Aarhus Universitet
Semesterprojekt 4
Gruppe 5

Test

CarnGo

Gruppemedlemmer:
Edward Hestnes Brunton
(201705579)
Marcus Gasberg
(201709164)
Martin Gildberg Jespersen
(201706221)
Mathias Magnild Hansen
(201404884)
Tristan Moeller
(201706862)
Erik Mowinckel
(20107667)
Hamza Ben Abdallah
(201609060)

Vejleder:
Jesper Michael Kristensen, Lektor

29. maj 2019

1 Versionshistorik

Ver.	Dato	Initialer	Beskrivelse
1.0	23-04-2019	MH	Opsætning og første udkast

Indhold

2 Indledning

Dette dokument beskriver, hvordan softwareapplikationen CarnGo testes. For at sikre, at softwaren opfører sig som forventet, opfylder kravene og kører uden fejl er det essentielt at teste softwaren grundigt. Software test skal i denne sammenhæng forstås som at man systematisk og objektivt verificerer en software enheds tilstand og opførsel baseret på nogle objektive og prædefinerede kriterier eller krav. Med systematisk og objektivt menes der, at testen skal være målbar, og det skal være muligt for en anden at gentage testen med identiske resultater. At kriterierne skal være objektive betyder, at enhver vil kunne afgøre, hvorvidt testen er udført, og der er således ikke plads til fortolkning af resultatet.

I det følgende beskrives de redskaber, frameworks og principper som danner grundlaget for CarnGo's software tests. Der indledes med en beskrivelse af Unit Testing og Continuous Integration, og herefter diskuteres en række begreber, som har relation til kvaliteten af testen. Endelig beskrives integrationstest for projektet.

3 Unit Testing

En unit test er et automatiseret stykke kode som kalder den 'unit of work', der bliver testet.[1] Den checker nogle antagelser omkring et enkelt slutresultat for denne unit. En unit test består ofte af tre hovedaktioner eller trin (mønsteret er også kendt som AAA og er anvendt for hver enkelt test case):

1. *Arrange* objekter, opret og opsæt dem efter behov
2. *Act* på et objekt, udfør aktiviteten som skal testes
3. *Assert* at noget er som forventet

Der anvendes oftest et framework til at skrive unit tests. Til at teste CarnGo er anvendt unit testing frameworket NUnit[2]. Dette har visse fordele, eftersom det gør test cases lettere at skrive og hurtige at eksekvere ved tryk på en knap. Derudover sikrer det, at testene er pålidelige, lette at forstå og lette at vedligeholde.

4 Continuous Integration

Hvis ikke man er opmærksom på løbende at integrere softwaren, kan man meget vel løbe ind i problemer, når man nærmer sig slutningen af udviklingsprocessen. For at imødekomme integrationsproblemer er der i projektet anvendt en praksis kaldet Continuous Integration[3]. Denne praksis er ofte anvendt inden for softwareudvikling og går ud på, at teamets medlemmer integrerer ofte, og at hver integration verificeres af et automatisk build. Dermed opdages og korrigeres integrationsfejl tidligere i processen, og man minimerer risikoen for problemer senere hen. En anden fordel er, at man aldrig er langt fra et fungerende build, og som udvikler får man løbende feedback og opdaterede mål for kodens og testens kvalitet.

Til projektet er Github anvendt som remote repository og Jenkins er anvendt som build server[4][5]. Dette er konfigureret på en sådan vis, at Jenkins bliver triggered, når der pushes til Git serveren. Det medfører, at der startes et build job, hvor projektet bygges og tests køres. Først herefter kan der igen pulles fra serveren.

5 Test kvalitet

5.1 Coverage

For at sikre kvaliteten af testene er der løbende blevet set på, hvor meget af applikationens kode, der er dækket af tests. Dette betegnes Test Coverage, og det giver et billede af, hvilke områder af programmet, der bliver anvendt i applikationens test suite[6]. Dette giver et kvantitativt mål for, hvor 'godt' der testes, eller rettere i hvor høj grad testene dækker koden. Denne viden kan anvendes til systematisk at udvide eller tilføje test cases indtil alt applikationskoden er omfattet.

Der skelnes mellem forskellige former for Test Coverage. Line Coverage giver et mål for, hvor mange linjer kode, der nås af tests. Branch Coverage er derimod et mål for, hvor mange branches, der anvendes i tests, eller om 'decision points' såsom switches og if-statements er blevet aktiveret til fulde. Der er visse ulemper forbundet med de forskellige former for coverage, og generelt er fuld coverage ikke ensbetydende med ingen fejl. Det er dog et simpelt og objektivt mål for kvaliteten af tests, og det kan guide udvikleren i forhold til, hvor der skal lægges en ekstra indsats. Til at beregne Line Coverage for CarnGo anvendes værktøjet dotCover, og der sigtes efter 100% coverage[7].

5.2 Boundary Value Analysis

På grund af begrænsede ressourcer vil det oftest ikke være realistisk at lave udtømmende tests for alle sæt af test data, og særligt ikke, hvis der er mange mulige input kombinationer. Der er derfor behov for en måde, hvorpå man kan vælge sine test cases, så man arbejder med et sæt af *tilstrækkelige* og *nødvendige* test cases, frem for et udtømmende sæt.

Boundary Value Analysis har til formål at identificere de input værdier, hvor output ændrer værdi eller validitet[8]. Disse betegnes boundary values og ideen er at teste for inputs på begge sider af boundary values. Boundary Value Analysis har gjort det muligt at vælge logiske input værdier, hvilket øger sandsynligheden for at finde fejl. Der vælges typisk følgende input værdier:

- Minimum
- Lige over minimum
- En nominel (mellemliggende) værdi
- Lige under maksimum
- Maksimum

5.3 Equivalence Partitions

Boundary Value Analysis (BVA) anvendes altid i kombination med Equivalence Partitions (EP)[8]. Equivalence Partitions er ligesom BVA en form for black-box testing, da de udelukkende betrager input og forventet output. Målet er at reducere mængden af tests gennem analyse. En EP er en del af alle mulige inputs, som resulterer i det samme output eller den samme form for opførsel. En EP er defineret af et sæt af boundary values, hvorfor BVA udføres før EP. Når Equivalence Partitions er identificeret, testes der med mindst én værdi fra hver EP (*nødvendigt*). Der testes ikke med samtlige værdier fra en EP, da én eller få værdier vil være *tilstrækkeligt*. Tilsammen har Boundary Value Analysis og Equivalence Partitions hjulpet med at sikre, at der er udtænkt de rigtige test cases, og at applikationen ikke testes for meget, og de er således et godt supplement til Coverage.

5.4 ZOMBIE

ZOMBIE er et akronym, der fungerer som en guide for, hvad der skal testes i projektet, og de overvejelser der er gjort undervejs[9]. **Z** står for Zero, og indikerer, at de første test scenarier bør være simple post-konditioner for nyligt oprettede elementer. F.eks. kan det testes, at en tom collection indeholder nul elementer, eller at der er nul kald af et nyoprettet objekt. **O** står for One som er den næste case, der bør testes. Når man går fra nul til ét input/output/action er der en række nye scenarier, der bør afprøves. F.eks. at et nyoprettet objekt har den rette tilstand efter ét kald af hver metode, eller tests af input for en collection med ét element. Når Zero og One cases er testet generaliseres scenariet, og der testes nu et Many scenarie (**M**) med mange inputs/outputs/actions. Eksempelvis testes objektets tilstand efter mange funktionskald og inputs testes for en collection med mange elementer. Man bør altså følge **ZOM**, så man går fra simple til mere komplekse testscenarier.

BIE-delen af ordet står derimod for de overvejelser, man bør gøre sig undervejs, alt imens man sigter efter simplicitet i test scenarierne og løsningerne. **B** står for boundaries, da man bør teste grænseværdier og aktivt gøre brug af førnævnte Boundary Value Analysis og Equivalence Partitions. **I** står for Interfaces, hvilket indikerer, at man bør teste alle metoder, events, kaldte interfaces og afhængigheder. Begrebet Coverage som blev diskuteret ovenfor, kan være et brugbart hjælpemiddel til at sikre, at dette punkt er opfyldt. **E** står for Exceptional Behaviour, som dækker over uforventet opførsel, timeouts og generel fejlhåndtering.

Nedenfor ses en test case, som anvender NUnit og NSubstitute frameworks[2][10]. Den tester, at der navigeres til startsiden, når bruger logger ind på applikationen. Enheden der testes (uut) er en instans af LoginPageViewModel. Først sikres det, at brugerens e-mail og password består valideringen (Arrange), og herefter kaldes en Login Command (Act). Endelig tjekkes der på, at der modtages et funktionskald af GoToPage() med StartPage som argument (Assert). I forhold til ZOMBIE testes der på metoder/afhængigheder (**I**).

```
[Test]
public async Task Login_LoginNavigatesToLogin_GoToStartpage()
{
    // Arrange
    _fakeEmailValidator.Validate(Arg.Any<string>()).Returns(true);
    _fakePasswordValidator.Validate(Arg.Any<SecureString>())
    .Returns(true);

    // Act
    await _uut.Login();

    // Assert
    _fakeApplication.Received()
    .GoToPage(Arg.Is<ApplicationPage>
    (page => page == ApplicationPage.StartPage));
}
```

6 Integrationstest

De vigtigste dele af applikationen er integrationstestet. Det vil sige, at der testes interaktioner og afhængigheder mellem flere moduler[11]. Frem for at lave udtømmende tests for hele applikationen, er det valgt at fokusere på de dele af systemet, hvor der er størst sandsynlighed for, at fejl kan forekomme. Derfor er der fokuseret på at teste modellaget og data access laget sammen, herunder database, queries og konvertering af data. Der er taget udgangspunkt i en Bottom up integrationsstrategi, hvilket vil sige at de moduler eller klasser, som har færrest afhængigheder testes sammen først, og herefter udvides der med flere moduler, indtil alle er omfattet af tests[12]. Denne strategi anvendes fordi, at det vurderes, at fejl ofte vil opstå i de laveste niveauer af CarnGo applikationen, da der ofte skal interageres med databasen og konverteres data. Med denne fremgangsmåde bliver fejlene opdaget og korrigeret tidligt i processen.