PROF. DR. TANJA E.J. VOS

Zoeken naar fouten

Op weg naar een nieuwe manier om software te testen

Open Universiteit
www.ou.nl

Prof. dr. Tanja E.J. Vos

Zoeken naar fouten

Op weg naar een nieuwe manier om software te testen

Open Universiteit
www.ou.nl

Copyright © Tanja E.J. Vos, Heerlen, 2017
tanja.vos@ou.nl
Niets uit deze uitgave mag worden gereproduceerd zonder schriftelijke
toestemming van de auteur.

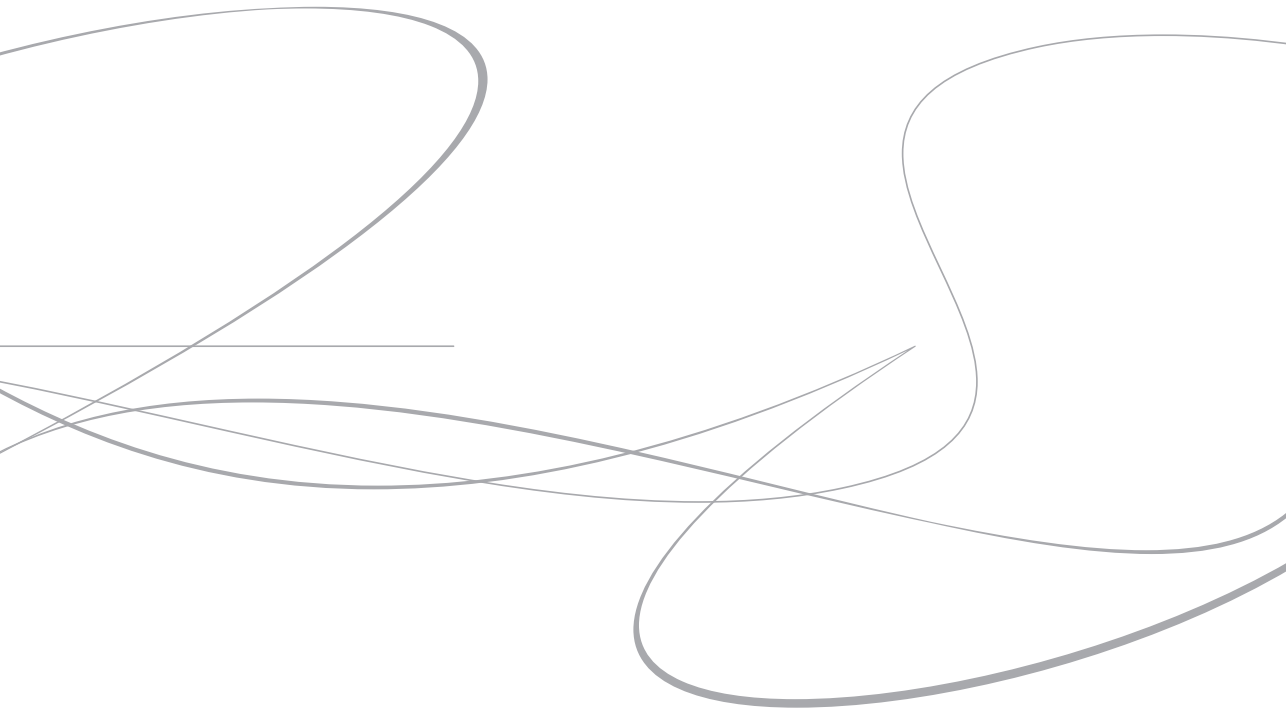
ISBN 978 94 92231 53 6

Ontwerp omslag:
Sumon Dutta en Vivian Rompelberg,
Afdeling Visuele Communicatie, Open Universiteit

Printed in The Netherlands

Inhoudsopgave

Lijst van figuren



Zoeken naar fouten

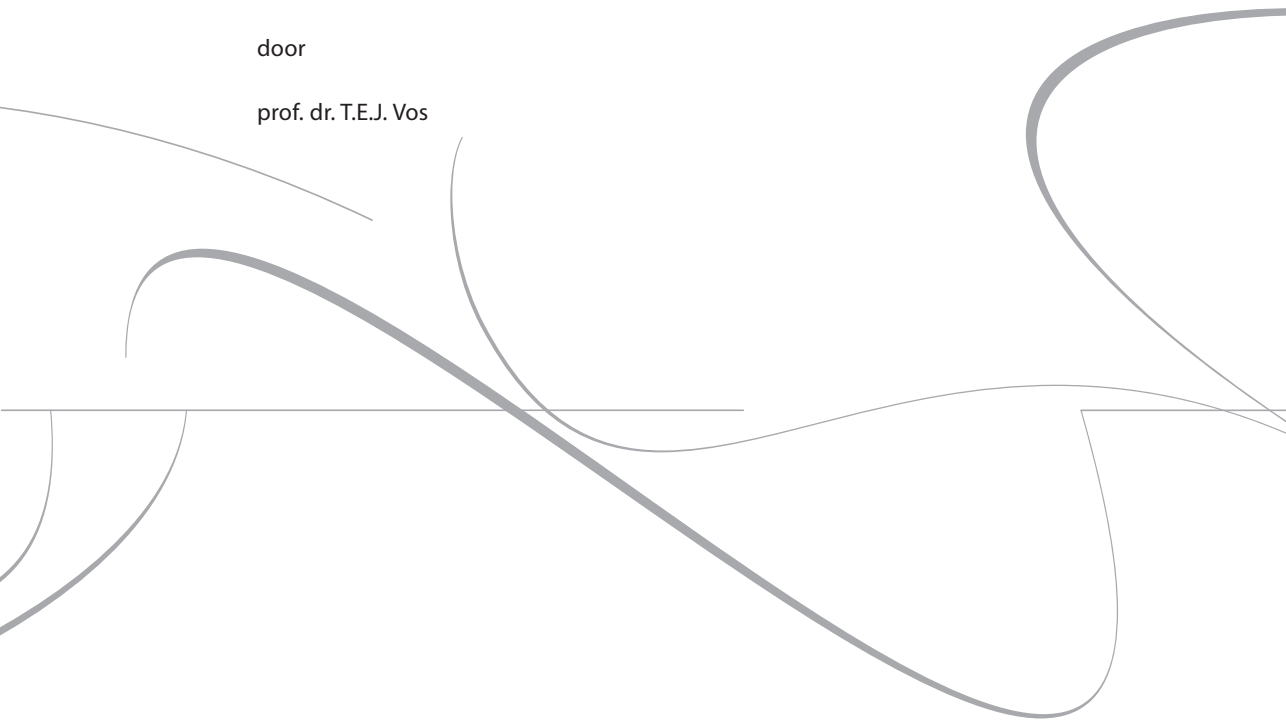
Op weg naar een nieuwe manier om software te testen

Rede

in verkorte vorm uitgesproken bij de openbare aanvaarding
van het ambt van hoogleraar Software Engineering
aan de Open Universiteit op donderdag 22 juni 2017

door

prof. dr. T.E.J. Vos



1 Opening

GEACHTE rector, familie, vrienden en collega's. Welkom in Heerlen op de Brightlands Smart Services Campus. Deze campus is opgericht in 2015 met als doel kennis, talent en ondernemerschap op het gebied van data science, informatietechnologie en servicedesign in de regio te binden en te verbinden. Ik vind het een eer om hier, in deze indrukwekkende zaal, mijn oratie te mogen uitspreken.

Op 1 Januari 2016 ben ik aangesteld als hoogleraar Software Engineering aan de Open Universiteit Nederland. Anderhalf jaar lang waren velen in afwachting van mijn oratie. Vandaag 22 juni 2017 is de dag daarvoor eindelijk aangebroken.

Voor mij is het afgelopen anderhalf jaar werken bij de Open Universiteit omgevlogen. Mogen werken met slimme en enthousiaste mensen, die gedreven zijn om de beste resultaten af te leveren, hart hebben voor hun vak en altijd klaar staan om iets aan te pakken, is een waar privilege.

De Open Universiteit, of OU, is een niet-traditionele, publieke universiteit. Het onderwijsmodel staat voor persoonlijk en activerend online-onderwijs, met veel flexibiliteit voor studenten. Voor het bachelor-onderwijs zijn er geen vooropleidingseisen en het onderwijs is ingericht op studeren in deeltijd.

De wetenschappelijke opleidingen krijgen veel waardering van studenten en experts. In de Nationale Studenten Enquête van 2016, scoren vijf bacheloropleidingen (waaronder de bachelor informatica) en vier masteropleidingen van de Open Universiteit de eerste plaatsen in de ranglijsten van hun studierichtingen. Van alle veertien Nederlandse universiteiten geven studenten deze opleidingen het hoogste cijfer van alle opleidingen in deze studierichtingen. Als instelling scoort de Open Universiteit de tweede plaats in de ranglijst van de veertien universiteiten.

Vandaag zal ik iets vertellen over het soort onderzoek dat ik met mijn groep doe op het gebied van software-engineering. Ik zal uitleggen wat we doen en waarom, en wat de uitdagingen zijn. Ik zal enkele onderzoeksresultaten presenteren en proberen een beeld te schetsen van wat we de komende jaren willen gaan doen op het gebied van de software-engineering. Ook zal ik iets vertellen over hoe ik in mijn rol als docent en programmaleider van de bachelor informatica wil bijdragen aan een

cultuur van software-kwaliteit. Het is mijn voornemen om aan de OU het informaticaonderwijs zodanig in te richten dat testen en kwaliteit als een rode draad door het curriculum lopen en zo als een onlosmakelijk onderdeel worden gezien van software-engineering.

2 Software is overal om ons heen

LATEN we eerst eens nagaan hoeveel software we allemaal al hebben gebruikt voordat we hier in Heerlen aan konden komen. Bijna iedereen heeft een smartphone. Daar staat een heleboel software op. We kunnen in een oogwenk berichtjes de hele wereld oversturen, nieuws lezen, e-mails checken en nog veel meer. Misschien heeft u vanmorgen uw tanden wel gepoetst met een tandenborstel die uw poetspatronen bijhoudt? Misschien ook heeft u zich wel gedoucht onder een intelligente douchekop die probeert water te besparen en kalkaanslag te voorkomen. Als u met de trein bent gekomen, of met de auto, dan heeft u zich al door heel veel verschillende software laten bedienen. Autobedrijven, net zoals de banken, zijn softwarehouses geworden: de hoeveelheid software die in een auto zit is de afgelopen jaren verdubbeld. Er is nu niet alleen software voor bijvoorbeeld cruisecontrol of centrale deurvergrendeling. Onze auto's kunnen tegenwoordig online zelf updates doen van applicaties voor navigatie, automatische regendetectie, assistentie bij het remmen of bij het tussen de belijningen van de weg blijven. Verleden jaar kwamen de eerste auto's op de markt die zelfstandig van A naar B kunnen rijden. Vliegtuigen worden bijna altijd via een automatische piloot op de grond gezet.

We worden dus omringd door software. Het is overal en ons leven wordt er steeds meer door bepaald. Maar toch is goedwerkende software niet vanzelfsprekend. Iedereen is wel eens in aanraking gekomen met fouten in software. Wie kent het niet, het bekende blauwe scherm van het Windows besturingssysteem dat aangeeft dat de hele computer is gecrasht omdat er een probleem was met een software-applicatie? Wie heeft er niet eens foutmeldingen gezien op schermen op het vliegveld, supermarkt, of als je geld aan het pinnen was? Bij wie is zijn tekstverwerker (Word bijvoorbeeld) niet eens precies gecrasht na een belangrijke aanpassing die je nog niet had bewaard?

Hoe komt dat?
Is dat nou echt zo erg?
En wat kunnen we ertegen doen dan?

Daar wil ik het vandaag over hebben, in de context van mijn onderzoek naar software-testen. Laten we beginnen met kijken wat software nu precies is en hoe daar fouten in komen.

3 Wat is software? En hoe komen er fouten in?

SOFTWARE is een algemene term voor de programma's die kunnen werken op een computersysteem. Die programma's zijn geschreven in een bepaalde programmeertaal en geven eigenlijk instructies aan de computer over wat die onder bepaalde condities moet doen.

Kleine foutjes maken als je een programmeertaal gebruikt, kunnen soms grote gevolgen hebben. Een voorbeeld dat ik vaak tegen kom [Ger15] in het Engels gaat over een telegram dat antwoord moest geven op de vraag of een miljoenendeal wel of niet door moest gaan. Er moest staan:

NO, PRICE IS TOO HIGH.

De komma was vergeten en er stond dus:

NO PRICE IS TOO HIGH.

Eén gemiste komma kostte de afzender van het telegram miljoenen!

Softwareprogramma's zijn ook geschreven in een taal, een programmeertaal. Zinnen zijn daarin dus de instructies die aangeven wat er moet gebeuren als bepaalde condities gelden. Bijvoorbeeld de pinautomaat waarmee je je geld pint, zal misschien een stukje programmacode bevatten dat een beetje lijkt op:

```
if (saldo + kreditlimiet > het bedrag dat klant wil opnemen)
then (de klant mag geld opnemen)
else (zet op het scherm dat het saldo onvoldoende is)
```

Dit heet een if-then-else-instructie. De expressie die na de if komt heet een predicaat en afhankelijk van de waarde van het predicaat (True of False) wordt het then-gedeelte if het else-gedeelte uitgevoerd.

Up to 300,000 heart patients may have been given wrong drugs or advice due to major NHS IT blunder

- Computer is system widely used by GPs is miscalculating patients' risk
- Means some people have no idea they have raised risk of heart attack
- May have led to some adults needlessly being prescribed statins, and enduring severe side effects - or not being given medication they need
- See more news on the NHS IT blunder at www.dailymail.co.uk/nhs

By SOPHIE BORLAND HEALTH EDITOR FOR THE DAILY MAIL
PUBLISHED: 16:54 BST, 11 May 2016 | UPDATED: 01:03 BST, 12 May 2016

Bug in GP software may have coughed up wrong data on heart disease risk

Watchdog probes how many patients were potentially misprescribed medication due to flaw.

Statins glitch means thousands may have been incorrectly prescribed

Patients with little risk of heart disease may have been needlessly prescribed statins

Figuur 1: *Voorbeeld van een foute berekening bij het voorschrijven van medicijnen (linksboven* Daily Mail *[Dai16], linksonder* Technica *[UK16], rechts* TheTelegraph *[The17])*

In je tandenborstel zit misschien zoiets als:

```
if (gepoetste tijd < 2 min)
then (zet op scherm dat je minimaal 2 minuten moet poetsen)
```

Of om te bepalen of iemand de juiste leeftijd heeft om te mogen stemmen:

```
if (geboortedatum < verkiezingsdatum - 18 jaar)
then (mag je stemmen)
```

In dit soort programmeerregels kunnen op vele manieren fouten komen. Bijvoorbeeld:

- een typefout of een vergissing met de operatoren < of > of + om berekeningen uit te voeren of om condities te checken. Hetzelfde als de vergeten komma van daarnet, die miljoenen kostte. Stel dat we per ongeluk een 'groter dan' neerzetten als een 'kleiner dan' bedoeld werd, of een plus in plaats van een min! Dan krijgen we incorrecte berekeningen. Dat was bijvoorbeeld wat er mis ging bij de NHS in

een programma om te berekenen wat het risico was dat een patiënt hartziekten had (zie Figuur 1). Als gevolg van deze fout kregen mensen die het niet nodig hadden medicijn toegediend, terwijl mensen die het wel nodig hadden, niets kregen.

- Het niet in acht nemen van bepaalde situaties die zich toch wel voor kunnen doen. Dat is wat er recentelijk misschien is gebeurd bij onze belastingdienst (zie Figuur 2). Bij het invullen van de voorlopige aanslag in de situatie dat je meer dan 3 inkomens uit vroegere dienstbetrekking hebt gehad, werd het belastbare inkomen verkeerd berekend. Misschien dat men er geen rekening mee heeft gehouden dat iemand meer dan 3 inkomstenbronnen zou kunnen hebben en is dat dus niet goed afgehandeld in de programmeercode.
- Verkeerd representeren van gegevens, zoals bijvoorbeeld een datum. Als je maar 2 cijfers neemt om het jaartal van een datum te representeren, dan krijg je problemen als je van eeuw wisselt! Dit voor de meesten van ons bekende probleem stond in de jaren '90 bekend als de millenniumbug. Maar verleden jaar nog haalde dit soort fouten weer de krant (zie Figuur 3). Een Zweedse vrouw die is geboren in 1912 kreeg een bericht dat ze naar de kleuterschool kon gaan, dus net zoals de rest van de kinderen die in 2012 waren geboren! Er zijn nog meer van dit soort voorbeelden te vinden [BBC15]. En er komen er nog een paar aan! Bijvoorbeeld in het jaar 2038 krijgen Unix systemen een probleem met de *signed 32-bit integer*-representatie van tijd die maar genoeg is tot `03:14:07 UTC on Tuesday, 19 January 2038` [Spi06].
- Soms hoef je helemaal niets te doen en leidt het blijven gebruiken van eerder goed werkende software later tot fouten. Doordat er inmiddels bijvoorbeeld bepaalde regels zijn veranderd, die dan in de software niet goed worden aangepast. Dat gebeurt ook vaak bij de belastingdienst, waar de regels erg vaak aangepast worden.

Fouten kunnen dus om heel veel redenen software binnenkomen. Het voorgaande is maar een kleine opsomming van mogelijke situaties die fouten kunnen veroorzaken; er zijn nog veel meer factoren die fouten kunnen introduceren. Daarbij komt dat systemen steeds groter en complexer worden. Software moet ook in steeds minder tijd gemaakt worden (*reduce the*

Figuur 2: *Foute berekening in software voor voorlopige belastingaanslag [De 17].*

Foutje: 104-jarige Zweedse mag naar kleuterschool

De Zweedse Maja Bergström mag naar de kleuterschool. Een week geleden kreeg ze daar een oproep voor. Een probleem: Bergström werd op 21 januari 1912 geboren en is dus 104 jaar oud, schrijft de krant Hallands Nyheter. De gemeente noemde het in de oproep „een goed moment" om een kleuterschool te vinden voor volgend schooljaar.

104-jarige krijgt brief van gemeente: je mag naar de kleuterschool!

Maja met de brief van de gemeente. © Hallandsposten/Jan Bergman

Geboren in '12: een goed moment om naar de kleuterschool te gaan. Zo'n bericht kreeg de Zweedse Maja Bergström van de gemeente Hylte. Maar de Zweedse is niet van 2012, maar van 1912.

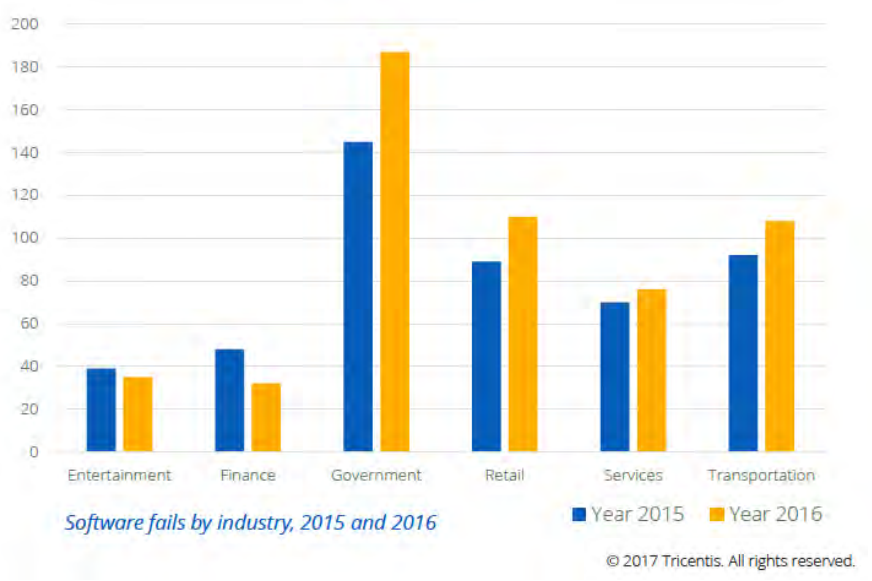
Figuur 3: *Voorbeeld van een recente millenniumbug in de krant rechts* De Gelderlander *[Gel16], links* RTL nieuws *[RTL16])*

time to market). Bovendien groeit en verandert het aantal technologieën dat gebruikt kan worden om het te maken in een hoog tempo. Ook het aantal platformen waarop ze moeten functioneren breidt steeds meer uit, van computer, mobiele telefoon, horloge tot tandenborstel.

4 Fouten zijn menselijk! Is het echt zo erg?

JA het is erg! Sterker nog, als we zo doorgaan zouden we weleens heel dichtbij een softwarekwaliteitscrisis kunnen uitkomen. Een crisis waarin het gebrek aan softwarekwaliteit meer kost dan de software zelf en de besparing die het gebruik ervan zou moeten opleveren.

Laatst las ik een artikel waarin stond dat uit een recent onderzoek [Eva15] was gebleken dat 75% van de mobiele apps met 1 tot 10 bugs gelanceerd wordt. Ook de *Software Fail Watch* [Tri16] analyseert elk jaar alle softwarefouten die in Engelstalige nieuwsartikelen te vinden zijn. In 2016 beoordeelden ze 1159 nieuwsartikelen. Het totaal geschatte aantal mensen dat door softwarefouten werden getroffen, is in het rapport 4.4 miljard. Dat is meer dan 50% van de wereldbevolking! De totale kosten als



Figuur 4: *Software-falen in 2015 en 2016 per sector (uit [Tri16]).*

gevolg van die fouten, liggen rond de 1.1 biljoen USD. In Figuur 4 zien we een voorbeeld van de resultaten die ze presenteren. Er is te zien dat het aantal nieuwsartikelen met gerapporteerde fouten in de meeste sectoren toegenomen is, met als absolute uitschieter de overheidssector. Het rapport concludeert dan ook:

> *With 4.4 billion people and 1.1 trillion in assets impacted by software failures in 2016, it's hard to argue that "more of the same" is the best path forward for software development professionals. —- Wolfgang Platz Founder and CPO of Tricentis, 2017.*

Soms is falende software slechts eventjes irritant voor de gebruiker. Als je een keuze hebt, haal je die applicatie gewoon weer weg van je computer of telefoon en zoek je een alternatief dat wel werkt. En als er geen alternatief is voor die applicatie, dan gebruik je het toch gewoon niet. Maar wat als het gaat om een applicatie die je 'verplicht' moet gebruiken voor je werk bijvoorbeeld? Dan kun je er niet omheen. Dan leidt irritatie al snel tot

productiviteitsverlies. Wat als het gaat om een applicatie die verkeerd jouw belastingaanslag berekent? Dan gaat het mogelijk om het verlies van geld! Of wat als het gaat om een applicatie die de hoeveelheid medicijnen die je moet gebruiken verkeerd berekent? Dan gaat dat ten koste van je gezondheid. Of wat als het gaat om software die in auto's, treinen of vliegtuigen zit? Dat wordt het een kwestie van onze veiligheid (en die van anderen) en kunnen er zelfs levens op het spel staan!

5 Wat is eraan te doen?

ENERZIJDS kunnen we de kans op fouten reduceren door beter te werken. Anderzijds moeten we automatiseren wat er te automatiseren valt, zodat mensen efficiënt kunnen doen waar ze, tot nu toe nog, beter in zijn dan machines: nadenken.

Het is niet dat we niet weten welke aspecten van software-ontwikkeling belangrijk zijn, boekenkasten vol kunnen we vullen met boeken die beschrijven hoe software gebouwd moet worden. Technieken en methodologieën zijn voorhanden en iedereen is het met elkaar eens wat belangrijk is. Om er maar een paar te noemen:

- Bewerkstellig meer communicatie tussen de gebruiker en de ontwikkelaar van de software.
- Schrijf meer en duidelijkere specificaties die beter te onderhouden zijn.
- Maak een ontwerp voordat je gaat programmeren.
- Doe risico-analyses. Door goed van te voren over mogelijke risico's na te denken kunnen er al een hoop onverwachte gebeurtenissen voorkomen worden.
- Houd je aan bepaalde programmeerstijlen, waarvan aangetoond is dat ze resulteren in betere programmacode.
- Becommentarieer programmacode om informatie mee te geven, waarin bijvoorbeeld uitgelegd wordt wat de programmacode doet, waarom die zo geschreven is en wat de verschillende variabelen betekenen.

- Doe reviews van programmacode.
- Test! Ga continu op zoek naar fouten zodat ze opgelost kunnen worden voordat de software in productie gaat.

Het is net zoals met het gaan naar de sportschool en fruit eten. Ook van die dingen weten we heus wel dat ze goed en gezond zijn. Maar we doen ze niet, of niet vaak genoeg, om allerlei redenen. We moeten een cultuur van kwaliteitsdenken kunnen creëren als het gaat om software-ontwikkeling. En die begint in het onderwijs! Programmeeronderwijs moet onlosmakelijk verbonden worden met kwaliteit. Vanaf het begin weten dat je je programma's moet becommentariëren, testen. En dat daarbij goede specificaties van belang zijn. Testen, bijvoorbeeld, moet niet langer een van de technieken zijn die je kunt gebruiken om je code te checken. Testen is iets dat je moet doen als je programmeert, no test, no code! En net zoals met mijn dochter Samanta die tijdens dit schrijven 7 jaar is. Zij gaat niet doen wat ik haar zeg, zij gaat doen wat ik ook doe! Goed voorbeeld doet goed volgen.

Een ander belangrijk aspect dat ik hiervoor al aangaf is automatiseren wat er te automatiseren valt, zodat mensen efficiënt kunnen doen waar ze, tot nu toe nog, beter in zijn dan machines: nadenken. Boris Beizer zei in de jaren 90:

> *One of the saddest sights to me has always been a human at a keyboard doing something by hand that could be automated. It's sad but hilarious.*
>
> –Boris Beizer in Black-Box Testing: Techniques for Functional Testing of Software and Systems 1995

Vanuit mijn leerstoel wil ik aan deze twee punten mijn steentje bijdragen. De afgelopen vijftien jaar heeft mijn onderzoek zich gericht op het beter testen van software en dan met name door het te automatiseren. In mijn rol als docent en programmaleider van de bachelor informatica kan ik het programmeeronderwijs zodanig sturen, dat testen en kwaliteit als onlosmakelijke onderdelen worden gezien van het programmeren.

Open Universiteit

6 Wat is software testen?

SOFTWARE testen is het uitvoeren van de software met als doel daarbij de kwaliteit daarvan te kunnen meten en te waarborgen. Dat klinkt misschien makkelijk, maar dat is het allerminst. Testen moet niet alleen aan het einde gebeuren, maar moet tijdens het ontwikkelen van de software op verschillende niveaus plaats vinden. Op het laagste niveau, dat is waar de programmacode geschreven wordt, moeten *unit-testen* worden gedaan. Daar moet dan voor kleine stukjes programmacode getest worden of die doen waarvoor ze geschreven zijn.

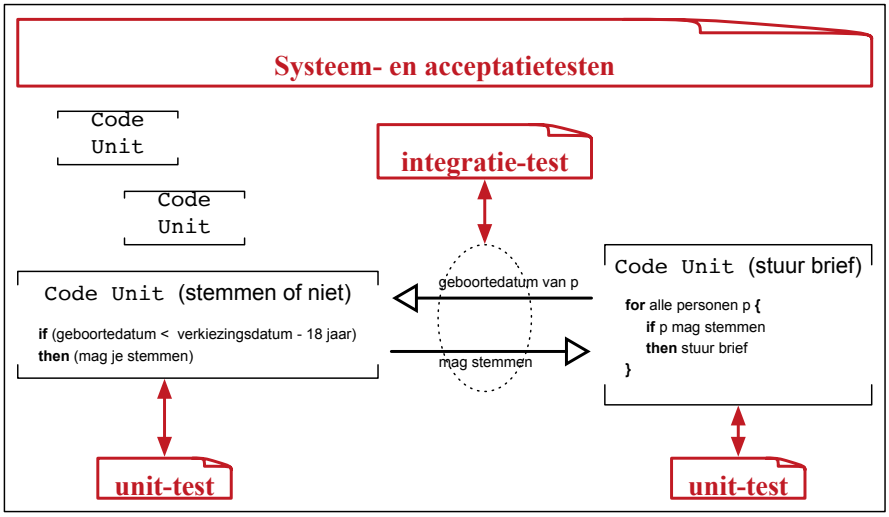
We zagen eerder een stukje programmeercode:

```
if (geboortedatum < verkiezingsdatum - 18 jaar)
then (mag je stemmen)
```

Op unit-testniveau zouden we dit stukje programmacode kunnen testen met verschillende *testwaarden*, bijvoorbeeld:

test-geval	**geboorte-datum**	**verkiezings-datum**	**resultaat**	**commentaar**
1	08-10-1971	22-06-2017	stemmen	
2	04-03-2010	15-05-2019	niet stemmen	
3	29-02-2012	13-04-2017	niet stemmen	schrikkeljaar
4	11-11-1947	29-02-2011	foutmelding	dag bestaat niet
5	29-05-2015	11-11-2011	niet stemmen	nog niet geboren
6	45-02-2015	22-06-2017	foutmelding	dag bestaat niet
7	16-02-1939	22-13-2017	foutmelding	maand bestaat niet
...	...	...	...	...

Dit is maar een voorbeeld van enkele testwaarden. Als we even doordenken kunnen we de lijst makkelijk langer maken wanneer we alle uitzonderingen goed zouden willen testen (denk aan jaartallen die niet bestaan, of die heel ver in het verleden zijn of juist weer in de toekomst, meer combinaties van schrikkeljaar met wel of niet stemmers, enz.). Merk op dat we hier niet alleen testen of de functionaliteit goed is geïmplementeerd, maar ook of uitzonderingen goed zijn opgevangen, bijvoorbeeld worden schrikkeljaren herkend en worden datums die niet bestaan adequaat afgehandeld.



Figuur 5: *Verschillende niveaus waarop getest moet worden*

Als we alle stukjes programmacode op unit-testniveau hebben getest, dan moeten we testen of al die stukjes wel op de juiste manier samenwerken, *integratie-testen* noemen we dat. Stel er is een ander stukje programmacode dat verantwoordelijk is voor het sturen van de stemkaarten naar de stemgerechtigden (zie Figuur 5). Hier zal het eerder genoemde stukje programmacode worden aangeroepen met een geboortedatum. Het antwoord zal gebruikt worden om te bepalen of er wel of niet een stemkaart gestuurd moet worden. In de integratie-testen zal de juiste werking van deze interactie getest moeten worden (worden de datagegevens goed ontvangen? Worden de foutmeldingen bij onjuiste datums goed afgehandeld? Enz.).

Na de unit-testen en de integratie-testen kunnen we het hele systeem in zijn geheel testen door middel van *systeem- en acceptatie-testen*. Dan komen er nog andere aspecten bij kijken. Hier testen we niet alleen de functionaliteit van een applicatie, maar ook onder andere of die veilig is, bruikbaar, robuust, gebruikersvriendelijk en tot goede presentaties leidt.

Systeem- en acceptatie-testen kun je bijvoorbeeld doen via de *Grafische User Interface* (GUI). Als we via de GUI testen, moeten we niet alleen testwaarden bepalen (bijvoorbeeld de datum van de verkiezingen), maar

ook de volgorde van de actie-sequentie die we op de GUI moeten uitvoeren. Stel we hebben een GUI om stemkaarten te versturen naar de personen die stemgerechtigd zijn. Daar is eerst een menu waar we uit moeten kiezen dat we de stemkaart willen versturen. Daarna moeten we de datum van de verkiezingen invullen. En vervolgens op de knop 'sturen' drukken. Een test-sequentie zou er dan als volgt uitzien:

```
1.  Klik op het menu met label:  Communicatie
2.  Selecteer optie:  Stemkaart
3.  Vul in het daarvoor bestemde veld de verkiezingsdatum in
4.  Klik op de knop:  Stuur stemkaart
```

7 We kunnen niet alles testen

EDSGER Dijkstra, de Nederlandse computerpionier, zei het al in 1974:

> *Testing may be convincingly demonstrate the presence of bugs, but can never demonstrate their absence*
>
> —- Edsger W. Dijkstra (1930-2002) in "Programming as a discipline of mathematical nature", Am. Math. Monthly, 81 (1974), no. 6, pp.608-12.

We kunnen dus nooit na het testen zeggen: deze software werkt foutloos! Aangezien we niet weten hoeveel fouten er in totaal in de software kunnen zitten, kunnen we nooit weten of we alle fouten hebben gevonden en of we eigenlijk wel goed getest hebben.

Laten we wat voorbeelden bekijken waarmee snel duidelijk wordt dat het onmogelijk is om een software-programma compleet te testen.

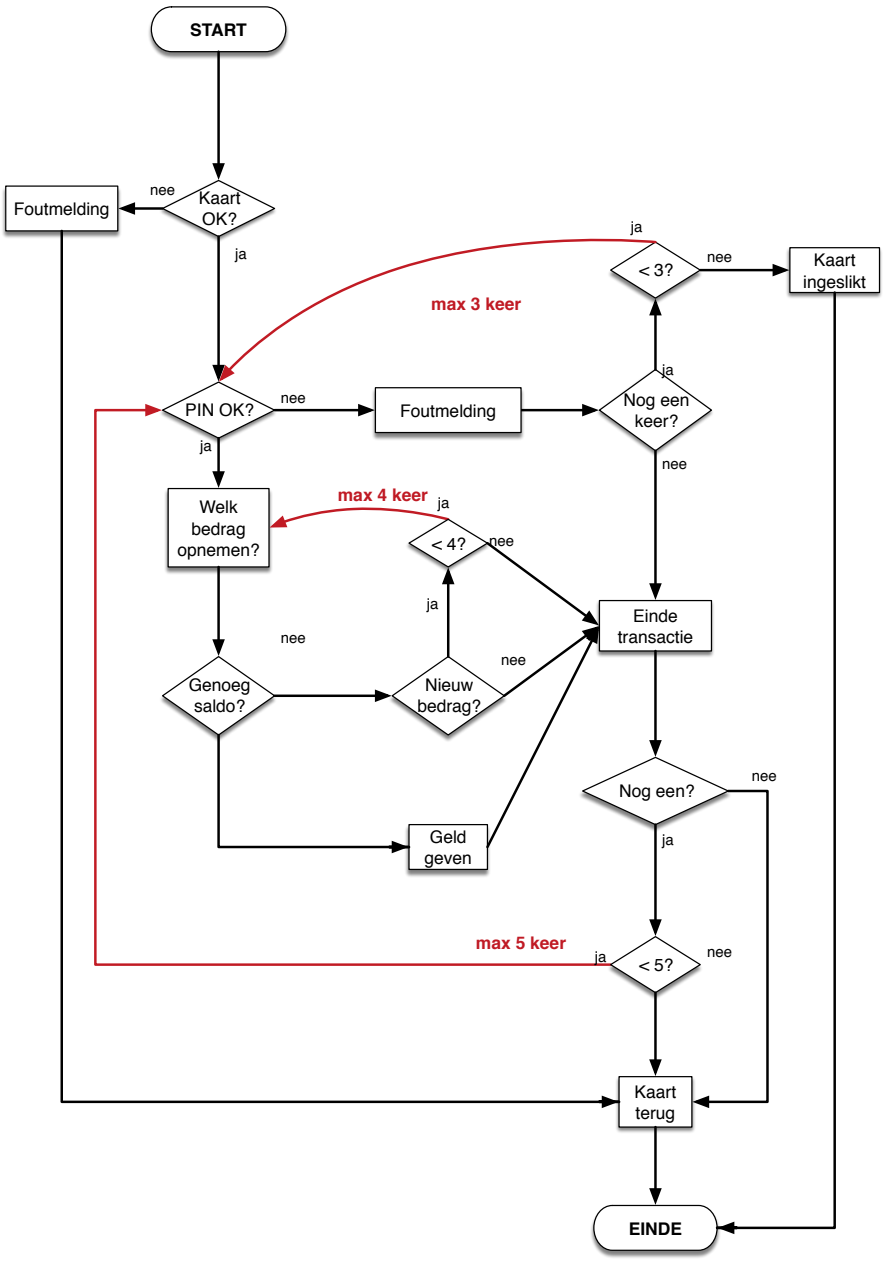
Er zijn te veel waarden die ingevoerd kunnen worden. Eerder hebben we een programma gezien om te bepalen of iemand stemgerechtigd is. Het programma heeft een datum nodig en op basis van die datum maakt het een bepaalde berekening. We zagen hiervoor al dat we snel veel testgevallen konden verzinnen. Maar als we het compleet zouden willen testen, dan zouden we alle datums die er bestaan één voor één moeten invoeren om te kijken of er geen fouten optreden.

Bovendien zouden we ook verkeerde datums moeten invullen om te kijken of het programma dat adequaat afhandelt. Dat is bijna onmogelijk en kost heel veel tijd.

Er zijn te veel sequenties die je door een software programma kunt belopen. Een sequentie, of pad, door een programma zijn alle stappen (kliks, scrolls, typen van tekst, beslissingen, enz.) die je maakt totdat je klaar bent met een taak of een programma. Laten we bijvoorbeeld naar Figuur 6 kijken waar een heel versimpeld stroomdiagram staat om geld op te nemen uit een PIN-automaat. Een 'ruit' stelt daar een keuze of beslissing voor in het programma. Als aan de conditie is voldaan dan gaan we de kant van de "ja"-pijl op en anders die van de "nee"-pijl. In een 'rechthoek' staat iets dat de software doet (bijvoorbeeld iets laten zien op het scherm, geld geven of de kaart inslikken). Eerst wordt gecheckt of de bankkaart OK is. Als dat niet het geval is, dan krijg je een foutmelding te zien, krijg je jouw kaart terug en kun je het nog een keer proberen. Als de kaart wel OK is wordt jouw PIN-code gevraagd. Als de PIN-code niet OK is kun je dat tot maximaal 3 keer opnieuw proberen, daarna wordt de kaart door het apparaat ingeslikt. Als de PIN-code wel goed is, kun je het bedrag invoeren dat je wilt opnemen. Als je genoeg saldo hebt, krijg je jouw geld, pak je jouw kaart terug en kun je die handelingen desgewenst herhalen. Als je niet genoeg saldo hebt, dan mag je tot 4 keer toe een nieuw bedrag invoeren waar jouw saldo dan mogelijk wel toereikend voor is.

Laten we eens kijken op hoeveel verschillende manieren we van START naar EIND kunnen gaan:

- We kunnen op $(3 \times (4+4+1)) + 3 = 30$ verschillende manieren van START naar Einde transactie gaan.
- Al deze manieren kunnen we tot 5 keer herhalen.
- Dus dat zijn $30^1 + 30^2 + 30^3 + 30^4 + 30^5 = 25.137.930$ mogelijke sequenties om van START naar Einde transactie te gaan.
- We kunnen op 1 manier van START naar Kaart terug via de foutmelding.



Figuur 6: *Stroomschema dat aangeeft hoe een PIN-automaat kan werken*

- De enige manier om naar Kaart ingeslikt te komen is door middel van een sequentie die 3 keer een verkeerde PIN ingeeft:
  PIN OK? $\rightarrow_{nee}$ Foutmelding $\rightarrow$ Nog een keer? $\rightarrow_{ja}$ 1<3 $\rightarrow_{ja}$
  PIN OK? $\rightarrow_{nee}$ Foutmelding $\rightarrow$ Nog een keer? $\rightarrow_{ja}$ 2<3 $\rightarrow_{ja}$
  PIN OK? $\rightarrow_{nee}$ Foutmelding $\rightarrow$ Nog een keer? $\rightarrow_{ja}$ 3<3 $\rightarrow_{nee}$
  Kaart ingeslikt. En van Kaart ingeslikt gaan we meteen door naar EINDE.
- We kunnen op $1 + 30^1 + 30^2 + 30^3 + 30^4$ manieren bij PIN OK? komen en vanaf daar dus 3 keer een verkeerde PIN invoeren. Dat zijn 837.931 manieren.
- Dus dat zijn dus totaal 25.137.930 + 1 + 837.931 = 25.975.862.
- Als het 5 minuten zou kosten (wat optimistisch is) om één test te bedenken, uit te voeren en te evalueren, dan komt dat neer op ongeveer 2.164.655 uur of 54.116 werkweken van 40 uur.
- Als er 42 werkweken in een jaar zitten dan komt compleet testen dus neer op meer dan 1288 jaar!

Dat is natuurlijk te lang. Daarbij komt, zulke simpele stroomdiagrammen staan alleen in instructieboeken. In werkelijkheid is het stroomdiagram voor geld pinnen veel uitgebreider, met nog meer paden.

Er zijn te veel combinaties van inputwaarden, paden en verschillende platformen. Denk aan verschillende:

- browsers (Firefox, Chrome, Internet Explorer, Safari, enz.);
- besturingssystemen (Windows, Linux, macOS, Android, iOS, enz.);
- en hun versies (Windows XP, Vista, Windows 7, Windows 8, Windows 10, enz.);
- of distributies (Suse, Redhat, Ubuntu, Debian, enz.).

De voorbeelden hiervoor tonen aan dat het voor hele simpele applicaties al onmogelijk is om door middel van testen alle gevallen af te vangen. Laat staan echte applicaties die vele malen complexer zijn dan de hiervoor

gegeven voorbeelden. En we zijn er nog niet: software-applicaties worden alsmaar complexer en het aantal apparaten en systemen waarop we ze kunnen gebruiken, groeit razendsnel. Testen wordt dus ook steeds maar complexer.

8 Hoe kunnen we dat aanpakken?

HET probleem is dus duidelijk. Software bevat fouten. We kunnen door gestructureerd en netjes te werken sommige fouten voorkomen. Maar software is te complex en mensen zijn geen perfecte denkers, dus foutloos zal het nooit zijn [Ger08]. Testen is dus altijd nodig. Maar, er is te veel om alles te testen: veel verschillende aspecten van software, op verschillende niveaus, met heel veel testwaarden en test-sequenties en hun combinaties leiden tot oneindig veel mogelijke testgevallen. Daarbij komt, dat we nooit kunnen weten wat voor soort en hoeveel fouten er in een software-applicatie kunnen zitten.

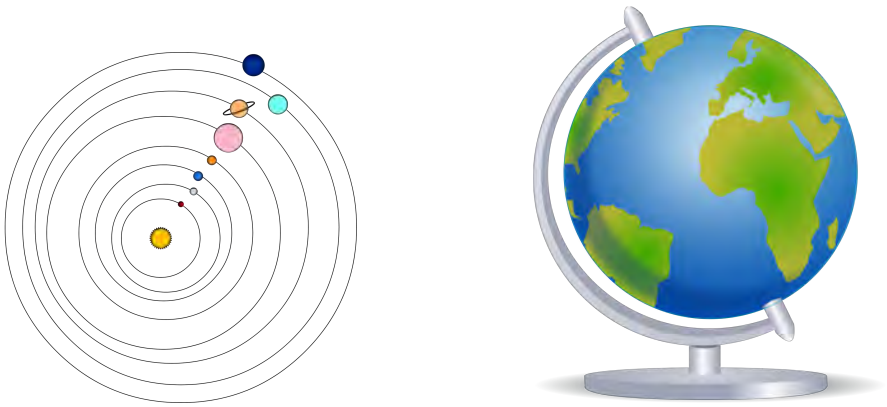
Testen is dus het zoeken naar een onbekende hoeveelheid onbekende fouten in een software-applicatie, door middel van het kiezen van een beperkt aantal testgevallen uit een oneindig aantal mogelijkheden.

Dat klinkt niet eenvoudig en dat is het ook niet. Dus, hoe pak je dat dan aan? We hebben hier twee problemen op te lossen:

1. We hebben een oneindig aantal testgevallen. Hoe kunnen we nu die gevallen kiezen die je zou moeten gaan gebruiken voor het uitvoeren van jouw testen?
2. We weten niet hoeveel fouten er in onze software kunnen zitten. Dus hoe weten we achteraf of we goed getest hebben?

8.1 Testontwerptechnieken

Als je gestructureerd op zoek gaat naar fouten dan pas je een testontwerptechniek toe die je in staat stelt een weloverwogen keuze te maken voor de testgevallen die je kiest. Er bestaan een heleboel van die technieken.



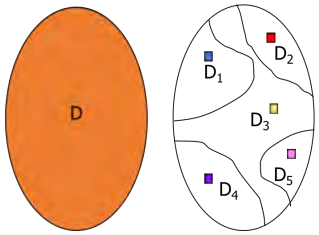
Figuur 7: *Modellen: bruikbare simplificaties van de werkelijkheid.*

Helaas is er nog niet echt consensus bereikt over een eenduidige manier om die testtechnieken te benoemen en in de literatuur te beschrijven. De vele boeken die geschreven zijn over software-testen (bijv. een kleine selectie [Bei90, Bei95, Bin00, PTV02, Cop04, KvdABV06, Ryb07, MSB11, Jor14, SLS14, AO17]) gebruiken nog allemaal hun eigen manier en definities om veelal dezelfde soort technieken te beschrijven. Ook zijn er nog niet echt veel richtlijnen die aangeven welke techniek het beste en snelste werkt, afhankelijk van welke software en welke testdoelen je hebt. Er is hier voor een hoogleraar Software Engineering dus ook nog veel te doen op testgebied !

De meeste technieken die beschreven worden in de literatuur komen echter neer op:

1. maak een model;
2. kies een dekkingscriterium (of *coverage* criterium in het Engels);
3. genereer je testgevallen gebaseerd op het criterium.

In het dagelijks leven en in de de wetenschap worden altijd modellen gebruikt om onze complexe realiteit weer te geven [Wei01, Ryb07]. In Figuur 7 zien we twee voorbeelden. Ook al zijn er daarbij een heleboel details weggelaten, deze twee modellen zijn bruikbaar om bijvoorbeeld te begrijpen waarom het dag en nacht wordt, hoe ver het reizen is van het ene land naar het andere, welke temperatuur er heerst op de verschillende planeten, enz.



Figuur 8: *Partitiemodel*

In de vakdiscipline van het software-testen is dat niet anders. Verschillende modellen, met verschillende niveaus van detail en verschillende niveaus van formaliteit, geven aanleiding tot verschillende testontwerptechnieken. Een van de meest beschreven testontwerptechniek is gebaseerd op het model waarin het inputdomein D van het te testen programma kan worden opgesplitst in subdomeinen $D_1, D_2, \ldots, D_n$. Dit op een zodanige manier dat alle elementen in hetzelfde subdomein leiden tot een hetzelfde gedrag van het programma. Je gebruikt dus kennis over het System Under Test (SUT) om testgevallen die tot eenzelfde soort verwerking leiden, binnen het systeem te groeperen. Dit heet partitie-testen en werd voor het eerst gepubliceerd in de jaren '80 [WO80]. We komen het in de literatuur ook tegen onder de namen domein-testen, equivalentieklassentest, *boundary value*-analyse, grenswaardentest, datacombinatietest en categorie-testen.

Ook al is dit partitiemodel een simplificatie van de realiteit, het geeft ons een belangrijk handvat voor het testen. Het oneindige domein is nu verdeeld in een eindig aantal subdomeinen en het aantal te kiezen testgevallen krijgt dus een beheersbare omvang.

Even terug naar het stukje software dat verantwoordelijk was voor het sturen van de stemkaart. Een mogelijke splitsing van partities zou als volgt kunnen zijn.

	partitie	
geboortedatum	schrikkeljaar	P_1
	geen schrikkeljaar	P_2
verkiezingsdatum	schrikkeljaar	P_3
	geen schrikkeljaar	P_4
verkiezingsdatum - geboortedatum	< 0	P_5
	$\in [0, 18[$	P_6
	$\in [18, 146]$	P_7
	> 146	P_8

We nemen hier de karakteristieken van de datums in acht (schrikkeljaar of niet) en het gedrag zoals we verwachten van de applicatie voor de verschillende datums. Hier gebruiken we de domeinkennis dat de stemgerechtigde

leeftijd 18 jaar en ouder is en dat er geen mensen zijn die ouder zijn geworden dan 146 jaar[HLN17]. Om het kort te houden nemen we alleen even de valide inputwaarden in acht en laten we de niet-valide (bijv. niet bestaande datums) even buiten beschouwing.

Als het model is gemaakt volgt stap twee, het kiezen van een dekkingscriterium. Testdekking geeft aan hoe de testgevallen de verschillende onderdelen van het partitiemodel raken. We kunnen dan dus de test-effectiviteit ("hoe goed hebben we getest") meten in termen van de gekozen dekking van het partitiemodel. Bijvoorbeeld, de tester kan één testgeval per subdomein kiezen, d.w.z. *each choice coverage*. Voor het hiervoor gegeven voorbeeld zou dat neerkomen op de volgende 6 testgevallen waarin elke partitie minstens één keer is gedekt:

test-geval	**geboorte-datum**	**verkiezings-datum**	**resultaat**	**dekt partitie**
1	08-10-1971	22-06-2017	stemmen	P_2, $P4$, P_7
2	04-03-2010	15-05-2019	niet stemmen	P_2, $P4$, P_6
3	29-02-2012	13-04-2017	niet stemmen	P_1
4	04-03-2010	29-02-2012	niet stemmen	P_3
5	29-08-2016	13-04-2014	foutmelding	P_5
6	13-05-1880	22-06-2017	foutmelding	P_8

Een ander dekkingscriterium bestaat uit alle combinaties van partities, d.w.z. het *all combinations coverage*. Dat zou betekenen dat we testgevallen moeten hebben die de 16 combinaties dekken voor (geboortedatum, verkiezingsdatum, verkiezingsdatum-geboortedatum):

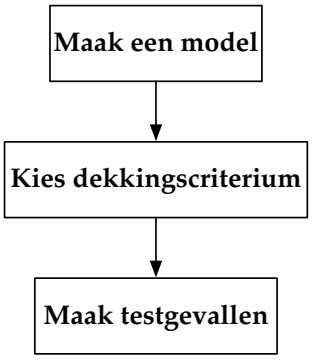
(schrikkeljaar, schrikkeljaar, < 0)
(schrikkeljaar, schrikkeljaar, $\in [0, 18[$)
(schrikkeljaar, schrikkeljaar, $\in [18, 146]$)
(schrikkeljaar, schrikkeljaar, > 146)
(schrikkeljaar, geen schrikkeljaar, < 0)
(schrikkeljaar, geen schrikkeljaar, $\in [0, 18[$)
(schrikkeljaar, geen schrikkeljaar, $\in [18, 146]$)
(schrikkeljaar, geen schrikkeljaar, > 146)
(geen schrikkeljaar, geen schrikkeljaar, < 0)
(geen schrikkeljaar, geen schrikkeljaar, $\in [0, 18[$)
(geen schrikkeljaar, geen schrikkeljaar, $\in [18, 146]$)

(geen schrikkeljaar, geen schrikkeljaar, > 146)
(geen schrikkeljaar, schrikkeljaar, < 0)
(geen schrikkeljaar, schrikkeljaar, $\in [0, 18[$)
(geen schrikkeljaar, schrikkeljaar, $\in [18, 146]$)
(geen schrikkeljaar, schrikkeljaar, > 146)

Bovenstaande testgevallen zijn zogenaamde logische testgevallen. Ze hebben nog geen concrete waarden voor de inputparameters: geboortedatum en verkiezingsdatum, die moeten nog berekend worden. Zodra we die concrete waarden wel hebben, noemen we ze fysieke testgevallen.

Dit zijn maar twee voorbeelden van dekkingscriteria. Er zijn nog vele meer, bijvoorbeeld *pairwise coverage* en het meer algemene *T-wise coverage*. Maar het gaat te ver om daar nu allemaal op in te gaan. Mocht u geïnteresseerd zijn dan verwijs ik naar één van de publicaties daarover [KvdABV06, Jor14, SLS14, AO17].

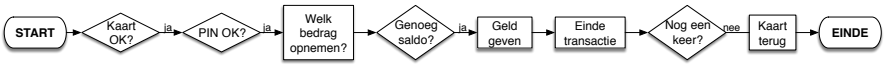
We hebben nu één testontwerptechniek in detail gezien, zodat u een idee krijgt wat het betekent "een testontwerptechniek toe te passen". Het moge duidelijk zijn geworden dat het best veel werk is om zo handmatig een testtechniek toe te passen. En weer geldt, dit was maar een heel simpel stukje programmacode met twee variabelen! Om met de complexiteit van echte software-applicaties om te gaan, kunnen we natuurlijk meer abstractieniveaus aanbrengen in ons model, maar het blijft een arbeidsintensief werkje om handmatig testgevallen te maken.



Figuur 9: *Drie stappen*

Er zijn nog vele andere testtechnieken, maar simpel bekeken volgen ze eigenlijk allemaal dezelfde drie stappen (zie Figuur 9): voor verschillende soorten modellen en verschillende soorten dekkingscriteria. Grafen zijn een veel gebruikt model. In Figuur 6 hebben we een voorbeeld gezien van een graafmodel. Een graaf bestaat uit een verzameling punten, knopen genoemd, waarvan sommige verbonden zijn door lijnen, ook wel zijden, kanten of takken genoemd. De grafen die we ge-

bruiken voor testen, hebben tenminste één beginknoop en één eindknoop. In het stroomdiagram van Figuur 6 zijn de knopen de 'rechthoeken' en de 'ruiten'. Omdat de lijnen de knopen verbinden kun je een pad door een graaf lopen. Bijvoorbeeld:



(PIN-automaat pad 1)

geeft het pad weer dat een gebruiker doorloopt als hij succesvol in één keer geld kan opnemen. Het volgende pad uit de graaf van Figuur 6 wordt doorlopen als er geen geld opgenomen kan worden omdat de pinpas niet geaccepteerd wordt.

(PIN-automaat pad 2)

Deze paden vormen twee logische testgevallen voor de software die door de graaf wordt beschreven. Dekkingscriteria voor grafen zijn onder andere gebaseerd op de elementen die in de paden voorkomen (bijv. knopen en kanten) of het aantal paden zelf. Bijvoorbeeld: knopendekking (of *node coverage* in het Engels), alle knopen worden minstens één keer geraakt in de testgevallen. Voor het testgeval weergegeven door pad 1 is de knopendekking 10 van de in totaal 18 knopen (dat is 55,6%). De knopendekking van pad 2 is 27,8%. Samen doorlopen ze 11 van de 18 knopen (61,1%). Met hoeveel testgevallen kunnen we 100% knopendekking bereiken?

Een ander dekkingscriterium is kantendekking (of *edge coverage* in het Engels): alle kanten worden bij kantendekking minstens één keer geraakt in de testgevallen. Voor het testgeval weergegeven door pad 1 van hiervoor is de kantendekking 9 van de in totaal 26 kanten (dat is 34,6%). De kantendekking van pad 2 is 15,4%. Samen doorlopen ze 11 van de 26 kanten (42,3%). Met hoeveel testgevallen kunnen we 100% kantendekking bereiken?.

Hiervoor hadden we al gezien dat er 25.975.862 paden zijn van START naar EIND in de graaf van Figuur 6. Complete padendekking (of *path coverage* in het Engels) zou dus bestaan uit hetzelfde aantal testgevallen. We hadden echter al gezien dat een dergelijke complete test onhaalbaar is.

Ik denk dat ik me een beetje heb laten meeslepen in het beschrijven van gestructureerd testen. Mijn bedoeling was om te laten zien hoe complex het testen eigenlijk kan zijn, als je het goed wilt uitvoeren. Ik denk dat dat wel aangetoond is.

Hoe past dat nu in het onderzoek waar ik me de afgelopen 10 jaar mee heb bezig gehouden?

Om testontwerp allemaal handmatig te doen, dat kost een hoop tijd. Handig zou het dus zijn om dit te automatiseren, zo veel als we kunnen. Helaas komt er bij het ontwerpen van het benodigde testmodel dezelfde creativiteit kijken als bij het ontwikkelen van de software zelf. Dit is een creativiteit die door het imperfecte menselijk brein goed gedaan kan worden (al is het met hier en daar een foutje), maar het automatiseren ervan is erg uitdagend. Met die uitdaging houdt mijn onderzoek zich bezig.

8.2 Willekeurig, ofwel random, testen

Een manier om testgevallen te kiezen die relatief makkelijk te automatiseren zijn, is willekeurig, ofwel random, kiezen. We gebruiken geen domeinkennis, we maken geen creatief testmodel. Maar we selecteren gewoon willekeurig testgevallen uit het inputdomein van een software-applicatie. Herinnert u zich de twee problemen die we eerder hebben gezien, die testen zo complex maken:

1. We hebben een oneindig aantal testgevallen. Hoe kunnen we nu die gevallen kiezen die je zou moeten gaan gebruiken voor het uitvoeren van jouw testen?
2. We weten niet hoeveel fouten er in onze software kunnen zitten. Dus hoe weten we achteraf of we goed getest hebben? Of met andere woorden, welke dekkingscriteria kunnen we gebruiken als we random testen?

Probleem nummer 1 is opgelost als we lekker makkelijk random kiezen. Het andere probleem blijft echter bestaan. En zoals het vaak betaamt in het leven, het oplossen van het ene probleem resulteert weer in een ander probleem. Aangezien we geen domeinkennis gebruiken tijdens het kiezen van de testgevallen:

3. Hoe weten we dan eigenlijk wat een fout is? Hoe kunnen we die fouten herkennen?

Dit laatste staat bekend als het orakelprobleem in de testwereld [Wey82, BHM+15, OKN15]. Een orakel[1] is een mechanisme dat correct en incorrect gedrag van een softwaresysteem van elkaar kan onderscheiden. Als we automatisch testen dan kunnen er in korte tijd heel veel testgevallen uitgevoerd worden. Het is vervolgens natuurlijk niet te doen om alle resultaten van die testen handmatig te checken. De orakels moeten in dit geval dus ook geautomatiseerd worden.

Het automatiseren van orakels is nog grotendeels een open probleem. Het is niet heel vreemd dat dit nog een open probleem is, want opnieuw komt bij het maken van orakels diezelfde creativiteit kijken die je nodig hebt om de software of een testmodel te ontwikkelen. Stel bijvoorbeeld dat het mogelijk is om een compleet geautomatiseerd orakel te schrijven dat in staat is om alle fouten in een programma te kunnen vinden. Het is dan niet meer nodig om het programma te schrijven want het orakel kon dan gewoon gebruikt worden [OKN15].

De ontbrekende orakels zijn een belangrijk knelpunt voor effectieve testautomatisering en het op grote schaal oppakken van geautomatiseerd testmethoden en testtools door bedrijven. Ik kom daar straks nog even op terug. Eerst even verder met random testen.

Random testen is namelijk erg omstreden in 'onze' geschiedenis. In de jaren '70 waren de meningen daarover erg verdeeld. Girard en Rault (1973) [GR73] noemden het een:

waardevol schema om testgevallen te genereren.

Dit wordt bevestigd door Thayer, Lipow en Nelson (1978) [TLN78] waarin ze schrijven dat het de:

noodzakelijke laatste stap is na alle testactiviteiten.

Echter, Glenford Myers (1979) in zijn baanbrekende boek op het gebied van Software Testing [Mye79] duidt random testen aan als:

waarschijnlijk de slechtste testmethode.

In 1984, echter, voerden Duran en Ntafos [DN84], een reeks experimenten uit waarin ze aantoonden dat random testen effectiever zou kunnen zijn dan meer gestructureerd testen. Hamlet en Taylor [HT88] herhaalden meer experimenten en kwamen tot dezelfde resultaten. Weyuker met Jeng vergeleken beide testmethoden vanuit een analytisch oogpunt [WJ91]. Echter, de resultaten wezen weer in dezelfde richting: een duidelijke superioriteit van meer gestructureerd testen kon niet worden aangetoond.

Deze resultaten openden de deuren naar een grote hoeveelheid onderzoek naar de eigenschappen en de voordelen van random testen. Veel auteurs onderzochten de voorwaarden waaronder random testen effectiever kon zijn dan gestructureerd testen of andersom (zie bijvoorbeeld [TDN93, CY94, Gut99, AB11, AIB12]). Een recente studie [BP16] waarin de tijd die het testen kost ook als factor wordt meegenomen, concludeert dat:

zelfs de meest effectieve testtechniek inefficiënt is in vergelijking met random testen als het genereren van een testcase relatief te lang duurt.

Hier ligt nog een onontgonnen onderzoeksterrein, waar ik de komende jaren bij mijn onderzoek op het gebied van softwaretesten zeker nog het één en ander in zal gaan verkennen. Het is namelijk zo dat de testtool TESTAR[2] die we de afgelopen jaren heb ontwikkeld ook gebaseerd is op de krachten van het random testen. Het is belangrijk om te onderzoeken wat we achteraf - na random getest te hebben - weten over de kans dat er nog fouten in de software achter zijn gebleven. En ook hebben we natuurlijk orakels nodig.

Maar voordat ik op TESTAR inga wil ik eerst nog even wat vertellen over ander onderzoek dat ik heb gedaan op het gebied van automatisch testen. Uiteindelijk komen die technieken namelijk allemaal samen in het toekomstige onderzoeksplan van TESTAR.

8.3 Zoekgebaseerd testen

In plaats van de testgevallen willekeurig te kiezen, kunnen we ook proberen ze op zo een manier te kiezen dat we een bepaald doeleinde optimaliseren.

Als we het over testen hebben is dat doel bijvoorbeeld het vinden van bepaalde fouten of het optimaliseren van een bepaald dekkingscriterium.

Voor optimalisatie-problemen in grote en complexe zoekruimten worden sinds de jaren 70 al met veel succes evolutionaire algoritmen [Gol89, Hol75] gebruikt. De term *evolutionaire algoritmen* omvat alle stochastische zoekalgoritmen die zijn gebaseerd op *reproductie* + *selectie* van kandidaatoplossingen. Evolutionaire algoritmen bootsen de processen van de natuurlijke genetica en Darwin's theorie van biologische evolutie na.

Het toepassen van evolutionaire algoritmen op testen wordt vaak evolutionair testen genoemd en is als eerste toegepast door Wegener in 2000 [WG00]. Dit lijkt een goede *match* te zijn; de afgelopen 15 jaar is er een hoop onderzoek naar gedaan (goede overzichtsartikelen vind je hier [WBS02, HHH+02, McM04, McM11, HJZ15] of neem een kijkje in de hele *repository* (database van artikelen) [ZHM17]). Aangezien het onmogelijk is om op alle mogelijke fouten in een stuk software te anticiperen, kunnen evolutionaire algoritmen een prominente rol spelen bij het testen. Evolutionaire algoritmen maken zeer weinig aannames over het onderliggende probleem dat ze proberen op te lossen. Daarnaast zijn stochastische optimalisatie- en zoektechnieken adaptief en kunnen die hun gedrag aanpassen wanneer ze geconfronteerd worden met nieuwe onvoorziene situaties. Deze twee eigenschappen - hun vrijheid van beperkende veronderstellingen en hun inherente adaptiviteit - maken evolutionaire testbenaderingen ideaal voor het omgaan met complexe software.

Evolutionaire algoritmen worden gekenmerkt door een iteratieve procedure die in parallel werkt aan een verzameling van potentiële oplossingen. Deze verzameling wordt *populatie* genoemd en de potentiële oplossingen *individuen*. Individuen hebben een *fitness*-waarde die aangeeft hoe goed de potentiële oplossing is die ze representeren. In de context van evolutionair testen zijn de individuen bijvoorbeeld de testgevallen en de fitness geeft aan hoe goed ze een bepaald deel van de software dekken. Een typische procedure van een evolutionaire algoritme bestaat uit:

1. initialiseer een populatie van potentiële oplossingen
2. evalueer de fitness van elk individu
3. selecteer twee individuen uit de populatie volgens een vooraf gedefinieerde selectiestrategie

4. combineer deze twee individuen om een nieuwe te produceren (analoog aan biologische voortplanting)
5. pas mutatie toe indien gewenst
6. evalueer de fitness van het nieuwe individu
7. beslis op basis van een vooraf gedefinieerde herintroductie-strategie of de nieuwe individuen geschikt zijn om een de volgende iteratie te maken
8. herhaal tot het optimum is bereikt, of een andere stopwaarde is vervuld.

Evolutionaire algoritmen zijn generiek en kunnen toegepast worden op een breed scala aan optimalisatieproblemen. Om een evolutionair algoritme voor een specifiek probleem te ontwikkelen, moet men een probleemspecifieke fitness-functie definiëren. De fitness-functie vergelijkt en contrasteert oplossingen van de zoekopdracht met betrekking tot het zoekdoel. Met behulp van deze informatie is de zoekopdracht gericht op potentiële veelbelovende gebieden van de zoekruimte.

Om softwaretesten te automatiseren met behulp van evolutionaire algoritmen, moet het testdoel worden omgezet in een optimalisatietaak. Afhankelijk van welk testdoel wordt nagestreefd, moeten er fitnessfuncties worden gedefinieerd om de testgevallen te evalueren.

Als een geschikte fitnessfunctie kan worden gedefinieerd voor het testdoel en evolutionaire algoritmen worden gebruikt als de zoektechniek, dan volgt de Evolutionaire Test de stappen zoals hierboven beschreven. De initiële populatie-testgegevens worden gegenereerd, meestal willekeurig. Maar in plaats van willekeurige initialisatie kan men ook testgegevens gebruiken die zijn verkregen door een vorige systematische test. Op deze manier kan de evolutionaire test profiteren van de kennis van de tester over de SUT. Na initialisatie vertegenwoordigt elk individu binnen de populatie een testgeval waarmee de SUT wordt uitgevoerd. Voor elke testgeval wordt de fitnesswaarde bepaald door de SUT uit te voeren met dat testgeval. Vervolgens worden testgevallen met hoge fitnesswaarden geselecteerd met een hogere kans dan die met een lagere waarde en worden die, zoals eerder uitgelegd, onderworpen aan combinatie- en mutatieprocessen om

nieuwe nakomelingen te genereren. Een nieuwe populatie van testgegevens wordt gevormd door nakomelingen en oudere individuen samen te voegen volgens de vastgestelde overlevingsprocedures.

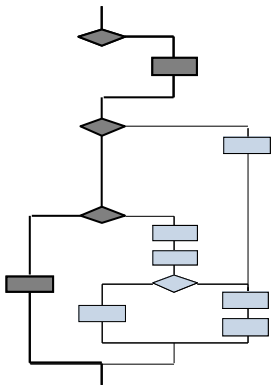
Hieraan heb ik gewerkt tijdens een Europees project dat ik heb gecoördineerd van 2006 tot 2010. EvoTest heette dat project, een acroniem voor *Evolutionary Testing* ofwel EVOlutionair TESTen [Vos09, SVW09]. Dit was een project gesubsidieerd door de Europese Commissie[3]. In het project hebben we evolutionair testen toegepast op twee soorten testen:

1. Structureel, of *white-box*, testen, dat als doel heeft het dekken van de programmacode van de software [GKWV09, VBL+10].
2. Functioneel, of *black-box*, testen, dat als doel heeft fouten te vinden in de implementatie van een bepaalde functionaliteit [VLW+13].

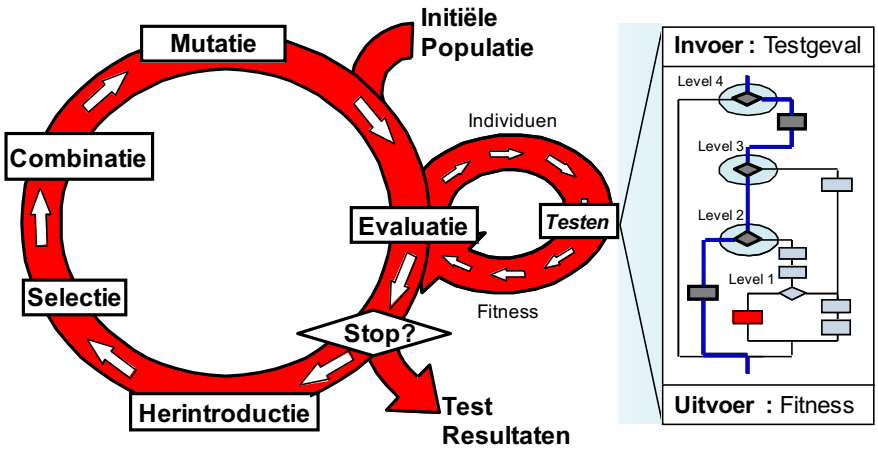
Ik ga hier in de volgende subsecties even op in.

8.3.1 Structureel evolutionair testen

Om verschillende dekkingscriteria van de programmacode te definiëren wordt vaak het zogenaamde programmastroomdiagram (of *control-flow graph* in het Engels) van de broncode gebruikt. Dat is een abstracte voorstelling van alle mogelijke opeenvolgingen van programmacode-elementen (bijv. instructies en beslissingen) die zich kunnen voordoen tijdens de uitvoering van een stuk programmacode. Zie bijvoorbeeld Figuur 10, daar zijn de 'rechthoeken' instructies en de 'ruiten' stellen de beslismomenten voor. De takken (in het Engels *branches*) tussen de verschillende programmacode-elementen kunnen samen paden vormen die door een programma doorlopen kunnen worden. In Figuur 10 is er één pad bij wijze van voorbeeld grijs gekleurd.



Figuur 10: *Stroomdiagram*



Figuur 11: *Workflow van evolutionair structureel, of* white-box*, testen*

De meeste gebruikte dekkingscriteria zijn:

Instructiedekking of programmaregeldekking (*statement coverage* in het Engels) is het percentage van instructies die zijn uitgevoerd door een verzameling testgevallen.

Beslissings- of takkendekking (*decision coverage* of *branch coverage* in het Engels) is het percentage van beslissingsuitkomsten die zijn uitgevoerd door een verzameling testgevallen.

Programmapaddekking (*path coverage* in het Engels) is het percentage programmapaden dat door een verzameling testgevallen is uitgevoerd. 100% programmapaddekking impliceert zowel 100% beslissingsdekking als 100% instructiedekking.

In het EvoTest-project hebben we samengewerkt met Joachim Wegener, de pionier op het gebied van evolutionair testen. We hebben evolutionaire algoritmen toegepast om takkendekking te optimaliseren [GKWV09, VBL+10]. In Figuur 11 wordt dat geïllustreerd. De individuen (d.w.z. de testgevallen) bestaan uit inputwaarden voor een stuk programmacode. Een afzonderlijke zoekopdracht wordt uitgevoerd voor elke onontdekte tak (de doeltak) die is vereist voor volledige takkendekking.

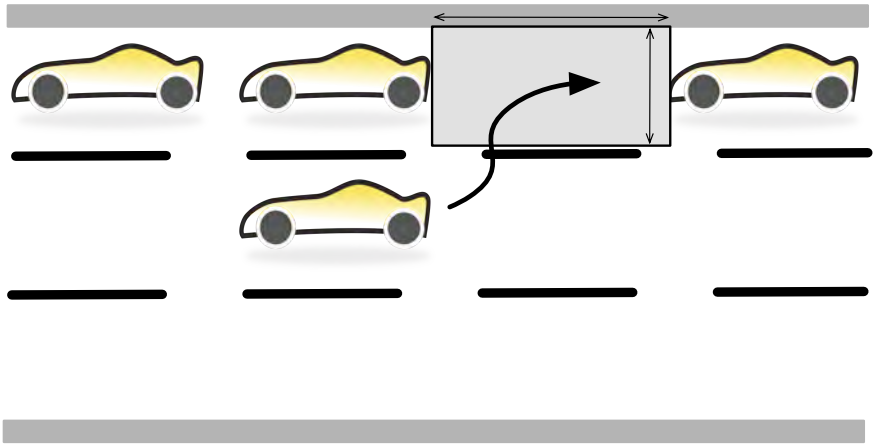
De fitness-evaluatie van een testgeval wordt berekend door het programma uit te voeren met het desbetreffende testgeval. Deze uitvoering resulteert in het bewandelen van een programmapad (bijvoorbeeld het blauwe pad in Figuur 11). Als de doeltak (de tak die naar het rode 'rechthoekje' leidt in Figuur 11) deel uitmaakt van het programmapad, is de tak gedekt en de fitnesswaarde is minimaal (dus 0) en is voor deze deelzoekopdracht het optimum behaald. Wanneer echter de doeltak gemist wordt, wordt het kritische predicaat bepaald: dat is de beoordeling die hoort bij het beslissingspunt in het programmastroomdiagram waarbij het bewandelde programmapad afwijkt van de richting naar de doeltak.

De variabele voor het benaderingsniveau(*approach_level*) is gedefinieerd als de afstand tussen het beslissingspunt waar het kritieke predicaat zich bevindt en de doeltak. Vervolgens wordt de afstand tussen de takken (de variabele *branch_dist*) bepaald door de afstand tussen de werkelijke waarden van de variabelen bij het kritieke predicaat te berekenen en de waarden die nodig zijn om de kritische predicaat die waarde te laten hebben dat het programmapad wel de kant van de doeltak opgaat.

De (minimaliserende) fitnessfunctie heeft de volgende basisvorm: t is de doeltak van de zoekopdracht en i het individu dat geëvalueerd wordt:

$$\mathit{fitness}(i,t) = \mathit{approach\_level}(i,t) + \mathit{branch\_dist}(i,t)$$

De *approach_level* is een natuurlijk (dus geheel) getal en de *branch_dist* is een reëel getal in het interval $[0:1]$. We hebben deze evolutionaire testen met succes toegepast en gevalideerd op een aantal programmacodefragmenten geschreven in de programmeertaal C door autofabrikant DaimlerChrysler [VBL+12]. Het ging hier om onder andere programmacodefragmenten voor rem-assistenten die als *'embedded'* software in de auto's wordt gebruikt. In het artikel [VBL+12] kunt u de positieve resultaten van deze studie nalezen. We laten daar zien hoe we automatisch en zonder domeinkennis testgevallen kunnen genereren die hoge dekkingspercentage op programmacodeniveau bereiken! Voor veiligheidskritische systemen (waaronder auto's) wordt een bepaalde mate van programmacodedekking verplicht door veiligheidsnormen en voorschriften (bijvoorbeeld [ISO08, IEC98, RTC92]). Automatisch en evolutionair testen, zodat deze dekking wordt verkregen, is dus erg interessant voor dit soort software.



Figuur 12: *Automatisch inparkeren*

8.3.2 Functioneel evolutionair testen

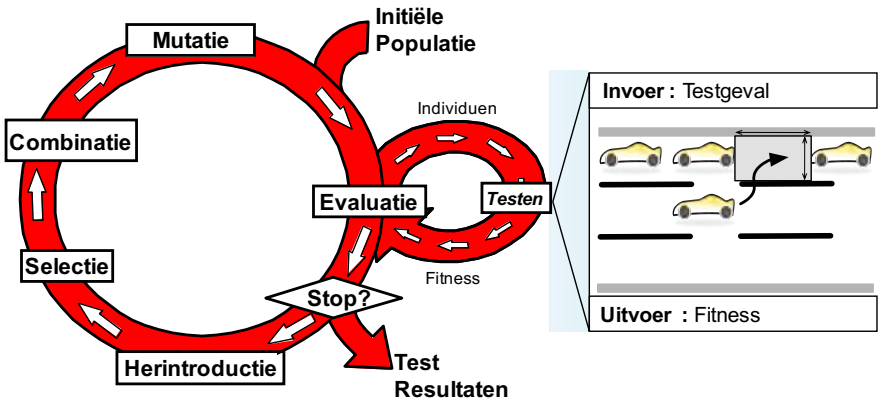
Voor functioneel testen is het testdoel een fout te vinden ten opzichte van een geselecteerde functionele eigenschap. De fitnessfunctie moet dus zo worden gedefinieerd dat het mogelijk is om te meten hoe dicht een testgeval in de buurt is om de geselecteerde eigenschap te breken. Ook dit hebben we gedaan in het EvoTest-project [BVD10, VLW+13]. Het voorbeeld aan de hand waarvan dit soort testen het makkelijkste kan worden uitgelegd is door middel van een voorbeeld uit het dagelijks leven: het parkeren van een auto. Bij DailerChrysler, een partner van het EvoTest-project, is jarenlang gewerkt aan een autonome parkeermodule, die bedoeld is om een auto te parkeren in een scenario zoals in Figuur 12 [BW04]. Voor dit doel is de auto uitgerust met omgevingssensoren die voorwerpen registreren rondom de auto. Bij het passeren kan het systeem voldoende grote parkeerplaatsen herkennen en kan aan de bestuurder signaleren dat er een parkeerplaats is gevonden. Als de bestuurder besluit om te parkeren, kan dit automatisch gebeuren. De parkeersoftwaremodule maakt gebruik van de geregistreerde gegevens van de parkeerplaats, samen met de positie van de auto. De software manoeuvreert de auto in de parkeerplaats door de snelheid en de stuurhoek te regelen. Dit moet natuurlijk gebeuren zonder ergens tegenaan te botsen! Dat is een fundamentele functionele eigen-

schap van deze software en die moet dan ook goed getest worden voordat die daadwerkelijk opgenomen kan worden in auto's die in productie gaan.

Handmatig testen van dit complete systeem is duur en tijdrovend. Elk testgeval zou bestaat uit het opbouwen van een parkeerscenario met echte auto's en iemand zou de autonome parkeerassistent elke keer in de parkeerplek moeten laten manoeuvreren. Geautomatiseerde functionele tests in een gecontroleerde simulatie-omgeving zijn veel minder duur en daarvan kunnen er dan ook meer worden uitgevoerd. Als we evolutionair testen willen toepassen moeten we het testen dus vertalen naar een optimalisatieprobleem. Eerst moeten we dan definiëren hoe we de testgevallen kunnen generen. Daarna moeten bepalen hoe we die testgevallen kunt evalueren met behulp van een fitnessfunctie.

De testgevallen (de individuen van de evolutionaire zoektocht) bestaan uit de waarden die nodig zijn om een parkeermanoeuvre te simuleren: de grote van de parkeerplek en de positie van de auto. Een simulatie-omgeving bootst dan een parkeermanoeuvre na, gebruikmakende van de software die we aan het testen zijn.

Zoals uitgelegd, bij evolutionair testen, wordt de evaluatie van de testgevallen uitgevoerd door de fitnessfunctie. De fitnesswaarde geeft een indicatie over hoe goed of slecht de parkeermanoeuvre is. Deze fitnesswaarde vertegenwoordigt de kwaliteit van het bijbehorende testgeval en is bedoeld om de evolutionaire zoektocht naar een richting van foutieve invoersituaties te leiden. Een foute parkeermanoeuvre is wanneer de auto botst met andere voorwerpen, of, in andere woorden, als de afstand tussen de in te parkeren auto en andere auto's of voorwerpen 0 is. Om fouten in de parkeermodule te vinden willen we dus testen of het mogelijk is die afstand te minimaliseren! In Figuur 13, die erg lijkt op Figuur 11, wordt geïllustreerd hoe de evolutionaire lus het systeem uitvoert, de fitness-waarden evalueert en individuen evolueert. De resultaten van de evolutionaire testen van dit systeem kunt u lezen in [BW04]. Op soortgelijke manier hebben we in [BVD10, VLW+13] evolutionaire testen gedaan van automodules voor Daimler's Adaptieve Cruise Control (ACC)-software en een Anti-Lock Braking System (ABS).



Figuur 13: *Workflow van evolutionair functioneel, of* black-box*, testen van parkeermodule*

9 TESTAR: Automatisch testen op GUI-niveau

Na het EvoTest-project had ik het voorrecht om een nieuw Europees project te mogen coördineren. Het FITTEST-project, ofwel het *"Future Internet Testing"*-project. Opnieuw een project gesubsidieerd door de Europese Commissie[4]. In dit project werd het toekomstige internet beschreven als een complexe interconnectie tussen diensten, applicaties, inhoud en media, eventueel aangevuld met semantische informatie. Het is gebaseerd op technologieën die een rijke gebruikerservaring bieden, waarmee de bestaande hyperlink-gebaseerde navigatie wordt uitgebreid en verbeterd. Steeds meer diensten worden aangeboden via het Internet, zelfs voor kritieke activiteiten zoals openbare diensten, sociale diensten, overheid, leren, financiën, zaken en entertainment. Testen van het toekomstige internet is dus erg belangrijk en het doel van FITTEST was testtechnieken en tools te ontwikkelen voor applicaties van het toekomstige Internet.

Met acht partners uit Nederland, Spanje, Verenigd Koninkrijk, Duitsland, Israël, Italië, Frankrijk en Finland hebben we gewerkt aan testtechnieken voor continue testen, regressie-testen, concurrency-testen, combinatorisch testen en automatisch testen op gebruikersinterface-niveau. Ik ga nu niet al die technieken de revue laten passeren, want daar is simpelweg geen tijd voor. U kunt de resultaten van het project en de genoemde test-

technieken nalezen in de volgende overzichten: [BLVW11, VTW+11, VLB14, VTW+13, VTP+14].

Ik wil het hier alleen hebben over de testtechnieken en gereedschappen voor het automatisch testen op gebruikersinterface-niveau. Daar heb ik mij veel mee bezig gehouden *tijdens* het project en *na* het project en ook mijn *toekomstige* onderzoekswerkzaamheden zullen daar over gaan. U raadt het al, het gaat over TESTAR, de eerder genoemde tool om automatisch te testen op het niveau van de Grafische User Interface (GUI) van een applicatie [VKC+15].

9.1 Wat is TESTAR?

TESTAR is een tool die applicaties automatisch kan testen via de GUI. In sectie 6 hebben we al een voorbeeld gezien van dit soort systeemtesten. We zagen dat testgevallen via de GUI bestaan uit test-sequenties die aangeven welke acties je moet doen op de GUI. Bijvoorbeeld:

```
1.  Klik op het menu met label:  Communicatie
2.  Selecteer optie:  Stemkaart
3.  Vul in het daarvoor bestemde veld de verkiezingsdatum in
4.  Klik op de knop:  Stuur stemkaart
```

Nadat deze acties zijn uitgevoerd in deze sequentie, moet er met behulp van een orakel gecheckt worden of de test geslaagd is of niet.

Er zijn een heleboel tools beschikbaar die de activiteiten van het testen van applicaties op het GUI-niveau ondersteunen. De meeste tools zijn gebaseerd op de zogenaamde *Capture and Replay* (CR)-methode, de "Opnemen en herspelen"-methode [SHS10, MS03, LCRT13, NB13].

Die tools werken als volgt. Stel een tester heeft een testgeval ontwikkeld, zoals in het voorbeeld hierboven: een sequentie van acties en invoerwaarden zoals klikken, toetsaanslagen, slepen, typen en neerzetten, enz. Terwijl de tester dit handmatig uitvoert, neemt de tool deze sequentie op in een script dat achteraf automatisch kan worden herspeeld. Denk bijvoorbeeld aan het volgende testscript:

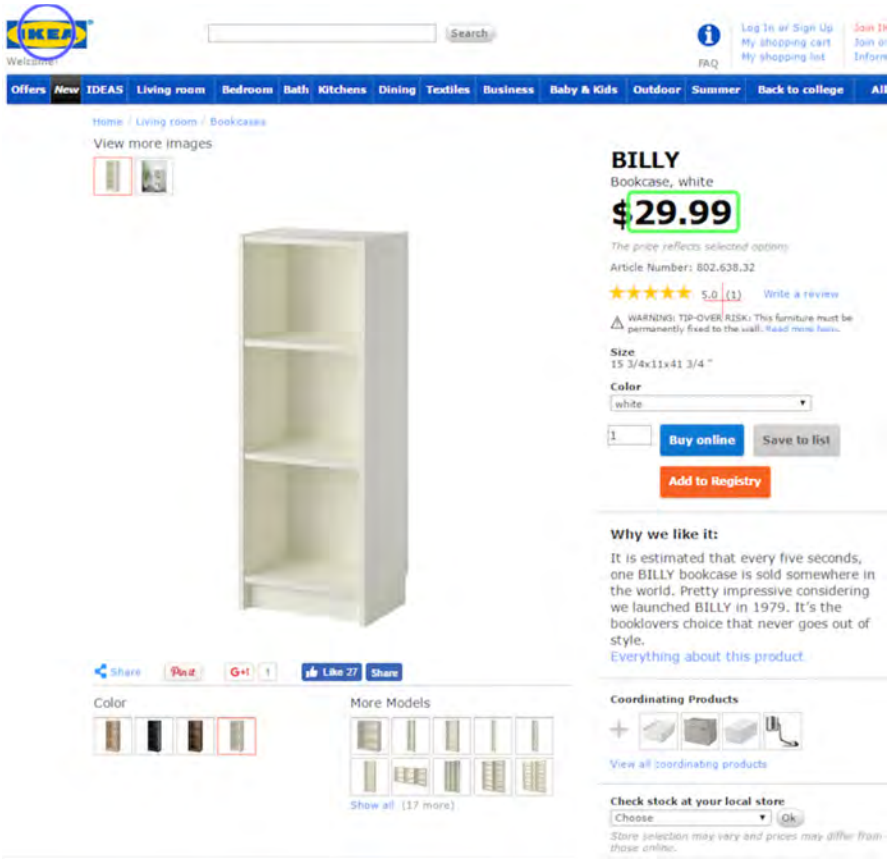
```
set_focus_to_window("$browser");
click_menu("Communication");
```

```
select_option("Stemkaart");
enter_text("field1", "22-06-2017");
click_button("Stuur stemkaart");
verify(get_field("outcome"), "Stemkaart gestuurd");
```

Dat testscript kan dan later weer automatisch door de computer worden uitgevoerd. Het belangrijkste probleem bij deze aanpak is dat de opgenomen sequenties vaak breken, door wijzigingen in de gebruikersinterface (een knop wordt bijvoorbeeld verwijderd, veranderd van naam of wordt op een andere plek gezet, enz.). Dit betekent dat de testers de opgenomen scripts voortdurend moeten repareren om de testgevallen te kunnen blijven onderhouden. In de praktijk resulteert dit in een tester die oude testgevallen bijschaaft, in plaats van nieuwe te maken die nieuwe fouten vindt! In de praktijk worden deze tools vaak aangeduid als "Shelfware"[Kan98, MS03, Mem02, Sil03]. Ze blijven meestal na aankoop op de plank liggen en worden niet gebruikt. Een ander probleem met deze soort tools is, dat ze zich uitsluitend richten op een bepaalde gebruikersinterfacetechnologie waardoor de testen dus zeer moeilijk zijn te porteren naar andere platformen.

Naast CR-tools zijn er zogenoemde Visuele GUI Test (VGT)-tools (bijvoorbeeld Eggplant [Tes17], Sikuli [Sik17], eyeAutomate[Auq17][5]). Deze tools zijn gebaseerd op beeldherkenning en worden visueel genoemd omdat ze de GUI-elementen visueel kunnen identificeren (of 'zien' zeg maar), net als een menselijke tester. VGT-tools kunnen testen automatiseren voor elk type gebruikersinterface, ongeacht de gebruikte technologieën en besturingssystemen [YCM09, ANO13].

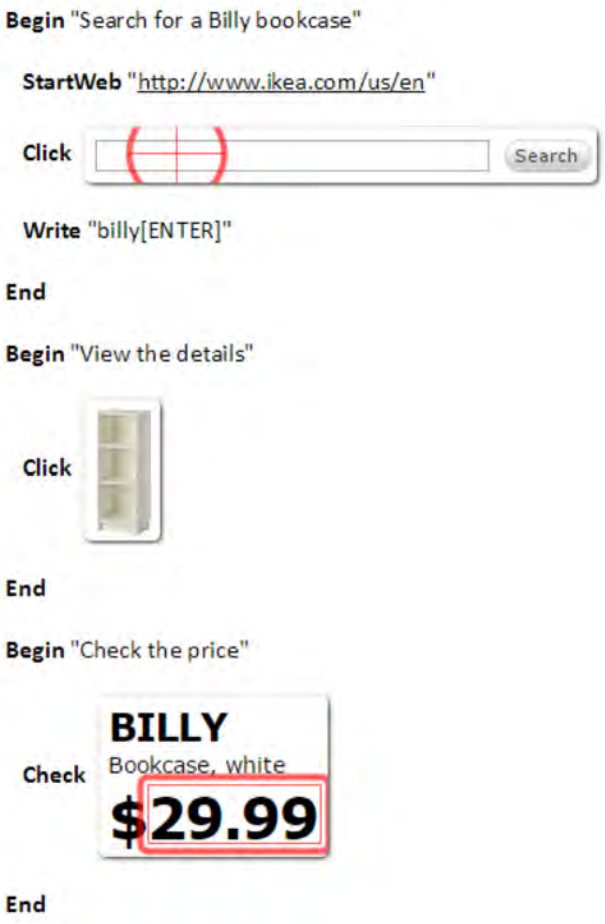
Laten we naar een heel simpel testvoorbeeld kijken, de website van IKEA (zie Figuur 14). Ik ken weinig mensen die geen 'Billy' thuis of op hun werk hebben staan. Stel we willen de prijs van die Billy checken. Een typisch visueel testscript in dit scenario ziet er dan uit als in Figuur 15[6]. Deze tools vertrouwen ook op de stabiliteit van de grafische weergave van de gebruikersinterface. Wijzigingen in een applicatie zijn vaak ook veranderingen in de gebruikersinterface, waardoor ook de visuele benadering bedreigd wordt. Ook bij deze aanpak zijn we tijd kwijt aan het onderhouden van de testscripts. Daarbij komt ook dat visuele aanwijzingen in de gebruikersinterface de beeldherkenning kunnen misleiden. Dit heeft als



Figuur 14: *Stel we willen het IKEA-web testen...*

gevolg dat er valse positieven (verkeerde UI-elementidentificatie) en valse negatieven (gemiste UI-elementen) ontstaan [AFR13, ASM16, AF17].

Het gebruiken van testscripts voor het automatiseren van testen op GUI-niveau zorgt dus voor een groot onderhoudsprobleem als de GUI wijzigt. En dat gebeurt nogal eens. GUI's veranderen tijdens de ontwerpfase (layout-veranderingen bijvoorbeeld), de ontwikkelfase (bijvoorbeeld door restricties van het platform of the technologieën die worden gebruikt) en tijdens de uitvoerfase (bijvoorbeeld omdat ze zich moeten aanpassen aan verschillende schermresoluties en -oriëntaties). De scripts kunnen vaak niet makkelijk hergebruikt worden en de tester is constant bezig om

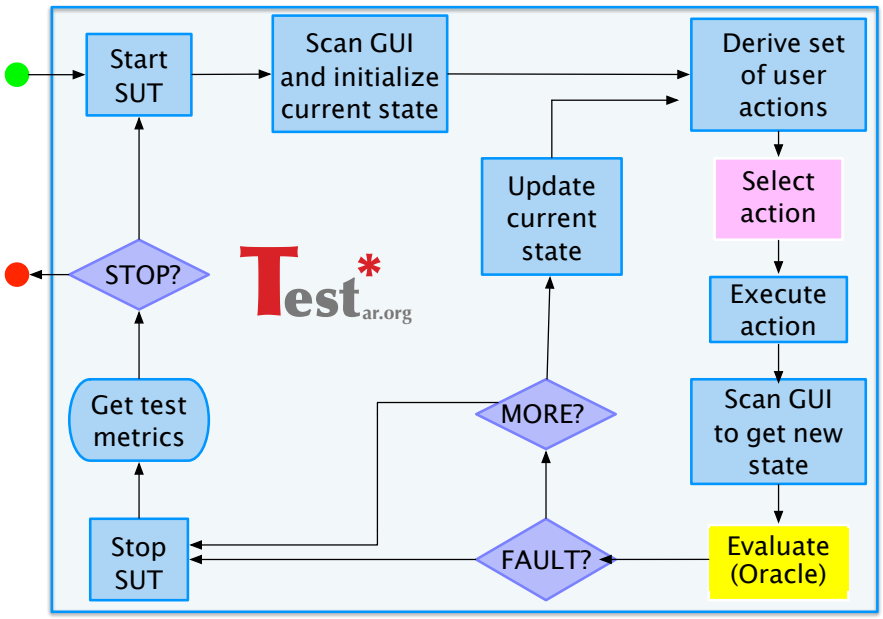


Figuur 15: *Voorbeeld van een visueel test-script*

bestaande tests werkend te houden.

Vandaar dat onze TESTAR aanpak *scriptless* is, dat wil zeggen er worden geen scripts opgenomen of geschreven om de testen te automatiseren. En wat er niet is, dat zorgt ook niet voor onderhoudsproblemen! Op deze manier kunnen testers zich concentreren op het vinden van fouten en raken ze geen tijd kwijt aan onderhoud.

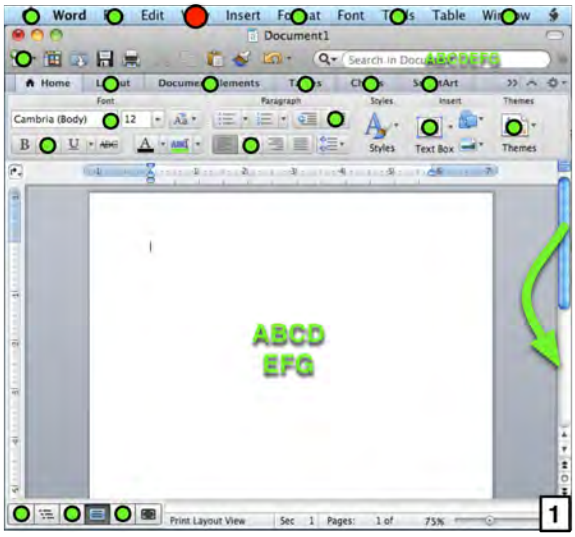
9.2 Hoe werkt TESTAR precies?



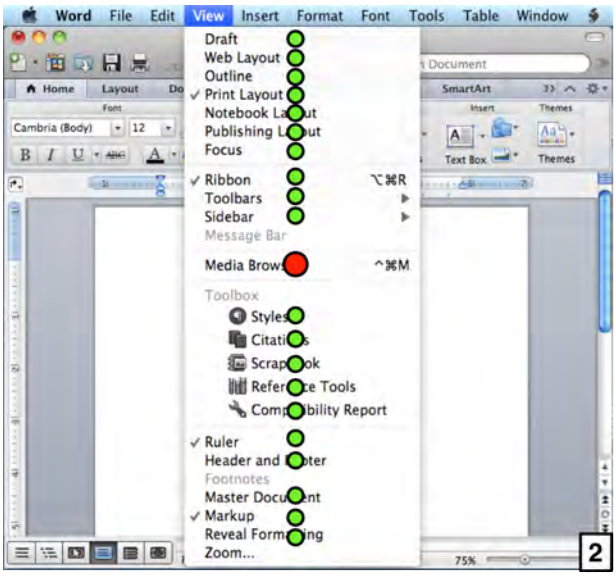
Figuur 16: *TESTAR*

TESTAR voert de stappen uit zoals die zijn te zien in Figuur 16:

1. Start de software die je wilt testen (Software Under Test (SUT)).
2. Scan de GUI en verkrijg de *toestand* waarin deze GUI van de SUT zich bevindt. Deze toestand bestaat uit alle *widgets* die in die bepaalde

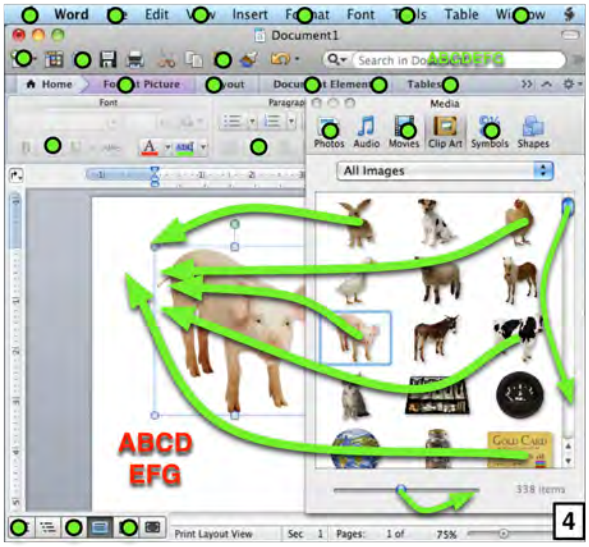


Figuur 17: *Voorbeeld van een toestand waarin de GUI van Word kan zijn.*



Figuur 18: *Voorbeeld van een toestand waarin de GUI van Word kan zijn.*

Figuur 19: *Voorbeeld van een toestand waarin de GUI van Word kan zijn.*



Figuur 20: *Voorbeeld van een van de toestanden waarin de GUI van Word kan zijn.*

toestand van de GUI zijn te zien (d.w.z. alle knoppen, menu's, tekstvelden, plaatjes, enz.) en al hun bijbehorende eigenschappen zoals: weergave positie, grootte, kleur, titel, enz. In de Figuren 17 tot en met 20 zie je voorbeelden van toestanden. De groene bolletjes, pijlen en teksten zijn voorbeelden van *widgets* waar de gebruiker in deze toestand op zou kunnen klikken, mee zou kunnen schuiven of in zou kunnen schrijven.

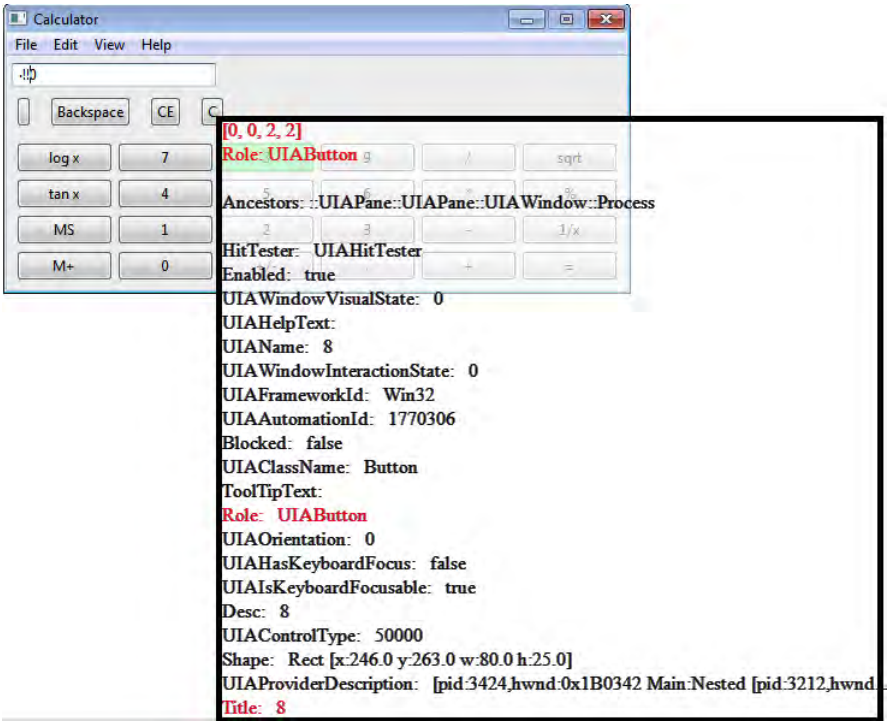
3. Leid een verzameling van mogelijke acties af die een gebruiker zou kunnen uitvoeren in deze specifieke toestand. Als we weer kijken naar Figuren 17 tot en met 20 zijn dat alle mogelijke groene bolletjes, pijlen, teksten, enz.

4. Selecteer één van deze acties (groene bolletjes) uit de verzameling. In Figuren 17 tot en met 20 is de geselecteerde actie het rood gekleurde bolletje. Hier komen we dan, zoals beloofd, terug op random (waar we het in Sectie 8.2 over hadden). In de standaard modus van TESTAR wordt deze uit te voeren actie random gekozen uit de verzameling. In de meer geavanceerde modus kunnen we slimmer selecteren (daar komen we hierna op terug).

5. Voer de geselecteerde actie uit (bijvoorbeeld klik, typ, schuif).

6. Pas de beschikbare orakels toe om de correctheid van de nieuwe toestand waarin de gebruikersinterface zich bevindt te controleren. Als er een fout is gevonden, stop dan de SUT en sla de testreeks op die de fout heeft gevonden. Als dat niet het geval is ga naar stap (2).

Om de toestand waarin de GUI van de SUT zich bevindt te verkrijgen, gebruikt TESTAR de toegankelijkheids-API (*Application Program Interface*) van een besturingssysteem. Oorspronkelijk is deze API bedoeld om de toegankelijkheid van softwareproducten te verhogen voor personen met een fysieke handicap, zoals een beperkte motoriek of een verminderd gezichtsvermogen. Deze API voorziet toegankelijkheidstoepassingen van informatie over een GUI die gebruikt kunnen worden om bijvoorbeeld de GUI voor te lezen of om te zetten in braille. TESTAR gebruikt deze informatie dus om te weten wat er op de GUI staat en welke mogelijke acties er geselecteerd kunnen worden om testreeksen te construeren.



Figuur 21: Informatie via de toegankelijkheids-API van Windows 7

Figuur 21 toont een voorbeeld van de informatie die is te verkrijgen via de GUI met de toegankelijkheids-API van het Windows 7-besturingssysteem voor een simpele rekenmachineapplicatie.

Met behulp van TESTAR kunt je dus meteen beginnen met random testen. Je hoeft vooraf geen testgevallen te specificeren, noch testscripts te schrijven of op te nemen. TESTAR genereert automatisch testreeksen op basis van de acties die mogelijk zijn in elke toestand waarin de GUI komt. Zoals aangegeven in Sectie 8.2 moeten we nu twee problemen oplossen:

- Dekkingscriteria - We weten niet hoeveel fouten er in onze software zitten, dus hoe weten we achteraf of we goed getest hebben?
- Orakels - Hoe weten we eigenlijk wat een fout is? Hoe kunnen we die fouten herkennen?

9.3 Hoe herkennen we fouten? Orakels in TESTAR.

Zonder iets te specificeren, kan TESTAR fouten detecteren in algemene systeemvereisten door middel van impliciete orakels.

Impliciete orakels zijn gebaseerd op algemene en impliciete kennis, die gebruikt wordt om te beslissen over juist of onjuist gedrag van een systeem. Deze impliciete kennis bevat situaties die bijna altijd fout zijn voor de meeste applicaties. Het gaat om orakels zoals:

- de SUT mag niet crashen;
- de SUT mag zich niet bevinden in een niet-responsieve toestand ('bevroren' zijn);
- de GUI-toestand mag geen *widget* bevatten met verdachte woorden zoals *error*, *problem*, *exception* enz.

Deze orakels vereisen geen domeinkennis, noch een formele specificatie. Ze zitten standaard in TESTAR en evalueren deze aspecten in elke toestand.

Andere orakels, die wel domeinkennis vereisen, zal de tester moeten toevoegen aan TESTAR.

Naast impliciete orakels, zijn er nog twee categorieën van mogelijke orakels: gespecificeerde orakels en afgeleide orakels.

Gespecificeerde orakels worden gemodelleerd aan de hand van een formele specificatie-taal. Er zijn door de jaren heen heel veel verschillende talen en formalismen gebruikt, die we zouden kunnen gebruiken om orakels te specificeren. Ik noem er hier een paar; voor een uitgebreide uiteenzetting verwijs ik naar [BHM+15, OKN15].

Formele specificatie-talen met namen als bijvoorbeeld Z [Spi89], B [Abr96] en VDM [Jon90] maken het mogelijk een programma te specificeren door middel van pre- en post-condities en invarianten. Pre-condities specificeren de eigenschappen waaraan de invoer van een applicatie moet voldoen. Post-condities definiëren het effect dat een programma (of een deel daarvan) heeft op de toestand van het programma. Een invariant is een eigenschap die blijft gelden tijdens de hele uitvoering van het software-programma.

Een probleem echter met dit soort specificaties is vaak dat ze te abstract zijn en te ver af liggen van de concrete uitvoer van een programma. Het is

dan moeilijk om een post-conditie te gebruiken als orakel om automatisch de juistheid of onjuistheid te berekenen van een concrete uitvoer van een programma [Aic99]. Voor invarianten geldt hetzelfde.

Een ander voorbeeld van een veel gebruikt formalisme voor het specificeren van software is een toestandsdiagram. Voorbeelden zijn formalismen als IOCO [Tre08], I/O Automata [LSV03] en Mealy/Moore-machines [LY96]. Het toestandsdiagram als formalisme beschrijft de software als een graaf. Het gedrag van de applicatie wordt beschreven door middel van de toestanden waarin de software kan zijn en de transities die er gemaakt kunnen worden van de ene toestand naar de andere. Het woord 'toestand' heeft hier echter een andere betekenis dan de toestand van de GUI waarover we het hebben bij TESTAR. Een GUI-toestand zoals in de vorige sectie werd uitgelegd is een *'snapshot'* (momentopname) van een concrete toestand waarin het systeem zich bevindt na het uitvoeren van een actie. Een toestand in een toestandsdiagram is een abstractie van verschillende *'snapshots'*. Ook hier geldt dat de concrete uitvoer van het systeem vertaald moet worden naar het abstractieniveau van de toestandsdiagrammen, willen we ze kunnen gebruiken als orakel in TESTAR.

Een algemeen overkoepelend probleem met het gebruiken van specificaties als orakels is dat ze vaak niet beschikbaar zijn. Informele specificaties zijn er vaak niet eens, laat staan formele. Daarbij komt ook dat als die er wel zijn, ze vaak verouderd zijn en niet meer actueel zijn met het systeem. Daarom is er veel onderzoek gedaan naar het afleiden van orakels.

Afgeleide orakels kunnen worden afgeleid uit informatie die kan worden verkregen uit aanwezige documentatie, bijvoorbeeld een gebruikershandleiding of eerdere versies van het systeem. In het eerder genoemde FITTEST-project hebben we orakels afgeleid door de executies van het systeem te loggen [EPH+15, NMC+13]. Aangezien orakels die afgeleid zijn uit een systeem beschrijven wat het systeem doet, kunnen we deze orakels alleen gebruiken om te testen of nieuwe versies van het systeem nog steeds voldoen aan de eigenschappen van het oude systeem (tijdens de zogenaamde regressietesten). Maar ook deze orakels zijn vaak niet compleet en het kan duur zijn om ze af te leiden. Andere methoden zijn *specification mining* en *learning*-technieken, maar ik heb voor dit moment simpelweg geen tijd om alles gedetailleerd te beschrijven. Voor een overzicht verwijs ik weer naar [BHM+15, OKN15].

Bovenstaande technieken zijn gericht op het aanwezig zijn van een 'artefact' (een reeds beschikbaar gegeven, zoals specificaties, handleidingen en zelfs de software zelf) als de basis voor het maken of afleiden van orakels. Maar als die artefacten er niet zijn, of niet voldoen, of weer voor andere onderhoudsproblemen zorgen, dan moeten we simpelweg de menselijke tester inschakelen als orakel om de (on)juistheid van de uitvoer van een SUT te bepalen wanneer we een bepaalde testreeks uitvoeren. Waar we dus naar moeten streven, is om de tester zo veel mogelijk te helpen dit op een efficiënte manier te kunnen doen [MSH10, SGH12].

Dat is de richting die we met TESTAR in de toekomst op willen gaan. Denk bijvoorbeeld aan:

- Het makkelijker maken om test-orakels te schrijven.
- Het makkelijker maken om testresultaten te evalueren.

Op dit moment moet het schrijven van een test-orakel in TESTAR gebeuren door middel van het programmeren van een stukje Java-code. Voorbeelden van impliciete orakels staan in Figuur 22.

Elk orakel krijgt als parameter de huidige toestand van de GUI van de applicatie (`State`). Deze toestand wordt gebruikt om een oordeel te geven over de (on)juistheid van die toestand (bijvoorbeeld voor de eerder genoemde impliciete orakels: er is een crash, de applicatie antwoordt niet of er is een verdacht woord gevonden in een titel van een *widget*).

In de toekomst willen we werken aan een formele specificatie-taal waarmee orakels geschreven kunnen worden die abstract genoeg zijn om eenvoudig te onderhouden, maar ook concreet genoeg zijn om automatisch de (on)juistheid te berekenen van een concrete uitvoer van een programma. Die specificatie-taal moet het de tester makkelijk maken om efficiënt orakels te schrijven en zelf te bepalen welke *trade-offs* hij of zij wil en kan maken tijdens het specificeren van de orakels, bijvoorbeeld complexiteit versus effectiviteit.

TESTAR levert de testreeksen op waarmee fouten zijn gevonden in de software samen met een toestandsgraaf met alle toestanden waarin de software is geweest tijdens de uitgevoerde testen. Deze testreeksen worden snel te groot om efficiënt handmatig te analyseren en de testen te kunnen evalueren. Door deze uitvoer slim te reduceren kunnen we de

```
protected Verdict oracle_Crash (State state){
   if(!state.get(IsRunning,false))
    return new Verdict("System offline! It crashed?");
}
protected Verdict oracle_Responsiveness (State state){
   if(state.get(NotResponding, true))
    return new Verdict("System is unresponsive!");
}
protected Verdicts oracle_SuspiciousTitles(State state){
    verdicts = new Verdicts():
    String titleRegEx = settings().get(SuspiciousTitles);

   // search all widgets for suspicious titles
   for(Widget w : state){
        String title = w.get(Title, "");
        if(title.matches(titleRegEx)){
        verdicts.add(new Verdict("......"));
   }
   return verdicts;
 }
```

Figuur 22: *Impliciete orakels in TESTAR*

tester misschien tijd besparen zodat hij of zij sneller de resultaten van de testen kan interpreteren. Ook willen we gaan werken aan de presentatie van die uitvoer. We kunnen bijvoorbeeld alleen de *big picture* laten zien zodat de tester kan inzoomen op bepaalde details, mocht hij of zij dat nodig hebben voor het vinden van meer onjuistheden in de software. Ook kunnen we bijvoorbeeld alle resultaten van de testen opslaan in een database zodat de tester ze kan raadplegen om bepaalde patronen te ontdekken of details te onderzoeken. Ook kunnen we profileringsinformatie verzamelen, denk aan gebruik van hulpbronnen (in het engels *resources*) zoals geheugen (RAM), de centrale processor eenheid (CPU), enz. die kunnen aangeven waar dingen misgaan en waarom.

9.4 Hoe weten we achteraf wat we getest hebben?

We hebben TESTAR tot nu toe bij vijf bedrijven in vier verschillende landen toegepast om te evalueren hoe goed of slecht TESTAR werkt als we echte software gaan testen in een echte industriële omgeving. En dat is nog maar het begin; steeds meer bedrijven beginnen interesse te krijgen in deze nieuwe aanpak van testen.

Clave Informatica[7], is een Spaans bedrijf gevestigd in Alicante. Zij hebben de afgelopen twintig jaar een ERP (Enterprise Resource Planning)-desktop-applicatie ontwikkeld die ze verkopen aan meer dan duizend klanten op de Spaanse markt. De applicatie is geschreven in de programmeertaal *Visual Basic* voor het Windows-besturingssysteem. Deze applicatie hebben we getest met TESTAR; de resultaten zijn gepubliceerd in [BdRV14]. De random actie-selectie en de impliciete orakels van TESTAR gaven goede resultaten bij dit bedrijf: we waren in staat om tien kritieke fouten te vinden in de software die nog niet bekend waren bij dit bedrijf zelf. Dat hadden de testers bij dit bedrijf niet verwacht. De mankracht die het heeft gekost om alles op te zetten en te analyseren zat rond 28 uur; de automatische testen met TESTAR hebben 91 uur achterelkaar gedraaid.

Berner & Mattner, dat nu bekent staat onder de name Assystems[8], is een Duits bedrijf, dat partner was in het FITTEST-project. Naast vele andere services, verkoopt dit bedrijf een testtool onder de naam TESTONA[9]. Deze testtool helpt de tester een testmodel te maken in de vorm van een combinatorische boom, zodat het daarna makkelijker wordt een testsuite te genereren. TESTONA hebben wij getest met TESTAR; de resultaten zijn gepubliceerd in [VKC+15]. Ook hier gaven random selectie van de acties en de impliciete orakels hele bevredigende resultaten voor het bedrijf, aangezien er binnen een paar uur getest kon worden en na een nacht TESTAR te hebben gedraaid er meteen een kritieke fout werd gevonden.

Softeam[10] is een Frans bedrijf dat een product heeft ontwikkeld dat *Modelio SAAS* heet. Dit product is geschreven in de programmeertaal PHP en heeft een webinterface waar een systeembeheerder verschillende accounts kan beheren voor gebruikers en software-projecten die worden gemaakt met behulp van hun andere open source-product *Modelio UML Modeling Tool*[11][12]. Wij hebben *Modelio SAAS* getest met TESTAR; de resultaten zijn gepubliceerd in [BVC+14]. Deze studie gaf ons de mogelijkheid om meer

aspecten van TESTAR te evalueren dan alleen het aantal gevonden fouten.

Ten eerste hadden we hier ook de mogelijkheid om de testresultaten van TESTAR te vergelijken met de handmatige testaanpak die het bedrijf hanteert om de software te testen.

Ten tweede was het mogelijk om fouten uit vorige versies in de software te her-injecteren in de huidige versie [VMEM12]. Op deze manier kennen we in elk geval een aantal fouten die in de software zitten en kunnen we de FoutDetectieRatio (FDR) berekenen:

$$\text{FDR} = \frac{\text{Aantal gevonden fouten}}{\text{Aantal geïnjecteerde fouten}} \times 100\%$$

De effectiviteit en efficiency van TESTAR kon in deze studie zeker concurreren met de handmatige aanpak. Gebruikmakende van random testen en de standaard impliciete orakels, was de FDR van de handmatige testsuite 83% en die van TESTAR 61%. Hoewel het testresultaat voor TESTAR iets minder is, is het op zich indrukwekkend, als je je realiseert dat TESTAR automatisch en random test, terwijl de handmatige aanpak gebaseerd is op een testsuite die gemaakt was met het doel om functionaliteiten te dekken met de testen.

Indenova[13] is een Spaans bedrijf gevestigd in Valencia. Ook zij verkopen een ERP-oplossing. Ze zijn begonnen op de Spaanse markt en langzaam uitgebreid naar de Latijns-Amerikaanse markt. Aangezien ze initieel in Spanje zijn begonnen, zijn veel woorden in de applicatie Castiliaans Spaans; de Zuid-Amerikaanse klanten klagen hierover.

Bijvoorbeeld:

Nederlands	Castiliaans Spaans	Latijns-Amerikaans Spaans
Mobiel	Móvil	Celular
Vakantie	Festivo	Feriado
Computer	Ordenador	Computadora

Helaas zitten deze woorden hard in hun programmeercode gebakken, dus het enige dat dit bedrijf kon doen was zoeken naar de ongewenste woorden en ze veranderen voor de desbetreffende markt. TESTAR biedt een goede uitkomst voor dit probleem. Als we aan het impliciete orakel, dat test op verdachte woorden, de Castiliaanse boosdoeners toevoegen, kunnen we de zoektocht automatisch voortzetten! De resultaten van deze studie zijn gepubliceerd in [MER+16].

De 5de studie wordt gedaan in de context van een Master of Science-thesis van twee studenten aan de OU: één werkt bij Cap Gemini en de ander bij Prorail. Ze passen TESTAR toe op een applicatie voor Prorail. Deze studie is nog in volle gang tijdens dit schrijven en hun resultaten zijn nog niet gereed, noch gepubliceerd. We kunnen wel alvast verklappen dat TESTAR met random actie-selectie en impliciete orakels tot meer functionele dekking leidt dan de bestaande handmatige testsuites. Bovendien zijn er een aantal belangrijke fouten gevonden die met de handmatige testen niet boven water kwamen. Ook al deinzen de testers eerst een beetje terug van TESTAR (omdat het niet past in de traditionele kijk die ze hebben op testen), al snel zien ze de voordelen van compleet automatische testen die ook daadwerkelijk fouten vinden!

De studies die hiervoor beschreven zijn, hebben TESTAR allemaal laten testen met random actie-selectie, gebruik makend van de impliciete orakels. De resultaten waren zeker hoopgevend! Hoe we in de toekomst TESTAR willen uitbreiden met technieken die de tester zo veel mogelijk helpen om op een efficiënte manier meer effectieve orakels te maken hebben we hiervoor uitgelegd. In de volgende sectie kijken we hoe we in de toekomst willen werken aan een slimmere manier om de acties te selecteren.

9.5 TESTAR slimmer maken

Een TESTAR die willekeurig acties selecteert werkt dus in de gevallen die we hebben gezien best goed! Maar, wat als we die random selectie kunnen verbeteren?

Wat als we, in plaats van willekeurig, de acties slimmer kiezen zodat we een bepaald doeleinde optimaliseren?

Dat is de vraag waar ons onderzoek zich de laatste tijd mee heeft bezig-gehouden - en in de toekomst mee gaat bezighouden. En sterker nog, in plaats van zelf te bedenken hoe we acties slim kunnen kiezen:

Wat als we TESTAR zelf laten leren wat de beste manier is om acties te selecteren?

Voor de optimalisatie kunnen we meta-heuristische zoekmethoden gebruiken, zoals bijvoorbeeld evolutionaire algoritmen, mierenkolonie-optimalisatie, enz. [BDGG09]. Om TESTAR te laten leren, kunnen we technieken gebruiken als Machinaal Leren (ML) (*Machine Learning* in het Engels) [Mit97] .

Eerst moeten we dan nadenken over wat we kunnen gaan leren of optimaliseren om meer fouten te vinden. Aangezien we het aantal fouten nooit van te voren weten, hebben we bij het testen altijd surrogaat-metrieken nodig die, als we ze optimaliseren, hopelijk leiden naar meer fouten [Mun03]. Surrogaat-metrieken hebben bijvoorbeeld te maken met: meer diversiteit, meer dekking en meer nieuwigheid. Voorbeelden zijn dat we de acties zo kiezen dat we proberen zoveel mogelijk:

- *verschillende acties* uit te voeren tijdens de testen;
- in *verschillende toestanden* van de SUT te komen (ook al weten we niet van te voren hoeveel er zijn);
- in *nieuwe toestanden* te komen waar we nooit eerder zijn geweest;
- *verschillende events* te laten aanroepen;
- programmeercodedekking te verkrijgen (als we toegang hebben tot de source code kunnen we kijken in welke programmaregeldekking, beslissingsdekking, programmapaddekking, enz. het testen met TESTAR resulteert).

Er is nog veel onderzoek nodig naar meer surrogaat-metrieken die ons iets kunnen vertellen over de kwaliteit van de testen en de kans op het vinden van fouten. In het werk van McMaster en Memon [MM08] wordt een surrogaat-metriek beschreven die *Maximale Call Stack* (MCS) wordt genoemd. Een *call stack* is een datastructuur (een stapel of *stack* dus) die in het geheugen van een computer wordt bijgehouden tijdens de uitvoering van een programma. Terwijl het programma wordt uitgevoerd, worden allerlei gegevens die nodig zijn tijdens de uitvoering van het programma op elkaar gestapeld (denk bijvoorbeeld aan de inhoud van registers die tijdelijk hergebruikt worden). In hun werk [MM08] tonen ze empirisch aan dat grotere *call stacks* kunnen leiden tot het vinden van meer fouten. In [BWW11] hebben we geprobeerd dat toe te passen in TESTAR. We hebben

mierenkolonie-optimalisatie [CDM92, DB05] toegepast om de testreeksen zo te genereren dat de MCS wordt geoptimaliseerd. Het mierenkolonie-optimalisatiealgoritme is gericht op het zoeken naar een optimale weg gebaseerd op het gedrag van mieren die een pad zoeken tussen hun kolonie en een bron van voedsel. Mieren beginnen normaliter willekeurig te wandelen op zoek naar voedsel. Bij het vinden van voedsel - en als ze terugkeren naar hun kolonie - leggen ze feromonen neer om te communiceren aan de andere mieren dat dit een goede weg is op zoek naar eten. Als andere mieren een pad vinden met feromonen, dan zullen ze waarschijnlijk niet willekeurig reizen, maar in plaats daarvan het feromonentraject volgen. Als ze uiteindelijk voedsel vinden en dan terugkeren, leggen ze meer feromonen op het pad. De feromonen accumuleren en andere mieren weten zo dat dit het beste pad is om te kiezen.

Als we dit vertalen naar onze testuitdaging, dan zijn:

- de keuzes die de mieren moeten maken op hun pad naar voedsel: de acties die TESTAR moet kiezen op weg naar het zoeken van fouten,
- de paden die de mieren afleggen: de testreeksen die TESTAR uitvoert.

Alle acties krijgen een initiële feromoon-waarde toegekend, en TESTAR wordt gestart. Acties worden geselecteerd op basis van hun feromoon-waarde. Na elke run van TESTAR worden de feromoon-waarden geactualiseerd: acties die in testreeksen voorkomen die een hoge MCS bereiken krijgen een hoge feromoon-waarde, zodat goede reeksen feromonen kunnen accumuleren en dus vaker gekozen worden. De resultaten in [BWW11] laten zien dat mierenkolonie-optimalisatie MCS waarden bereikt van 144.082 terwijl random niet verder komt dan 91.587. In de toekomst moeten we uitzoeken of we door deze metriek te optimaliseren met TESTAR ook daadwerkelijk meer fouten kunnen vinden.

Een andere surrogaat-metriek die we hebben geoptimaliseerd, is het aantal verschillende soorten acties die geselecteerd worden. Dit hebben we gedaan door TESTAR te laten leren welke acties beter zijn om te kiezen als het doel is steeds meer verschillende acties te kiezen [BV12, Vos14, EAAM+16, AEMR16]. Hiervoor hebben we een simpel *Machine Learning* algoritme gebruikt dat bekend staat onder de naam *reinforcement learning* of *Q-learning* [WD92]. Het idee is als volgt:

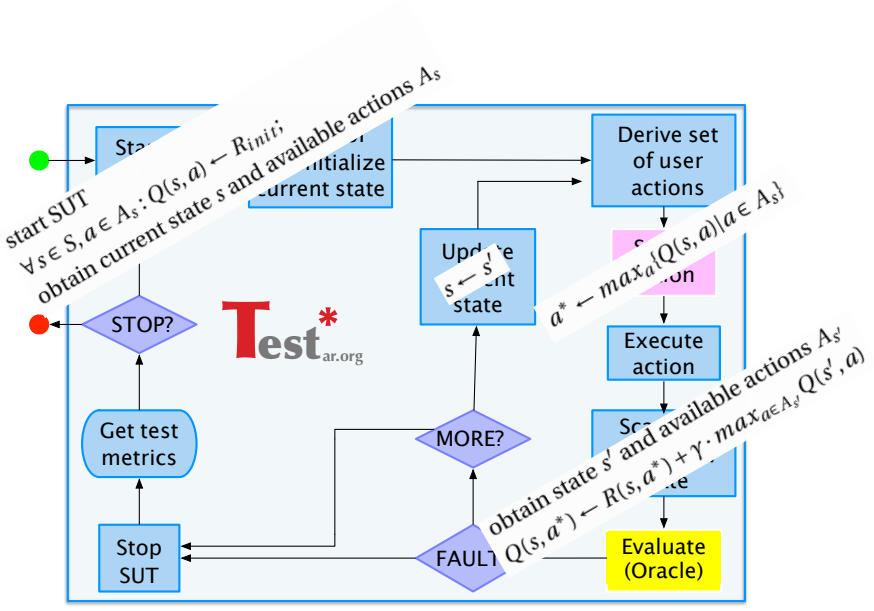
Input: R_{init} /* initiële beloning */
Input: $0 < \gamma < 1$ /* discount factor */
begin
  start SUT
  $\forall s \in S, a \in A_s : Q(s,a) \leftarrow R_{init}$;
  obtain current state s and available actions A_s
  repeat
    $a^* \leftarrow max_a\{Q(s,a) | a \in A_s\}$ /* kies de beste */
    execute a^*
    $ec(a^*)++$ /* teller */
    obtain state s' and available actions $A_{s'}$
    $Q(s,a^*) \leftarrow R(s,a^*) + \gamma \cdot max_{a \in A_{s'}} Q(s',a)$ /* leer */
    $s \leftarrow s'$
  until *stopping criteria met*
  stop SUT
end

Figuur 23: *Q-Learning*

- We hebben een verzameling S van mogelijke toestanden waar in een SUT zich kan bevinden.
- Voor elke toestand $s \in S$ hebben we een verzameling acties A_s die we in deze toestand kunnen uitvoeren.
- Als we willen proberen zoveel mogelijk verschillende acties te kiezen, moeten we dus acties die al een keer uitgevoerd zijn zoveel mogelijk vermijden. Om dat te weten moeten we dus per actie een teller bijhouden die aangeeft hoe vaak die actie al is gekozen en uitgevoerd.

  Laten we die teller bijvoorbeeld voor alle acties $a \in A_s$ weergeven met $ec(a)$.
- We kunnen dan als volgt een *reward* (of belonings)-functie definiëren voor de acties:

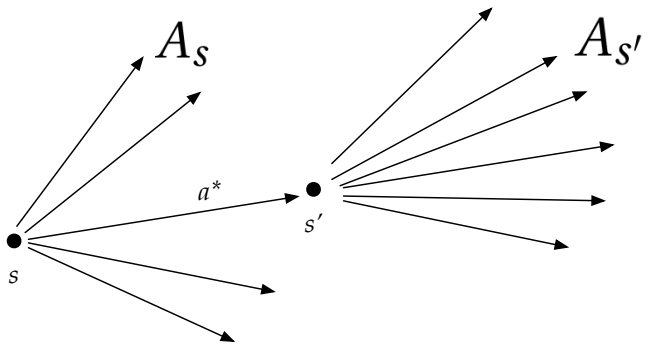
$$\forall s \in S, a \in A_s : R(s,a,) = \begin{cases} R_{init} & \text{, if } ec(a) = 0 \\ \frac{1}{ec(a)} & \text{, else} \end{cases}$$



Figuur 24: *TESTAR en Q-learning*

Acties die nog nooit zijn uitgevoerd hebben een initiële beloning (R_{init}). Naarmate de $ec(a)$ omhoog gaat omdat de actie a wordt uitgevoerd, gaat de beloning om die actie te kiezen naar beneden.

- Het *Q-Learning* algoritme werkt zoals te zien is in Figuur 23. In Figuur 24 is te zien hoe dat precies past in de TESTAR-aanpak. In plaats van willekeurig te kiezen, wordt die actie gekozen met de maximale Q-waarde die stap voor stap wordt geleerd in de instructie in regel 10. De Q-waarde van actie a^* in toestand s is gelijk aan de beloning R van die actie a^* in toestand s plus de maximale Q-waarde van acties die in toestand s' gekozen kunnen worden. Op die manier kunnen we als we een actie kiezen in toestand s (zie Figuur 25) één stap vooruitkijken om acties met een hoge beloningswaarde in s' niet te missen. Als de waarde van parameter γ dicht bij 0 is zijn we gretig en willen we alleen snel in toestand s al de hoogste beloning hebben. Naarmate γ naar de waarde 1 gaat, nemen we ook meer eventuele beloningen in acht die in de toestand s' erna kunnen komen.

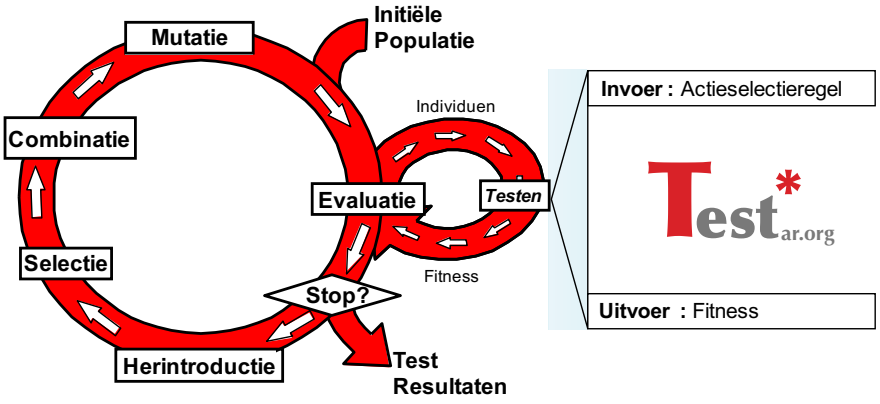


Figuur 25: *Toestandsovergangen*

In [EAAM+16, AEMR16] hebben we deze Q-lerende versie van TESTAR losgelaten op een webapplicatie ODOO[14] en Powerpoint[15]. Om de kwaliteit van de testen te meten hebben we gekeken naar metrieken als:

1. het aantal fouten dat gevonden is met de impliciete orakels;
2. het aantal verschillende GUI-toestanden dat is doorlopen tijdens de testen;
3. lengte van de langste testreeks die bestaat uit unieke toestanden zonder herhalingen (zo kunnen we zien of TESTAR ook daadwerkelijk wat dieper de applicatie is ingegaan, in plaats van alleen maar wat aan de oppervlakte te klikken);
4. maximale en minimale actiedekking per toestand, dat wil zeggen, in een bepaalde toestand s kunnen we misschien A_s verschillende acties kiezen. De actiedekking in die toestand is gedefinieerd als het percentage van acties uit A_s die ook daadwerkelijk een keer gekozen zijn tijdens de testen.

Op al deze metrieken scoorde Q-learning beter dan puur random kiezen, mits de parameters (R_{init} en γ) goed gekozen waren. Ook hier moet toekomstig werk opleveren of we door deze metrieken te optimaliseren met TESTAR ook daadwerkelijk meer fouten kunnen vinden. En, zoals al gezegd, is er ook meer onderzoek nodig naar meer metrieken.



Figuur 26: *Evolueren van actieselectieregels*

We hebben dus twee verschillende manieren geïmplementeerd om acties, in plaats van willekeurig, slimmer te selecteren. We zouden nog heel veel andere selectiemechanismen kunnen bedenken en ze allemaal proberen met verschillende meta-heuristieken en *Machine Learning*-algoritmen. Maar ons doel voor de toekomst is:

TESTAR zelf te laten leren wat de beste manier is om acties te selecteren!

Drones, auto's en robots worden niet geprogrammeerd met regels die aangeven hoe ze moeten werken. Ze worden geprogrammeerd zodat ze kunnen leren hoe ze moeten werken. Dat is wat we hierboven hebben aanggeven als Machinaal Leren, een breed onderzoeksveld binnen de Kunstmatige Intelligentie, dat zich bezighoudt met de ontwikkeling van algoritmes en technieken waarmee computers kunnen leren [Mit97].

Met onderzoek om TESTAR te laten leren zijn we het afgelopen jaar begonnen en het is ons plan om daarmee door te gaan. De eerste resultaten hebben we al gepubliceerd in [EARV17]. Daar laten we zien hoe we evolutionaire algoritmen hebben toegepast om actieselectieregels te evolueren. Deze keer zijn de individuen dus niet de testgevallen maar de actieselectieregels, zoals in Figuur 26. En de fitness-functie het aantal unieke toestanden waarin de GUI kan komen te optimaliseren. De resultaten zijn veelbelovend en laten zien dat de geëvolueerde selectie-regels daadwerkelijk meer fouten vinden dan random.

10 Onderwijs

Ik wil het tot slot nog even hebben over mijn plannen in het onderwijs. Zoals eerder aangegeven denk ik dat we een cultuur van kwaliteitsdenken moeten creëren als het gaat om software-ontwikkeling. En die moet beginnen in het onderwijs. We weten namelijk welke aspecten van belang zijn om software van kwaliteit te ontwikkelingen, alleen het lukt ons nog niet om studenten zo te vormen dat ze die ook daadwerkelijk gaan gebruiken.

Software-engineering-vakken in een informatica-opleiding zijn meestal geneigd zich meer te concentreren op de creatieve aspecten van programmeren en codering. Dit is natuurlijk niet verwonderlijk. Veel technieken om de kwaliteit van de software te verbeteren zijn moeizaam en geven het gevoel van het doen van "dubbel werk". Denk bijvoorbeeld aan het schrijven van een unit-test. Je hebt dan bijvoorbeeld al een methode geschreven om een bepaald probleem op te lossen, moet je dan toch nog een methode schrijven die dan ook test dat het wel werkt zoals je verwacht? Dat kost dan toch twee keer zoveel tijd, of niet? En vaak moet je dan je methode aanroepen met lange reeksen van invoeren die een klein beetje verschillen om randvoorwaarden te testen. Denk bijvoorbeeld aan de 16 testgevallen op pagina 20 die we hadden ontworpen om alle combinaties van partities te dekken voor het stukje software dat verantwoordelijk is voor het sturen van de stemkaart. Om die allemaal in een stukje code te zetten, dat is helemaal niet zo enerverend. Je herhaalt gewoon 16 keer bijna dezelfde aanroep en dat is het dan. Je kunt het ook andersom doen: als je de *test-eerst*-methode toepast: eerst een test schrijven die de methode die je nog niet hebt geprogrammeerd kan testen zodra je hem wel hebt geschreven. Veel studenten hebben veel meer zin om meteen nieuwe code te schrijven die iets doet, in plaats van een code die iets test. Ook als we denken aan het becommentariëren van code: werkt je stukje code net, moet je nog eens op gaan schrijven hoe en waarom. Vaak wordt dat uitgesteld en gebeurt het nooit meer.

En natuurlijk in het klaslokaal, waar de levensduur van de code die we schrijven vaak niet langer is dan de tijd die nodig is om het vak te halen, voelen we vaak niet echt de pijn van niet of slecht testen en becommentariëren. Ook haal je soms met spaghetticode nog wel een magere voldoende.

Om het toenemende probleem van slechte softwarekwaliteit aan te pak-

ken, hebben we daarom een fundamentele verandering nodig in de manier waarop software-engineering wordt geleerd. Toekomstige softwarebouwers moeten worden opgeleid met een waardering voor softwarekwaliteit en kwaliteitsborgingstechnieken moeten een natuurlijk aspect van softwareontwikkeling worden, in plaats van een niche-onderwerp. Studenten moeten bewust gemaakt worden van de schoonheid van compacte oplossingen, korte methoden, veelzeggende namen en modulair opgebouwde systemen. Dit vereist een fundamentele verandering in de houding van de docenten en de studenten en van hun perceptie van dit onderwerp en de manier waarop het wordt onderwezen.

Programmeeronderwijs moet onlosmakelijk verbonden worden met kwaliteit. Vanaf het begin weten dat je je programma's moet becommentariëren en testen. Een student die niet aantoonbaar zijn of haar programma uitvoerig heeft getest, kan en mag geen voldoende beoordeling krijgen. Programmeren moet netjes gebeuren; studenten moeten zo onderwezen worden dat ze hun spaghetticode niet eens durven in te leveren.

Om dit voor elkaar te krijgen zal ons lesmateriaal veranderd moeten worden. Standaard tekstboeken die bijvoorbeeld de programmeertaal Java leren aan studenten, behandelen kwaliteit veel te laat. Eerst worden studenten de syntactische constructen van de taal geleerd, dan hoe ze methoden kunnen schrijven die een bepaald probleem oplossen, hoe ze die methoden kunnen groeperen in klassen en vervolgens hoe hun systeem bestaat uit al die klassen. Meestal wordt er dan snel nog iets gezegd over hoe je netjes moet programmeren en dan..., dan laten we de studenten los op hun eerste programmeeropdracht. Veel later pas wordt er dan verteld dat ze ook hadden moeten testen. Maar vaak is het dan al te laat. De studenten hebben geproefd van de lekker snel gemaakte spaghetti. Veelzeggende namen? Waarom? Ik begrijp wat er staan! En het werkt toch? Kortere methodes? Ik vind dit best kort en volgens mij is het niet op te splitsen! En ... het werkt toch? Testen? Waarom? Dat kost toch veel extra tijd? En ..... het werkt toch? Modulair? Het is toch opgedeeld in klassen? En ..... het werkt toch?

We moeten dus het lesmateriaal veranderen en kwaliteit en testen veel eerder introduceren. Software-engineering-technieken moeten niet in een gelijknamig vak allemaal tegelijk de revue passeren. Deze technieken moeten als een rode draad door het hele curriculum doorlopen, te beginnen bij

de eerste lessen in het programmeren.

Maar we moeten ook de studenten de pijn laten voelen van slechte en niet geteste code. We moeten de studenten bewust maken van de voordelen en de schoonheid van goede code. En dat alles moeten we proberen zo te doen dat het voor de studenten leuk en motiverend is. Naast het veranderen van lesmateriaal moeten we ook de manier van lesgeven veranderen. Het introduceren van gamificatie in het onderwijs is een van de oplossingen waaraan ik wil werken. Gamificatie is het gebruiken van spelprincipes en speeltechnieken in een niet-spelcontext (zoals onderwijs of wetenschappelijk onderzoek) om menselijk gedrag op een positieve wijze te sturen [DDKN11].

Spellen en gamificatie zijn heden ten dage populaire benaderingen in het onderwijs. En er zijn bewijzen uit verschillende domeinen dat spelomgevingen de motivatie en de betrokkenheid van de student vergroten. Hierdoor wordt effectiever leren bevorderd [US14]. Het gebruik van gamificatie-concepten in het software-engineering-onderwijs is tot op heden is echter zeer beperkt. Een reden hiervoor is dat de vertaling naar spelconcepten alleen voor creatieve activiteiten zoals programmeren eenvoudig is [RF16, Lak13, TdHXB13], maar niet voor analytische kwaliteitsborgende technieken zoals bijvoorbeeld testen. Dit zal een van de belangrijke punten zijn om mee te beginnen. Stel we laten studenten zoeken naar fouten in de programmeercode van een medestudent. Wie het snelste tien fouten vindt heeft gewonnen en kan daarmee munten verdienen. Wie het eerste boven een bepaalde score komt mag dan door naar het volgende level: het toevoegen van fouten aan de software die niet gevonden worden door de bijgeleverde testen. Elke niet gevonden fout levert munten op. Elke test die toegevoegd wordt en die de fout wel vindt levert nog meer munten op. Munten kunnen bijvoorbeeld ook verdiend worden met het zoeken naar de verschillen tussen een mooie modulaire oplossing en spaghetticode oplossing. Deze munten kun je dan uiteindelijk gebruiken als je een nieuwe functionaliteit moet toevoegen aan een bestaand systeem. Met verkregen munten kun je betere kwaliteit-code verkrijgen met meer en uitgebreider commentaar, waardoor het uitbreiden van het systeem makkelijker wordt en je veel eerder klaar bent.

Het zijn maar een paar ideeën. In de komende jaren gaan we ze vormgeven en inpassen in het online- en activerende onderwijs aan de OU.

Tot slot

Software van slechte kwaliteit. Tijdens het schrijven van deze oratie kwamen er verschillende nieuwsartikelen voorbij waarin datgene wat ik net had opgeschreven weer bevestigd werd. Het meest recente, de problemen bij British Airways – een *major IT system failure* – zorgde voor chaos alom. Jeff Offutt, collega, docent aan de George Mason University in Virginia USA, en schrijver van een aantal boeken over testen, kon het niet beter verwoord hebben:

We don't need terrorists to disrupt our infrastructure when we are too cheap to TEST the software.

Er is nog een heleboel te onderzoeken en te evalueren. De problemen bij British Airways zullen niet de laatste zijn. Maar bij de Open Universiteit zijn we op weg, op weg naar een nieuwe manier om software te testen.

Ik heb gezegd.

Notes

[1] Deze terminologie werd in testen geïntroduceerd in 1978 door het werk van Williams Howden [How78].
[2] www.testar.org
[3] met contractnummer FP6-IST-33472
[4] met contractnummer FP7-ICT-257574
[5] Eerder bekend onder de naam JAutomate.
[6] Met dank aan Michell Nass die dit op mijn verzoek voor me heeft gemaakt met de eyeAutomate tool [Auq17]
[7] https://www.clavei.es/
[8] http://www.assystem.com/
[9] http://www.testona.net/
[10] http://www.softeam.com/
[11] https://www.modelio.org/
[12] https://www.youtube.com/watch?v=wVNJPuRSJVs
[13] https://www.indenova.com/
[14] https://github.com/odoo/odoo
[15] https://en.wikipedia.org/wiki/Microsoft_PowerPoint

Referenties

[AB11] Andrea Arcuri and Lionel Briand. Adaptive random testing: An illusion of effectiveness? In *International Symposium on Software Testing and Analysis*, ISSTA, pages 265–275, NY, USA, 2011. ACM.

[Abr96] J.-R. Abrial. *The B-book: Assigning Programs to Meanings*. Cambridge University Press, New York, NY, USA, 1996.

[AEMR16] Francisco Almenar, Anna Isabel Esparcia-Alcázar, Mirella Martínez, and Urko Rueda. Automated testing of web applications with TESTAR - lessons learned testing the odoo tool. In *Search Based Software Engineering - 8th International Symposium, SSBSE 2016, Raleigh, NC, USA, October 8-10, 2016, Proceedings*, pages 218–223, 2016.

[AF17] Emil Alégroth and Robert Feldt. On the long-term use of visual gui testing in industrial practice: a case study. *Empirical Software Engineering*, pages 1–35, 2017.

[AFR13] Emil Alegroth, Robert Feldt, and Lisa Ryrholm. Visual gui testing in practice: Challenges, problems and limitations. *Empirical Software Engineering Journal*, 2013.

[AIB12] A. Arcuri, M. Z. Iqbal, and L. Briand. Random testing: Theoretical results and practical implications. *IEEE TSE*, 38(2):258–277, 2012.

[Aic99] Bernhard K. Aichernig. *Automated Black-Box Testing with Abstract VDM Oracle*, pages 250–259. Springer Berlin Heidelberg, Berlin, Heidelberg, 1999.

[ANO13] E. Alegroth, M. Nass, and H.H. Olsson. Jautomate: A tool for system- and acceptance-test automation. In *Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on*, pages 439–446, March 2013.

[AO17] Paul Ammann and Jeff Offutt. *Introduction to software testing*. Cambridge University Press, 2017.

[ASM16] E. Alégroth, M. Steiner, and A. Martini. Exploring the presence of technical debt in industrial gui-based testware: A case study.

In *2016 IEEE Ninth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 257–262, April 2016.

[Auq17] Auqtus. Eyeautomate. `http://eyeautomate.com/eyeautomate.html`, 2017. [Online; accessed 9-5-2017].

[BBC15] BBC future. The number glitch that can lead to catastrophe. `http://www.bbc.com/future/story/20150505-the-numbers-that-lead-to-disaster`, 2015. [Online; accessed 1-6-2017].

[BDGG09] Leonora Bianchi, Marco Dorigo, Luca Maria Gambardella, and Walter J. Gutjahr. A survey on metaheuristics for stochastic combinatorial optimization. *Natural Computing*, 8(2):239–287, 2009.

[BdRV14] S. Bauersfeld, A de Rojas, and T.E.J. Vos. Evaluating rogue user testing in industry: An experience report. In *Research Challenges in Information Science (RCIS), 2014 IEEE Eighth International Conference on*, pages 1–10, May 2014.

[Bei90] Boris Beizer. *Software testing techniques*. Van Nostrand Reinhold, 2nd edition, 1990.

[Bei95] Boris Beizer. *Black-box testing - techniques for functional testing of software and systems*. Wiley, 1995.

[BHM+15] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo. The oracle problem in software testing: A survey. *IEEE Transactions on Software Engineering*, 41(5):507–525, May 2015.

[Bin00] R.V. Binder. *Testing Object-oriented Systems: Models, Patterns, and Tools*. Addison-Wesley object technology series. Addison-Wesley, 2000.

[BLVW11] Arthur I. Baars, Kiran Lakhotia, Tanja E. J. Vos, and Joachim Wegener. Search-based testing, the underlying engine of future internet testing. In *Proceedings of Federated Conference On Computer Science and Information Systems (FedCSIS '11)*, pages 917–923, Szczecin, Poland, 18-21 September 2011. IEEE.

[BP16] M. Böhme and S. Paul. A probabilistic analysis of the efficiency of automated software testing. *IEEE TSE*, 42(4):345–360, 2016.

[BV12] Sebastian Bauersfeld and Tanja EJ Vos. A reinforcement learning approach to automated gui robustness testing. In *4th Symposium on Search Based-Software Engineering (SSBSE2012), Fast Abstracts, 28-30 September, Riva del Garda, Trento, Italy*, pages 7–12, 2012.

[BVC+14] Sebastian Bauersfeld, Tanja E. J. Vos, Nelly Condori-Fernández, Alessandra Bagnato, and Etienne Brosse. Evaluating the TESTAR tool in an industrial case study. In *2014 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM '14, Torino, Italy, September 18-19, 2014*, page 4, 2014.

[BVD10] Arthur I. Baars, Tanja E. J. Vos, and Dimitar M. Dimitrov. Using evolutionary testing to find test scenarios for hard to reproduce faults. In *Third International Conference on Software Testing, Verification and Validation, ICST 2010, Paris, France, April 7-9, 2010, Workshops Proceedings*, pages 173–181. IEEE Computer Society, 2010.

[BW04] O. Bühler and J. Wegener. Automatic testing of an autonomous parking system using evolutionary computation. In *Proc of SAE 2004 World Congress*, pages pages 115–122, March 2004.

[BWW11] Sebastian Bauersfeld, Stefan Wappler, and Joachim Wegener. *A Metaheuristic Approach to Test Sequence Generation for Applications with a GUI*, pages 173–187. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[CDM92] Alberto Colorni, Marco Dorigo, and Vittorio Maniezzo. Distributed optimization by ant colonies. In *Proceedings of the First European Conference on Artificial Life*, pages 134–142. Elsevier, 1992.

[Cop04] L. Copeland. *A Practitioner's Guide to Software Test Design*. Software Testing. Artech House, 2004.

[CY94] T. Y. Chen and Y. T. Yu. On the relationship between partition and random testing. *IEEE Trans. Softw. Eng.*, 20(12):977–980, December 1994.

[Dai16] Daily Mail UK. Up to 300,000 heart patients may have been given wrong drugs or advice due to major NHS IT blunder. `http://www.dailymail.co.uk/health/article-3585149/Up-300-000-heart-patients-given-wrong-drugs-advice-major-NHS-blunder.html`, 2016. [Online; accessed 4-2-2017].

[DB05] Marco Dorigo and Christian Blum. Ant colony optimization theory: A survey. *Theoretical Computer Science*, 344(2):243 – 278, 2005.

[DDKN11] Sebastian Deterding, Dan Dixon, Rilla Khaled, and Lennart Nacke. From game design elements to gamefulness: Defining "gamification". In *Proceedings of the 15th International Academic MindTrek Conference: Envisioning Future Media Environments*, MindTrek '11, pages 9–15, New York, NY, USA, 2011. ACM.

[De 17] De Belastingsdienst. Voorlopige aanslag 2017 mogelijk verkeerd berekend. `https://www.belastingdienst.nl/wps/wcm/connect/bldcontentnl/berichten/verstoringen/voorlopige-aanslag+2017-mogelijk-verkeerd-berekend`, 2017. [Online; accessed 4-2-2017].

[DN84] J. W. Duran and S. C. Ntafos. An evaluation of random testing. *IEEE TSE*, SE-10(4):438–444, July 1984.

[EAAM+16] Anna I. Esparcia-Alcázar, Francisco Almenar, Mirella Martínez, Urko Rueda, and Tanja E.J. Vos. Q-learning strategies for action selection in the TESTAR automated testing tool. In *Proceedings of META 2016 6th International Conference on Metaheuristics and Nature Inspired computing*, pages 174–180, 2016.

[EARV17] Anna Isabel Esparcia-Alcázar, Francisco Almenar, Urko Rueda, and Tanja E. J. Vos. Evolving rules for action selection in automated testing via genetic programming - A first approach.

In *Applications of Evolutionary Computation - 20th European Conference, EvoApplications 2017, Amsterdam, The Netherlands, April 19-21, 2017, Proceedings, Part II*, pages 82–95, 2017.

[EPH+15] A. Elyasov, W. Prasetya, J. Hage, U. Rueda, T. Vos, and O.N. Condori-Fernandez. *AB=?A: execution equivalence as a new type of testing oracle.*, pages 1559–1566. ACM, 2015.

[Eva15] Evans Data. Driekwart van de apps wordt met bugs gelanceerd. `http://appworks.nl/2015/10/13/driekwart-van-de-apps-wordt-met-bugs-gelanceerd/`, 2015. [Online; accessed 19-4-2017].

[Gel16] De Gelderlander. Foutje: 104-jarige Zweedse mag naar kleuterschool. `http://www.gelderlander.nl/bizar/foutje-104-jarige-zweedse-mag-naar-kleuterschool~ab175865/`, 2016. [Online; accessed 4-2-2017].

[Ger08] Gerald M. Weinberg . *Perfect Software and other illusions about testing.* Dorset House, 2008.

[Ger15] Gerald M. Weinberg . *Errors Bugs, Boo-boos, Blunders.* Leanpub, 2015.

[GKWV09] Hamilton Gross, Peter M. Kruse, Joachim Wegener, and Tanja E. J. Vos. Evolutionary white-box software test with the evotest framework: A progress report. In *Second International Conference on Software Testing Verification and Validation, ICST 2009, Denver, Colorado, USA, April 1-4, 2009, Workshops Proceedings*, pages 111–120. IEEE Computer Society, 2009.

[Gol89] D. E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison Wesley, 1989.

[GR73] E. Girard and J. C Rault. A programming technique for software reliability. 1973.

[Gut99] Walter J. Gutjahr. Partition testing vs. random testing: The influence of uncertainty. *IEEE Trans. Softw. Eng.*, 25(5):661–674, September 1999.

[HHH+02] M. Harman, L. Hu, R. Hierons, A. Baresel, and H. Sthamer. Improving evolutionary testing by flag removal. In *GECCO*, pages 1233 – 1240, NY, 2002. Morgan Kaufmann.

[HJZ15] M. Harman, Y. Jia, and Y. Zhang. Achievements, open problems and challenges for search based software testing. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, pages 1–12, April 2015.

[HLN17] HLN. Oudste mens ter wereld overleden op 146-jarige leeftijd. urlhttp://www.hln.be/hln/nl/959/Bizar/article/detail/3146696/2017/05/01/Oudste-mens-ter-wereld-overleden-op-146-jarige-leeftijd.dhtml, 2017. [Online; accessed 1-6-2017].

[Hol75] J. H. Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor, 1975.

[How78] William E. Howden. Theoretical and empirical studies of program testing. In *Proceedings of the 3rd International Conference on Software Engineering*, ICSE '78, pages 305–311, Piscataway, NJ, USA, 1978. IEEE Press.

[HT88] D. Hamlet and R. Taylor. Partition testing does not inspire confidence. In *[1988] Proceedings. Second Workshop on Software Testing, Verification, and Analysis*, pages 206–215, Jul 1988.

[IEC98] IEC 61508-3, functional safety of electrical / electronic / programmable electronic safety-related systems, 1998.

[ISO08] Iso/cd 26262: Road vehicles-functional safety. committee draft, work in progress, Sep 2008.

[Jon90] Cliff B. Jones. *Systematic Software Development Using VDM (2Nd Ed.)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1990.

[Jor14] Paul C. Jorgensen. *Software Testing: A Craftsman's Approach*. CRC Press, Inc., Boca Raton, FL, USA, 4th edition, 2014.

[Kan98] Cem Kaner. Avoiding shelfware: A manager's view of automated gui testing. In *STAR'98 Conference, Orlando, FL.*, 1998.

[KvdABV06] Tim Koomen, Leo van der Aalst, Bart Broekman, and Michiel Vroon. *Tmap Next - voor resultaatgericht testen*. Tutein Nolthenius, Den Bosch, The Netherlands, 1st edition, 2006.

[Lak13] Kiran Lakhotia. En Garde: winning coding duels through genetic programming. In Tanja Vos and Simon Poulding, editors, *Sixth IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW 2013)*, pages 421–424, Luxembourg, 22 March 2013. IEEE. Refereed Fast Abstract.

[LCRT13] Maurizio Leotta, Diego Clerissi, Filippo Ricca, and Paolo Tonella. Capture-replay vs. programmable web testing: An empirical assessment during test case evolution. In *WCRE*, pages 272–281, 2013.

[LSV03] Nancy Lynch, Roberto Segala, and Frits Vaandrager. Hybrid i/o automata. *Inf. Comput.*, 185(1):105–157, August 2003.

[LY96] D. Lee and M. Yannakakis. Principles and methods of testing finite state machines-a survey. *Proceedings of the IEEE*, 84(8):1090–1123, Aug 1996.

[McM04] P. McMinn. Search-based software test data generation: A survey. *Software Testing, Verification and Reliability*, 14(2):105–156, 2004.

[McM11] Phil McMinn. Search-based software testing: Past, present and future. In *Proceedings of the 4th International Workshop on Search-Based Software Testing (SBST '11)*, pages 153–163, Berlin, Germany, 21-21 March 2011. IEEE.

[Mem02] Atif M Memon. Gui testing: Pitfalls and process. *IEEE Computer*, 35(8):87–88, 2002.

[MER+16] Mireilla Martinez, Anna Esparcia-Alcázar, Urko Rueda, Tanja E. J. Vos, and Carlos Ortega. Automated localisation testing in industry with test ^* _*. In *Testing Software and Systems - 28th IFIP WG 6.1 International Conference, ICTSS 2016, Graz, Austria, October 17-19, 2016, Proceedings*, pages 241–248, 2016.

[Mit97] Thomas M. Mitchell. *Machine Learning*. McGraw-Hill, Inc., New York, NY, USA, 1 edition, 1997.

[MM08] S. McMaster and A. Memon. Call-stack coverage for gui test suite reduction. *IEEE Transactions on Software Engineering*, 34(1):99–115, Jan 2008.

[MS03] Atif M. Memon and Mary Lou Soffa. Regression testing of guis. *SIGSOFT Softw. Eng. Notes*, 28(5):118–127, September 2003.

[MSB11] Glenford J. Myers, Corey Sandler, and Tom Badgett. *The Art of Software Testing*. John Wiley & Sons, Inc., New York, NY, USA, 3rd edition edition, 2011.

[MSH10] Phil McMinn, Mark Stevenson, and Mark Harman. Reducing qualitative human oracle costs associated with automatically generated test data. In *Proceedings of the First International Workshop on Software Test Output Validation*, STOV '10, pages 1–4, New York, NY, USA, 2010. ACM.

[Mun03] J.C. Munson. *Software Engineering Measurement*. CRC Press, 2003.

[Mye79] Glenford J. Myers. *The Art of Software Testing*. John Wiley & Sons, Inc., New York, NY, USA, 1st edition, 1979.

[NB13] Stanislava Nedyalkova and Jorge Bernardino. Open source capture and replay tools comparison. In *Proceedings of the International C* Conference on Computer Science and Software Engineering*, C3S2E '13, pages 117–119, New York, NY, USA, 2013. ACM.

[NMC+13] Cu D. Nguyen, Bilha Mendelson, Daniel Citron, Onn Shehory, Tanja E.J. Vos, and Nelly Condori-Fernandez. Evaluating the fittest automated testing tools: An industrial case study. In *Proceedings of the International Symposium on Empirical Software Engineering and Measurement, ESEM 2013, 7-11 October 2013, Baltimore, Maryland,USA*, pages 332–339. ACM, 2013.

[OKN15] Rafael A. P. Oliveira, Upulee Kanewala, and Paulo A. Nardi. Automated test oracles: State of the art, taxonomies, and trends. *Advances in Computers*, 95:113–199, 2015.

[PTV02] Martin Pol, Ruud Teunissen, and Erik VanVeenendaal. *Software Testing: A Guide to the TMap Approach*. Addison-Wesley, 2002.

[RF16] José Miguel Rojas and Gordon Fraser. Code defenders: A mutation testing game. In *Proc. of The 11th International Workshop on Mutation Analysis*. IEEE, 2016. To appear.

[RTC92] Inc RTCA. Do-178b, software considerations in airborne systems and equipment certification, Jan 1992.

[RTL16] RTL nieuws. 104-jarige krijgt brief van gemeente: je mag naar de kleuterschool! `http://www.rtlnieuws.nl/nieuws/opmerkelijk/104-jarige-krijgt-brief-van-gemeente-je-mag-naar-de-kleuterschool`, 2016. [Online; accessed 4-2-2017].

[Ryb07] Torbjörn Ryber. *Essential Software Test Design*. Fearless Consulting, 2007.

[SGH12] Matt Staats, Gregory Gay, and Mats P. E. Heimdahl. Automated oracle creation support, or: How i learned to stop worrying about fault propagation and love mutation testing. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 870–880, Piscataway, NJ, USA, 2012. IEEE Press.

[SHS10] Zafar Singhera, Ellis Horowitz, and Abad Shah. A graphical user interface (gui) testing methodology. In *Web Engineering Advancements and Trends: Building New Dimensions of Information Technology*, pages 160–176. IGI Global, 2010.

[Sik17] Sikuli. `http://www.sikuli.org/`, 2017. [Online; accessed 9-5-2017].

[Sil03] M. Silverstein. Logical capture/replay. *STQE Magazine*, 5(6):36–42, 2003.

[SLS14] A. Spillner, T. Linz, and H. Schaefer. *Software Testing Foundations: A Study Guide for the Certified Tester Exam*. Rocky Nook, 2014.

[Spi89] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[Spi06] D. Spinellis. *Code Quality: The Open Source Perspective*. Effective Software Development Series. Pearson Education, 2006.

[SVW09] Holger Schlingloff, Tanja E. J. Vos, and Joachim Wegener, editors. *Evolutionary Test Generation*, number 08351 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2009. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany.

[TdHXB13] Nikolai Tillmann, Jonathan de Halleux, Tao Xie, and Judith Bishop. Pex4Fun: A web-based environment for educational gaming via automated test generation. In *Proc. 28th IEEE/ACM International Conference on Automated Software Engineering (ASE 2013), Tool Demonstrations*, November 2013.

[TDN93] M. Z. Tsoukalas, J. W. Duran, and S. C. Ntafos. On some reliability estimation problems in random and partition testing. *IEEE Trans. Softw. Eng.*, 19(7):687–697, July 1993.

[Tes17] Testplant. Eggplant. `https://www.testplant.com/eggplant/testing-tools/`, 2017. [Online; accessed 9-5-2017].

[The17] TheTelegraph. Statins glitch means thousands may have been incorrectly prescribed . `http://www.telegraph.co.uk/news/2016/05/11/statins-glitch-means-thousands-may-have-been-incorrectly-prescri/`, 2017. [Online; accessed 4-2-2017].

[TLN78] T. A. Thayer, M. Lipow, and E. C. Nelson. *Software Reliability*. North-Holland Pub. Co, Amsterdam, 1978.

[Tre08] Jan Tretmans. Model based testing with labelled transition systems. *Formal methods and testing*, pages 1–38, 2008.

[Tri16] Tricentis. Software fail watch: 2016 in review. `https://www.tricentis.com/resource-assets/software-fail-watch-2016/`, 2016. [Online; accessed 19-4-2017].

[UK16] ARS Technica UK. Bug in GP software may have coughed up wrong data on heart disease risk. `http:`

//arstechnica.co.uk/security/2016/05/bug-in-gp-heart-risk-calculator-tool-tpp/, 2016. [Online; accessed 4-2-2017].

[US14] V. Uskov and B. Sekar. Gamification of software engineering curriculum. In *2014 IEEE Frontiers in Education Conference (FIE) Proceedings*, pages 1–8, Oct 2014.

[VBL+10] Tanja E. J. Vos, Arthur I. Baars, Felix F. Lindlar, Peter M. Kruse, Andreas Windisch, and Joachim Wegener. Industrial scaled automated structural testing with the evolutionary testing tool. In *Third International Conference on Software Testing, Verification and Validation, ICST 2010, Paris, France, April 7-9*, pages 175–184. IEEE Computer Society, 2010.

[VBL+12] Tanja E. J. Vos, Arthur I. Baars, Felix F. Lindlar, Andreas Windisch, Benjamin Wilmes, Hamilton Gross, Peter M. Kruse, and Joachim Wegener. Industrial case studies for evaluating search based structural testing. *International Journal of Software Engineering and Knowledge Engineering*, 22(8):1123–, 2012.

[VKC+15] Tanja E. J. Vos, Peter M. Kruse, Nelly Condori-Fernández, Sebastian Bauersfeld, and Joachim Wegener. TESTAR: tool support for test automation at the user interface level. *IJISMD*, 6(3):46–83, 2015.

[VLB14] Tanja E. J. Vos, Kiran Lakhotia, and Sebastian Bauersfeld, editors. *Future Internet Testing - First International Workshop, FITTEST 2013, Istanbul, Turkey, November 12, 2013, Revised Selected Papers*, volume 8432 of *Lecture Notes in Computer Science*. Springer, 2014.

[VLW+13] Tanja E. J. Vos, Felix F. Lindlar, Benjamin Wilmes, Andreas Windisch, Arthur I. Baars, Peter M. Kruse, Hamilton Gross, and Joachim Wegener. Evolutionary functional black-box testing in an industrial setting. *Software Quality Journal*, 21(2):259–288, 2013.

[VMEM12] Tanja E. J. Vos, Beatriz Marín, María José Escalona, and Alessandro Marchetto. A methodological framework for evaluating software testing techniques and tools. In *12th Internati-*

onal Conference on Quality Software (QSIC), Xi'an, Shaanxi, China, August 27-29, pages 230–239. IEEE, 2012.

[Vos09] Tanja E. J. Vos. Evolutionary testing for complex systems. *ERCIM News*, 2009(78), 2009.

[Vos14] Tanja Vos. Test Automation at the User Interface Level. In Vadim Zaytsev, editor, *Pre-proceedings of the Seventh Seminar in Series on Advanced Techniques and Tools for Software Evolution (SATToSE 2014)*, page 5. Dipartimento di Informatica Università degli Studi dell'Aquila, L'Aquila, Italy, jul 2014. Invited Talk.

[VTP+14] Tanja E. J. Vos, Paolo Tonella, Wishnu Prasetya, Peter M. Kruse, Alessandra Bagnato, Mark Harman, and Onn Shehory. FITTEST: A new continuous and automated testing process for future internet applications. In *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering, CSMR-WCRE 2014, Antwerp, Belgium, February 3-6, 2014*, pages 407–410, 2014.

[VTW+11] Tanja E. J. Vos, Paolo Tonella, Joachim Wegener, Mark Harman, Wishnu Prasetya, Elisa Puoskari, and Yarden Nir-Buchbinder. Future internet testing with fittest. In *15th European Conference on Software Maintenance and Reengineering, CSMR 2011, 1-4 March 2011, Oldenburg, Germany*, pages 355–358. IEEE Computer Society, 2011.

[VTW+13] Tanja E.J. Vos, Paolo Tonella, Joachim Wegener, Mark Harman, Wishnu Prasetya, and Shmuel Ur. *Testing of Future Internet Applications Running in the Cloud.*, pages 305–321. IGI Global, Scott Tilley, Tauhida Parveen, 2013.

[WBS02] Joachim Wegener, André Baresel, and Harmen Sthamer. Suitability of evolutionary algorithms for evolutionary testing. In *26th International Computer Software and Applications Conference (COMPSAC 2002), Prolonging Software Life: Development and Redevelopment, 26-29 August 2002, Oxford, England, Proceedings*, pages 287–289, 2002.

[WD92] Christopher J. C. H. Watkins and Peter Dayan. Q-learning. *Machine Learning*, 8(3):279–292, 1992.

[Wei01] G.M. Weinberg. *An Introduction to General Systems Thinking*. Dorset House, 2001.

[Wey82] Elaine J. Weyuker. On testing non-testable programs. *The Computer Journal*, 25(4):465, 1982.

[WG00] Joachim Wegener and Matthias Grochtmann. Evolutionärer test von realzeitsystemen. *Inform., Forsch. Entwickl.*, 15(3):151–160, 2000.

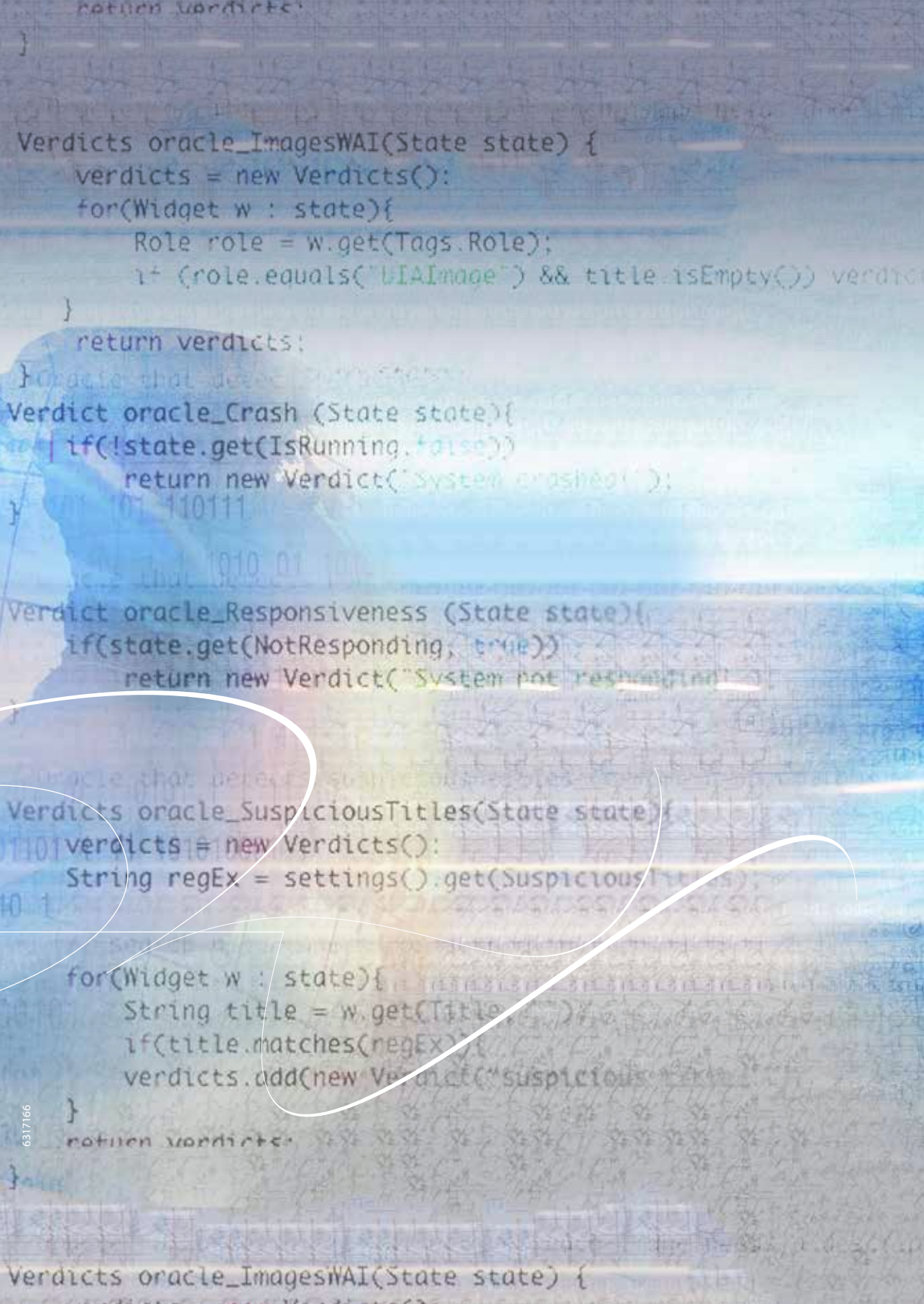
[WJ91] E. J. Weyuker and B. Jeng. Analyzing partition testing strategies. *IEEE TSE*, 17(7):703–711, Jul 1991.

[WO80] E. J. Weyuker and T. J. Ostrand. Theories of program testing and the application of revealing subdomains. *IEEE TSE*, SE-6(3):236–246, May 1980.

[YCM09] Tom Yeh, Tsung-Hsiang Chang, and Robert C. Miller. Sikuli: Using gui screenshots for search and automation. In *Proceedings of the 22Nd Annual ACM Symposium on User Interface Software and Technology*, UIST '09, pages 183–192, New York, NY, USA, 2009. ACM.

[ZHM17] Yuanyuan Zhang, Mark Harman, and Afshin Mansouri. The SBSE repository: A repository and analysis of authors and research articles on search based software engineering, 2017. `crestweb.cs.ucl.ac.uk/resources/sbse_repository/`.

Index


Verdicts oracle_ImagesWAI(State state) {
verdicts = new Verdicts();
for(Widget w : state){
Role role = w.get(Tags.Role);
if (role.equals("UIAImage") && title.isEmpty()) verdicts
}
return verdicts;
}
Verdict oracle_Crash (State state){
if(!state.get(IsRunning, false))
return new Verdict("System crashed");
}
Verdict oracle_Responsiveness (State state){
if(state.get(NotResponding, true))
return new Verdict("System not responding");
}
Verdicts oracle_SuspiciousTitles(State state){
verdicts = new Verdicts();
String regEx = settings().get(SuspiciousTitles);
for(Widget w : state){
String title = w.get(Title, "");
if(title.matches(regEx)){
verdicts.add(new Verdict("suspicious
}
return verdicts;
}
Verdicts oracle_ImagesWAI(State state) {

6317166