# Gutenberg rapport Cheesy Choppin - Andreas Larsen & Emil Pilgaard

Vores læringsmål var

For this project we will be combining the Database (Gutenberg Data Mining Project) and Test course projects. As per requested by the project description for the Database, we will be using 2 databases; MySQL and Neo4j.

·        We will be using test driven development, and get some practical experience.

·        We will be performing integration tests and unit test on the databases, mocking away

specific parts. This will require us to build the code, so that mocking is possible.

·        We will make api-test to test our rest api.

·        We will also make use of Continuous integration and travis, so we can get some Practical experience and see if it will improve our workflow.

# Testdriven Development

Igennem dette forløb har vi benyttes os af testdriven Development, forstået på den måde at det kode der blev skrevet skulle testes, inden det blev skubbet af sted til vores Github repository, Det der ellers oftest forbindes med TDD er test first, hvor testen skrives først, og koden bagefter, dette valgte vi dog at afvige fra, da den anden tilgang ligger lidt tættere på vores daglige arbejdsmønster med kodning, og chancen for at vi ville være i stand til at holde os til planen med at få skreven testene var derfor større. Det meste af tiden lykkedes det faktisk også for os at holde os til planen, og gav os et godt workflow uden at vi skulle hen og lave alt for mange ændringer, og fejlrettelser. Når det så er sagt skal det også nævnes at vi et par gange har været lidt tidspresset af nogle deadlines vi havde sat for os selv og derfor var faldet tilbage i vores gamle arbejdsmønstre, med at skubbe koden af sted og krydse fingre for at det virker, hvilket det ikke altid gjorde, så blev der lige rette den ting der fejlet og så blev det skubbet af sted igen. Her efterfølgende kan vi se med alt den tid vi har brugt på at rette fejl, refaktureringer, og få skrevet test til koden, har det at vi ikke tilbage til vores gamle arbejdsmønster haft den modsatte effekt af det som vi prøvede på at opnå, nemlig at vi brugte endnu mere tid. Derfor har vi øjnene op for at TDD er en effektiv arbejdsmetodik som vi bestemt bør arbejde mere hen imod at mestre.

# Integration tests and unit test

Vi har igennem forløbet lavet både unit- og integrationstests, når der er blevet færdiggjort en ny komponent. Projektets backend kan deles op i 2 dele: Data Mining logik og Database-interaktion/logik. Begge disse indeholder logik, der er vigtig at blive testet.   
Data Mining logikken er afhængig af, der findes en ressource, der kan mines fra.  
Der er derfor blevet oprettet ”dummy” .txt filer, som den kan læse fra, så logikken fra de relevante klasser kan blive unit testet.  
Database logikken har været afhængig af at, der både har været en database og at databasen er sat op korrekt. Til testing af database interaktionen, har vi unit testet metoderne i de enkelte database-facader, ved at mocke databaseforbindelsen væk, og lavet stubs af returværdierne. Vi har også lavet integrationstests af database-facaderne, ved at undlade at mocke databasen væk. Unit testsne har været gode til at hjælpe os med at finde huller i vores businesslogik, og integrationstestne har været gode til at finde problemer med mulige afhængighedsproblemer. Testene har også gjort det muligt for os at have et success-kriterie, når der har skullet refaktoreres.   
Testing har naturligvis også haft en positiv indflydelse på design og struktur, især i forhold til afhængighed imellem de forskellige klasser.

# API test

I undervisningen har vi arbejdet lidt med apitest hvor vi har benyttede os af Jmeter,.Til dette projekt har vi i stedet valgt at bruge postman, da vi føler det var bedre egnet til opgaven.

Postman er en editor som er designet til at teste api‘er, og er grafisk og intuitiv at bruge, vi fik derfor meget hurtigt et overblik over om vi overhovet havde kontakt til serveren, og kunne derefter kigge nærmere på om vi så også fik det til bage som vi forventede.

Den erfaring vi har fået med apitest er at der ikke er lige så mange fejl at finde her, når man allrede har fundet mange af fejlene på unit og integration niveauet. De fejl der så har været her på api niveauet har også kun været nogle småfejl, og har derfor også været meget hurtige at rette.

# Continuous integration

Igennem dette forløb har vi benyttet os af Continuous integration, det har vi gjort med det værktøj der hedder Travis, hvilket gør det utrolig let at komme i gang. Log ind på travises hjemmeside med din Github konto, vælg hvilket projekt den skal oprette en byggeserver for, og så er du i gang. Fremover når man så pusher noget til Github bliver det automatisk testet på Travis, og man får besked hvis der var noget der fejlede. Når det så er sagt så begynder der til gengæld at være lidt mere arbejde i det når man skal have testet andet end unit og integration test som ikke mock’er databasen væk. Det gik fornuftigt nok med at få sat Travis op til at bruge mysql, derimod har det ikke lykkedes for os at få Travis til at sætte en Neo4j database op. Vi har via en udvidelse til post kunne køre vores api test igennem terminalen og derfor også på Travis, vi har bare haft problemer med at testene fejler fordi vi ikke har kunnet starte vores api uden det får Travis til at stoppe med alt andet. Hvis vi ser bort fra de småproblemer vi har haft, har det været en succes at bruge et CI værktøj da det har kunnet lave regression testning for os, og sparet os for at vi har skullet køre alle testene igennem hver gang.