

Refleksion over læringsmål i Test

Vi vil udvide vores erfaring med TDD, ved at udvikle dataminereren med TDD. Målet er at vores TDD, vil efterlade os med en test-suite af unittests, der afspejler projektets funktionalitet.

Vores mål om at arbejde med TDD er overordnet set lykkedes efter vores ønske. Vores approach til at få hul på projektet var måske en smule omvendt af hvad der giver mest mening ud fra et test-driven synspunkt. Det ville måske give mest mening at lave en test der testede om en liste af bøger blev hentet korrekt ud på en forfatters navn. Herefter implementere noget kode der med en mock eller stub fik noget data fra hvad der senere skulle blive en database.

Vores tankegang var omvendt. Vi følte at projektets komplicerede dele lå i selve de forskellige queries til databaserne. Derfor startede vi i Sprint 0, med at opsætte en stærk struktur, der ved hjælp af interfaces tillod os at lade vores logiske lag, være ligeglad med hvilken databasetype den benyttede som datakilde. Herefter startede vores TDD. Opret connection, få en workable database, håndter tilfælde af ingen databaseadgang osv.

Dernæst kunne vi teste og implementere vores egentlige queries. Vi kan nu se at dette har været en "omvendt" tilgang, da vi på en måde er startet i "bunden". Eftersom al kompleksiteten lå i disse queries, følte vi heller ikke at vi kunne mocke vores forespørgsler væk. Det kan derfor diskuteres om de tests vi troede var unittests, rent faktisk er unittests, da de har direkte kontakt med en database. Om det så er en in-memory eller testdatabase er irrelevant da der er tale om en egentlig connection. Man kan derfor argumentere for at vi startede med TDD hvor vi skrev integrationtests, fremfor unittests.

Integrationtests eller ej, så fik vi det ønskede ud af TDD. Vi fik en ganske fin test-suite, der testede de komplekse dele af programmet, og sikrede et testable design af koden. Vi føler at dette design har styrket programmets struktur, og formentligt gjort det mere maintainable og forhindret at vi akkumulerer teknisk gæld, hvis projektet havde fortsat.

Vi vil gå i dybden med at implementere en CI/CD-chain, ved hjælp af et relevant værktøj.

Vi er lykkedes med at få opsat en fin CI/CD-chain til projektet. Valget faldt på "Travis", dette var mest af alt et ønske om at prøve en service vi ikke før havde prøvet. Maven spiller også en stor rolle i denne chain, da det er et værktøj der gør at vi kan afvikle og kontrollere vores build lokalt, før vi lader Travis gøre det online. Vores opsætning af projektet og Maven's commandlinetool, tillader os at vi kan køre forskellige typer tests i suites efter ønske. Dette gav enormt god mening for os, da vi kunne køre vores hurtige unittests lokalt når vi udviklede, og når vi var klar til at skubbe buildet, kunne Travis stå for at køre de langsommere og mere omfattende integrationtests. Selve deployment delen af vores chain, virkede også efter ønske. Det var muligt at give vores git pushes 'tags', således at Travis forstod at der var tale om egentlige release. Disse releases endte ud i at vi havde eksekverbare jar-filer. Udbyttet af vores chain, var nok mere i form af læring, end "business-value" for vores projekt. Da dette er et skoleprojekt, var det ikke så afgørende om

vores builds fik chainen til at fejle og blive rød. Læringen derimod har været positiv. Eftersom vi formentlig kommer til at arbejde i agile miljøer, er det visse ting vi kan tage med fra denne chain.

Hvis du ønsker at arbejde agilt, ved du at du skal tage overkommelige arbejdsopgaver der helst ikke strækker sig over en dag. Dette kan en CI/CD-chain hjælpe dig med at overholde. Hvis du ikke vil for langt væk fra master-branchen, må du lave opgaver af en størrelse der gør at du på én dag ikke breaker buildet. Tager du en for stor mundfuld, breaker du buildet, eller undlader at pushe. Vi oplevede at ved at skubbe builds vi godt var klar over failede, overlader du dine kollegaer/kammerater til et stykke kode der er forkert. Det kan godt være at du selv er klar over at en specifik ting fejler pga en specifik linje du har tilføjet, men det gør din makker ikke nødvendigvis. Og pusher du ikke dine ting ofte, så ender du med at komme langt væk fra basen af din branch. Her er der så stor chance for at du eller andre skal solve merge-conflicts fordi I ender med at arbejde på de samme ting.

Vi vil udvide vores erfaring med problematikken med test af databaser, i samspil med resten af programmet.

Dette læringsmål bygger på en problemstilling vi er stødt ind i mange gange. Vi har formået at komme nærmere en løsning på problemet. Måden vi gjorde det på var ved at benytte virtuelle maskiner. Disse VMs tillader os at opsætte de præcis samme rammer i vores tests, som i produktion. Ved at afvikle de samme scripts i vores VMs som i vores produktionsmiljø, sikre vi os at forholdene er de samme. I vores tests kan vi så afvikle scripts der importerer test-data. Dette betyder selvfølgelig at vi ikke manuelt skal gøre noget før at vores testdatabase er i den korrekte state for at testen kan starte. Udover vores lokale VMs, der kørte testene hurtigt, har vi benyttet Travis. Travis har en bred vifte af databaser, der gør at vores database tests også kører på vores CI/CD chain. Igen ved hjælp af scripts, bliver Travis' databaser sat op præcis som vores lokale tests.

Undervejs i vores læring, afprøvede vi også in-memory databaser. Dette gjorde vi fordi at de kører enormt hurtigt. Og en forudsætning for at man ikke bliver træt af at skrive tests når der udvikles lokalt, er at det går hurtigt. Hagen ved dette var dog at det betød at når vi skubbede vores tests, kørte vi vores in-memory databaser på Travis. På Travis betyder det ikke noget at testene kører lidt langsommere, og vi mener at det giver mere value at lade de "langsomme" tests køre på egentlige databaser, frem for RAM. Et muligt fix til dette problem, kunne være at skrive ens tests på en dynamisk måde, der tillader testen at identificere sit eget miljø, og derefter finde ud af om den skulle connecte på en in-memory eller "ægte" database. Som sagt valgte vi blot at opsætte lokale VMs der spejlede Travis og vores produktionsdatabaser.

En anden læring har været at finde en grænse for hvornår at det ikke længere giver mening at mocke en database væk. Det er klart at når der skrives tests i logic-laget, så giver det ikke mening at ens metoder henter data fra en rigtig database. Her skal der selvfølgelig bare sættes en mock eller en stub ind. Vores erfaring siger os nu at, på et tidspunkt når vi ned til et sted hvor vores komplekse queries er nød til at blive testet op mod en rigtigt database. Selvom den unit vi arbejder på er databaseQuery, er vi nu ifølge tommelfingerreglerne ovre i en integrationstest. Og det er vi fordi vores unit er afhængige af en database udefra. Men ved at sørge for at databaseQuery er en "ægte" unittests, så "skal" vi mocke en database væk.

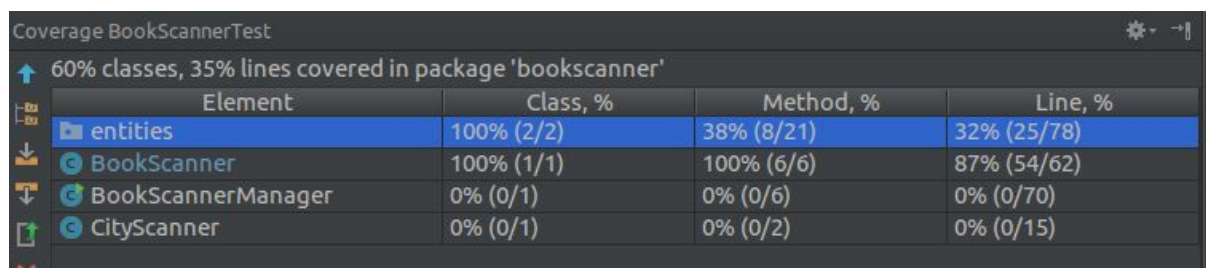
Dette giver for os ingen value, da vi er så langt nede i datatilgangen, at vi reelt set tester vores mock/stub, og selve querien er ligegyldig.

Vi har tænkt at opsætte code-coverage, for at kunne identificere områder af projektet der eventuelt mangler testing.

Da vores projekt blev udviklet test-driven, medførte det ganske høj code-coverage fra start af. Der var dog steder i projektet vi prioriteret højere end andre. Entitets klasser mv. lavede vi ganske få test af, da det som oftest ikke er her fejlene opstår.

Vi valgte i stedet for, at fokusere mest på områder hvor fejlene oftest opstår. Ved at kigge på andres erfaringer, findes som regel 80 % af fejlene i 20 % af koden, nemlig forretningslogikken. Komplex forretnings logik har oftest større chance for at indeholde fejl, og det kan derfor være ekstra vigtigt at koden her er vel gennemtestet.

Vores mest komplekse logik findes i vores bog-skanner. Dette service står for at udtrække bynavne fra bøger, mappe til filer mm. Derfor et sted hvor høj code-coverage er at foretrække.



Coverage BookScannerTest

60% classes, 35% lines covered in package 'bookscanner'

Element	Class, %	Method, %	Line, %
entities	100% (2/2)	38% (8/21)	32% (25/78)
BookScanner	100% (1/1)	100% (6/6)	87% (54/62)
BookScannerManager	0% (0/1)	0% (0/6)	0% (0/70)
CityScanner	0% (0/1)	0% (0/2)	0% (0/15)

Vores code-coverage på BookScanner var 100% på klassen og metoderne. Der var enkelt linjer der ikke var dækket af test i klassen, og her var code-coverage yderst hjælpsom og grafisk viste hvilke linjer der ikke var testet. Dette hjalp os til at lave bedre test cases, da man oftest ikke medtænker alle mulige scenarier for en given metode. På billede nedenfor markeret med rødt, er der et specielt tilfælde hvor en liste længde kan nul. Code-coverage værktøjet viser at der netop ikke findes nogle test for netop dette tilfælde.



```
67         addAuthorToMap(authorID, author);
68         book.addAuthor(author);
69         book.setId(file.replace( charSequence: ".txt", charSequence1: ""));
70     }
71 }
72
73 if(nList.getLength()==0){
74     Author author = new Author( id: "49", name: "Unknown");
75     addAuthorToMap( authorID: "49", author);
76     book.addAuthor(author);
77     book.setId("49");
78 }
79
80 nList = document.getElementsByTagName( $: "dcterms:title");
81 if (nList.getLength() > 0) {
```

Gennem vores projekt har code-coverage været et godt værktøj til både visuelt og statistisk at hjælpe os med at skrive bedre test. Vi havde en general code-coverage på af hele projektet på omkring 80 %. Når det så er sagt, skal man heller ikke stole blindt på code-coverage bare fordi man har 100% coverage af en kompleks metode. Der kan stadigvæk være situationen og scenarier som man ikke har overvejet eller tænkt på. Det

kunne være ting som forkert input mm. som kan skabe uforudsete fejl. Som nævnt ofte "A fool with a tool, is still a fool."

Vi vil gerne stifte praktisk erfaring med begreberne "ility-testing" og general non-functional-testing.

Får at starte vores performance-tests satte vi jMeter op til at køre med en af de jar filer vi havde buildet. I dette build havde vi lavet en klasse der hedder jMeterClass, som indeholder nogle metoder til at tilgå databasen. Vi lavede en test plan ved hjælp af jMeter's GUI, som indeholdte tre thread groups, én for hver database. Ved brug af BeanShell samplers fik vi så adgang til de tre metoder, og kunne køre dem uafhængigt af hinanden, eller oven i hinanden hvis det var det vi søgte.

Det viste sig dog, at vi ikke kunne køre mere end 7 kald til vores neo4j database af gangen. Mongo var en lille smule stærkere med et max på omkring 10. Hvor MySql kunne håndtere omkring 100. Det var tydeligt at se der var noget galt, og vi fandt hurtigt ud af at det var fordi at vores Neo4jConnection ikke korrekt lukker forbindelsen til databasen. Med mongo, løb vores VM tør for RAM, hvilket tyder på det er samme problem som i Neo4j. MySql kunne køre ikke køre mere end 100 af gangen, fordi der var sat en maximum antal af samtidige connections til 100. Det ville vi kunne ændre efter behov.

Fordi vi havde disse problemer, kunne vi ikke lave meget mere på vores tests, da vores VM'er løb tør for RAM. VM'en brød sammen af alle de forbindelser som ikke blev lukket korrekt, og det var bestemt en vigtig ting at få ud af vores test.

Omkring scalability skal vi huske at vores VM er ikke den stærkeste maskine, og i produktion ville man aldrig bruge en VM med 8gb RAM til at holde sine databaser. Så med den nuværende opsætning kan vores maskine håndtere ca. 7 brugere til Mongo og Neo. I produktion ville det være mere sandsynligt at de kunne håndtere 100, da en server ville blive sat op, som måske indeholdt 56gb RAM eller lign. Og som kun brugte kræfter på en af de 3 databaser.

Vi ville gerne have lavet et par forskellige typer af performance-tests, men udover at vores problemer med maskinerne, havde vi svært ved at sætte nogle grænser, når nu der ikke var nogen kravspecifikation. Havde vi haft en liste af krav, så som: "Databasen skal kunne håndtere 50 brugere på samme tid, og hver query må højst tage 10 sekunder" eller lign, havde vi nemmere kunne sætte nogle tests op. Vi ville gerne have lavet tests som tjekkede responstid, altså tiden det ville tage for et enkelt kald i databasen, eller concurrency/throughput. Selvom vi på en måde fik testet concurrency ved at tjekke hvor mange kald der kan være aktive til de tre databaser samtidig, ville vi alligevel gerne have lavet nogle load tests eller lign, som tjekkede antal af kald gennemført, altså throughput.