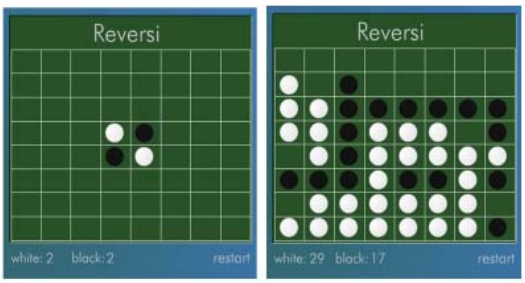
TOOLS

Niemand wil software met veel fouten gebruiken. Niet alleen omdat het vervelend is, maar fouten kunnen ook ernstige gevolgen hebben. Het is daarom belangrijk dat ontwikkelaars hun software testen voor ze deze op de markt zetten. Dit is echter geen makkelijke klus. Elke test is uniek: het controleert hoe de software reageert op een unieke reeks van input. Dit betekent dat heel veel tests nodig zijn om allerlei mogelijke configuraties en interacties van de software te controleren. De tests zijn vaak met de hand geschreven. Het is dus zeer arbeidsintensief en duur.

Automatisch testen van Java klassen

Wishnu Prasetya
(wishnu@cs.uu.nl)

Een eenvoudig voorbeeld rond het spel Reversi. Het is een klein strategiespel van twee spelers op een 8x8 bord. Om de beurt leggen de spelers een schijfje van de eigen kleur (zwart of wit) op het bord. De schijven van de tegenstander die door de zet omsloten raken, worden omgezet naar de speler's kleur. Het spel is afgelopen als beide spelers geen zet meer kunnen doen. Dankzij nog een paar andere regels is het toch een vrij complex spel, met circa 10^{28} mogelijke posities.
In Figuur 1 twee screenshots van een online versie van dit spel.



Figuur 1: Links het begin van het spel en rechts een situatie na 46 zetten.

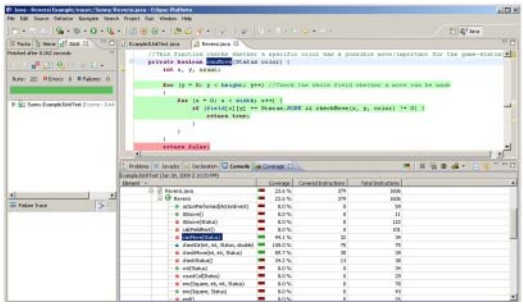
Als het spel in Java is geïmplementeerd, is het in twee lagen gesplitst: de spellogicalaag, en de user-interfacelaag. Deze splitsing maakt het mogelijk om de lagen afzonderlijk te testen. De spellogica-laag is het brein van een applicatie, dus het meest kritische deel van de applicatie. In Java gebruikt men vaak JUnit om tests te schrijven. JUnit maakt het schrijven van tests en het analyseren van het resultaat eenvoudiger, maar het is geen automatische tool: je moet nog steeds de tests zelf schrijven.

```
@Test t1(){
Reversi R = new Reversi() ;

R.move(new Square(3,2)) ;
assert R.getSquare(3,2) == WHITE
assert R.getSquare(3,3) == WHITE
}

@Test t2(){
Reversi R = new Reversi() ;
R.move(new Square(4,2)) ;
assert R.getSquare(4,2) == EMPTY ;
}
```

Figuur 2: twee tests, t1 en t2, op de spellogica van Reversi, geschreven in Java/JUnit.

In Figuur-2 staan twee voorbelden van JUnit tests van de spellogica van Reversi. De eerste test t1() doet een zet (wit is als eerste aan de beurt) op het vakje (3,2). Dit is het vakje op de vierde kolom van links en derde rij van onder op het bord. Het controleert vervolgens dat er op dat vakje nu inderdaad een witte schijf zit en dat op het vakje (3,3), die door en zwarte schijf bezet was, nu een witte schijf zit. De tweede test t2() probeert een andere zet, namelijk (4,2). Volgens de regels van Reversi is de zet echter illegaal. De test controleert dus of op dat vakje inderdaad geen schijf is geplaatst.
Deze twee tests zijn erg eenvoudig, maar er zijn meer tests nodig. Een belangrijke, maar lastige vraag is: hoeveel tests heb je eigenlijk nodig? Reversi heeft naar schatting 10^{58} mogelijke speelpartijen. Een partij is een volledige reeks van zetten van beide spelers, vanaf een beginconfiguratie tot het spel eindigt. Het is ondoenlijk om ze allemaal te testen. Daarbij is Reversi een klein programma; grotere programma's hebben nog grotere aantallen van configuraties en interacties om te testen.

Pragmatisch gezien is het doel van testen dus niet het vinden van alle fouten, maar het vinden van de meeste fouten binnen een redelijke tijd. Het percentage van de delen van een programma die door een test worden uitgevoerd heet de dekking van de test. Hogere dekking geeft meer kans om fouten te vinden. Omdat dekking meetbaar is, is het in de praktijk ook het meest gebruikte criterium van testvolledigheid. Er zijn ook tools om testdekking te meten, bijvoorbeeld Emma en Cobertura. Allebei zijn open source. Emma heeft ook mooie plugins voor Eclipse en Netbeans; zie de screenshot in Figuur 3. Je moet echter goed beseffen dat zelfs 100% dekking geen garantie is voor foutloze software. Je kunt ervoor kiezen om fijnere aspecten van het programma ook te dekken, maar daar staat tegenover dat je meer tests moet schrijven.



Figuur 3: JUnit en Emma in Eclipse. JUnit houdt automatisch bij welke tests slagen of falen. De groene balk linksboven betekent dat ze allemaal slagen. Emma houdt de dekking van de tests bij. Het kleurt de broncode (bovenblad): groene regels zijn al gedekt, rode nog niet. Het maakt ook een handige overzichtslijst (onderblad) van het dekkingspercentage van verschillende klassen en methodes.

Eigenlijk dekken de twee tests in Figuur 2 al vele delen van Reversi. De verleiding is groot om hier te stoppen, maar vasthoudendheid is bij het testen geen overbodige luxe. Bugs loeren vaak in moeilijk te dekken plekken van de code. Dit is ook waar testen lastig wordt. Om de implementatie van het eidnspel te testen zou een test nodig zijn met een lange reeks van zetten die het spel helemaal uitspeelt. De grootst mogelijke partij telt 64 zetten; het kost veel werk om de zetten te bedenken en om ze op te schrijven. Een specifieke situatie zoals gelijkspel is ook erg moeilijk handmatig te reconstrueren. Een andere lastige situatie is waar een speler zijn beurt moet overslaan, omdat hij geen zet kan doen.

Dit is niet het enige probleem van handmatig testen. Testcodes zoals in Figuur 2 zijn ook fragiel. Stel dat je besluit om de spelregels te veranderen. In bedrijven is dit een realistisch scenario. Een bedrijf kan haar beleid aanpassen. Bedrijven kunnen fuseren. De regering kan regels en wetten veranderen. Deze kunnen allemaal leiden tot aanpassingen in een bedrijfsapplicatie. Na de aanpassingen kunnen delen van de tests onbruikbaar worden omdat ze de applicatie op verkeerde manieren aansturen, of omdat ze de reacties van de applicatie met verkeerde waarden vergelijken. De tests moeten dus worden bijgewerkt en dit is vaak net zo lastig als opnieuw schrijven.

Automatisch Testen

Je kunt kosten besparen door tests automatisch te genereren. Er is veel onderzoek gedaan naar dit onderwerp en dit heeft geleid tot tools zoals FindBugs (University of Maryland), JCrasher (Georgia Institute of Technology), Korat (MIT), Agitar (commercieel), SpecExplorer (Microsoft Research), en T2. De laatste is van Universiteit Utrecht, met een unieke feature dat het een Java klasse in haar geheel test. Het test een methode en niet geïsoleerd, zoals in 'unit testen' gebeurt. Unit testen is het testen van een 'unit'. Een unit is meestal een functie of een methode. Omdat een unit klein is, heb je beter overzicht van hoe het werkt en kun je ook beter testen. Voor een objectgeoriënteerde taal komt deze definitie van 'unit' echter te kort. Zie het voorbeeld onder:

```
class Caldendar{
private int day = 1 ;
public void nextDay() { day++ ; }
public int getWeek() { return day/7 ; }
public void reset() { day=0 ; }
}
```

De waarde die c.getWeek() teruggeeft, hangt van de toestand van het object c af; dit is op zijn beurt afhankelijk van wat je met c hiervoor deed. Als je c.reset() uitvoert net voor de aanroep c.getWeek(), gaat de tweede crashen. Deze fout komt niet aan het licht als getWeek geïsoleerd wordt getest. T2 genereert daarom 'testreeksen'. Elke reeks begint met het aanmaken van een doelobject; dit is een object uit de klasse die getest moet zijn. Elke stap in de reeks is een methode aanroep op het doelobject, of een update op een van zijn velden. Op deze manier simuleert T2 interacties tussen methodes. Bovendien controleren methodes in dezelfde reeks ook elkaar, en heeft T2 dus een betere kans om fouten te vinden. Per stap controleert T2 dat er geen assertieovertredingen of andere onverwachte excepties zijn. Afhankelijk van de doelklasse kan T2 duizenden testreeksen binnen één seconde genereren. Dit zijn heel veel tests met slechts één druk op de knop! T2 heeft ook geen probleem om lange testreeksen voor Reversi te genereren en de eerder genoemde lastige spelsituaties te dekken.

T2 is open source met een GPL-licentie. Je kunt het gebruiken als stand alone of als een bibliotheek vanuit JUnit. Het werkt dus ook in elke IDE die JUnit ondersteunt (zoals Eclipse of Netbeans).

Specificatie-gebaseerd Testen

In handmatige testcode zoals in Figuur-2 worden vaak concrete waarden gebruikt om de verwachtingen uit te drukken, zoals in:

```
assert R.getSquare(3,2) == WHITE
```

Je verwacht dat er in het vakje (3,2) een witte schijf zit. Als de zetten anders moeten zijn, bijvoorbeeld omdat de spelregels veranderen, moeten de 'verwachtingen' ook worde aangepast.
Dit probleem los je op door formele specificaties te schrijven met een 'specificatietaal'; een taal die speciaal voor dat doel is ontworpen. Voorbeelden: Object Constraint Language (OCL, een onderdeel van UML), Java Modelling Language (JML), of Z. Één van de spelregels van Reversi is dat het spel eindigt als beide spelers geen zet meer kunnen zetten. In OCL kan men dit bijvoorbeeld zo uitdrukken:

```
context: Reversi
inv: possibleMoves(BLACK).isEmpty() and
possibleMoves(WHITE).isEmpty()
      implies
      status() != ONGOING
```

Het beschrijft een voorwaarde, een zogenaamde klasse-invariant, voor de klasse Reversi, namelijk dat het waar moet zijn na elke aanroep op de methodes van Reversi. Een specificatie kan men blijven hergebruiken in alle tests. Als de spelregels veranderen hoeft je alleen de specificaties bij te werken in plaats van dat te moeten op alle tests. Specificaties zijn dus veel robuuster.

Bedrijven zijn echter vaak huiverig om specificaties te schrijven. Het is moeilijk om daarvoor de juiste programmeurs te vinden. Bestaande specificatietalen integreren ook niet volledig met populaire programmeertalen. Dit leidt tot onderhoudsproblemen. Het is dus goed voor te stellen dat een bedrijf dit te riskant vindt. Specificaties kunnen echter ook in een programmeertaal zoals Java worden geschreven. Het heet in-code specificatie. Het bezwaar is echter dat ze lelijk zijn.
Hier valt iets te leren van de Embedded Domain Specific Language-aanpak (EDSL). Een domeinspecifieke taal is een (kleine) taal specifiek voor het uitdrukken van uitspraken uit een bepaald domein. In plaats van een aparte vertaler voor de taal te schrijven, wat heel veel werk kost, kun je vaak een kleine taal in een rijke taal zoals Java simuleren door een bibliotheek van methodes te schrijven, die de basiswoorden van de taal vormen. Zinnen bouw je door deze woorden te combineren. Specificaties van een applicatie vormen eigenlijk ook een eigen domeinspecifieke taal, en kunnen dus ook 'embedded' in Java gecodeerd. Qua syntax ziet het resultaat er redelijk netjes uit (het voorbeeld onder). Echt mooi zoals in OCL zullen ze inderdaad nooit worden, maar daar staat tegenover dat er geen integratieprobleem meer zijn.

Stel dat je de klasse Reversi uitbreidt met de methodes cnt(c), possibleMoves(c), en status(). Deze geven respectievelijk het aantal van de schijfjes van de speler c, de mogelijke zetten van c en de status van het spel terug. Je kunt dan zeggen dat ze nu de woordenschat zijn waarmee je de specificaties van Reversi gaat schrijven. De spelregel die eerder in OCL was, kun je nu ook in Java schrijven; het ziet er bijna net zo abstract uit als de OCL versie:

```
boolean classinv(){
   if (possibleMoves(BLACK).isEmpty() &&
posibleMoves(WHITE). isEmpty())
return status() != ONGOING ;
   else return true ;
}
```

De eerder genoemde tool T2 controleert ook klasse-invarianten. Dus het begrijpt specificaties zoals boven. Hier is er nog een:

```
boolean classinv(){
   if   (status() == BLACKWIN) return
(cnt(BLACK)>cnt(WHITE)) ;
else if (status() == WHITEWIN) return
(cnt(BLACK)<cnt(WHITE)) ;
   else return (cnt(BLACK)==cnt(WHITE)) ;
}
```

Deze klasse-invariant codeert de spelregel die de winaar van het spel bepaalt.

Op Zoek naar 100% Dekking

Zou een automatische tool zoals T2 100% testdekking kunnen leveren? In principe niet. Onderzoekers noemen dit probleem onbeslisbaar. Geen computer kan het voor alle gevallen kan oplossen, ongeacht hoe slim de computer is. Een tool kan veel werk besparen, maar het is hoe dan ook geen volledige vervanger van handmatige tests. De ontwikkeling van de tool zorgt natuurlijk wel voor een betere dekking. De basismachine van T2 genereert testreeksen op randombasis. Dit is de makkelijkste en snelste manier. Deze levert vaak 70-80% dekking.

Maar hoe hoog de dekking precies is, is voor de onderzoekers van Universiteit Utrecht nog een open vraag. In een onlangs uitgevoerd experiment werd geprobeerd klassen te instrumenteren om de executiepaden van een test bij te houden. Zo kun je een groep van testreeksen selecteren die het dichtst bij een nog ongedekt deel van een methode komen. Met een bepaalde heuristiek worden de parameters van deze testreeksen gevarieerd om nieuwe reeksen te maken. Dit werkt heel goed, maar het algoritme is nog niet snel genoeg. Overal in de wereld is het dekkingprobleem een hot onderzoekitem. Er wordt van alles geprobeerd: adaptieve random, symbolische executie, genetische algoritmen. Er wordt dus hard gewerkt. Hopelijk leidt dit ook tot spannende doorbraken. «

Bedrijven zijn vaak huiverig om specificaties te schrijven

Wishnu Prasetya
(wishnu@cs.uu.nl)