt te groot geacht ten opzichte van de waarde die unit tests toevoegen.

Gezien het feit dat er momenteel geen bereidheid is om structureel unit tests te schrijven, zijn methodes als TDD en BDD op dit moment ook niet inzetbaar voor de opdrachtgever.

Voor performance testing wil de opdrachtgever gebruik gaan maken van JMeter. Er moet per project beoordeeld worden, in overleg met de klant, op welke schaal een functionaliteit gebruikt moet kunnen worden. Ook moet per applicatie vastgesteld worden wat de maximaal acceptabele responsetijd is. Zodra hier concrete cijfers voor vastgesteld zijn, moeten deze bij het project opgeslagen worden, en later automatisch in JMeter ingeladen kunnen worden.

Implementatie

In de laatste fase wordt een plan gemaakt voor het implementeren van de gekozen oplossingen. Voor elk van de gekozen producten en technieken wordt beschreven hoe deze ingezet moet worden binnen het testproces. De uitkomst van deze fase vormt de basis voor de teststrategie van de opdrachtgever.

De manier van integreren is natuurlijk voor alle gekozen methodes en producten verschillend, elke oplossing moet op een eigen punt binnen het proces zijn plek krijgen. We bekijken daarom per aspect van het totale traject welke producten geïmplementeerd moeten worden, en hoe.

Integratie in het ontwikkelproces

Het grootste deel van de implementatie komt natuurlijk te liggen binnen het ontwikkelproces. Bij het beginnen van de bouw van nieuwe functionaliteit zal een inschatting gemaakt moeten worden wat er getest moet worden, en hoe. Programmeurs moeten hier op gecoacht worden.

De beste optie is om in een nieuw project te beginnen met geautomatiseerd testen. Er is dan geen kans op legacyproblemen waardoor tests niet geïmplementeerd kunnen worden, of niet betrouwbaar werken. Ook kan dan in de opzet rekening gehouden worden met het feit dat de code automatisch getest wordt.

Bij aanvang van een project zullen de benodigde libraries, zoals de Selenium runtime en nUnit / phpUnit runtime, in het project toegevoegd moeten worden. Later kan dit in de standaardprojecten van de opdrachtgever worden opgenomen, zodat programmeurs dit niet zelf hoeven te doen.

Voordat het daadwerkelijke programmeren begint, zal een lijst gemaakt moeten worden van de functionaliteiten waarvoor tests gemaakt moeten worden. In eerste instantie zal dit beperkt zijn tot formulieren, workflows, JSON- en AJAX-templates. Door in te schatten wat getest moet worden, wordt ook beter nagedacht over wat de functionaliteit uiteindelijk moet doen.

Na het ontwikkelen van een formulier of workflow kan de programmeur deze doorlopen met de Selenium IDE browserplugin ingeschakeld. Er wordt dan een macro opgenomen van bijvoorbeeld het invullen van het formulier. Deze macro kan geëxporteerd worden naar een C#-script, en automatisch in de ontwikkelomgeving worden ingeladen. Hier kan de programmeur de testcode eventueel nog handmatig bewerken. Zodra de test in het project staat, moet de programmeur deze ook uitvoeren, om na te gaan of deze slaagt. Is dat het geval, dan kunnen zowel functionaliteit als test ingecheckt worden in het source control-systeem.

Zie voor een voorbeeld van een dergelijke test bijlage G: Gebruiksvoorbeeld automated UI test.

Na het ontwikkelen van een JSON- of AJAX-template moet een programmeur hier een test bij schrijven. Deze test zal over het algemeen een webrequest doen naar een URL binnen de webapplicatie in kwestie, en de response hiervan zal geparsed moeten worden om te kijken of aan bepaalde voorwaarden voldaan wordt. Zodra de test slaagt, kan ook deze functionaliteit en bijbehorende test in source control ingecheckt worden.

Voor een voorbeeld voor het inzetten van geautomatiseerde functionele tests, zie bijlage H: Gebruiksvoorbeeld automated functional test.

In het geval dat een programmeur code schrijft die hij of zij zelf vrij complex acht, of waarvan de uitkomst niet voldoende transparant is, kan in overleg met het hoofd development besloten worden hier een unit test voor te schrijven. Dit zorgt er ook voor dat de complexe code voldoet aan principes als Single Responsibility (het ontwerpprincipe dat stelt dat een enkele functie slechts één taak mag hebben) en Separation of Concerns (een verwant ontwerpprincipe dat zorgt voor modulaire opbouw van applicaties).

In bijlage I: Gebruiksvoorbeeld unit test wordt een voorbeeld gegeven van hoe de ontwikkelaars van de opdrachtgever dergelijke unit tests in projecten kunnen implementeren.

Voor functionaliteiten en pagina's binnen een webapplicatie die zwaarder dan normaal zijn (queries met veel joins, of aanroepen van een externe webservice) moeten performancecriteria opgesteld worden. Er moet ingeschat worden hoeveel gebruikers tegelijk de bijbehorende pagina aan moeten kunnen roepen, en wat de maximale laadtijd van de pagina dan mag zijn.

Eveneens moet voor functionaliteiten waarbij grote hoeveelheden data verwacht kunnen worden, een script geschreven worden voor het genereren (en later weer verwijderen) van testdata. In de frontend moet er rekening mee gehouden worden dat dergelijke testcontent nooit in productieomgeving getoond wordt.

Na het doorlopen van één of enkele projecten waarbij automatisch getest wordt, kunnen de tests geanalyseerd worden op veel gebruikte constructies. Hiervoor kunnen helpers of sjablonen geschreven worden, die deze manier van werken makkelijker maken. Ook moet er aan programmeurs gevraagd worden wat ze handig of bruikbaar zouden vinden bij het schrijven van automatische tests. Het heeft geen nut om een framework te creëren zonder te weten wat deze precies moet kunnen.

Integratie in het deploymentproces

Het voornaamste bij het integreren van programmatische tests in het deploymentproces is het verder ontwikkelen van de proof of concept voor de test runner (zie bijlage D). Deze zal zowel UI tests als functionele tests en eventuele unit tests automatisch moeten kunnen uitvoeren.

De test runner moet te runnen zijn vanaf een commandline, en moet commandlineparameters kunnen verwerken die onder andere aangeven welk base URL gebruikt wordt (om aan te geven of de tests moeten draaien op de productie- of acceptatieomgeving), en hoe de resultaten gepresenteerd moeten worden. Opties hiervoor zijn commandline output, versturen via mail, en opslaan in een database.

Voor performance testing kan danwel automatisch na deployment, danwel handmatig, een JMeter-instantie gestart worden met een configuratiebestand specifiek voor het zojuist gedeployde project. Zowel de testparameters als de acceptatiecriteria zijn in overleg met de klant vastgesteld.

In verband met de tijd die performance testing kost, zal dit niet na iedere deployment uitgevoerd worden. Voor de meeste functionaliteiten en aanpassingen zal dit ook niet relevant zijn. Dit moet van tevoren per functionaliteit beoordeeld worden. Als een functionaliteit door het verantwoordelijke team behoorlijk zwaarder dan gemiddeld wordt geacht, moet gekozen worden om hier performance tests op los te laten.

Presentatie naar de klant

Het test dashboard moet geïntegreerd worden in een reeds in ontwikkeling zijnde portal waarin uptime-data voor de diverse projecten van een klant getoond wordt. In een apart tabblad binnen deze applicatie moet een menu te vinden zijn, waardoor de tijdstippen staan waarop de tests gerund zijn. Als op een tijdstip geklikt wordt, moet er getoond worden welke tests in die testrun geslaagd zijn, en welke niet.

Daarnaast is er een teststrategie geschreven: een document voor klanten, wat beschrijft hoe de opdrachtgever omgaat met testen, wat de verschillende varianten van geautomatiseerd testen inhouden, en hoe dit geïmplementeerd wordt. Het is de bedoeling dat dit de bewustwording van klanten over dit onderwerp vergroot, en ervoor zorgt dat klanten bijvoorbeeld begrijpen wat wel en niet mogelijk is met automatisch testen.

Dit document is bijgevoegd in bijlage F: Teststrategie.

Conclusies en aanbevelingen

Het product van dit afstudeertraject is een opzet voor het schrijven van programmatische tests van onze webapplicaties, en een leidraad voor het hierbij gebruiken van de gekozen hulpmiddelen. Verder kunnen geschreven tests automatisch uitgevoerd worden tijdens het build- en deployment-proces, waardoor het testen met succes in het ontwikkeltraject geïntegreerd is.

Er zijn beargumenteerde keuzes gemaakt voor specifieke oplossingen, op basis van het onderzoek in dit afstudeerproject, en voor al deze oplossingen is uitgewerkt hoe deze geïmplementeerd kunnen worden.

De gebruiksvoorbeelden voor de diverse types tests tonen aan dat geautomatiseerde tests met succes ingezet kunnen worden in de projecten van de opdrachtgever. Verder is het mogelijk om deze tests buiten een ontwikkelomgeving (bijvoorbeeld tijdens deployment) te laten uitvoeren, door middel van de ontwikkelde Test Runner.

Hoewel de eigen Test Runner nog niet uitontwikkeld is, en de implementatie van de gekozen oplossingen nog loopt, kan al wel begonnen worden met het schrijven van tests. Er kan stapsgewijs begonnen worden met het coachen van programmeurs voor het schrijven van tests, waarbij uitleg van het doel, goede begeleiding en verzamelen van feedback centraal staan.

De ontwikkelde teststrategie zorgt ervoor dat geautomatiseerd testen een centrale plaats krijgt binnen het ontwikkeltraject van de opdrachtgever. Er is een solide basis gelegd voor het creëren en uitvoeren van geautomatiseerde tests voor de C#-webapplicaties.

Hoewel de implementatie op korte termijn geslaagd is, moet deze na een periode van gebruik wel geëvalueerd worden. Of de teststrategie op langere termijn ook het gewenste resultaat heeft, dat wil zeggen, de kwaliteit van de producten ook daadwerkelijk ten goede komt, is pas te controleren nadat deze enige tijd in gebruik is. Het is daarom van belang om na elke oplevering in de komende twee maanden, die de nieuwe teststrategie gebruikt, een kleine evaluatie te doen. Na die twee maanden kunnen die evaluaties samengevoegd worden, om te controleren of de strategie het gewenste resultaat heeft.

Hiervoor kunnen we ook kijken naar het gemiddelde aantal gemelde defects, of de totale tijd die na release besteed is aan het oplossen van bugs. Door dit te vergelijken met een oplevering van voor de invoering van de teststrategie kunnen we controleren of het aantal bugs in de geproduceerde code daadwerkelijk is afgenomen. Het is belangrijk om dit doel niet uit het oog te verliezen.

Hoewel functionele tests kunnen verificeren of een functionaliteit aan de buitenkant naar behoren werkt, kunnen ze natuurlijk geen fouten in logica aan het licht brengen. Nu de keuze is gemaakt om slechts incidenteel unit tests in te zetten, is het interessant om nog een

mogelijkheid te hebben om code te valideren: door code reviews. Een simpele blik van een collega kan er vaak voor zorgen dat logicafouten al verholpen worden voor de code in het source control-systeem komt. Het enige wat hiervoor nodig is, is coaching hiervoor aan de ontwikkelaars.

Wellicht is het voor de opdrachtgever interessant om over een jaar nog eens te kijken naar de mogelijkheden van unit testing in het algemeen, en Test Driven Development in het bijzonder. De structuur die een dergelijke ontwikkelvorm biedt, kan de kwaliteit van de producten van de opdrachtgever mijns inziens naar een hoger niveau tillen.

Bronnen

[1] McWherther J., Hall B.: *Testing ASP.NET Web Applications*
Wiley Publishing, Inc., Indianapolis, Indiana, 2010
ISBN: 978-0-470-49664-0

[2] Nguyen H.Q., Johnson B., Hackett M.: *Testing Applications on the Web*
Wiley Publishing, Inc., Indianapolis, Indiana, 2003
Second Edition
ISBN: 978-0-471-20100-7

[3] Kaner C., Bach J., Pettichord B.: *Lessons Learned in Software Testing*
John Wiley & Sons, Inc., New York City, New York, 2002
ISBN: 978-0-471-08112-8

[4] MSDN .NET Framework 4.5 Class Library
http://msdn.microsoft.com/en-us/library/gg145045.aspx

[5] The Three Rules of TDD - Robert C. Martin
http://butunclebob.com/ArticleS.UncleBob.TheThreeRulesOfTdd

[6] Wikipedia, the free encyclopedia
http://en.wikipedia.org/wiki/Main_Page

Evaluatie

Ik heb veel geleerd van het onderzoek naar testen. Het verbaasde me in de loop van het project, hoeveel er met testen te doen valt, en hoe weinig ik daar over wist. In mijn ervaring is dit onderdeel van software-ontwikkeling tijdens de opleiding veel te onderbelicht gebleven. Ik zou de Hogeschool Rotterdam dan ook aanraden om hier meer tijd aan te besteden. Rond een onderwerp als Test Driven Development zou prima een vak gevormd kunnen worden.

Het viel me tegen hoeveel tijd het kost om zaken goed uit te zoeken. Simpelweg het correct integreren van Selenium-code in nUnit-tests heeft me twee dagen gekost, waar ik geschat had dat dit direct had moeten werken. Ook het theoretisch onderzoek heeft langer geduurd dan de bedoeling was, omdat ik te breed aan het zoeken was. Door het ontbreken van basiskennis was het lastig om specifiek te zoeken.

Het schrijven van het afstudeerverslag heb ik ervaren als een zeer zware opgave. Dit heeft er onder andere mee te maken dat ik al een aardige tijd fulltime aan het werk ben, en het schrijven van verslagen niet meer gewend ben. Het kostte vaak veel moeite en tijd om zelfs de simpelste dingen op papier te zetten.

Vooral het verwerken van de feedback bleek lastig, omdat ik hierbij tegen een soort writer's block aan liep. Ook is het lastig de structuur van een onderzoek te veranderen, nadat het uitgevoerd is.

Bijlage A

Woordenlijst

Om verzekerd te zijn van duidelijke interne communicatie met betrekking tot een onderwerp waar niet iedereen even veel kennis van heeft, is er een simpele woordenlijst opgesteld met veelgebruikte termen. Zo wordt er voor gezorgd dat termen in de juiste context gebruikt worden, en er dus geen misverstanden ontstaan.

Als basis voor deze lijst dient de paragraaf 'Testing terminology' van hoofdstuk 1 van het boek 'Testing ASP.NET Web Applications'. Deze lijst zal aangevuld worden, indien nodig, naarmate het project vordert.

Test
Een systematische procedure om vast te stellen dat een specifiek deel van een applicatie correct werkt.

Unit test
Een methode om vast te stellen dat een kleinste eenheid van een applicatie correct werkt. Als kleinste eenheid wordt vaak een enkele methode in de code gebruikt.

Functionele test
Een functionele test controleert of een deel van een applicatie (meer dan één methode) voldoet aan een enkele requirement. Wordt dus getest per functionaliteit.

Regressietest
Een manier om te controleren of bestaande functionaliteit na een aanpassing nog steeds correct werkt.

Mock class
Een klasse die slechts gebruikt wordt binnen de geautomatiseerde tests, meestal binnen unit tests. Een mock class heeft hetzelfde uiterlijk als een klasse die in productie gebruikt wordt, maar heeft geen afhankelijkheden, zoals een database waar informatie uit gehaald moet worden, en vaak geen interne verwerkingslogica.

Bijlage B

Vergelijking methodes en technieken

Bij het kiezen van de te gebruiken methodes en technieken voor het geautomatiseerd testen, kijken we naar specifieke eigenschappen. De criteria die hierbij gebruikt worden, zijn zeer afhankelijk van het doel en de omvangrijkheid van de te beoordelen techniek of methode, en als zodanig niet vooraf te benoemen.

Performance testing
Performance testing moet grotendeels los gezien worden van alle andere vormen van testen, omdat het doel in essentie anders is dan bij andere vormen van testen. Performance testing kijkt er niet naar of iets goed gaat, maar hoe lang het duurt.

Het grootste nadeel van performance testing is dan ook gelijk dat het in de basis niet slaagt of faalt. Welke resultaten wel en niet acceptabel zijn, moet door de tester worden vastgesteld. Vaak verschilt dit per project. Dit is een belangrijk punt om in gedachten te houden bij het implementeren van performance tests.

Performance testing zorgt wel voor resultaten die anders niet snel aan het licht zouden komen. Problemen met performance van een webapplicatie doen zich vaak voor bij zwaar gebruik, of bij grote hoeveelheden data. In beide scenario's is het vaak te laat om het probleem correct op te lossen. Als een site bij publiciteit down gaat door performanceproblemen, is de schade al geleden. Bij grote hoeveelheden data is het vaak lastig om een oplossing te bieden die de data intact laat.

UI testing
UI testing simuleert het gebruik van een webapplicatie door een browser, en komt dus het dichtste bij de activiteit van een echte gebruiker. Een User Interface-test controleert alle lagen van een applicatie, dus de gebruikersinterface plus alle afhankelijkheden. Dit heeft als voordeel dat er veel tegelijk getest wordt, maar dat heeft weer als nadeel dat een niet geslaagde test geen duidelijkheid geeft over wat er fout ging.

UI testing is van nature omslachtig en langzaam in de uitvoering, door de overhead van een user interface, bijvoorbeeld het instantiëren van een browser.

Door propriëtaire, productgebonden syntax zorgen UI testing tools vaak voor slechte onderhoudbaarheid van tests. Het is vaak makkelijk om een test te exporteren naar een programmeertaal naar keuze, maar het aanpassen van deze code vereist intieme kennis van het gebruikte framework.

Functional testing
Functional testing is ideaal voor het verifiëren van data-integriteit, bijvoorbeeld bij feeds en API-calls. Daar komt bij dat functional testing geen rekening hoeft te houden met een presentatielaag, waardoor data ook veel beter te controleren valt.

Een voordeel én nadeel van functional testing is dat het zowel top-level functies (van buitenaf beschikbaar) plus alle afhankelijkheden test. Dit zorgt ervoor dat er veel tegelijk getest kan worden, maar leidt er ook toe dat de oorzaak van een gefaalde test minder makkelijk te achterhalen is.

Unit testing
Unit tests hebben daarentegen helemaal niets te maken met dependencies, wat ervoor zorgt dat een gefaalde test relatief precies aangeeft waar het probleem zit. Een goede unit test controleert in principe slechts één functionaliteit.

Het gebruiken van unit tests zorgt er ook voor dat programmeurs volgens verschillende programmeerprincipes te werk gaan. Als een functie bijvoorbeeld meer dan één taak heeft, zal dit al blijken bij het geven van een naam aan de unit test.

Een nadeel van structureel unit testen is de tijdsinvestering. Het kost relatief veel tijd om code te schrijven die te unit-testen is, en levert op zich weinig op. Een enkele unit test biedt weinig garanties dat het eindproduct ook echt doet wat het moet doen.

Integration testing
Een integration test zegt al meer over de uiteindelijke correctheid van een functionaliteit. Door de samenwerking tussen blokken te controleren, komen veel problemen in logica boven water.

Een nadeel bij integration testing is weer de benodigde tijd: een integration test heeft alleen nut, als de componenten die moeten samenwerken ook al unit tests ondergaan hebben.

Test-driven Development
TDD is een volledige projectaanpak gericht op testen, en zorgt, indien correct geïmplementeerd, voor aanzienlijk beter opgezette projecten en goed gebruik van programmeerprincipes. Door eerst een test voor een functionaliteit te schrijven, zal afgedwongen worden dat er goed nagedacht wordt over elke functie die geschreven wordt.

Hier tegenover staat de grote hoeveelheid tijd die per project nodig is voor TDD: door voor elk stuk logica een test te schrijven, wordt effectief anderhalf keer zoveel code voor een project geschreven. Ook zullen programmeurs moeten wennen aan een relatief strak stramien van programmeren, en zelfs over simpele functionaliteiten goed na moeten denken om dit volgens de regels te doen.

Behavior-driven Development
Het grote voordeel van BDD is de mogelijkheid om eindspecificaties direct te testen. In principe kan degene die de specificaties opstelt, ook de tests schrijven, en zodoende geautomatiseerd de acceptatiecriteria bepalen.

Nadeel hierbij is de vaak stijle leercurve van een syntax voor tests, en het is altijd maar de vraag of er bij een klant de bereidheid is om hier tijd in te steken. Ook voor de programmeurs is dit een lastige ontwikkelmethode, door de omvangrijke frameworks die bij deze manier van werken om de hoek komen kijken.

Verder zal behavior-driven development altijd doorboorduren op test-driven development, waardoor alle voordelen en nadelen daarvan ook op BDD van toepassing zijn.

Bijlage C

Productvergelijking

Om een afweging te maken voor de te gebruiken producten, is een lijst met criteria opgesteld waar de bekendste producten op beoordeeld worden. De beoordelingscriteria zijn als volgt.

Integratiemogelijkheid
De mogelijkheden van het product om aan te sluiten op andere producten en processen. Bijvoorbeeld of een project zonder al te veel moeite ingepast kan worden in de bestaande ontwikkelomgeving of deployment scripts, en (indien van toepassing) of het product goed programmatisch aan te sturen is.

Leercurve
De moeite die het voor de gebruiker kost om een product onder de knie te krijgen. Hierin wordt ook meegenomen of een gebruiker in één keer het hele product moet begrijpen, of eerst met de basis aan de slag kan en vervolgens steeds de kennis van het product uitbreidt.

Ondersteuning
De mogelijkheid om bij problemen met een product hulp te krijgen en vragen te kunnen stellen, bij voorkeur aan de partij die het product ontwikkeld heeft.

Kosten
De financiële implicaties van het gebruik van het product, zoals licenties en benodigde hardware en infrastructuur.

Overige
Andere criteria die van toepassing zijn. Betreft specifieke eigenschappen van het product. Indien dit criterium niet van toepassing is, zal deze achterwege gelaten worden.

De mogelijke beoordelingen zijn ++ (zeer goed), + (goed), 0 (neutraal of niet van toepassing), - (slecht) en – (zeer slecht).

nUnit
Integratiemogelijkheid ++
nUnit heeft goede integratie met Visual Studio en biedt voldoende mogelijkheden voor integratie in het build- en deploymentproces. Ook andere producten (waaronder Selenium) bieden integratie met het nUnit-framework.

Leercurve +
Het nUnit-framework heeft een duidelijke syntax en goede documentatie, waardoor dit snel aan te leren moet zijn voor developers.
Ondersteuning ++
nUnit is het breedst ondersteunde testframework beschikbaar voor .NET-programmeren.
Kosten ++
nUnit kan gratis gebruikt worden. Overige + nUnit heeft qua structuur grote overeenkomsten met PHPUnit, wat er voor zorgt dat tests in PHP en C# met een vergelijkbare structuur opgezet kunnen worden. Dit biedt voordelen met betrekking tot overdraagbaarheid van code.

xUnit.net
Integratiemogelijkheid 0
Het gebruik van xUnit.net binnen Visual Studio is zeer makkelijk, maar ondersteuning door andere programmatuur en de mogelijkheid tot gebruiken vanuit het build- en deploymentproces is minder goed.
Leercurve ++
xUnit.net heeft een schonere syntax dan nUnit, en zorgt ervoor dat tests in minder code geschreven kunnen worden.
Ondersteuning +
xUnit.net heeft goede documentatie, maar nog geen grote actieve userbase.
Kosten ++
Het gebruik van xUnit.net is gratis.
Overige -
Doordat xUnit.net relatief nieuw is, zitten er nog wat kinderziektes in het framework. Ook is de continuïteit van het product niet zeker, zoals dat bij bijvoorbeeld nUnit het geval is.

WatiN
Integratiemogelijkheid 0
WatiN kan tests zeer goed naar C#-code exporteren, maar niet naar andere programmeertalen. WatiN werkt alleen in combinatie met Internet Explorer. WatiN integreert redelijk met nUnit.
Leercurve ++
Het gebruik van WatiN is intuïtief en simpel, en zal voor ontwikkelaars snel op te pakken zijn.
Ondersteuning -
WatiN wordt door relatief weinig mensen gebruikt, waardoor de ondersteuning ook minder is.
Kosten ++
Het gebruik van WatiN is gratis.

Selenium
Integratiemogelijkheid +
Selenium werkt goed samen met xUnit-frameworks, en kan tests naar diverse programmeertalen exporteren. Bij exporteren naar C# zijn er nog enkele instructies die soms niet goed vertaald worden.
Leercurve +
De syntax van Selenium is simpel en transparant, en bouwt indien gewenst voort op de teststructuur van een gekozen xUnit-framework.
Ondersteuning ++
Selenium is het meest gebruikte platform voor automated UI testing, en als zodanig heeft het de grootste actieve gebruikerscommunity. Op deze manier bieden gebruikers ondersteuning aan

elkaar.
Kosten ++
Het gebruik van Selenium is kosteloos.

ReSharper
Integratiemogelijkheid –
ReSharper is gemaakt als plugin voor Visual Studio, dus van nature is de integratie goed. De Express-versie van Visual Studio, welke bij de opdrachtgever gebruikt wordt, ondersteunt echter geen plugins, en ReSharper werkt hier dus niet mee samen.
Leercurve +
ReSharper is makkelijk in gebruik, zonder de ontwikkelaar in de weg te zitten.
Ondersteuning ++
ReSharper heeft prima commerciële ondersteuning, en werkt naadloos samen met alle bekende frameworks en test-tools.
Kosten -
ReSharper vereist een éénmalige licentie per programmeur, waarvan de kosten per licentie 314 euro excl. BTW bedragen.

JMeter
Integratiemogelijkheid 0
JMeter heeft op ontwikkelniveau geen mogelijkheid voor integratie, maar kan wel vanuit het build- en deploymentproces aangeroepen worden, eventueel met een configuratie specifiek voor het project in kwestie.
Leercurve –
De interface van JMeter is zeer uitgebreid, waardoor het voor de gebruiker lastig is om 'even snel' een performance test op te zetten.
Ondersteuning +
JMeter is één van de meest gebruikte tools voor performance testing, en heeft als zodanig een zeer goede ondersteuning vanuit zijn gebruikerscommunity.
Kosten ++
JMeter is kosteloos te gebruiken.

Bijlage D

Proof of concept: Test runner

Om de tests van een project te draaien op een server, moet een applicatie geschreven worden die alle tests in een project doorloopt. In de gecompileerde DLL-file wordt gekeken naar alle klasses die volgens specificatie een test zijn.

Voor de .Net-variant hiervan kunnen we gebruik maken van een mechanisme genaamd Reflection. Dit mechanisme maakt het mogelijk om gecompileerde code te gebruiken zonder daadwerkelijk te weten hoe deze eruit ziet. Er kan dynamisch gekeken worden welke klassen een namespace bevat, en welke members een klasse heeft. Deze members kunnen ook aangeroepen worden.

Hierdoor kunnen we in de DLL van een gecompileerd project alle klassen doorlopen, en controleren of deze voldoen aan een van tevoren gespecificeerde interface. Als dat het geval is, dan is de klasse blijkbaar een geautomatiseerde test. Dit impliceert dan ook dat de klasse een functie heeft om de test te draaien, en dat deze vervolgens een concreet testresultaat terug geeft.

Hieronder de gedeelde code, die zowel in een project als in de test runner beschikbaar moet zijn.

```
using System;
using System.Collections.Generic;
using System.Web;
namespace W3S.Testing {
    public interface IAutomatedTest {
        string Name { get; }
        TestResult Run (string [] p_arrArguments);
    }
    public class TestResult {
        public enum TestStatusValue {
            Pass,
            Fail
        }
        public TestStatusValue TestStatus;
        public string TestName;

        public TestResult (IAutomatedTest p_objAutomatedTest,
            Exception p_objException) {
            this.TestStatus = (p_objException == null) ?
                TestStatusValue.Pass :
                TestStatusValue.Fail;
            this.TestName = p_objAutomatedTest == null ?
                "Unidentified test" :
```

```
                    p_objAutomatedTest.Name;
            }
        }
    }
```

Hieronder de code voor het concept van de test runner, en benodigde helper classes.

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Reflection;
using System.Text;
using System.Threading.Tasks;
using W3S.Testing;
namespace W3S.TestRunner {
    public class Program {
        string path =
        public static void Main (String [] p_arrArguments) {
            List <TestResult> l_arrTestResults =
                new List<W3S.Testing.TestResult> ();
            Assembly l_objAssembly = Assembly.LoadFile (
"C:\\inetpub\\wwwroot\\DeelgemeentePrinsAlexander\\SportPortaal\\TestRunner\\
bin\\Debug\\SportPortaal.dll");
            Type l_objAutomatedTestType = typeof (IAutomatedTest);
            List<Type> l_arrTestClasses = new List<Type> ();
            foreach (Type l_objType in l_objAssembly.GetTypes ()) {
                foreach (Type l_objInterface in l_objType.GetInterfaces ()) {
                    if (l_objInterface == typeof (IAutomatedTest)) {
                        l_arrTestClasses.Add (l_objType);
                    }
                }
            }
            int l_intFailed = 0;
            foreach (Type l_objType in l_arrTestClasses) {
                IAutomatedTest l_objAutomatedTest = (IAutomatedTest)
                    Activator.CreateInstance (l_objType);
                TestResult l_objResult;
                try {
                    l_objResult = l_objAutomatedTest.Run (p_arrArguments);
                } catch (Exception p_objException) {
                    l_objResult = new TestResult (l_objAutomatedTest,
                        p_objException);
                }
                l_arrTestResults.Add (l_objResult);
                if (l_objResult.TestStatus == TestResult
                    .TestStatusValue.Fail) {
                        l_intFailed++;
                }
                Console.WriteLine (
                  String.Format (
                    "{0}: {1}",
                    l_objResult.TestName,
                    l_objResult.TestStatus.ToString ()));
            }
            Console.WriteLine ("{0} tests run, {1} passed, {2} failed.",
                l_arrTestResults.Count,
                l_arrTestResults.Count - l_intFailed,
                l_intFailed);
            Console.ReadLine ();
        }
    }
}
```

Bijlage E

Proof of concept: Crawler

Om snel te achterhalen welke pagina's zich in een site bevinden, en of deze geen serverfouten geven, kan een crawler gebouwd worden. In de basis kan deze bestaan uit een commandline-applicatie waar als parameter een URL aan meegegeven wordt.

De crawler doet vervolgens een webrequest naar het URL, en haalt de content daarvoor op. Uit deze content worden links gehaald naar andere pagina's binnen de site. Die worden vervolgens ook weer opgehaald. Dit proces herhaalt zich tot alle pagina's (waarnaar gelinkt wordt in de site) doorzocht zijn.

Onderstaande code demonstreert hoe een simpele crawler kan worden opgebouwd. Als besloten wordt tot inzetten van een crawler, dan moet hier logging, foutafhandeling, en uitbreiding van de output in gemaakt worden.

Nieuwe pagina's om te crawlen worden alleen aan de lijst toegevoegd als deze nog niet in de lijst staan. Er wordt hierbij uitgegaan van het principe, dat een collectie (zoals een ArrayList) bij het aanroepen van de Contains-methode alle reeds opgeslagen objecten zal vergelijken met de parameter aan de hand van de Equals-methode van het object.

```
using System;
using System.Collections;
using System.IO;
using System.Net;
using System.Text.RegularExpressions;
namespace W3S.Crawler {
    class Program {
        static ArrayList m_arrPages = new ArrayList();
        static string m_strBaseUrl;
        static void Main(string[] p_arrArguments) {
            m_strBaseUrl = "http://sportportaal.dev";
            Webpage l_objPage = new Webpage("/");
            m_arrPages.Add(l_objPage);
            while ((l_objPage = GetNextPage()) != null) {
                try {
                    Console.WriteLine(l_objPage.Url);
                    Crawl(l_objPage);
                } finally {
                    SetProcessed(l_objPage);
                }
            }
            Console.WriteLine(String.Format("{0} pages crawled on {1}",
```

```
                m_arrPages.Count,
                m_strBaseUrl));
            Console.ReadLine();
        }
        static void Crawl(Webpage p_objPage) {
            try {
                HttpWebRequest l_objRequest = (HttpWebRequest)
                    HttpWebRequest.Create(m_strBaseUrl + p_objPage.Url);
                string l_strResponseText = new StreamReader(
                    l_objRequest.GetResponse()
                    .GetResponseStream()).ReadToEnd();
                MatchCollection l_mcLinks =
                  Regex.Matches(l_strResponseText,
@"<a.+?href\=[""|']?(?<link>.*?)[""|'](?=[>\s]).*?>(?<lnkText>.+?)</a>",
                  RegexOptions.Singleline | RegexOptions.IgnoreCase);
                foreach (Match l_objMatch in l_mcLinks) {
                    string l_strUrl = l_objMatch.Groups[1].Value
                        .ToLower();
                    if (l_strUrl.IndexOf('#') > -1) {
                        l_strUrl = l_strUrl.Remove(l_strUrl
                            .IndexOf('#'));
                    }
                    if (l_strUrl.StartsWith("/")) {
                        Webpage l_objPage = new Webpage(l_strUrl);
                        if (!m_arrPages.Contains(l_objPage)) {
                            m_arrPages.Add(l_objPage);
                        }
                    }
                }
            } catch { }
        }
        private static Webpage GetNextPage() {
            Webpage l_objReturn = null;
            foreach (Webpage l_objPage in m_arrPages) {
                if (!l_objPage.Processed) {
                    l_objReturn = l_objPage;
                    break;
                }
            }
            return l_objReturn;
        }
        private static void SetProcessed(Webpage p_objWebpage) {
            m_arrPages.Remove(p_objWebpage);
            p_objWebpage.Processed = true;
            m_arrPages.Add(p_objWebpage);
        }
    }
    class Webpage {
        public string Url { get; private set; }
        public bool Processed { get; set; }
        public Webpage(string p_strUrl) {
            Url = p_strUrl.ToLower().Trim();
            Processed = false;
        }
        public override bool Equals(Object p_objPage) {
            return this.Url == ((Webpage) p_objPage).Url;
        }
    }
}
```

Bijlage F

Teststrategie

W3S hecht grote waarde aan de kwaliteit en veiligheid van uw website. Om deze facetten te waarborgen, wordt uw website op diverse manieren uitvoerig getest.

Uiteraard wordt alle functionaliteit, voordat deze naar de klant opgeleverd wordt, gecontroleerd door een medewerker. Om partijdigheid te voorkomen, zal dit handmatig testen nooit door de programmeur zelf gedaan worden.

Naast handmatig testen maakt W3S echter ook gebruik van zogenaamde programmatische tests. Deze tests zijn als het ware scripts die automatisch bepaalde functies van uw websites controleren. Het schrijven van programmatische tests gebeurt altijd in overleg.

W3S maakt bij programmatische tests onderscheid tussen unit tests, functionele test, en geautomatiseerde User Interface-tests.

Unit testing Het schrijven van unit tests is wenselijk bij complexe logica, die veelal missiekritiek is. Om een stuk code te kunnen unit testen, moet deze daar specifiek voor geschreven worden. Daarom is het belangrijk dat complexiteit bij de oorspronkelijke bouw in te schatten is. Het is lastig om unit tests later toe te voegen.

Functional testing Voor te bouwen feeds, API's en andere databronnen kunnen, in overleg met de klant, functionele tests geschreven worden. Deze tests zorgen ervoor dat data te allen tijde goed uitgeleverd wordt. Dit is bijvoorbeeld belangrijk als externe partijen afhankelijk zijn van deze data.

Automated UI testing Geautomatiseerde User Interface-tests kunnen achterhalen of veelgebruikte gebruikersacties nog werken. Denk hierbij aan het testen van een login-functionaliteit, of kijken of de zoek- functie nog werkt. User Interface-tests simuleren het complete gebruik van uw site, inclusief browser. In overleg kan voor elke gebruikerfunctionaliteit een geautomatiseerde test geschreven worden. W3S kan u hierover adviseren.

Meerwaarde van automatisch testen Door bepaalde zaken automatisch te testen, in plaats van met de hand, kan veel tijd bespaard worden. Ontwikkelaars kunnen hierdoor meer tijd besteden aan het daadwerkelijk ontwikkelen van nieuwe functionaliteiten.

Daarnaast heeft u de zekerheid dat een functionaliteit na een nieuwe deployment nog steeds werkt. Als er door een aanpassing iets stuk gaat, zullen de geautomatiseerde tests dit uitwijzen, lang voordat er een gebruiker aan te pas komt.

Performance testing

Naast het testen van specifieke functionaliteiten maakt W3S gebruik van performance testing, om na te gaan of een applicatie wel voldoende schaalt. Het is belangrijk om na te gaan, of een site met 100 gebruikers nog steeds net zo goed werkt als met één gebruiker. Daarnaast zal de hoeveelheid data op een site door de loop van de tijd ook groeien, wat in software soms tot onvoorziene problemen kan leiden.

Door performance tests op te stellen, kan vastgesteld worden of uw website ook bij grote publiciteit de druk nog aan kan. Mocht dit niet het geval zijn, dan kunnen aanpassingen gedaan worden voordat dit een probleem is. Zo bent u een stuk zekerder van de beschikbaarheid van uw website.

Acceptabele resultaten voor performance testing zullen altijd voor het begin van een ontwikkeltraject, in overleg met de klant worden opgesteld. W3S maakt hiervoor een eerste opzet op basis van verwacht gebruik en de gebruikte serveromgeving (shared of dedicated, en hoeveelheid rekenkracht). Als de klant niet de technische expertise heeft om hier inschattingen over te maken, kan W3S hier in adviseren.
Wilt u meer informatie over de mogelijkheden van geautomatiseerd testen, neem dan contact op met W3S.

Bijlage G

Gebruiksvoorbeeld automated UI test

Voorbeeld van een geautomatiseerde UI test, zoals deze door de opdrachtgever ingezet kan worden.

Een geautomatiseerde UI test omhelst voor een browserinstantie die programmatisch kan worden aangestuurd. Selenium levert een controle-object voor deze browserinstantie, waarmee er vervolgens opdrachten aan deze instantie gegeven kunnen worden door de testcode.

Onderstaande code test de mogelijkheid om in te loggen met een specifieke testaccount. De code initialiseert Selenium, maakt een browserinstantie aan, vult de accountgegevens in, en controleert vervolgens of de browser na inloggen op de account-pagina uit komt.

Onderstaande klasse is zo opgebouwd, dat deze herkend en verwerkt kan worden door de Test Runner uit bijlage D.

```
using System;
using System.Collections.Generic;
using System.Web;
using W3S.Testing;
using NUnit.Framework;
using OpenQA.Selenium;
using OpenQA.Selenium.Firefox;
using OpenQA.Selenium.Support.UI;
using System.Text;
using System.Text.RegularExpressions;
using System.Threading;
namespace SportPortaal.App_classes.Tests {
    public class LoginUITest : W3S.Testing.IAutomatedTest {
        #region declare
        private IWebDriver driver;
        private StringBuilder verificationErrors;
        private string baseURL;
        #endregion
        public LoginUITest() {
            driver = new FirefoxDriver();
            baseURL = "http://stowa2013.dev/";
            verificationErrors = new StringBuilder();
        }
        public String Name {
            get {
                return "Login_UI_test";
            }
```

```
        }
        public TestResult Run (string [] args) {
            try {
                driver.Navigate ().GoToUrl (baseURL + "/");
                driver.FindElement (By.LinkText ("login")).Click ();
                driver.FindElement (By.Id ("email")).Clear ();
                driver.FindElement (By.Id ("email")).SendKeys ("stefan@w3s.nl");
                driver.FindElement (By.Id ("password")).Clear ();
                driver.FindElement (By.Id ("password")).SendKeys ("1234");
                driver.FindElement (By.CssSelector ("input[type=\"submit\"]"))
                  .Click ();
                Assert.IsTrue (Regex.IsMatch (
                  driver.FindElement (By.CssSelector ("BODY")).Text,
                  "^[\\s\\S]*Verenigingsgegevens[\\s\\S]*$"));

                try {
                    driver.Quit ();
                } catch (Exception) {
                    // Ignore errors if unable to close the browser
                }
                Assert.AreEqual ("", verificationErrors.ToString ());
                return new TestResult (this, null);
            } catch (Exception p_objException) {
                return new TestResult (this, p_objException);
            }
        }
    }
}
```

Bijlage H

Gebruiksvoorbeeld automated functional test

Voorbeeld van een functionele test, zoals die door de opdrachtgever ingezet zou kunnen worden.

Deze test haalt een JSON-feed met kalenderitems op, en probeert deze vervolgens te parsen.

Er vindt in deze test geen assert plaats, dat wil zeggen, er is geen controle op het resultaat. Als het parsen niet lukt, zal er een exception optreden, waardoor vervolgens een negatief testresultaat teruggegeven wordt. De Assert-logica van testframeworks werkt hetzelfde: als het resultaat niet is wat verwacht wordt, geeft deze ook een exception.

Onderstaande klasse is zo opgebouwd, dat deze herkend en verwerkt kan worden door de Test Runner uit bijlage D.

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Net;
using System.Web;
using System.Web.Script.Serialization;
using W3S.Testing;
namespace SportPortaal.App_classes.Tests {
    public class FunctionalTest1 : W3S.Testing.IAutomatedTest {
        public FunctionalTest1 () { }
        public String Name {
            get {
                return "Calendar feed functional test";
            }
        }
        public TestResult Run (string [] args) {
            try {
                HttpWebRequest l_objRequest = (HttpWebRequest)
                  HttpWebRequest.Create (
                  "http://prinsalexanderinternet.dev/templates/json/calendar.aspx");
                string l_strResponse = new StreamReader (
                  l_objRequest.GetResponse ().GetResponseStream ())
                  .ReadToEnd ();
                Object l_objResult = new JavaScriptSerializer ()
                  .Deserialize (l_strResponse, typeof (Object));
                return new TestResult (this, null);
            } catch (Exception p_objException) {
```

```
                return new TestResult (this, p_objException);
            }
        }
    }
}
```

Bijlage I

Gebruiksvoorbeeld unit test

Voorbeeld van een unit test, zoals die door de opdrachtgever ingezet zou kunnen worden.

De code instantiëert een nieuwe order in een webshop, en voegt hier een product aan toe. Vervolgens wordt gecontroleerd of er inderdaad één product in de zojuist aangemaakte order aanwezig is.

Voor het product wordt een mock object gebruikt, een klasse die specifiek geschreven is om mee te testen. Dit voorkomt afhankelijkheden, zoals gegevens van het product ophalen uit de database. Deze klasse wordt alleen in testcode gebruikt.

Onderstaande klasse is zo opgebouwd, dat deze herkend en verwerkt kan worden door de Test Runner uit bijlage D.

```
using NUnit.Framework;
using System;
using System.Collections.Generic;
using System.Web;
using W3S.Testing;
using OtibV.Webshop;
namespace OtibV.App_classes.Tests {
    class AddProductUnitTest : W3S.Testing.IAutomatedTest {
        public AddProductUnitTest () { }
        public String Name {
            get {
                return "Add product unit test";
            }
        }
        public TestResult Run (string [] args) {
            Order l_objOrder = new Order ();
            Product l_objProduct = new TestProduct (); // Mock class
            l_objOrder.Add (l_objProduct);
            Assert.AreEqual (l_objOrder.Products.Length, 1);
            return new TestResult (this, null);
        }
    }
}
```