Programmeerplagiaatdetectie met Marble

Jurriaan Hage

Department of Information and Computing Sciences, Utrecht University

Technical Report UU-CS-2006-062

www.cs.uu.nl

ISSN: 0924-3275

Programmeerplagiaatdetectie met Marble

Jurriaan Hage

Department of Information and Computing Sciences, Utrecht University
P.O.Box 80.089, 3508 TB Utrecht, The Netherlands
{jur}@cs.uu.nl

Abstract. In Utrecht gebruiken we bij verscheidene programmeervakken al enkele jaren een plagiaatdetector, momenteel genaamd Marble. Ik beschrijf in dit artikel hoe we met plagiaat en andere vormen van fraude omgaan in Utrecht, hoe Marble werkt en gebruikt kan worden, en de resultaten van een experiment om de werking van Marble te testen. Voorts geef ik een overzicht van de vakken waar Marble is toegepast en welke technieken zijn gebruikt om Marble beknopt en efficiënt te krijgen. Cruciaal hierbij is het gebruik van reguliere expressies.

1 Inleiding

In 2002 werd er begonnen met het ontwikkelen van een eenvoudige tool die behulpzaam moest zijn bij het detecteren van plagiaat gepleegd door studenten bij het vak Gedistribueerd Programmeren. Met het groeien van het aantal studenten, en dus ook het aantal ingeleverde opdrachten, werd het steeds moeilijker om op structurele wijze de aanwezigheid van plagiaat uit te sluiten. Tot op dat moment was de ontdekking van plagiaat vooral het gevolg van het feit dat de geplagieerde opdracht iets op een aparte wijze deed en er opvallend genoeg een tweede inlevering was waarbij hetzelfde gebeurde. Niet alleen is dit sterk afhankelijk van de oplettendheid van de nakijker, worden opdrachten vaak verdeeld over verschillende nakijkers (wat de kans om gesnapt te worden behoorlijk verkleint), maar erger nog is dat plagiaat vaak gepleegd wordt met programma's die in eerdere jaren werden gemaakt en vervolgens door de student op een website werden gezet ter illustratie van zijn kunnen, alwaar ze eenvoudig kunnen worden teruggevonden door de nieuwe lichting studenten.

Het doel van de tool, Marble genaamd, is *niet* om fraudeurs aan te wijzen. Het geeft voor een lijst vergeleken Java-bestanden aan in welke mate die op elkaar lijken, en wel zo dat de paren bestanden met de grootste overeenkomst vooraan staan. De gebruiker van Marble dient dan handmatig deze paren te vergelijken om te constateren of er inderdaad sprake is van plagiaat. De praktijk wijst uit dat dit vrijwel altijd eenvoudig te zien is (op basis van de manier van inspringen, de inhoud van commentaar, gebruikte constanten, etc.). Het doel van het tool is vooral zoveel mogelijk paren uit te sluiten die (waarschijnlijk) *geen* plagiaat vormen.

Alle programma's die door Marble zijn bekeken worden bewaard, zodat als jaren later dezelfde opdracht wordt gebruikt (of een sterk vergelijkbare), dan kunnen nieuwingeleverde programma's ook worden vergeleken met die oudere inleveringen. De huidige praktijk wijst uit dat juist hier het meeste plagiaat te vinden is. Een andere reden om historische vergelijkingen te maken is het volgende scenario: student P. en Q. maakten in 2003 een opdracht samen, met een voldoende resultaat. P. haalde echter het vak niet en doet het vak opnieuw in 2005. Nu levert hij een sterk gelijkende uitwerking van dezelfde opdracht in, samen met student R. Ook dit wordt door Marble ontdekt, zodat de student R. aangesproken kan worden op zijn meeliftgedrag dan wel fraudegedrag.

Marble is sinds 2003 ingezet bij verschillende vakken (hoewel we ook kunnen vergelijken met nog oudere programma's, teruggaand tot het cursusjaar 1999), en heeft de nodige gevallen van plagiaat ontdekt. Vooral bij eerstejaarsvakken is er nog sprake van een behoorlijk aantal plagiaatgevallen. Recentelijk bij een opdracht van Imperatief Programmeren acht gevallen op zo'n tweehonderd inleveringen. En dit terwijl studenten op voorhand werden gewaarschuwd dat plagiaatdetectie plaats zou vinden. Bij vakken in latere jaren vindt veel minder plagiaat plaats: bij een vak als Gedistribueerd Programmeren is sinds de introductie van de plagiaatdetector het aantal

gevallen gedaald tot eentje per twee à drie jaar. Meer resultaten en gegevens over de inzet van het programma vindt de lezer in Sectie 3 en Sectie 5.

De vraag is altijd in welke mate studenten in staat zijn om een plagiaatdetector te omzeilen, door middel van programmatransformaties die de semantiek van een programma identiek laten, maar anderzins de broncode in sterke mate veranderen. Hierbij valt te denken aan het vertalen van commentaar van Nederlands naar Engels, het veranderen van identifiernamen, het verwijderen of juist toevoegen van commentaar, maar in het geval van bijvoorbeeld Java ook het verwisselen van methodes en interne klassen. Natuurlijk bestaan er al de nodige tools om programma's onherkenbaar te maken, de zogenaamde obfuscators. Gelukkig hebben wij hier het voordeel dat als een student gebruik maakt van een dergelijke tool, dat dit bij het nakijken direct op zal vallen. Deze realisering is essentieel: we hoeven alleen maar na te gaan hoe waarschijnlijk het is dat verschillende inleveringen afkomstig zijn van dezelfde bron en mogen ons hierbij concentreren op inleveringen die voor een nakijker als zinnig worden ondervonden. Dat wil zeggen dat operaties die sterk negatief bijdragen aan de begrijpelijkheid en modulariteit zoals loop-unrolling en inlining, en het vervangen van identifiernamen door willekeurige andere namen, slechts in beperkte mate plaats kunnen vinden.

Zoals de resultaten van Marble duidelijk maakt, dient een bruikbaar tool wel om te kunnen gaan met verschillen in commentaar, identifiernamen, inspringen en volgorde van methodes en dergelijke. Een echte studie over wat er met bestaande refactoringtools nog meer bereikt kan worden en waardoor Marble eventueel omzeild kan worden is nog niet uitgevoerd, maar in Sectie 5 doen we kort verslag van een experiment dat door twee studentassistenten werd uitgevoerd om na te gaan in welke mate Marble in staat is om plagiaat te detecteren.

We beginnen echter in Sectie 2 met een beschrijving hoe wij in ons departement met fraude en plagiaat omgaan, terwijl Sectie 3 een overzicht geeft waar Marble is ingezet. We vervolgen in Sectie 4 met een beschrijving van het plagiaatdetectieproces en in Sectie 5 geven we de resultaten van een experiment, bedoeld om Marble te valideren. In Sectie 6 beschrijven we kort hoe Marble geïmplementeerd is, welke keuzen hierbij zijn gemaakt en wat de gevolgen hiervan zijn. Hoewel deze laatste twee delen toegespitst zijn op plagiaatdetectie voor Java, zijn de principes direct over te dragen naar andere programmeertalen. Tot slot, geven we in Sectie 7 een overzicht van wat er nog aan Marble toegevoegd zou kunnen worden.

2 Omgang met plagiaat en fraude in ons departement

Ter introductie beschrijf ik nu kort hoe men in het Departement Informatica (waaronder ook de studie Informatiekunde valt) van de Universiteit Utrecht omgaat met fraude en plagiaat. Onder *plagiaat* versta ik, in de lijn van de Docenthandleiding van ons departement [2],

> het in een scriptie of ander werkstuk gegevens of tekstgedeelten van anderen overnemen zonder bronvermelding.

Onder werkstukken vallen niet alleen geschreven stukken tekst, maar ook (computer)programma's en andersoortig werk. In dit artikel zal technisch gezien alleen ingegaan worden op plagiaatdetectie voor computerprogramma's.

Het begrip *fraude* is iets ruimer, en omvat onder andere het spieken tijdens tentamens, het fingeren van onderzoeksresultaten en het zich voor de tentamendatum in het bezit stellen van een tentamen. In deze sectie zal ik voor het gemak plagiaat beschouwen als een vorm van fraude, terwijl alle volgende secties zich exclusief bezighouden met plagiaat.

De examencommissie van het departement is verantwoordelijk voor het bepalen dat fraude heeft plaatsgevonden, zoals beschreven staat in Artikel 4.9 van de Onderwijs- en Examenregeling 2006 voor de bacheloropleiding, en voor de Masteropleiding binnen de graduate school Artikel 3.10 van haar Onderwijs- en Examenregeling 2006. Deze regelingen zijn alle terug te vinden op de website van het departement [3]. De examencommissie begint haar werk te doen als een examinator van een vak haar attendeert op een mogelijk geval van fraude. Voordien heeft de examinator de

student al schriftelijk op de hoogte gesteld van zijn vermoeden van fraude, en stelt de student in staat hierop te reageren. De uitkomsten hiervan worden aan de examencommissie verstrekt.

Mocht de examencommissie inderdaad besluiten dat het in het onderhavige geval om fraude gaat, dan wordt de student hier schriftelijk van op de hoogte gebracht. In ieder geval wordt van de constatering nota gemaakt in het dossier van de student, en wordt het werk, waarbij fraude werd geconstateerd, ongeldig verklaard. Verder kan de examencommissie overgaan tot één of verscheidene sancties, zoals uitsluiting van het vak voor de duur van een jaar (dit is in de praktijk de standaardsanctie) of zelfs volledige uitsluiting van deelname aan alle tentamens en vormen van toetsing voor de periode van een jaar. Indien de student, volgens zijn dossier, al eerder betrapt is op fraude, is de examencommissie verplicht deze laatste sanctie op te leggen, alsmede de student te adviseren de opleiding te verlaten. Deze regeling geldt niet alleen voor het Departement Informatica, maar voor de gehele Graduate School of Natural Sciences waarbinnen onze masteropleidingen vallen. Zelfstandig heeft het departement besloten die regeling over te nemen voor de eigen bacheloropleidingen.

De vraag is of een dergelijke straf zelfs bij herhaling verplicht opgelegd dient te worden. Het gevolg hiervan zal, is de mening van de auteur, eerder zijn dat sommige gevallen van fraude niet als zodanig worden aangemerkt (en daardoor de zaak zullen vertroebelen), dan dat hiervan een afschrikkende werking uit zal gaan. Een voorbeeldscenario is eenvoudig te construeren: student S. komt in zijn eerste jaar tijd te kort en kopieert een programma van een medestudent. Plagiaat wordt geconstateerd en de student wordt conform de regeling gestraft. In zijn derde jaar dient hij een korte scriptie te schrijven over een complex technisch onderwerp, waarbij hij plaatjes gebruikt van een website of uit een boek. Hij vergeet echter correcte referenties op te nemen naar plaatjes. In principe, is dit alweer een vorm van fraude, zelfs als de student wel de bron in zijn literatuurlijst opneemt. Conform de regels zou de student nu voor een jaar uitgesloten dienen te worden van deelname aan vakken en geadviseerd te worden de opleiding te verlaten.

Enkele jaren geleden al is er vanuit de Examencommissie van het departement actie ondernomen om fraude en plagiaat structureel tegen te gaan, grotendeels in de lijn van wat hierboven werd beschreven. Tot die tijd werd fraude niet gecentraliseerd bijgehouden, wat het erg eenvoudig maakte voor studenten om bij verschillende vakken fraude te plegen en daar dan vaak nog redelijk eenvoudig vanaf te komen. Het bijhouden van een centraal dossier helpt dit te voorkomen en maakt ook duidelijk aan de student dat we fraude zeer serieus nemen. Sinds de invoering van de dossiers, is tot nog toe slechts tweemaal een student herhaaldelijk op fraude betrapt. In ieder geval één van die twee gevallen ging het dan nog om onbegrip over wat nu precies toegelaten is bij het schrijven van scripties op basis van het werk van anderen.

Als reactie op dit laatste fenomeen heeft de auteur een lezing voorbereid hoe aantijgingen van plagiaat vermeden kunnen worden. Deze lezing wordt viermaal per jaar gegeven in het kader van het vak Overdragen van Informatica, waarbij de studenten onder andere een scriptie moet schrijven over een informaticaonderwerp op basis van externe bronnen (het daadwerkelijk doen van onderzoek past niet binnen dit vak). In de lezing worden concrete aanwijzingen gegeven wat wel en niet toegestaan is, zoals het rechtstreeks kopieren, maar ook het parafraseren of vertalen van de teksten van anderen, zonder hierbij bronvermeldingen op te nemen. Dit laatste voorkomt in ieder geval aantijgingen van plagiaat, hoewel de student daarnaast ook nog de nodige eigen inbreng moet tonen om tot een voldoende te komen. Ook hierin geeft de lezing de nodige hints: verzin eigen voorbeelden, gebruik zoveel mogelijk verschillende bronnen, vergelijk de verschillende bronnen kritisch en herstructureer de tekst (in sterke mate). Een constatering die hierbij wordt gemaakt, is dat de grenzen tussen eigen woorden en kopieerwerk niet duidelijk te stellen zijn, en dat het bij twijfel wijs is vooraf bij de examinator te rade te gaan. Het uitgangspunt bij dit vak is uiteindelijk dat de student moet laten blijken aan de examinator dat hij in staat zou zijn een begrijpelijk verhaal op te hangen over iets waarbij hij *niet* beschikt over welke externe bron dan ook.

3 Gebruik van Marble binnen het onderwijs

Het Marble systeem is al sinds 2003 bij ons departement in gebruik, met name bij de vakken Imperatief Programmeren (IMP) en Gedistribueerd Programmeren (GDP). Voor het vak Internetprogrammeren (INP) zijn ook versies van het programma gemaakt die, hoewel minder precies, vergelijkbare functionaliteit hebben voor het detecteren van plagiaat in Perl, XSLT en PHP programma's. Het verschil zit hem hierbij voornamelijk in hoe het normalisatieproces verloopt (zie Sectie refplagiaatdetectie).

De volgende tabel geeft weer waar en in welke mate Marble is ingezet: Met de taal Java hebben we de meeste ervaring. De afkorting APA staat voor Automatische Programma-analyse en ALG voor Algoritmiek.

Naam	# incarnaties	# inleveringen	# bronbestanden	vak
mandelbrot	7	762	840	IMP
tournament	1	62	248	INP
animatedquicksort	5	187	1043	GDP
reversi	7	662	1141	IMP
treeroamer	2	46	335	GDP
monotoneframeworks	1	2	38	APA
petersonshortcut	5	104	578	GDP
sensornetwork	1	36	278	GDP
webshopservlets	1	47	112	INP
changroberts	1	40	210	GDP
spanningtree	4	87	411	GDP
prettyprint	2	95	217	ALG
threadedmergesort	4	78	482	GDP

Met niet-Java-programma's hebben we wat minder ervaring. Merk op dat in het geval van XSLT de ingeleverde bestanden letterlijk werden vergeleken. Zelfs dit leverde al iets op.

Naam	# incarnaties	# inleveringen	vak	taal
phpdiscussiongroup	1	66	INP	PHP
cartoonperl	1	56	INP	Perl
website	1	36	INP	PHP
javaperl	1	53	INP	Perl
xmlitunes	1	31	INP	XSLT

Helaas is er door de jaren heen niet structureel bijgehouden hoeveel gevallen van plagiaat er zijn ontdekt. Dit hangt samen met het feit dat de resultaten van Marble aan de docenten wordt teruggegeven, waarna het aan hen is om actie te ondernemen.

Bij GDP werd er dit jaar een tweetal ontdekt waarvan één van beide deelnemers poogde mee te liften op werk dat de ander in eerdere jaren had gedaan. Bij de prettyprint opgave werd bij de eerste incarnatie ook minstens één geval van plagiaat ontdekt; dit bleek een geval te zijn dat de docenten zelf ook al was opgevallen. Bij de tweede incarnatie, uitgevoerd in December 2006, werden geen gevallen ontdekt. Ook bij de XML iTunes opgave werd een geval van plagiaat ontdekt, hoewel dit uiteindelijk een verregaande mate van samenwerking bleek te zijn. Bij de laatste incarnatie van Reversi voor IMP werden er ruim vijf gevallen van plagiaat ontdekt. Het precieze aantal is nog niet duidelijk, omdat de docent nog bezig is met de communicatie met de studenten in kwestie. Potentieel ging het hier om acht verdachte gevallen. Uit het feit dat plagiaat bij de andere vakken incidenteel is, blijkt dat juist het eerstejaarsvak over programmeren het vak is waarbij plagiaat veelvuldig voorkomt. Dit lijkt vooral te maken hebben met het feit dat er bij dit vak een vrij veel studenten meedoen die erg veel moeite hebben met programmeren, maar die door het verplichte karakter van het vak hier niet aan kunnen ontkomen.

4 Programmeerplagiaatdetectie

Hoewel Sectie 2 grotendeels ging over het schrijven in Engels en Nederlands en niet over programmeren, zijn er wel de nodige overeenkomsten. Zo is ook hier sprake van duidelijke gevallen van plagiaat, duidelijke gevallen waarin het geen plagiaat betreft en een flink grijs gebied. Zo kunnen twee inleveringen op elkaar lijken, omdat de studenten met elkaar overlegd hebben hoe één en ander aangepakt zou kunnen worden, maar dat ze dit uiteindelijk zelfstandig in een programma hebben omgezet. Dit is één van de redenen dat de examinator altijd gevonden overeenkomsten handmatig dient te beoordelen, en ook de student(en) de ruimte moet geven om te reageren. In de praktijk komt dit neer op het laten langskomen van de studenten en door middel van ondervraging na te gaan of ze het geïmplementeerde werk in voldoende mate begrijpen. Deze zelfde methode kan ook worden gebruikt bij het opsporen van meeliftgedrag. Onze ervaring is dat studenten die kopiëren, niet de moeite nemen het gekopieerde werk ook echt te begrijpen. Plagiaat komt bijna altijd voort uit een gebrek aan tijd of een gebrek aan kunde in het programmeren, en niet zozeer de onwil om zelf iets te bouwen.

Plagiaatdetectie met Marble gaat ruwweg op de volgende wijze. Uitgaande van een vaste layout voor de inleveringen, worden allereerst de nieuwe inleveringen geprepareerd. Vervolgens wordt het plagiaatdetectieproces gestart welke ook een vergelijking maakt met inleveringen uit eerdere incarnaties van het vak.

4.1 Layout van het filesysteem

Er wordt uitgegaan van een filesysteem met de volgende layout:

```
hallowereld/        -- opdrachtnaam
  ...
  0405p1/           -- incarnatienaam, periode 1 van cursusjaar 2004/2005
    ...
  0405p3/
    ...
  0506p1/
    jur/            -- nakijkernaam
      inlevering1/  -- meestal de inlognamen van de studenten
        Dag Wrede Wereld.java
        src/
          Hallo Wereld.java
      inlevering2/
        ...
    nakijker2/
    ...
```

Per programmeeropdracht is er een directory voor iedere incarnatie van het vak. Binnen elke incarnatie vinden we een subdirectory voor iedere nakijker, en daarbinnen weer een directory voor iedere inlevering (dit is normaalgesproken een opsomming van de namen van de studenten verantwoordelijk voor de inlevering, bijvoorbeeld `student1-student2`). Op de interne structuur van de inlevering hebben we geen invloed. Dit kan een heel platte structuur zijn met alle broncode op toplevel, of een uitgebreide geneste structuur met packages en dergelijke. Wij maken hierover daarom geen enkele aanname.

Voor het gemak is deze layout precies degene die gebruikt wordt door het zogenaamde Submit systeem dat wij in ons departement gebruiken [4]. Met dit systeem kunnen studenten via een webinterface opdrachten inleveren, en kunnen de examinator en zijn assistenten op eenvoudige wijze de resultaten van het nakijken invoeren en communiceren naar de studenten. De verdere ondersteuning die het systeem biedt aan de examinator om het nakijkwerk te verspreiden over zijn nakijkers is wat Marble gebruikt om zijn werk te doen. Examinators die Submit gebruiken (en dat zijn veruit de meesten), kunnen zo op eenvoudige wijze kun materiaal gereed krijgen voor Marble.

Als alles in bovenstaande vorm aanwezig is, kan het plagiaatdetectieproces beginnen. Deze valt uiteen in twee fasen: de normalisatiefase en de plagiaatdetectiefase. In de normalisatiefase wordt de inlevering van elke student waarvoor de plagiaatdetectie plaatsvindt geprepareerd voor het eigenlijke onderlinge vergelijken dat plaatsvindt in de plagiaatdetectiefase.

In wat nu volgt gaan we er vanuit dat de incarnatie genaamd `0506p1` de meest recent toegevoegde is. De andere incarnaties zijn normaliter al in een eerder stadium genormaliseerd en onderling vergeleken. De bedoeling van deze "ronde" is nu om de incarnatie `0506p1` toe te voegen, te normaliseren, alle genormaliseerde bronbestanden van `0506p1` onderling te vergelijken, en deze ook te vergelijken met alle genormaliseerde bronbestanden van voorgaande incarnaties.

4.2 De normalisatiefase

Beperken we ons voor het gemak tot Java, dan omvat normalisatie van de inlevering het volgende: normaliseer de naamgeving van bronbestanden, splits de Java-broncode op zodat elk bronbestand slechts één klasse-definitie bevat (deze worden simpelweg genummerd zoals beneden in het voorbeeld aangegeven), en transformeer de Java-broncode naar een speciale vorm (in het geval van Java zetten we deze zelfs om naar twee verschillende vormen). Deze getransformeerde versies krijgen allemaal hun eigen extensie. Voor elke speciale vorm is er een extensie, zodat ze in de detectiefase eenvoudig terug te vinden zijn. In het geval van Java zijn dit `.nf` (normalized form) en `.nfs` (normalized form sorted).

In het geval van het bovenstaande voorbeeld zal normalisatie het volgende opleveren.

```
hallowereld/
  ...
  0405p1/
    ...
  0405p3/
    ...
  0506p1/
    jur/
      inlevering1/
        Dag Wrede Wereld.java
        Dag@Wrede@Wereld.java.nf.0
        Dag@Wrede@Wereld.java.nfs.0
        Dag@Wrede@Wereld.java.nf.1
        Dag@Wrede@Wereld.java.nfs.0
        src!Hallo@Wereld.java.nf.0
        src!Hallo@Wereld.java.nfs.0
        src/
          Hallo Wereld.java
      inlevering2/
        ...
    nakijker2/
    ...
```

We veronderstellen hier dat het bronbestand `Dag Wrede Wereld.java` bestaat uit twee Java-klasse-definities, en dat `Hallo Wereld.java` er slechts eentje bevat. Merk op dat we alleen bestanden toevoegen, zodat we normalisatie in een later stadium opnieuw toe kunnen passen, bijvoorbeeld omdat we het algoritme hebben aangepast. Voor een goede historische vergelijking dienen we voor elk bronbestand hetzelfde algoritme te gebruiken. Zoals reeds gememoreerd worden voor elke Java-klasse twee genormaliseerde varianten bepaald: eentje waarbij de methoden, interne classes en attributen op kanonieke wijze worden geordend (`.nfs`) en eentje waarbij dat niet gebeurt (`.nf`).

Voor andere programmeertalen kan het bovenstaande enigszins verschillen, maar het resultaat is hetzelfde: voor elke inlevering vinden we na afloop een lijst van genormaliseerde bronbestanden,

herkenbaar aan een speciale extensie, eventueel gevolgd door een volgnummer. Zo is het voor PHP niet noodzakelijk om bronbestanden op te splitsen (en is het gebruik van volgnummers dus overbodig), maar dient men wel zowel `.inc` en `.php` bestanden in de vergelijking te betrekken, inclusief varianten zoals `.php3`.

4.3 Normalisatie van broncode

Het basisidee van normalisatie is dat details in de broncode die eenvoudig kunnen worden veranderd zonder het gedrag van het programma te veranderen zoveel mogelijk verwijderd worden. Alweer beperken we ons hier tot een beschrijving van het geval voor Java, maar hetzelfde principe is eenvoudig toe te passen op andere programmeertalen.

Om te voorkomen dat we een volledige parser voor de taal moeten definiëren, (dit zou teveel werk zijn), is er voor gekozen om niet verder te gaan dan een lexicale analyse[1] van het programma (door middel van reguliere expressies, zie Sectie 6). De lexicale structuur van de meeste programmeertalen is een orde van grootte minder ingewikkeld dan de context-vrije structuur. Een ander voordeel is dat veel programmeertalen een vergelijkbare lexicale structuur kunnen hebben, wat hergebruik bevordert. Ook is het eenvoudiger om de normalisator aan te passen aan taalveranderingen. Een ander punt is dat normalisatie ook werkt op programma's die niet syntactisch correct zijn, hoewel dat bij Marble niet echt van belang is.

Commentaar, overvloedige white-space en stringconstanten worden geheel verwijderd (dit laatste is een gevolg van een praktische overweging; meer hierover in Sectie 6). Karakterconstanten, hexadecimale getallen, getallen, identifiers worden respectievelijk vervangen door de karakters `L, H, N` en `X`. Symbolen zoals de accolades, de comma en plus, keywords zoals `class, while, private, super` en `throws`, maar ook veelgebruikte klassenamen (e.g., `String, System, File, Graphics, Socket, JMenu`) en belangrijke methodes (e.g., `toString, wait, notify`) worden ook bewaard. Mochten studenten gebruik maken van synoniemen voor deze veelgebruikte identifiers dan zal dit een nakijker direct opvallen. Door deze identifiers niet weg te reduceren, hopen we dat het aantal valse positieven (gevallen die uiteindelijk geen plagiaat zijn, maar structureel wel sterk op elkaar lijken) een stuk minder wordt.

Vooruitkijkend op het gebruik van `diff` worden tokens zoveel mogelijk op verschillende regels gezet. Zo wordt een de klasse-definitie

```
class Bliep extends Zwiep {
   String glob (int z) {
     int cnt = x;
     cnt = cnt*2;
   }
}
```

omgezet naar

```
CLASS
X
EXTENDS
X
{
STRING
X
(
INT
X
```

[1] Een lexicale analyse transformeert een rij karakters naar een rij tokens. Tokens zijn de kleinste betekenishebbende eenheden in een programma. Keywords, getallen (geheel of floating point), stringconstanten en operatoren zijn voorbeelden van tokens.

```
){
INT
X
=
X
;
X
=
X
*
N
;
}
}
```

Tot slot, operators en symbolen worden zoveel mogelijk met rust gelaten, en dus ook niet altijd gesplitst, zoals blijkt uit de regel `){` in bovenstaan voorbeeld. Zoals al eerder is genoemd worden Java-bestanden gesplitst als ze verscheidene klasse-definities bevatten. `import` declaraties worden simpelweg weggegooid. Het is te eenvoudig voor plagieerders om hier aanpassingen te maken.

Het geschikt maken voor Java 1.5 (met generieke instantiatie) was niet veel werk; de lexicale structuur is grotendeels hetzelfde. Er moesten enkele nieuwe keywords worden toegevoegd, `enum` en `assert`. Eerder werd al aangegeven dat ook sommige belangrijke klassennamen bewaard dienden te blijven. De gekozen methode om dit voor elkaar te krijgen werkt echter niet in het bijzijn van generieke instantiatie. Zo wordt `LinkedList<HashTable<T,Y>>` omgezet naar `X<X<X,X>>`, zelfs als gespecificeerd is dat `LinkedList` en `HashTable` bewaard zouden horen te blijven. De reden hiervoor is de niet-reguliere (geneste) structuur van generieke instantiatie. Het behouden van deze typen zou nogal wat werk vergen terwijl het niet te verwachten is dat het bewaren ervan heel veel verschil gaat maken.

4.4 Volgorde-onafhankelijkheid

Zoals al eerder gememoreerd genereert de normalisatiefase voor een gegeven Java-klasse paren genormaliseerde bestanden: eentje met extensie `.nf`, de andere met extensie `.nfs`. Wat tot nog toe werd beschreven gold voor beide versies. We zijn nu gereed om te bespreken waarin de `.nfs` versie afwijkt van de oudere `.nf` versie. Zoals in de introduktie werd aangegeven hebben veel moderne talen de voor plagiaatdetectie wat vervelende eigenschap dat sommige delen van bronbestanden straffeloos omgewisseld kunnen worden, zonder het gevaar de semantiek van het programma te veranderen. Zo mogen in een Java-bronbestand klasse definities onderling worden gewisseld (bedenk dat een Java-bestand in principe verschillende klasse-definities mag bevatten) en mogen in een klasse definitie attributen (data members), methodes (method members) en interne klassen (inner classes) naar believen worden omgewisseld.

De volgorde-onafhankelijkheid voor toplevel klasse-definities wordt opgevangen door het eerdergenoemde splitsen van bronbestanden zodat ieder genormaliseerd bestand slechts (de genormaliseerde code) van een enkele klasse bevat. Hoe omgegaan wordt met de tweede complicatie maakt het onderscheid tussen de `.nf` en `.nfs` versie. In het tweede geval wordt voor een gegeven klasse bepaald waar precies de definities van methoden, interne klassen en attributen beginnen en eindigen. De genormaliseerde versies van deze worden vervolgens op een vaste manier geordend, in de hoop dat vergelijkbare methodes en interne classes, dicht bij elkaar eindigen. Het is echter zo dat kleine verschillen grote gevolgen kunnen hebben. Vandaar dat ook de `.nf` versie bepaald wordt, zodat op basis van deze vergeleken kan worden.

Momenteel is vanuit het oogpunt van eenvoud gekozen om de volgorde puur lexicografisch te houden. Dat wil zeggen dat methoden allereerst gesorteerd worden op het aantal tokens dat ze bevatten, en als ze even lang zijn wordt de volgorde bepaald op basis van de standaard alfabetische volgorde (van de genormaliseerde code). In de Sectie 6 wordt ingegaan op hoe precies methoden en dergelijke uit klasse-definities geëxtraheerd kunnen worden zonder te parsen.

4.5 De detectiefase

Na toepassing van normalisatie bevat elke inlevering op topniveau een rijtje genormaliseerde bronbestanden, herkenbaar aan hun extensie (en eventueel nog gevolgd door een volgnummer). Het plagiaatdetectieprogramma kan nu worden aangeroepen, welke voor een gegeven extensie (in bovenstaande voorbeeld dienen we te kiezen tussen `.nf` en `.nfs`) alle bestanden onderling gaat vergelijken.

Zoals al eerder werd aangegeven dienen alle betrokken bestanden (uit de incarnatie `0506p1`) onderling te worden vergeleken, en te worden vergeleken met de genormaliseerde bestanden uit voorgaande incarnaties. Het doel van normalisatie is dat de mate van gelijkenis tussen twee oorspronkelijke programmbestanden bepaald kan worden door de genormaliseerde versies met behulp van het standaard Unix programma `diff` te vergelijken. Dit programma vergelijkt twee tekstbestanden op basis van een algoritme dat probeert de langste gemeenschappelijke substrings te identificeren, en dan de verschillen over en weer als resultaat oplevert [1]. Door het aantal verschillende regels te vergelijken met het totale aantal regels, krijgen we een maat die aangeeft in welke mate de genormaliseerde bestanden verschillen. Dit is eigenlijk heel simpel. gebruikmakend van de standaard Unix commando's `diff`, `wc` (voor het tellen van het aantal regels) en de Unix shell operator `|` (voor het doorgeven van de uitvoer van `diff` aan `wc`):

```
verschillenderegels = diff file1 file2 | wc -l
lengte1 = wc -l file1
lengte2 = wc -l file2
maat = 100 * verschillenderegels / (lengte1 + lengte2)
```

Kortom, we rekenen uit hoe lang de bestanden zijn, en we rekenen uit op hoeveel regels ze verschillen. Als de bestanden identiek zijn is het aantal verschillende regels nul, en levert dit een maat van nul op. Als de bestanden compleet verschillen, dan levert `diff` elke regel van beide op en is het aantal verschillende regels de som van de lengtes. In dat geval is `maat` dus 100. De interpretatie is dientengevolge dat lage waarden voor `maat` een grotere waarschijnlijkheid van plagiaat weergeven.

4.6 Uitvoer en interpretatie

De uitvoer van Marble is een shell script, meestal `suspects.X` genaamd, waarbij `X` de extensie is van de genormaliseerde bestanden. Dit script bestaat momenteel uit een rij aanroepen van een editor waarin dan twee *originele* bronbestanden worden geladen, en tegelijkertijd worden in de shell waarvandaan het script wordt aangeroepen drie getallen afgedrukt: de mate waarin de twee bestanden overeenkwamen (0 (identiek) tot 100 (totaal verschillend), en twee getallen die een maat zijn voor hoe groot de vergeleken bestanden zijn. Een cruciale eigenschap van dit script is dat de regels oplopend zijn gesorteerd op het eerste getal, dat wil zeggen dat de paren met de sterkste overeenkomst eerst komen.

Een voorbeeld van een dergelijke regel is

```
echo 00 59 59 && vimdiff \
   ../testsetzelf/jur/origineel/QSortObserver.java \
   ../historie/testset/versie9/QuickSortObserver.java
```

Het tool `vimdiff` is een standaard Unix-programma dat bij uitstek geschikt is voor het bekijken van de overeenkomsten tussen tekstbestanden. Het geeft door middel van kleurcodes aan in waar de overeenkomsten en verschillen zitten, en als je scrollt in het ene bestand, dan scrollt het andere vanzelf mee.

Marble is zeker niet perfect. Hoewel de resultaten al aangeven dat het vrij moeilijk is om programma's te verbouwen zodat ze niet meer als plagiaat worden herkend, maar toch nog als zinnige inleveringen worden gezien, zit er in sommige gevallen aardig wat kaf tussen het koren. Hiermee wordt bedoeld dat vergelijkingen hoog eindigen, om redenen anders dan plagiaat. Allereerst is het zo dat kleine Java-klassen sterk de neiging hebben op elkaar te lijken. Zo lijkt elke klasse die een attribuut bevat en de bijbehorende getter- en setter-methode structureel heel sterk op elke

andere klasse van dit type. Echter, dit is nu eenmaal de manier waarop men dit soort klassen implementeert, en is geen indicate van plagiaat. Dit is de reden dat Marble alleen programma's vergelijkt met een zeker minimum aan tokens. Ook wordt er in het uitvoerscript aangegeven bij elke match, hoeveel tokens beide bestanden bevatten. Zo kan de examinator eerst de gevallen bekijken waarbij die aantallen hoog zijn, omdat de kans van toevallige overeenkomsten vrij klein is.

Overeenkomsten kunnen ook een direct gevolg zijn van het feit dat sommige algoritmen op een zekere manier worden uitgedrukt. Zo zullen implementaties van stapels, rijen en buffers allemaal niet gek veel uiteenlopen. Bij het vak Gedistribueerd Programmeren wordt tijdens het college als voorbeeld een gesynchroniseerde bounded buffer gebruikt. Het is dan niet gek dat alle studenten die klasse gebruiken in hun praktikum. In het algemeen dient de examinator bij gevonden overeenkomsten voor zichzelf te beslissen in welke mate de overeenkomsten toelaatbaar zijn.

Dit gaat met behulp van het gegenereerde script zee eenvoudig. Door het script te starten worden de paren mogelijke plagiaatgevallen één voor één aan de examinator aangeboden. Op een zeker moment zal een punt worden bereikt waarop de inleveringen op elkaar gaan lijken terwijl er toch geen sprake is van plagiaat. Dit punt is meestal zeer eenvoudig te herkennen: meestal zijn er voor lage scores (dus hoge plagiaatkans) maar weinig vergelijkingen die een dergelijke score hebben (plagiaat vindt meestal plaats tussen paren inleveringen, en betreft zelden meer dan dat). Hoe hoger de scores hoe groter het aantal klassevergelijkingen met die score, en dit aantal blijkt meestal op zeker punt explosief toe te nemen. Dit is het moment voor de examinator om te stoppen met de handmatige inspectie. De overeenkomsten zijn op dat moment meestal het gevolg van het feit dat de studenten in kwestie dezelfde opdracht maakten. Overigens ligt deze drempelscore voor elke opdracht weer anders. De auteur heeft opdrachten gezien waarbij deze grens net onder de 70 zat, maar meestal ligt de drempel zo rond de 40. Het is belangrijk zich niet dood te staren op een zekere waarde, maar in de gaten houden voor de score hoe de frequentie zich ontwikkeld.

5 Een experiment

In het studiejaar 2003/2004 deed ik samen met twee studentassistenten, Arie Middelkoop en Arjen Swart, een test. Ik gaf hen een uitwerking van een opdracht die hen bekend was, maar was gemaakt door studenten in het jaar 2001/2002. De opdracht was om de inlevering zo te verbouwen dat mijn plagiaatdetector het niet als plagiaat zou herkennen. Zo maakte Arjen Swart een serie van negen verbouwingen, die in steeds sterkere mate afweken van het origineel. De modificaties die hij hierbij achtereenvolgens maakte laten zich als volgt samenvatten:

Versie	gemaakte aanpassing	score
1	commentaar vertaald of verwijderd, layout veranderd	00
2	declaraties van methoden van plaats veranderd	00
3	sommige attribuutdeclaraties verwisseld	04
4	aanroepen van sommige GUI-methoden omgedraaid	13
5	imports veranderd	13
6	GUI tekst en kleuren van de GUI veranderd	14
7	namen van sommige variabelen en klassen vertaald	14
8	enkele expressies in de code herschreven	14
9	get/setmethode ge-inlined	14

Merk op dat voor een gegeven versie ook alle aanpassingen van de voorgaande versies zijn uitgevoerd.

Bij wijze van test is de plagiaatdetector gedraaid om het origineel te vergelijken met alle negen versies *en* een hele jaargang van inzendingen uit het jaar 2005/2006 (de controlegroep, vijftig inleveringen in totaal). De resultaten zijn uitermate bemoedigend.

Bekijken we eerst de resultaten van de variant waarbij de volgorde van methoden en dergelijke niet verandert, dan is 49 de laagste score voor `maat` die werd gevonden voor een vergelijking met

een klasse uit de controlegroep. Degene die wordt gevonden is dan ook nog eens voor een hele kleine klasse. Voor de bovenbeschreven versies is het verhaal heel anders: per versie is in bovenstaande tabel aangegeven wat de laagste score is waarbij een (significant grote) klasse van de versie overeenstemt met een klasse uit het origineel (in alle gevallen blijkt dit ook de corresponderende klasse te zijn). De getallen zijn allemaal een heel stuk lager dan 49. Tot nog toe is gebleken dat overeenkomsten tussen significant grote Java-klassen met een score van onder de 30 op zijn minst wijzen op verregaande samenwerking. Bij sommige klassen, vooral als ze zeer groot zijn, kan het zelfs nog zijn dat er sprake van plagiaat is bij scores die een stuk hoger liggen, soms zo hoog als zeventig.

Voor het geval dat de methoden, interne klassen en attributen worden gesorteerd op de eerder beschreven manier zijn de resultaten nog een stuk beter: de scores die worden behaald voor, alweer significant grote, Java-klassen zijn nu op zijn hoogst 02 (tegen 14 voor de ongesorteerde versie). Het effect van sortering wordt nog eens bevestigd door het relatief grote verschil in score tussen versie 3 en versie 4 in bovenstaande tabel, welke verdwenen is in de scores voor de gesorteerde versie. Opvallend genoeg zijn de scores voor vergelijkingen met de controlegroep bijna niet veranderd: 48 is nu de laagste score.

6 Implementatie

In deze sectie ga ik kort in op hoe Marble is geïmplementeerd. Marble is slechts een incidenteel zijpad in de dagelijkse bezigheden en constructie van en onderhoud aan het programma dienden niet al te tijdrovend te zijn. Door de keuze om de analyse te baseren op lexicale aspecten van de programma's, lag het voor de hand een taal te kiezen waarbij het gebruik van reguliere expressies en manipulatie van tekstbestanden goed en efficiënt is geregeld. Ondanks de nodige nadelen aan de taal zelf is er daarom voor gekozen om de taal Perl te gebruiken. Normalisatie is ondergebracht in twee Perl-programma's: eentje die de normalisatie uitvoert op een Java-bronbestand (`normalize.pl`, 339 regels code en uitgebreid commentaar), en een ander dat ervoor zorgt dat normalisatie wordt toegepast op de juiste bestanden in de juist subdirectories (`normalizeall.pl`, 67 regels code en commentaar). Plagiaatdetectie wordt uitgevoerd door een vrij simpel script (`detectfraud.pl`, 255 regels code en commentaar) dat de subdirectories afspeurt naar de juiste bestanden om onderling te vergelijken, `diff` aan te roepen en de uitvoer in een script beschikbaar te maken.

Het `normalize.pl` programma is van de drie scripts het interessants en technisch gezien het ingewikkeldst. Daarom gaan we er wat dieper op in. De normalisatie valt uiteen in drie delen: de lexicale pre-processing, het annoteren van de accolades, en het verder transformeren van de afzonderlijke klassen.

Lexicale pre-processing Tijdens de lexicale pre-processing wordt het programma het meest verbouwd. Uiteindelijk worden alle identifiers omgezet naar het karakter X. We weten dus nog wel dat er een identifier stond, maar niet meer hoe hij heette. Dit is logisch omdat we onafhankelijk willen zijn van de namen die men koos voor de identifiers. Echter, keywords lijken heel erg op identifiers en die willen we nu juist niet wegtransformeren. Hetzelfde geldt in minder mate voor namen van belangrijk methoden en speciale constanten zoals `null`, `true` en `false`; deze laatste drie zijn geen keywords in Java. Het programma bevat daarom een aantal eenvoudig uitbreidbare arrays met de identifiers die uiteindelijk bewaard dienen te blijven. Deze worden allemaal verzameld in een array, `untouchables`, en op volgorde van dalende lengte gesorteerd.

De lexicale preprocessing gaat dan als volgt: eerst worden alle ge-escapete (enkele en dubbele) quotes en backslashes uit de stringconstanten verwijderd. Vervolgens worden met een enkele, vrij complexe reguliere expressie tegelijk alle stringconstanten, alle single-line comments en alle multi-line comments verwijderd. Het is essentieeel dat dit tegelijkertijd gebeurt. Het is namelijk mogelijk dat een stringconstante iets bevat dat op commentaar lijkt. Echter, dit mag niet als commentaar worden gezien. Andersom kan in commentaar iets staan dat op een stringconstante lijkt. De regel is dat van links naar rechts in een programma gelezen dient te worden, gekeken moet worden welke

van de drie (stringconstante, single-line comments of multi-line comment) als eerste voorkomt, en dan deze eerst afgehandeld dient te worden alvorens het proces voort te zetten. Karakteristiek voor de uitdrukkingskracht is dat met een enkele reguliere expressie kan:

```
$prog =~ s/(\/\*(.|\n)*?\*\/)|(\/\/.*?\n)|(".*?")|('.*?')/ /g;
```

Vervolgens wordt al het overvloedige whitespace uit het programma verwijderd. Het doel is hierbij alleen het whitespace te bewaren dat betekenisvol is. Zo mag de spatie tussen twee identifiers nooit worden verwijderd, maar is de spatie tussen + en een identifier niet nodig, en zal dus worden verwijderd.

Vervolgens wordt er aanvang gemaakt met te zorgen dat alle belangrijke identifiers en keywords zullen worden bewaard. De grote truc is hier dat we het hele programma eerst omzetten naar kleine letters en vervolgens stukje bij beetje gaan transformeren naar hoofdletters. Alles wat in hoofdletters wordt omgezet zal *nooit* meer matchen met welke reguliere expressie dan ook. Het bewaren van de keywords en de speciale constanten, methoden en klassenamen is dan ook voornamelijk een kwestie van de juiste identifiers omzetten naar hoofdletters. Uiteraard moeten we er wel zorg voor dragen dat als we `for` omzetten naar `FOR` we niet ook `fork` naar `FORk` omzetten (enzovoorts). Gelukkig kennen de reguliere expressies in Perl hiervoor de nodige hulpmiddelen.

Het bewaren van de speciale klassenamen is een stukje moeilijker, omdat klassenamen ook als gewone variabelenamen voor kunnen komen. De regel die we hier gebruiken, is dat de enige identifiers die na whitespace gevolgd kunnen worden door nog een identifier, de namen van typen zijn. Nu zijn de basistypen zoals `int` al reserved keywords en dus al vanzelf bewaard, en zorgt deze tweede conversie ervoor dat klassenamen (vaak) worden bewaard. Zoals al eerder is opgemerkt werkt dit niet voor klassenamen waaraan type-parameters worden meegegeven, maar de invloed hiervan op de vergelijking is te verwaarlozen. Daarna wordt alle informatie over imports weggegooid.

We vervolgen met het wegwerken van nog enkele lexicale klassen. Allereerst is het de beurt aan de hexadecimale getallen (deze beginnen met `0x`). Vervolgens worden integer en floating point getallen omgezet naar het karakter N. De pre-processing eindigt met het splitsen van operators en identifiers, zodat `X=N+N` gewoon `X = N + N` wordt.

Annotatie van accolades De volgende grote truc is het annoteren van accolades. Reguliere expressies zijn niet in staat om voor een gegeven accolade de bijbehorende sluitaccolade te vinden. Hier maken we dan ook gebruik van een while loop in Perl waarin we met behulp van reguliere expressies elke openings- en sluitaccolade verzien van een cijfer. De accolades op het hoogste niveau (dat zijn bij Java degene die de klassen demarkeren krijgen het cijfer 0, de accolades voor de bodies van methodes en interne klassen krijgen het cijfer 1, enzovoorts. Het basisidee is om te zoeken naar de eerstvolgende sluit- of openingsaccolade. Is het een openingsaccolade dan annoteren we die met de huidige diepte en hogen we de huidige diepte eentje op. Is het een sluitaccolade, dan lagen we de diepte eentje af en annoteren we de sluitaccolade. Het splitsen van een Java-bronbestand op basis van de klassen die erin worden gedefinieerd is nu heel eenvoudig: alle klassen lopen (zo ongeveer) tussen achtereenvolgende accolades geannoteerd met diepte nul.

Verdere transformaties van de klassen Voor de variant waarbij we de volgorde van methode en dergelijke onveranderd laten zijn we bijna klaar: we vervangen alle overgebleven identifiers (in kleine letters!) door het karakter `X` en verwijderen de annotaties bij de accolades. Tokens worden zoveel mogelijk op verschillende regels gezet om het leven door `diff` makkelijk te maken. Dat wordt zo een stuk minder gevoelig voor hoe de code over regels is verspreid.

In het geval dat de methoden en dergelijke wel gesorteerd dienen te worden, moeten deze allereerst uit de klasse-definitie worden gepeuterd. Dit kan echter redelijk eenvoudig, omdat de betreffende accolades geannoteerd zijn met het cijfer 1. Alle methodes, attributen en interne klassen worden allereerst verzameld in drie afzonderlijke arrays. Deze worden vervolgens op lexicografische wijze geordend, en daarna worden eerst de methode, dan de interne klassen en uiteindelijk de attributen in de uitvoer gezet. Deze worden gescheiden door speciale teksten om `diff` het leven

nog iets makkelijker te maken. Uiteraard moeten ook hier de identifiers vervangen worden door `X`, en worden de annotaties van de accolades weer verwijderd.

7 Verder met Marble

Marble is op verschillende manieren uit te breiden en robuuster te maken. Allereerst ontbreekt het momenteel aan een goede gebruikersinterface, waardoor er door leken nog niet op eenvoudige wijze gebruik van Marble kan worden gemaakt. Ook zouden er verschillende manieren moeten zijn om de genereerde scripts samen te vatten. Dit zou deels bereikt kunnen worden door een directe integratie met Submit.

Een technisch wat ingewikkelder opgave is het omgaan met sjablonen (templates). Bij sommige opdrachten dienen studenten een aangeleverd bronbestand aan te passen. Omdat een groot deel van de code voor iedere student hetzelfde is zullen al deze programma's hoog eindigen in de ordening. Het zou beter zijn de vergelijking toe te passen op die delen die aangepast dienen te worden. De implementatie hiervan is niet heel ingewikkeld, maar is nog niet gebeurd. Het is wel waarschijnlijk dat dit het gebruik van Marble nogal compliceert.

De ordening van methodes en interne klassen en dergelijke is nu vrij simpel geregeld: de genormaliseerde versies van de afzonderlijke methoden worden simpelweg gesorteerd op lengte, en vervolgens op alfabet. Door het toevoegen van zogenaamde debug-statements zouden studenten hier echter vrij eenvoudig om heen kunnen werken. Daarom lijkt een benadering die eerst uitgaat van het return type en het aantal parameters een logischere. Dit is echter nooit uitgewerkt, vooral omdat het lastig is dit door middel van reguliere expressies te doen.

Het is zeker wenselijk de verschillende versies van Marble (voor de verschillende programmeertalen) te integreren. Deze zou bijvoorbeeld uit kunnen gaan van één of andere codering van de lexicale structuur van de taal, zoals gebruikt bij compilerbouwtools. Het is echter waarschijnlijk dat er altijd wel sprake zijn van allerlei ad-hoc constructies per taal, bijvoorbeeld omdat in de ene taal vrijelijk methode- en functie-definities omgedraaid kunnen worden en in een andere weer niet.

Hoewel Marble voldoet in de zin dat het de nodige plagiaatgevallen heeft gevonden en ook overeind bleef tijdens een vrij uitgebreid experiment, is een degelijke studie om haar kwaliteiten in vergelijken met andere tools van deze soort, nooit ondernomen.

De vraag die bij de lezer gerezen kan zijn, is of het gebruik van Marble ook van toepassing is op scripties, dat wil zeggen, documenten geschreven in natuurlijke taal. In principe kan dit wel. Normalisatie kan heel simpel zijn, en zou enkel alle woorden op een afzonderlijke regel zetten, en dan dezelfde detectiemethode te gebruiken (dat wil zeggen, simpele `diff`s). Een groot verschil is echter dat juist bij scripties er de noodzaak is om niet alleen scripties onderling te vergelijken, en met scripties uit vroeger jaren, maar ook deels met bronnen op het Web. Dit verlangt een heel andere manier van werken, en is iets waar de auteur zover de tijd dat toelaat zijn aandacht aan besteed. Een programmeerbare interface tot een search-engine zoals Google is hiervoor essentieel.

Een voordeel, ten opzichte van talen als Java, is dat het bij geschreven tekst veel moeilijker is voor studenten om stukken tekst om te wisselen. Wat een (grote) extra uitdaging is, is om te detecteren dat een student een bekende bron letterlijk heeft vertaald.

8 Tot slot

Marble is ontstaan uit de wens van de auteur om op wat meer gestructureerde wijze na te gaan in welke mate er sprake van plagiaat was bij programmeerpraktika. Het programma heeft de nodige gevallen van plagiaat weten te ontdekken, zoals achteraf ook is geverifieerd aan de hand van de reacties van de betrokken studenten. De uitvoer van Marble dient altijd met de hand worden doorgelopen, tot op het punt dat de overeenkomsten het gevolg zijn van het feit dat in feite dezelfde programmeeropdracht wordt gemaakt. Marble is door middel van een experiment gevalideerd. In dit artikel beschreven we verder hoe het tool is opgebouwd en welke technieken zijn gebruikt om er een efficiënte implementatie van te krijgen, en welke keuzes zijn gemaakt om de implementatie met zo min mogelijk inspanning voor elkaar te krijgen. Tot slot gaven we een

overzicht van de mogelijke uitbreidingen op Marble welke de drempel tot het gebruik ervan sterk kunnen verlagen. Toepassing ervan op grotere schaal ligt dan binnen bereik.

Bedankt De auteur bedankt Arie Middelkoop en vooral Arjen Swart voor hun actieve deelname aan het valideren van de fraude-checker, en verder Dick Grune, Nikè van Vugt, Jeroen Fokker, Arthur van Leeuwen en vooral Lex Bijlsma voor hulp bij het schrijven van deze paper.

References

1. P. Eggert, D. MacKenzie, and R. Stallman. *Comparing and Merging Files with GNU Diff and Patch.* Network Theory Limited, Jan 2003. http://www.gnu.org/software/diffutils/manual/.
2. Departement Informatica. Docentenhandleiding. http://www.cs.uu.nl/bama/Docenten/Docentenhandleiding.pdf.
3. Departement Informatica. Onderwijs- en examenregelingen 2006. http://www.cs.uu.nl/education/regels/.
4. Departement Informatica. Submit systeem. http://www.cs.uu.nl/docs/submit/.