Testondersteuning in frameworks voor webapplicaties

Mirko van Ede(9902236)

Nijmeegs Instituut voor Informatica en Informatiekunde

Radboud Universiteit Nijmegen

Versie: 1.1, 11 februari 2009

Omvang: 3 ects

Samenvatting

Frameworks zijn een veelgebruikte basis voor het bouwen van een *webapplicatie*. Omdat deze applicaties ook steeds bedrijfskritischer worden, is het van belang dat zij goed getest kunnen worden. Daarom is de mate waarin een *framework* daar ondersteuning voor biedt een belangrijk gegeven.

Omdat er zoveel *webframeworks* zijn, is in dit onderzoek gekeken naar slechts drie, populaire, *frameworks*: *CakePHP*, *Django* en *Ruby on Rails*. Deze drie raamwerken zijn geanalyseerd op de mate van ondersteuning die zij bieden op het gebied van testen. De soorten tests die in deze analyse zijn onderzocht zijn een verzameling van de meest essentiële soorten *black*- of *graybox tests* die uitgevoerd kunnen worden.

Alle *frameworks* bieden in ruime mate ondersteuning voor dezelfde soorten tests. Zowel *model-based testing als capture-and-playback tests* worden door geen enkel *framework* ondersteunt. De grootste verschillen treden op bij de mogelijkheden tot analyseren van de output die naar de browser gaat. Daarbij hebben *CakePHP* en *Ruby on Rails* een voorsprong op *Django*.

Inhoudsopgave

H1. Inleiding

Inleiding

Om inzicht te geven waarom dit onderzoek wordt uitgevoerd en wat het belang ervan is, beschrijft dit hoofdstuk het projectkader van deze bachelorthesis.

Testen is een belangrijk onderdeel van alle moderne software ontwikkelingsmethodes en het belang ervan wordt door iedereen wordt onderschreven, hoewel het echter ook een onderdeel is dat er vaak, vanwege tijd- of geldgebrek als eerste bij inschiet.

Een ander bekend fenomeen is dat webapplicaties, applicaties die via het internet gebruikt worden, steeds groter, complexer en ook bedrijfskritischer worden. Dat betekent dus dat het van steeds groter belang wordt om deze applicaties op grondige en gestructureerde wijze te testen.

Omdat bij het bouwen van deze webapplicaties, zeker wanneer het grote en complexe applicaties betreft, bijna altijd *frameworks* worden ingezet, is het dus van belang dat deze *frameworks* ondersteuning bieden voor het testen van de code en programmatuur die hierin gemaakt worden.

Dit onderzoek zal daarom kijken naar de testondersteuning die dergelijke *frameworks* voor webapplicaties bieden.

Documentstructuur

In het eerste hoofdstuk van deze scriptie zal het probleem uit de doeken worden gedaan en aangegeven worden wat de relevantie van het probleem is. Daaruit zal de onderzoeksvraag volgen en de deelvragen die nodig zijn om deze onderzoeksvraag te beantwoorden.

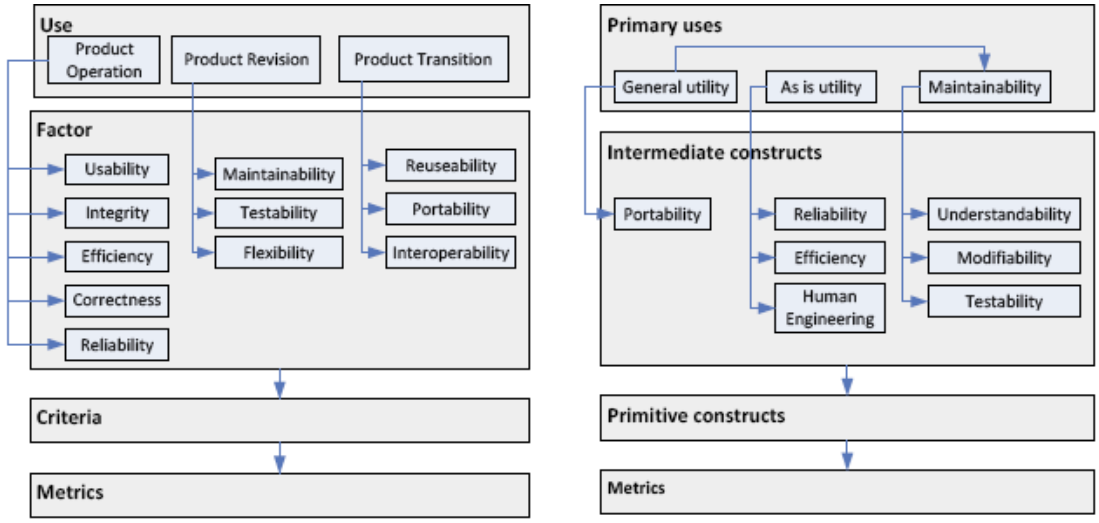
De hoofdstukken 3,4 en 5 zullen vervolgens de resultaten tonen die resulteren uit het onderzoek naar de drie opgestelde deelvragen. Daarnaast zal per hoofdstuk een kleine conclusie over de deelvraag worden getrokken, alsmede een stukje reflectie op de resultaten.

Het laatste hoofdstuk bevat vervolgens de conclusie van dit onderzoek, waarmee de onderzoeksvraag beantwoord is. Ook zal in dit hoofdstuk kort worden aangegeven of en welk vervolgonderzoek er eventueel gedaan kan worden.

H2. Probleemstelling en relevantie

Kwaliteit van software

Niet alleen is *testen* een onderdeel van alle recente softeware-ontwikkelingsmethodes, in speciaal voor software opgestelde kwaliteitsmodellen neemt het een belangrijke plaats in. De modellen van McCall (Cavano & McCall, 1978), origineel opgesteld voor de US Airforce en bedoeld om te bemiddelen tussen ontwikkelaars en gebruikers , en Boehm (Boehm, Brown, & Kaspar, 1978) zijn algemeen geaccepteerd en zien respectievelijk er als volgt uit (Woolderink, 2007):



Afbeelding 1: Kwaliteitsmodel van McCall (links) en Boehm

Zoals te zien, ruimen beide modellen een aparte plek in voor de *testability* van de software en onderkenden daarmee, al lang geleden, het belang van *testen* voor de kwaliteit van software. Ook het "modernere", de eerste versie stamt uit 1991, de meest recente uit 2001, standaard kwaliteitsmodel voor software, ISO 9126 (ISO, 2001), dat is gebaseerd op de modellen van McCall en Boehm en aangepast aan de behoeftes en ervaringen met die modellen, laat zien dat het *testen* van software een belangrijke plaats inneemt om te komen tot een kwaliteitsproduct.

Manieren van testen

Er zijn veel aspecten waarop software getest kan worden: hoe snel is het systeem (*performance*), hoe intensief kan het gebruikt worden (*stress*), hoe gaat het systeem om met onverwacht gedrag zoals verkeerde input van gebruikers (*robustness*), hoe veilig is het systeem (*security*), etcetera. Waar we in deze scriptie naar zullen kijken, is het functioneel testen, waarbij er getest wordt of het programma voldoet aan de functionele eisen die er zijn opgesteld, ook wel *conformance testing* genoemd.

Je kunt onderscheid maken tussen verschillende manieren van testen. Allereerst is er het verschil tussen *statisch* en *dynamisch testen*. *Statisch testen* wordt gedaan zonder de code daadwerkelijk uit te voeren, zoals bijvoorbeeld een compiler doet. Bij *dynamische testen* wordt de programmacode wel uitgevoerd. Tijdens deze scriptie zullen we alleen kijken naar de ondersteuning voor *dynamisch testen.*

Een ander veelgemaakt onderscheid is het verschil tussen *white box* en *black box* testen. Bij *black box testen* wordt het te testen object gezien als een zwarte doos, waar iets in gaat en weer uitkomt. Bij *white box testen* heeft men de beschikking over de programmacode en gebruikt men die om te testen. Er zijn natuurlijk vele combinaties mogelijk, *grey box testen* genoemd, bijvoorbeeld wanneer men weet uit welke modules een systeem is opgebouwd, maar de modules wel een black box zijn.

Samenvattend zullen we het in deze scriptie alleen hebben over *dynamisch black- en greybox testen.*

Frameworks voor webapplicaties

Een ander bekend fenomeen is dat er tegenwoordig meer en meer, vaak bedrijfskritische, applicaties worden ontwikkeld die volledig *webbased* zijn. Populaire talen voor dit soort applicaties zijn onder andere Java, Microsoft's .NET talen, Python, Ruby en PHP.

Vanwege de grote verscheidenheid aan talen beschikbaar zijn voor webapplicaties, zullen we de scope beperken tot de open source talen PHP, Ruby, en eventueel Python en Java.

Voor al deze talen zijn bestaan *frameworks*, die het de ontwikkelaar makkelijker maken, zorgen voor gestructureerde, gelaagde code en daarom veel gebruikt worden bij het ontwikkelen van webapplicaties. Bekende *frameworks* zijn bijvoorbeeld Ruby on Rails voor Ruby, Django voor Python, Struts en Spring voor Java en Symphony,, het Zend framework en CakePHP voor PHP.

Vanwege de beperkte omvang van dit onderzoek en het grote aantal *frameworks* dat er voor sommige talen beschikbaar is, zullen we twee raamwerken bekijken. Allereerst is dat CakePHP, zoals de naam al doet vermoeden is dit een PHP *framework* en we kiezen deze omdat PHP een zeer populaire taal is voor webapplicaties en CakePHP daarbinnen één van de grotere *frameworks* is. Het tweede *framework* dat we zullen bekijken is *Ruby on Rails*, voor de programmeertaal Ruby en tevens erg populair.

Omdat webapplicaties groter, bedrijfkritisch en complexer worden, is het van groot belang dat deze code van hoge kwaliteit en dus een goed testbaar is. Veel van deze *frameworks* hebben in meer of mindere mate ondersteuning ingebouwd voor het ontwerpen en uitvoeren van tests. De vraag is of deze specifieke testondersteuning zorgt voor goede *tests* van software die geschreven is in deze *frameworks*. De concrete vraagstelling die we kunnen formuleren is:

Welke ondersteuning voor testen bieden de veelgebruikte frameworks voor webapplicaties?

Onderzoeksvraag en deelvragen

Om de onderzoeksvraag te kunnen beantwoorden, is de volgende kennis nodig:

A. Wat voor type *tests* kunnen we onderscheiden binnen de aangegeven criteria?
B. Hoe zijn de geselecteerde *frameworks* opgebouwd?
C. Welke van de soorten *tests* worden door de geselecteerde *frameworks* ondersteund?

Onderzoek naar het antwoord op deelvraag A zal gedaan worden door middel van literatuuronderzoek.

Het antwoord op deelvraag B zal geschieden door middel van literatuuronderzoek.

Het antwoord op deelvraag C zal worden deels gezocht moeten worden in documentatie en deels door het maken van voorbeelden die laten zien hoe de eisen en randvoorwaarden, zoals gevonden bij deelvraag A, geïmplementeerd kunnen worden.

Het combineren van de antwoorden op de deelvragen verschaft ons vervolgens een inzicht in de mate van testondersteuning van deze *frameworks* .

H3. Type software tests

Inleiding

Dit hoofdstuk beschrijft een inventarisatie van mogelijke soorten *tests* die de *frameworks* zouden kunnen ondersteunen, binnen de kaders die gesteld zijn in hoofdstuk 2. Van elke type test zal een korte omschrijving worden gegeven.

Deze inventarisatie is tot stand gekomen door middel van een literatuuronderzoek en het nagaan van best practices op het gebied van software ontwikkeling met behulp van webframeworks. De lijst met soorten *tests* die dit hoofdstuk oplevert, zal worden gebruikt voor het vergelijken van de testondersteuning van de geselecteerde *frameworks* .

Inventarisatie

Handmatig testen

Het systeem ondersteunt het handmatig uitvoeren van *tests*, bijvoorbeeld via de commandline.

Capture and play back

Het test systeem onthoud de handmatig uitgevoerde tests en kan die op een later herhalen.

Code coverage analyse

Alhoewel dit geen *blackbox* test is, is de *code coverage analyse* een veelgebruikte test en tevens een indicatie of de opgestelde *testsuite* de code dekt.

Doctests

Testcases die in de documentatie worden geschreven van een methode of functie. Deze kunnen dan geautomatiseerd allemaal uitgevoerd worden. Hiermee kan dus uitsluitend de werking van de functie op zich worden getest, niet de interactie met andere methodes binnen de unit.

Geautomatiseerde unittests

Unittests worden gebaseerd op van tevoren vastgestelde input, eventueel ondersteunt door zogenaamde *fixtures.* Daarna kunnen ze geautomatiseerd, uitgevoerd worden. Unittests worden uitgevoerd op het niveau van het *model.*

Verder is het interessant welke opties er geboden worden om deze *unittests* in te richten:

- *Fixtures*; herbruikare definities van de testdata, waardoor deze door meerdere test gebruikt kan worden.
- *Mock objects*; het simuleren van objecten waar het te testen object van afhankelijk is.

Geautomatiseerde functionele tests

Ook functionele tests maken meestal gebruik van *fixtures*. In tegenstelling tot de *unittest*, wordt hier echter niet slechts één *model* getest, maar een compleet *request* uitgevoerd om te zien of dat het verwachte resultaat oplevert.

Binnen de functionele tests, zullen we kijken naar de volgende opties:

- *Database wijzigingen*; is het mogelijk om na het uitvoeren van een functionele test te analyseren of er wijzigingen in de database zijn aangebracht?
- *HTTP Headers*; is het mogelijk de HTPP headers te analyseren?
- *View variabelen/objecten*; omdat we het hier hebben over tests op controller-niveau, is het mogelijk om te zien welke data er aan de view doorgegeven wordt?
- *Authenticatie*; kan de test zich authenticeren als een bepaalde gebruiker tijdens het uitvoeren van de test?

Web tests

Omdat we in dit onderzoek specifiek kijken naar *frameworks* voor *webapplicaties*, is het ook interessant om te zien of er specifieke ondersteuning wordt geboden voor dit platform. We zullen de volgende opties onderscheiden:

- GET requests; het simuleren van HTTP GET requests.
- POST requests; het simuleren van HTTP POST requests, onder andere bruikbaar voor het testen van formulieren.
- *Bestandsuploads*: simuleren van bestandsuploads.
- *HTTP Headers*; is het mogelijk de HTPP headers te analyseren?
- *HTTP redirects*: kunnen HTTP-redirects gevolgd worden?
- *View variabelen/objecten*; is het mogelijk om te zien welke data er van de controller aan de view is doorgegeven?
- *Output test*: controleren of de HTML output is, zoals verwacht werd, eventueel gecombineerd met een parser om te zien of de output geldige (X)HTML oplevert.
- *Cookie ondersteuning*: het correct afhandelen van de cookies zoals die door de applicatie gebruikt worden.

Interface/in browser tests

Geautomatiseerd testen van de interface middels de browser, volgens het *capture and playback* principe, zodat de tests herhaald kunnen worden. Daarbij kunnen we kijken of deze tests browseronafhankelijk zijn, iets wat met name bij de interface van een webapplicatie nogal een struikelpunt kan zijn.

Model-based testing

Het volledig kunnen specificeren van het gedrag van de webapplicatie, zodat het testsysteem automatisch zijn eigen testcases kan afleiden om de applicatie volledig te testen.

Conclusie

Dit hoofdstuk heeft een lijst opgeleverd met basisopties die een *framework* kan ondersteunen op het gebied van het testen van de code en beantwoord daarmee deelvraag A:

> "Wat voor type *tests* kunnen we onderscheiden binnen de aangegeven criteria?"

Deze opties voldoen allemaal aan de eisen zoals we die eraan gesteld hebben in hoofdstuk 2, namelijk dat het *dynamisch, grey- en blackbox testen* zijn. Er zijn nog wel meer opties te bedenken die een *framework* ook kan ondersteunen en het is een subjectief oordeel of een bepaalde optie een standaardoptie behoort te zijn. Ik denk echter dat deze lijst vrij compleet is, en het ontbreken van bepaalde opties in deze lijst zal ook geen fundamenteel verschil maken voor het vervolg van het onderzoek. Eventueel vervolgonderzoek kan meer opties meenemen in het eindoordeel.

H4. Frameworks voor webapplicaties

Inleiding

Dit hoofdstuk beschrijft de selectie van *frameworks* die we zullen gebruiken in de rest van dit onderzoek. Van de geselecteerde raamwerken zal een korte beschrijving worden gegeven, waarbij er gekeken wordt naar de algemene werking van het *framework* aangevuld met een kort overzicht van de features die het *framework* biedt. Hiermee wordt zal deelvraag B beantwoord worden:

> "Hoe zijn de geselecteerde *frameworks* opgebouwd?"

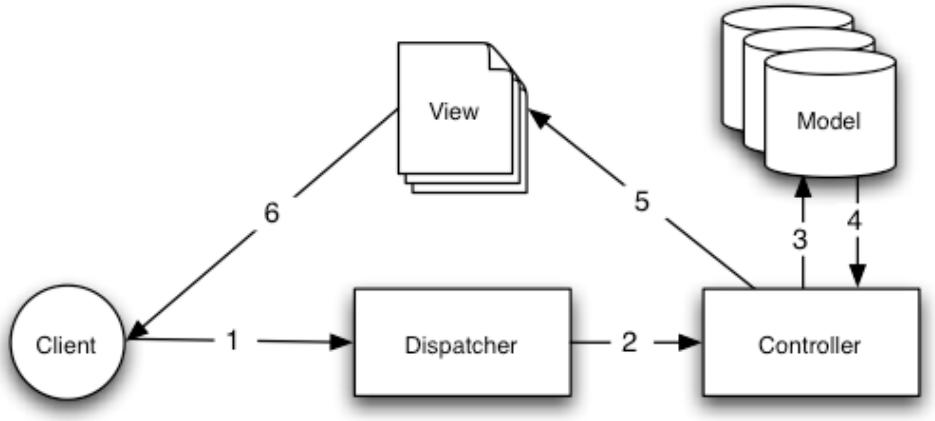
CakePHP

CakePHP is een object-georiënteerd, open source rapid development *framework* (Cake, 2001) geschreven in en voor de geïnterpreteerde, imperatieve programmeertaal *PHP* (PHPGroup). *CakePHP* is gestoeld op het design pattern Model-View-Controller (kortweg *MVC*, dat geïntroduceerd werd door Smalltalk) en dwingt de ontwikkelaars om hun applicatie op te delen in drie delen. Deze drie delen hebben elk hun eigen verantwoordelijkheden:

Het *model*; representeert de data waarop de applicatie werkt. Tussen de models kunnen verschillende type relaties bestaan. Typisch wordt deze data uiteindelijk opgeslagen in een database. Het *model* bevat de businesslogica van de applicatie en opslag en bewerken van de data verloopt via de *models*.

De *view*; verzorgt de presentatie van de data. Voor webapplicaties zijn dat meestal webpagina's, maar aan andere representaties, zoals XML of een PDF-document, valt ook te denken. De *view* doet geen bewerkingen op de data.

De *controller*, verwerkt de binnenkomende verzoeken (events). Deze zijn meestal afkomstig van handelingen van de gebruikers, maar zouden even zo goed op andere manieren (bijvoorbeeld via een SOAP interface) binnen kunnen komen. De controller voert ook applicatie-logica uit, zoals bijvoorbeeld de autorisatie. Een typisch MVC-request ziet er als volg uit (Cake, 2001):



Afbeelding 2: MVC request

1. Vanaf de client komt er een verzoek
2. De *CakePHP* dispatcher bekijkt het verzoek en stuurt het door naar de correcte *controller*.
3. De *controller* voert zijn taken uit, bijvoorbeeld authentificatie en autorisatie van de gebruiker die het verzoek doet. Daarna stuurt hij opdrachten (bijvoorbeeld verwijderen of aanpassen data) of verzoeken naar de benodigde *models*.
4. Het *model* geeft data terug
5. De *controller* stuurt alle data naar de view
6. De *view* gebruikt deze data om de output op te bouwen en stuurt de inhoud terug naar de client.

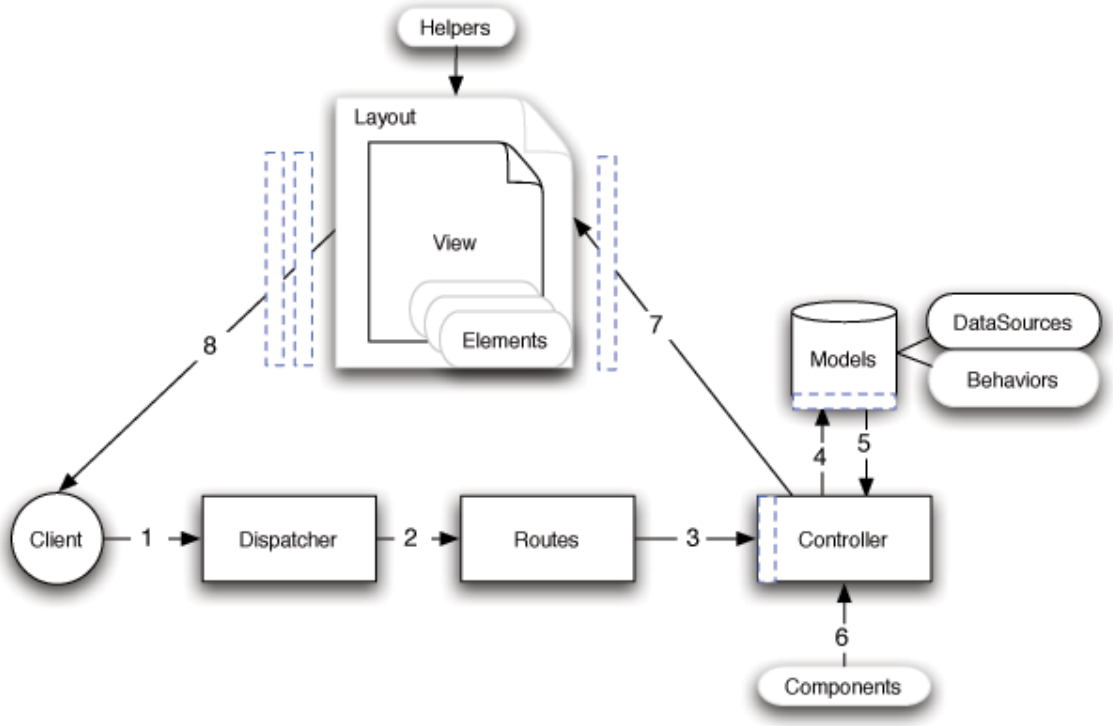
Features

Hieronder staat een lijst met features die het *CakePHP framework* biedt, zoals beschreven staat op de website van het project (Cake, 2001):

1. Compatibel met PHP4 en PHP5
2. Geïntegreerde CRUD (create, read, update en delete) voor database interactie
3. Applicatie scaffolding, het automatisch verwerken van standaardverzoeken, op basis van de gedefinieerde database tabellen. Bedoelt als tijdelijke, snelle start.
4. Code genereren, het automatisch genereren van de basis van de applicatie, op basis van de gedefinieerde database tabellen.
5. Vriendelijke (mens en zoekmachine) URL's en aangepaste routes
6. Ingebouwde validatie, validatie van ingevoerde gegevens, op basis van het model.
7. View Helpers voor AJAX, JavaScript, HTML-formulieren, zorgt ervoor dat niet alle HTML code handmatig uitgewerkt hoeft te worden.
8. Componenten, gedeelde uitbreidingen op de *controllers*, Standaard componenten als Email, Cookie, Security, Session, en Request .

9. Behaviors, gedeelde uitbreidingen op de *models.*
10. Flexibel toegangsbeheer (ACL, access control lists)
11. Flexibele Caching, op niveau van *models* en *views*.
12. Localisatie, het *framework* biedt ondersteuning voor multilingual applicaties.

Een standaard CakePHP-request is door de mogelijkheid van routing (punt 5), behaviors, componenten en de mogelijkheid om andere databronnen dan slechts de database te gebruiken, iets uitgebreider dan het hierboven beschreven standaard MVC-request. (Cake, 2001)



Afbeelding 3 Typisch CakePHP request afhandeling

Ruby on Rails

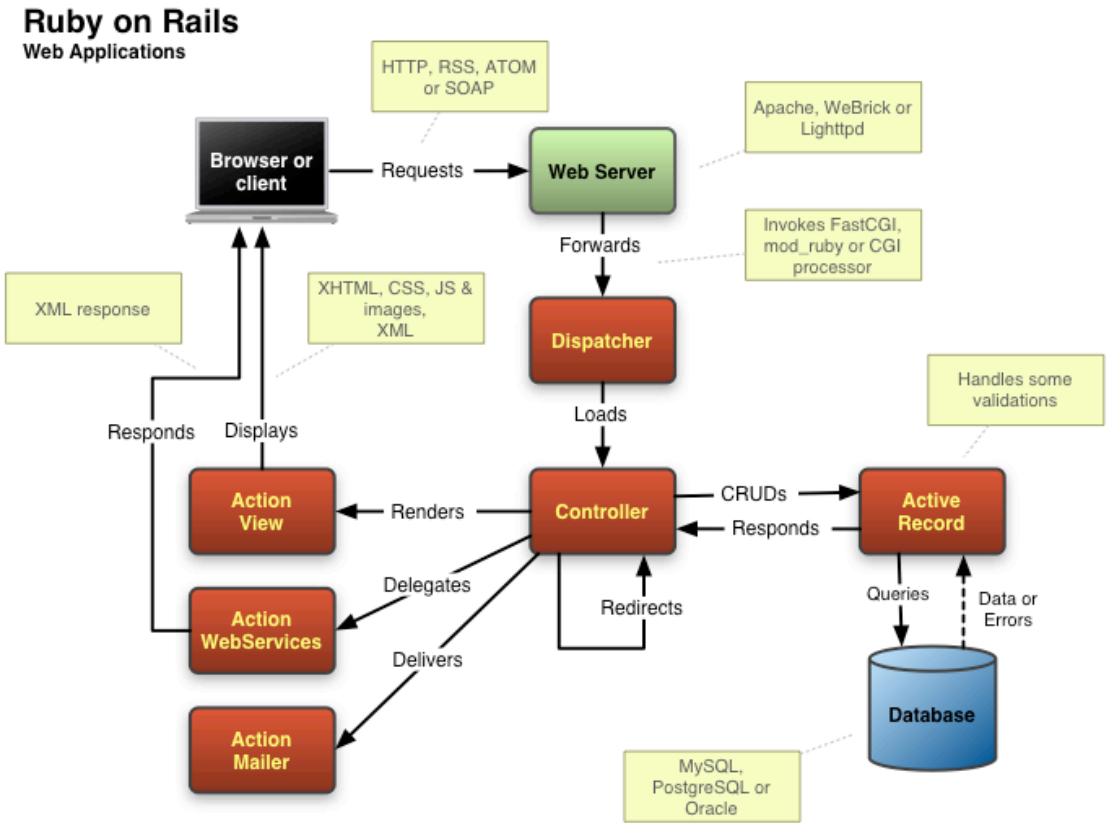
Ruby on Rails is een *MVC framework*, geschreven in en voor de programmeertaal *Ruby*. De taal *Ruby* is een geïnterpreteerde, puur object-georiënteerde taal die, aldus de kenners, erg lijkt op *Smalltalk*. De taal *Ruby* stamt uit 1995 (Matsumoto, 1995) en is, zoals gezegd puur object-georiënteerd.


```
5.times { print "We *love* Ruby -- it's outrageous!" }
```

Code voorbeeld: Alles is een object in Ruby, zelfs primitieve typen

Het *framework Ruby on Rails (RoR)* stamt uit 2003 (RoR, 2003)en gebruikt evenals *CakePHP het* Model-View-Controller patroon. De scheiding in drie delen werkt identiek als hierboven beschreven bij *CakePHP*. *RoR* komt, in tegenstelling tot *CakePHP*, met een eigen webserver, Webrick, maar kan ook gebruikt worden met andere webservers als Apache.

Een typisch RoR request ziet er als volgt uit (RoR, 2003):



Afbeelding 4. Ruby on Rails architectuur

Qua architectuur lijkt *RoR* dus erg veel op *CakePHP* en het handelt op vrijwel dezelfde manier verzoeken af.

Features

Ook qua features lijken *RoR* en *CakePHP* erg op elkaar. Op zich is dat ook niet verwonderlijk, als je bedenkt dat *CakePHP* in eerste instantie naar voorbeeld van *RoR* is gebouwd. De volgende standaard features maken deel uit van *RoR*

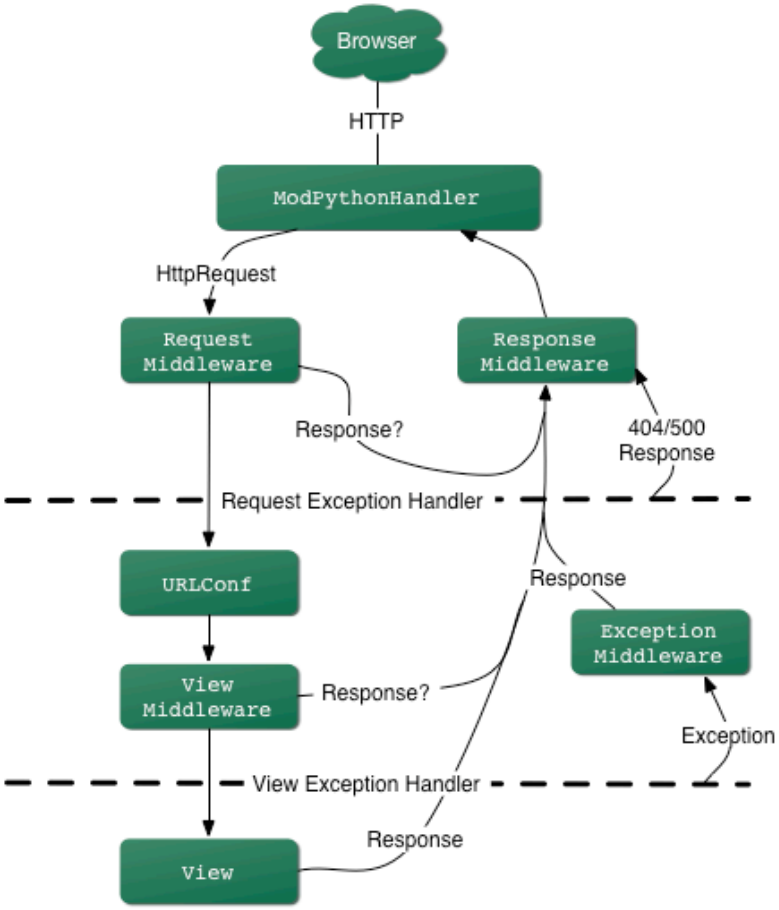
1. Eigen webserver, Webrick
2. Geïntegreerde CRUD (create, read, update en delete) voor database interactie
3. Applicatie scaffolding, het automatisch verwerken van standaardverzoeken, op basis van de gedefinieerde database tabellen. Bedoelt als tijdelijke, snelle start.

4. Code genereren, het automatisch genereren van de basis van de applicatie, op basis van de gedefinieerde database tabellen.
5. Vriendelijke (mens en zoekmachine) URL's en aangepaste routes
6. Ingebouwde validatie, validatie van ingevoerde gegevens, op basis van het model.
7. View Helpers voor AJAX, JavaScript, HTML-formulieren, zorgt ervoor dat niet alle HTML code handmatig uitgewerkt hoeft te worden.
8. Partials; hergebruik van delen van andere views
9. Standaard ondersteuning voor email, cookies, authorisatie
10. Flexibel toegangsbeheer (ACL, access control lists)
11. Flexibele Caching, op niveau van *models* en *views*.
12. Localisatie, het *framework* biedt ondersteuning voor multilingual applicaties.
13. Request profiler; nagaan waar in het afwerken van een request door het framework de bottleneck zit.

Django

Django is een *webframework* geschreven voor en in de programmeertaal Python (Python, 1991). De taal Python stamt uit het begin van de jaren negentig en is ontworpen door de Nederlander Guide van Rossum. Het project *Django* is ontstaan in 2005 en is gebaseerd op het MVC design pattern wat hierboven reeds is uitgelegd. Daarbij hanteert *Django* wel een andere benaming: wat in MVC de controller heet, heet in *Django* de view en wat in MVC de view is genoemd, noemt *Django* de template. Het idee hierachter is, is dat de view de data beschijft die getoond wordt aan de gebruiker, maar niet hoe die data er dan uitziet. De template bepaalt dan hoe de data er uiteindelijk uit komt te zien. Je zou het *framework* dan als een MTV *framework* aan kunnen duiden.

De manier waarop *Django* een request verwerkt, ziet er schematisch als volgt uit: (Holovaty & Kaplan-Moss, 2007)



Afbeelding 5 Request verwerking van Django

Dit ziet er iets anders uit de het schema van *CakePHP* en *RoR* en behoeft wat uitleg. Waneer er een HTTP-request binnen komt, wordt dit door het framework verwerkt tot een HTTPRequest object. Via de URLConf wordt hierna de juiste View aangesproken, die dit verzoek verder af dient te handelen. Wanneer er een fout optreedt, wordt dit door de ExceptionHandler opgepakt, en uiteindelijk gaat alles via een template langs de RedsponseMiddleware terug naar de client.

In feite lijkt dit dus erg op het MVC patroon van de andere twee frameworks, maar is een en ander wat anders benoemd.

Features

Hieronder staat een lijst van de meest opvallende features die *Django* biedt:

- Uitgebreide ondersteuning voor formulier verwerking en validatie
- Standaard beheer-interface, die automatisch beschikbaar is zonder code daarvoor te hoeven maken
- Standaard authenticatie systeem
- Eigen templatesysteem, gebaseerd op overerving

- Templatesyssteen is voorzien van tags en filters om eenvoudig het uiterlijk van de output te bepalen.
- Ondersteuning voor cookies
- Standaard CRUD ondersteuning
- Vriendelijke (mens en zoekmachine) URL's en aangepaste routes
- Caching
- Ondersteuning voor meertalige applicaties
- Security ondersteuning

Conclusie

Dit hoofdstuk biedt een beschrijving van alle frameworks die vergeleken zullen worden in het vervolg van dit onderzoek. Daarbij zijn uiteraard niet alle details besproken, dat is natuurlijk niet mogelijk, maar er is een goed beeld ontstaan over hoe deze frameworks in elkaar steken. Daarmee is de tweede deelvraag beantwoord.

H5. Vergelijking op testondersteuning van de frameworks

Inleiding

Dit hoofdstuk beschouwt de ondersteuning die de *frameworks*, zoals we die geïntroduceerd hebben in hoofdstuk 4, bieden aan de verschillende soorten *tests* die in hoofdstuk 3 staan beschreven en beantwoord hiermee de derde deelvraag zoals die is opgesteld in het eerste hoofdstuk. De resultaten zullen allereerst in tabelvorm gepresenteerd worden, waarna de opvallende verschillen, of onderdelen die dat behoeven, toegelicht zullen worden.

Test ondersteuning

Onderstaande tabel geeft een overzicht van welke soorten tests ondersteunt worden per framework. Hierbij is initieel alleen gekeken naar de ondersteuning die de frameworks “out-of-the-box” bieden, maar vanwege de modulaire opzet van sommige frameworks is ervoor gekozen om te vermelden wanneer een bepaalde feature middels een enkele plugin ondersteunt wordt.

Type test	**CakePHP**	**Django**	**Ruby on Rails**
Handmatig testen			
Capture and play back			
Code coverage analyse	++	*	*
Doctests		++	*
Unit tests	++	++	++
Fixtures	++	++	++
Object mocking	++	*	++
Functionele tests	++		++
Database wijzigingen	++		++
HTTP redirects	+		++
View variabelen/objecten	++		++
Authenticatie			
Web tests			
GET requests	++	++	++
POST requests	++	++	++
File uploads		++	++
HTTP Headers	++	++	++
HTTP redirects	+		++
View variabelen/objecten		++	++

Cookies		++	++
Output tests	++	+	++
Interface / in browser tests	**	*	*
Model-based testing			

+: biedt enige ondersteuning, ++: biedt ruime ondersteuning, *: biedt ondersteuning middels plugin

Toelichting op de resultaat tabel

Deze paragraaf bespreekt de opvallende verschillen en onderdelen die toelichting behoeven uit de resultaattabel die hierboven beschreven staat. Daarbij dient allereerst opgemerkt te worden dat de manier van testen van *CakePHP* en *Ruby on Rails* goed met elkaar vergelijkbaar zijn, omdat ze dezelfde indeling en type *tests* kennen, wat ook te verklaren is aangezien de opbouw van de beide *frameworks* sterke gelijkenis vertoond. *Django* echter kent twee soorten *tests*: standaard unittests, die eigenlijk direct uit Python komen, en *tests* die gebruik maken van Django's Client klasse. Die laatste bevat in feite de mogelijkheden die bij *CakePHP* en *RoR* verdeeld zijn over de *functionele tests* en de *webtests*.

Handmatig testen en capture and playback test

Geen van alle *frameworks* biedt ondersteuning van het handmatig, bijvoorbeeld via de commandline interface, uitvoeren van bepaalde *tests*. Daarmee is de ondersteuning van *capture and playback* van deze *tests* natuurlijk ook onmogelijk. Alle *frameworks* gaan uit van het uitvoeren van *tests* in een script.

Doctests

Alleen *Django* biedt standaard ondersteuning voor de zogenaamde *doctests. RoR* kan wel worden uitgerust worden om dit te ondersteunen, door het gebruiken van een plugin. Om een beeld te vormen, een *doctest* in *Django* ziet er als volgt uit:

```
def get_elem(a_list, index):
    """
    >>> a = ['John', 'Sjaak', 'Piet']
    >>> my_func(a, 0)
    'John'
    >>> my_func(a, 1)
     'Sjaak'
    """
    return a_list[index]
```

Code voorbeeld 1: doctest in Django

Unittests

Alle onderzochte frameworks bieden een ruime ondersteuning voor het uitvoeren van *unittests*. Daarbij bieden ook alle drie de *frameworks* de mogelijkheid tot het gebruik van *fixtures*, iets wat bijzonder handig, bijna noodzakelijk, is voor het efficiënt opstellen van de tests. Het is daarbij wel interessant om voor elk

framework een simpele unittest te laten zien, zodat de verschillen en overeenkomsten zichtbaar worden. We gaan daarbij uit van een simpel model genaamd Car, waarbij een auto een bepaalde kleur kan heeft.

```
class CarTest extends Car {
  var $name = 'CarTest';
  var $useDbConfig = 'test_suite';
}

class CarTestCase extends CakeTestCase {
  var $fixtures = array( 'car_test' );

  function testColor() {
    $this->CarTest =& new CarTest();

    $result = $this->CarTest->color("Ferrari");
    $this->assertEqual($result, 'red');
  }
}
```

Code voorbeeld 2: unittest in CakePHP

```
import unittest
from myapp.models import Car

class CarTestCase(unittest.TestCase):
  def setUp(self):
    self.ferrari = ↲
      Car.objects.create(name="Ferrari", color="red")

  def testColor(self):
    self.assertEquals(self.ferrari.color(), 'red')
```

Code voorbeeld 3: unittest in Django

```
require 'test_helper'

class CarTest < ActiveSupport::TestCase
  fixtures :cars

  def test_color
    color = Car.color('Ferrari')
    assert_equal color, 'red'
  end
end
```

Code voorbeeld 4: unittest in Ruby on Rails

Zoals te zien is, los van de verschillende taalconstructies, zijn dit soort simpele tests in alle *frameworks* op vergelijkbare manier op te zetten. De test van *Django* gebruikt, ter demonstratie, geen *fixtures* zoals de andere twee wel, maar heeft ook die mogelijkheid en dat geldt andersom ook.

Object mocking

Een ander opvallend punt in onze vergelijking is *object mocking*. Dit wordt gebruikt om afhankelijkheden bij het testen te reduceren en daarvoor bieden zowel *RoR* als *CakePHP* ondersteuning. Bij *object mocking* wordt er een nieuwe klasse gemaakt van een bepaalde referentieklasse, waarbij alle bestaande methodes nog bestaan, maar deze zijn allemaal leeg. Beide *frameworks* hebben ook nog eens de mogelijkheid tot zogenaamd *partial mocking*, zodat het mogelijk is om sommige methodes wel in oorspronkelijke staat te houden. *Django* ondersteunt *object mocking* niet standaard, maar kan middels een plugin hier wel toe uitgerust worden.

Functionele tests

Zoals al eerder opgemerkt, scharen *CakePHP* en *RoR* het testen van de controllers onder de *functionele tests.* De test die daarbij uitgevoerd wordt, betreft dus niet een simulatie van een HTTP request. *Django* kent deze manier van testen niet, views (wat de controllers worden genoemd in de andere twee *frameworks*) en templates worden getest middels simulatie van HTTP requests.

File uploads

Een belangrijk onderdeel bij het testen van webformulieren, is het kunnen testen van formulieren waarbij het uploaden van een bestanden een rol speelt. Dit is een situatie die bij webapplicaties natuurlijk niet ongewoon is en daarom ook getest zou moeten kunnen worden. *CakePHP* biedt hier als enige *framework* geen ondersteuning voor, hoewel het wel in de roadmap voor toekomstige releases staat opgenomen.

HTTP Redirects

Een controllertest die detecteert welke HTTP-headers de controller terugstuurt, is in *Ruby on Rails* wel mogelijk, maar niet in *CakePHP*. Wil een dergelijke test gedaan worden in CakePHP, dan is het wel mogelijk om daar een workaround voor te maken. Om dit te demonstreren staat hieronder eerst een voorbeeld van een dergelijke test in *RoR* en vervolgens in *CakePHP*.

```
def test_should_create_post
  assert_difference('Post.count') do
    post :create, :post => { :title => 'Hello',↲
      :body => 'This is my first post.'}
  end
  assert_redirected_to(:controller => "posts",↲
    :action => "index")
  assert_equal 'Post was successfully created.',↲
```

```
    flash[:notice]
end
```

Code voorbeeld 5: Functionele test in Ruby on Rails

```
class TestPostsController extends PostsController {
  var $name = 'Posts';
  function redirect($url, $status = null, $exit = true) {
    $this->redirectUrl = $url;
  }

  function _stop($status = 0) {
    $this->stopped = $status;
  }
}

class PostsControllerTestCase extends CakeTestCase {
  function __construct() {
    $this->Posts = new TestPostsController();
  }
  function testCreate() {
    $beforeCount = $this->Posts->Post->find('count');
    $this->Posts->data = array(
      'Post' => array(
        'title' => 'Hello',
        'body' => 'This is a test post!',
      )
    );
    $this->Posts->add();

    $afterCount = $this->Posts->Post->find('count');
    $this->assertEqual($beforeCount+1,$afterCount);

    $this->assertEqual($this->Posts->Session-> ↲
      read('Message.flash.message'), ↲
      'Post was successfully created.');

    $this->assertEqual($this->Posts->redirectUrl, ↲
      array('action' => 'index'));
   }
}
```

Code voorbeeld 6: Functionele test in CakePHP

Zoals uit dit voorbeeld blijkt, is de test in *Ruby on Rails* een stuk simpeler dan die in *CakePHP*. Dit verschil komt voornamelijk omdat in *CakePHP* bepaalde methodes van de controller overschreven moeten worden om ze goed te kunnen testen. In

RoR is dat niet nodig en zit die ondersteuning zonder extra moeite direct ingebakken.

Django biedt ook dergelijke ondersteuning. Zoals al opgemerkt, zit deze functionaliteit bij dit *framework* op een wat andere plaats, maar toch is een vergelijkbare test mogelijk:

```
from django.test import *
import doctest

class ViewTests(TestCase):
  def runTest(self):
    c = Client()
    response = c.post('/posts/add',{'title': 'Test title',↲
     'message': 'This is the actual post'})

    self.assertRedirects(response, '/posts/')
    self.assertEqual(self.response.context['flash'] ↲
      ['message'], 'Post was successfully created.')
```

Code voorbeeld 7: webtest in Django

Output tests

Tot slot kijken we nog naar de ondersteuning die de frameworks bieden op het gebied van output-analyse, iets wat bij webframeworks een belangrijk issue is. Hier kunnen we concluderen dat *CakePHP* en *Ruby on Rails* hier een uitgebreide ondersteuning voor beiden. Om de verschillen en overeenkomsten te demonstreren zetten we nogmaals een vergelijkbare test op voor alle *frameworks*.

```
class myTestWebTestCase extends CakeWebTestCase {
  function testHomepage() {
    $result = $this->get('/pages/home');
    $this->assertResponse(200);
    $this->assertText('Lorem ipsum dolor sit amet.');
    $this->assertTitle('Foobar :: Home');

    $this->assertElementsBySelector(
      ul#navigation > li',
      array('Menu item 1', 'Menu item 2')
     );
}}
```

Code voorbeeld 8: output test in CakePHP

```
def test_should_show_post
   get "/pages/home"
   assert_response :success
```

```
    # Assert_select zeer krachtig
    assert_select 'title', 'Foobar :: Home'
    assert_select 'ul.navigation' do
          ...
    end
  end
```

Code voorbeeld 9: output test in Ruby on Rails

```
def test_home(self):
    response = self.client.get('/pages/home')
    self.assertEqual(response.status_code, 200)

    # Zijn er 5 newsitems aan de template gestuurd?
     self. assertEqual(len(response.context['newsitems']), 5)
    self.assertEqual(response.template[0].name, 'home.html')
```

Code voorbeeld 10: output test in Django

Een aantal verschillen is de moeite waard om even te belichten. Zoals te zien valt, bieden *RoR* en *CakePHP* de mogelijkheid om de output, wat meestal HTML is, vergaand te analyseren. *Django* kan wel zien of er bijvoorbeeld een bepaalde string in de output zit, maar kan de HTML-structuur niet analyseren.

Django heeft wel de mogelijkheid om te zien welke variabelen de view heeft doorgegeven aan de template. *Ook Ruby on Rails* heeft die optie, maar met *CakePHP* valt dat alleen te testen met een functionele test.

In-browser tests

Alle frameworks hebben een sterretje bij in-browser tests, omdat er tools zijn die dit onafhankelijk van het framework kunne doen. Een voorbeeld van een dergelijke tool is *Selenium* (SeleniumHQ). Helaas is die niet browser onafhankelijk, iets wat bij webpagina's wel een belangrijk punt is. *CakePHP* heeft een ingebouwde koppeling voor *Selenium* en is daarmee iets beter uitgerust dan de andere twee *frameworks.*

Model based tests

Helaas zijn modelbased test nog een stukje te hoog gegrepen voor alle frameworks. Zover ik dat met zekerheid kan zeggen is er ook nog geen enkel webframework die dat soort ondersteuning wel biedt.

Conclusie

Het is lastig om tot een goede vergelijking te komen tussen de *frameworks*. Dit wordt ten eerste bemoeilijkt door de zeer verschillende wijze waarop de *frameworks* hun tests hebben ingedeeld. Kun je soms een bepaalde test niet linksom realiseren, dan kan het vaak wel rechtsom.

De tests zoals die beschreven zijn in hoofdstuk 2 zijn echter wel allemaal getoetst voor alle drie de raamwerken. Daarbij is de mate van ondersteuning die is toegekend een subjectieve waarde. Omdat er echter maar weinig nuances in die klassering zijn, is het echter niet waarschijnlijk dat deze compleet de plank misslaan.

H6. Conclusie

Conclusie

De onderzoeksvraag die beantwoord dient te worden middels dit onderzoek, luidt als volgt:

> *Welke ondersteuning voor testen bieden de veelgebruikte frameworks voor webapplicaties?*

Aan de hand van de deelvragen, is in hoofdstuk 5 een overzicht gekomen van de mate van testondersteuning die de drie *frameworks* bieden, waarmee de onderzoeksvraag beantwoord is. Hoewel het niet strikt de onderzoeksvraag is, is het natuurlijk interessant om te zien of je hier nog een conclusie aan kan verbinden die vertelt welke van deze *frameworks* dan het meeste of de beste ondersteuning biedt. Dat is echter nog niet zo eenvoudig, omdat het niet een simpele kwestie is van, bijvoorbeeld, de kruisjes uit de tabel tellen. Dat is niet mogelijk, omdat de gebieden van de software die de soorten tests dekken, elkaar overlappen. Kun je in een bepaald *framework* een bepaalde test dus niet op manier X uitvoeren, dan is er vaak wel een manier Y waarop het wel kan er waarmee je hetzelfde bereikt. Je kan voorzichtig zeggen dat de mate van testondersteuning bij deze drie onderzochte raamwerken ongeveer gelijk is, waarbij er opgemerkt kan worden dat *CakePHP* en *Ruby on Rails* een duidelijke voorsprong op *Django* hebben op het gebied van HTML-analyse.

Tot slot kun je je nog afvragen: als er betere testondersteuning is, kun je dan ook betere applicaties maken in het betreffende *framework*? Daar kunnen we kort over zijn: dat is niet zo. Ten eerste is dat niet zo, omdat we geconcludeerd hebben dat de verschillen niet erg groot zijn. Ten tweede, en dat is het belangrijkste, hangt het natuurlijk af van hoe zorgvuldig de software ontwikkelaar zijn tests opstelt.

Reflectie en aanbevelingen

Bij het beantwoorden van de deelvragen zijn er een paar opmerkingen gemaakt. Ten eerste dat de selectie van *frameworks* natuurlijk niet volledig zijn. Het kan voor de inhoud van dit onderzoek en de volledigheid daarvan interessant zijn om nog meer *frameworks* in de vergelijking mee te nemen. Een ander punt is de selectie van meest relevante soorten tests. Dat is een subjectieve norm en daarom zou deze selectie verder uitgebreid kunnen worden. Daarnaast is er natuurlijk alleen gekeken naar *black- en greybox testen*, maar bij vergelijking van de *frameworks* zijn er natuurlijk nog veel meer aspecten die van belang kunnen zijn, zoals bijvoorbeeld de (Cavano & McCall, 1978).

Al met al zijn er genoeg punten waar dit onderzoek uitgebreid kan worden om een gedetailleerdere vergelijking tussen de *webframeworks* te maken.

H7. Bibliografie

Beizer, W. (1996). *IEEE: Black Box Testing: Techniques for Functional Testing of Software and Systems.* Wiley.

Boehm, B., Brown, J., & Kaspar, J. (1978). *Characteristics of Software Quality.* Amsterdam, Nederland: American Elsevier.

Cake, S. F. (2001). *CakePHP Rapid Development PHP Framework, http://cakephp.org.*

Cavano, J., & McCall, J. (1978). *A framework for the measurement of software quality.* New York, USA: ACM Press.

Holovaty, A., & Kaplan-Moss, J. (2007). *The definitive guide to Django, webdevelopment done right.* Apress.

ISO. (2001). *ISO/IEC 9126: Software Engineering - Product Quality - Quality Model.*

Matsumoto, Y. (1995). *Ruby.* Japan.

PHPGroup. *PHP: Hypertext PreProcessor, http://www.php.net.*

Pressman, R. S. (2001). *Software Engineering: a practioner's approach (5th ed.).* McGraw-Hill.

Python. (1991). *Python Programming Language, http://www.python.org.*

RoR. (2003). *Ruby On Rails, http://rubyonrails.org.*

SeleniumHQ. *Web application testing system, http://seleniumhq.org.*

Woolderink, C. (2007). *Het bepalen van de onderhoudbaarheid van objectgeoriënteerde broncode door middel van metrieken.* Amsterdam, Nederland.