Test Rapport

Læringsmål

1. Opnå praktisk erfaring med TDD.
2. Erfaring med Data driven testing, ved at bruge forskellige bibliotekter der kan læse test data fra filer(cvs, Excel, etc.).
3. Mere erfaring med Unit testing og mocking, hvor der indgår objekter med afhængigheder.
4. Erfaring med Integration testing med henblik test af databasen(DBUnit etc).
5. Erfaring med problemstillinger omhandlende test af databasen.**(Opsætning af test data, mocking)**
6. Erfaring med Continuous Integration og de tilhørende værktøjer som er inkluderet i dette(Travis, Maven).

Contents

1. Opnå praktisk erfaring med TDD

Vi alle i gruppen har et lidt ambivalent forhold til TDD. Vi har de seneste projekter, blandt andet på datamatiker-studiet, blevet introduceret til TDD. Derfor har vi igennem tiden prøvet at have det som en fast rutine, når vi skulle udvikle et nyt projekt. Problemet er bare, at vi oftest ser bort fra det, da vi i udviklingen af projektet, som regel glemmer at skrive det vigtigste, nemlig alle de tests. Derfor var det en bestemt mulighed at give det en ekstra chance i dette semesterprojekt. Vi må alle indrømme, at vi er blevet positivt overrasket.
Vi har primært benyttet det når der skulle skrives helt enkle unit-tests, da det oftest var stort og bøvlet at benytte det til Integration Tests, hvor forskellige komponenter skulle inkluderes.
Vi følte at TDD var med til at holde produktiviteten høj under udviklingen af projektet.
Grundet til dette var, at den hjalp med at indsnævre vores fokus. Man skriver en test der fejler, hvorefter man udelukkende fokuserer på det for at få den til at blive grøn. Det tvinger en til at tænke på mindre stykker funktionalitet ad gangen snarere end projektet/applikationen som helhed, og man kan derefter gradvis bygge det op med tests, velvidende, at det virker, end at forsøge at få det hele til at virke til sidst.
Vi er dog kommet frem til en lille delkonklusion efter at have benyttet TDD.
Vi føler ikke at TDD er ikke en testfærdighed. Det er en færdighed i at designe en applikation korrekt. Det kan kun føre dig til en god, enkel, brugbar kode med praksis og en konstant bevidsthed om de designretninger, som man vil benytte i sin applikation. Hvis du skriver tests af hensyn til kodedækning, vil man lave tests, som ikke giver mening og er overflødige. Vi tror TDD er nyttigt, men vi vil ikke være dogmatiske omkring det. Hvis disse overflødige tests gør vedligeholdelsen vanskelig, kan man ligeså godt slette dem.

2. Erfaring med Data driven testing, ved at bruge forskellige bibliotekter der kan læse test data fra filer(cvs, Excel, etc.).

Data driven testing er relativt nyt for os alle, da det ikke er noget vi decideret har arbejdet med. Det har givet god mening for os at benytte det i dette projekt, da vi har brug for at lave CSV-filer, der skal importeres ind i en database. Der kan derfor spares tid ved at aflæse data fra filerne, i stedet for at skulle importere det i databasen for derefter at finde ud af om det virker. Når du har

mulighed for løbende at ændre testdata og bruge den med det samme, giver den dig en realistisk test case og kan potentielt finde andre fejl, du ikke ville finde ellers.

3. Mere erfaring med Unit testing og mocking, hvor der indgår objekter med afhængigheder.

I forlængelse med TDD og Continuous Integration føler vi, at Unit Tests har været med til at reducere antal fejl i koden. Unit test bliver kørt automatisk ved hvert build, hvorpå man straks får at vide, hvor koden fejler, hvis dette er tilfældet.
Med mocking har vi erfaret, man bør mocke ting, der afhænger af afhængighed eller eksternt for at forhindre testen i at være afhængig af noget eksternt.
Hvis afhængigheden ikke afhænger af noget eksternt, er den eneste fordel, du får ved at mocking det, at testen vil fungere ordentligt, selvom, at det er en forkert afhængighed - men det forudsætter at mock fungerer korrekt. Derfor skal man enten:
Skrive en mock, der emulerer afhængigheden helt eller skrive en mock, der emulerer afhængigheden af de specifikke tilfælde, der vil blive brugt i testen.
Den første mulighed er ikke særlig god - hvorfor skulle ens mock være bedre end den oprindelige afhængighed. Det er trods alt sikkert at antage, at vi har lagt mere tid og en større indsats i den oprindelige afhængighed end i mocken.
Den anden mulighed betyder, at ens mock kender de nøjagtige detaljer om implementeringen - ellers vil man ikke vide, hvordan implementeringen bruger afhængigheden, deraf ved man ikke, hvordan man mocker disse specifikke anvendelser. Det betyder, at testen ikke er rettet mod et af hovedformålene med Unit Test – nemlig at verificere, at koden fungerer korrekt efter ændringer i implementeringen.
Erfaringen er derfor, at ulemperne ved mocking er for store, og fordelene er for små - især i betragtning af, at man altid kan køre Unit Tests med afhængighederne for at kontrollere, om det fungerer korrekt

4. Erfaring med Integration testing med henblik test af databasen(DBUnit etc).

DBUnit er en tilføjelse til JUnit, som kan bruges til at sætte ens database i en kendt tilstand i mellem hver test. Det er blandt andet blevet brugt til at sikre, at databasen ikke bliver fyldt op med testdata. Vi har dertil også åbnet muligheden for at benytte en in-memory database, således, at vi ikke skal bruge databasen på vores virtuelle maskine. DBUnit har været en svær ting at få opsat grundet sine mange konfigurationer. Derfor er erfaringen -

1. Forbered miljøet
2. Kør test
3. Tjek testresultatet
4. Validér i forhold de forventede resultater.

   En værdi bør ikke opdateres et sted og vise en ældre værdi på et andet. Derfor er det vigtigt at ens DB test cases inkluderer en kontrol af dataene alle steder, således, at det er konsekvent det samme.

Med disse tests kan man kontrollere, om ændringerne har brudt eksisterende funktionalitet.

5. Erfaring med problemstillinger omhandlende test af databasen.(Opsætning af test data, mocking)

En Unit Test skal altid teste en klasse isoleret. En problemstilling kunne være, at der forekommer uønskede effekter fra andre klasser eller komponenter.

Derfor har vi erfaret at mocking er en god måde, at få en Unit Test isoleret på. Vi kan teste et mock objekt, der f.eks. simulerer vores database, hvor der samtidig er sikret, at forholdene altid er det samme. Der vil ikke kunne opstå en situation, hvor en test vil fejle fordi forbindelsen til databasen er gået tabt. Derfor har vi fundet det vigtigt at mocke alle de klasser væk, som snakker til

databasen.
En anden problemstilling er at sikre sig at tests af databasen ikke ender ud med at gøre således at andre tests fejler. DBUnit er også en stor hjælp her.

6. Erfaring med Continuous Integration og de tilhørende værktøjer som er inkluderet i dette (Travis, Maven).

Continous Integration(CI) har været en stor hjælp til at sikre os, at den kode som bliver kørt lokalt også virker når det bliver smidt op på en server. Vi har reelt ikke haft tilgang til nogen server og derfor har prøvet at simulere denne proces via branches og Travis. Vi har dertil udvidet processen lidt og benyttet os af Pull Requests hver gang vi havde en stor ændring til projektet. Her kom Travis frem og fortalte hvis der ville opstå problemer hvis Pull Requestet blev godtaget. Det har hjulpet en del, i forhold til Merge Conflicts. Travis gør Continous Integration let – en konfiguration af Travis blev udført på få minutter. Man skal bare hooke sit Git repository hop og tilføje en .travis.yml. Den store fordel ved Continous Integration viser sig, når noget stopper med at fungere. Man kan sætte det op således at man får en email, der informerer dig om problemet, og derefter kan du rette eventuelle fejl der måtte være. Jo hurtigere du skubber dine nye kode op, desto hurtigere får du feedback. Det meste af tiden vil det gå hurtigt nok til, at du stadig kan huske, hvad din sidste ændring handlede om. Det samme gælder for manglende filer. Hvis man glemmer at commite en fil, som man til tider gør, kan Travis fange den fejl og sende en påmindelse.
En ulempe har dog været, at det har været svært at integrere Travis op i mod en virtuel maskine. Travis har ikke tilgang til denne og vil fejle builds, da den vil fejle alle tests, som benytter sig af den virtuelle maskine. Det har betydet, at vi har måtte lave @Ignore på nogle tests - og nu tilbage til mocking.