INSTITUT FOR DATALOGI
SCIENCE AND TECHNOLOGY
AARHUS UNIVERSITET

Hovedopgave

Master i Informationsteknologi
linien i Softwarekonstruktion

BDD med SpecFlow vs. TDD

af
Martin Skov Nielsen

14-06-2012

Martin Skov Nielsen,
studerende

Henrik Bærbak Christensen,
vejleder

Institut for Datalogi
Aarhus Universitet
Åbogade 34
8200 Aarhus N

Tlf.: 87154112
Fax: 87154115
E-mail: cs@cs.au.dk
http://www.cs.au.dk/da

Resumé

Denne opgave sætter fokus på muligheden for at udvikle pålidelig software med Behaviuour Driven Development (herefter BDD) med værktøjet SpecFlow til C#. Rapporten beskriver erfaringen med BDD under udviklingen af en mobil applikation til Windows Phone 7.1 til registrering og uddelegering af arbejdsopgaver via telefonens kamera og GPS enhed sammenlignet med det forventede resultat gennem udvikling af den samme software gennem TDD.

Indhold

Figurer

1 Motivation

Et hold udviklere hos Avior A/S blev kontaktet af en kunde (herfter omtalt som kunden eller trafikselskabet), som bruger rigtig lang tid på at koordinere opgaver mellem mange medarbejdere via rundsending af emails med opgaver.

Kunden ønsker et system, som kan tildele opgaver til medarbejderne udfra deres geografiske lokationer. Medarbejdere og andre skal samtidig kunne indrapporterer nye opgaver gennem applikationen ved at tage et billede af opgaven. Det kunne i forhold til trafikselskabet være et ødelagt stoppested, læskur eller et hul i vejen.

Da vi hos Avior A/S har meget begrænset kendskab til udvikling af applikationer til mobile enheder, var det svært for os at komme med et kvalificeret tilbud. Denne opgave var derfor en kærkommen lejlighed til dels at undersøge domænet for applikationer til mobile enheder, men også til at undersøge BDD som alternativ til TDD.

Jeg har efter mit møde med TDD i fagpakken PaSOOS fundet stor glæde ved denne agile udviklingsmetode, men synes jævnligt at jeg mangler en forklaring på hvorfor en test er lavet. Hvilket scenarie dækker den over? Med BDD forventer jeg at kunne skrive mine tests som hele use cases. Det giver mig en forventning om en bedre dokumentation af test cases samt fastholdelse af scenariet omkring de forskellige tests.

2 Hypotese/Problemformulering

Kan man ved brug af SpecFlow opnå en situation, hvor kunden reelt kan formulere test scenarier gennem deres egne brugsmønstre? Hvor stort er overhead på konfiguration af SpecFlow [B], et værktøj til udvikling med Behaviour-Driven Development i .Net, kontra det selv at skrive test cases? Er BDD et fornuftigt alternativ til TDD? Eller kan BDD først bruges senere, da det operere på et bredere scope?

3 Ordforklaring

En række af ordene i denne rapport kan kræve en mindre forklaring, for at give mening for alle læsere. Disse ord vil være markeret med **fed** font igennem rapporten. Herunder er en liste med ordene og deres forklaring.

ORD	FORKLARING
Solution	En løsning i VS2010 [A] kaldes en solution. En solution kan bestå af et eller flere projekter
Projekt	Et projekt i VS2010 [A] udgør byggestenene i en .Net løsning. Et projekt kan være af forskellige typer (ex. Web, WCF, Class library). Forskellige projekter har forskellige egenskaber. Eksempelvis har et „Class labrary“ ikke noget grafisk brugergrænseflade og en „ASP.NET Web Application“ er beregnet til at præsentere indhold via en webserver.
DLL el. dll	En dll-fil eller et dynamic link-library er resultatet af et kompileret .Net projekt. Dll filer indeholder en samling af eksverbare metoder og data, som kan bruges i .Net applikationer
Gherkin	Ligesom YAML eller Python, er Gherkin en linje-orienteret sprog, der bruger indrykning til at definere strukturen. Linjeskift afslutter udsagn (kaldet trin) og enten mellemrum eller tabulering kan anvendes til indrykning. Endelig er de fleste linjer i Gherkin starte med en særlig søgeord, som „Given“, „Then“, „Scenario“ (Kilde:Behat.org [Beh]
CustomTool	Et CustomTool er en udvidelse af VS2010. SpecFlow anvender eksempelvis et CustomTool til oversætte features og scenarier til NUnit test kode
CS el. cs	Betegnelse for en fil der indeholder C# kildekode. (Forkortelse af C# source)
Regulære udtryk	Et regulært udtryk, også kaldet et mønster, er et udtryk som beskriver en mængde af tekststrenge som mønsteret passer til. De kan være nyttige når man skal afgøre om en tekststreng tilhører en godkendt mængde, når man vil søge efter mønsteret for en delstreng i en større tekst, eller hvis man desuden vil erstatte dele af en tekststreng med en anden. Regulære udtryk har en syntaks hvor tegn er delt op i almindelige tegn, der betyder hvad der står, og metategn, der har særlige betydninger. Der findes flere dialekter inden for regulære udtryk som afgør hvilke tegn der er specielle. Nedenfor er en dialekt givet som er kompatibel med flere implementeringer som benyttes i moderne programmeringssprog. [wikb]

Extension methods	Extension Methods er en af de nye grundlæggende sprogudvidelser der kommer med C# 3.0. Extension methods bruges i stor stil til at implementere LINQ funktionaliteten i .NET 3.5. Extension methods er statiske metoder, som du kan kalde med standard instansmetode syntaks. Extension methods er udelukkende et kompiler trick. [H]
Spaghettikode	Slangudtryk for et ustruktureret program. Betegnelsen refererer oprindeligt til et program, hvor den rækkefølge, som de enkelte sætninger skal udføres i, ikke går i nogenlunde lige linie fra top til bund i programmet, men bestemmes af et stort antal goto-sætninger, som giver hyppige hop frem og tilbage i programmet. At læse og forstå sådan et program kan sammenlignes med at skulle skaffe sig overblik over, hvor hvert enkelt bånd i en portion spaghetti begynder og slutter. (Kilde:Version2 [MT])
SCRUM	Scrum er en agile udviklingsmetode skabt i starten af 1990'erne, med meget fokus på projektledelse. [wikc]

4 Metode

I første omgang vil jeg bruge tid på at sætte mig ind i teorien bag BDD gennem nogle artikler fundet på nettet. Til den praktiske anvendelse vil jeg bruge dokumentation og screencast fra www.specflow.net. Sammenligningen med TDD vil være baseret på teorien som dels er gennemgået på PaSOOS kurset samt literaturen fra [Chr07] og [Bec03]

Når teorien er på plads vil jeg anvende den i praksis. Da opgaven kun er en ide fra en kunde vil jeg lade mine kollegaer beskrive de features og scenarier, som skal anvendes til udviklingen af applikationen. På baggrund af den erfaring jeg får i forbindelse med udviklingen af applikationen vil danne baggrund for min analyse af BDD som udviklingsmetode.

4.1 BDD - Behaviour Driven Design

Som beskrevet på det officielle website for BDD [Bdd10], så er BDD funderet på tankerne omkring TDD. Et af argumenterne for at udvikle en ny metode er at TDD ikke hjælper udviklerne til at opnå indsigt i hvordan det system de udvikler på skal bruges. Forstået på den måde at fokus ligger på at lave test istedet for at udvikle egentlig funktionalitet. Man kan med andre ord komme til at lave tests, der tester features, som i almindeligt brug aldrig eller sjældent vil blive brugt. Derved bruges resourcerne måske på mindre væsentlig funktionalitet.

Forklaringen på Dan Norths syn på problemet omkring resource forbruget, skal i virkeligheden nok findes i det samme miljø, som gav ham ideen til BDD. Det var studerende som oplevede de udfordringerne. Den normale tilgang til TDD foreskriver at man indvoldvere interessenterne i processen. Eksempelvis med **SCRUM** hvor man ved afslutning af hvert sprint præsentere produktet for de forskellige stakeholders. I et undervisningsmiljø er det min erfaring, at det kan være svært at placere denne stakeholder-rolle. Hvilket kan give fornemmelsen af, at man render lidt i blinde og spilder resourcer.

Et andet argument er at man med TDD kan opnå meget høj code coverage og rigtig mange gode tests, men man kan mangle det forkrommede overblik, som giver helhedsbilledet af et scenarie. Da BDD netop beskriver features og scenarier, så har man testen beskrevet med ord og der er en forklaring på alle tests.

BDD defineret af Dan North 2009

BDD is a second-generation, outside–in, pull-based, multiple-stakeholder, multiple-scale, high-automation, agile methodology. It describes a cycle of interactions with well-defined outputs, resulting in the delivery of working, tested software that matters.

Dan North beskrive på sin artikel „Introducing BDD"[Nor06], hvordan han i sin undervisning af TDD oplevede at de studerende var i tvivl om hvor de skulle starte, hvor meget de skulle teste og hvad de **ikke** skulle teste - *When is enough enough*. Dele af TDD som jeg antager mange slås med, når man i første omgang møder metoden. I følge Kent Beck, så er det en del af indlæringenskurven på TDD.

En anden ting som Dan North nævner, er at test cases ofte skrives som sætninger, de starter som regel med „should"[Nor06] - og hvis ikke, så kan det tages som et tegn på, at der er noget galt med test casen. Altså følger test cases en bestemt mønster. Et sigende navn gør den lettere at spore, når noget fejler.

Typisk opstår fejl - i følge Dan North - når en af tre ting sker:

- Introduktion af en bug
- Den ønskede adfærd er stadig relevant, men er flyttet
- Den adfærd, som testes, er ikke længere ønsket

Som det fremgår af artiklen, så er sidstnævnte en ting, som jævnligt forekommer i agile udvikling. Det er jo netop styrken ved at være agile.

BDD opstilles efter samme formel som en acceptance test: „Som en [role] ønsker jeg, at [feature], så jeg opnår [mål]." En acceptance test omsættes til en feature, og en feature kan så yderligere nedbrydres i scenarier. En feature implementeres som om klassen, der beskriver den adfærd som bliver testet her.

På samme måde tilsvare et scenarie en test case. Scenarier skrives med **Gherkin**-syntax: „Givet [context] når [handling] så sker [resultat]" Denne anskuelse af tests skulle gøre det nemmere for alle deltagere i et projekt at opnå en fælles forståelse for, hvilke krav der er og hvad der bliver testet for.

En pointe her kunne være at i det tilfælde hvor kunden selv opstiller sine features og scenarier, så kommer vi let til at mangle de negative cases. Alle de scenarier, som ikke beskriver „happy path“. Disse scenarier skal vores udviklere stadig kunne spotte.

4.2 Eksperimentet

Eksperimentet vil i første omgang bestå af en meget simpel prototype baseret på en feature beskrevet af en kollega.

Feature 1: Opgave uddeler

- Som tekniker
- ønsker jeg at angive min position
- så jeg får de nærmeste 5 opgaver

Denne feature skal så deles op i nogle scenarier:

Scenarie 1: Der er mere end 5 opgaver

- *Givet* der er opgaver
- *og* bruger er tekniker
- *og* der er +5 opgaver
- *så* når tekniker beder om opgaver
- *returneres* liste med nærmeste 5 opgaver

Scenarie 2: Der er mindre end 5 opgaver

- *Givet* bruger er tekniker
- *så* når tekniker beder om opgaver
- *returneres* liste med tilgængelige opgaver

Scenarie 3: Bruger er ikke tekniker

- *Givet* bruger ikke tekniker
- *så* når bruger beder om opgaver
- *returneres* fejl besked

Disse tre scenarier vil udgøre første eksperiment med SpecFlow

4.3 SpecFlow

Opsætning af SpecFlow [B] i MS Visual Studio 2010 [A] er meget simpel. Download og installer seneste SpecFlow msi-pakke fra GitHub. Msi-pakken sørger for at installere en række templates - som anvendes i arbejdet med SpecFlow - i MS Visual Studio 2010 (herefter VS2010).

4.3.1 Opsætning af en solution i VS2010 med SpecFlow

I modsætning til et MS Test **projekt**, så er der ikke en projekt skabelon til SpecFlow. Så for at komme igang oprettes et „Class library" - den mest basale projekt i VS2010. For at kunne anvende SpecFlow skal der refereres to **dllér** den ene er NUnit frameworket [C] og den anden er selve SpecFlow pakken. Når disse dll'er er på plads, så er vores SpecFlow projektet klar.

Med projektet på plads er vi klar til at indtaste vores features og scenarier.

4.3.2 SpecFlow Feature filer

Specflow installere nogle forskellige fil templates i VS2010, som anvendes til SpecFlow projekter. Den første vi har brug for er „SpecFlow Feature File". Feature filen er hvor vi angiver feature og hvilke scenarier i tekst form [1] udfra de tidligere nævnte formler.

```
1  Feature: TaskHandout
2      In order to keep track of my tasks
3      As a technician
4      I want to turn in my location
5      so that I get the X closest assingments
6
7  @mytag
8  Scenario: Get 5 closest task
9      Given a technician enters a location
10     When request for assignments are sent
11     Then the result should be a list of the 5 closest assignments
12
```

Figur 1: Eksempel på feature fil

Faktisk består feature templatet i VS2010 af 2 filer. Dels den vi selv skriver i, men også en bagved liggende fil, som i virkeligheden indeholder alle vores test cases, der bliver genereret af SpecFlow via et **CustomTool**, når vi gemmer vores feature fil. Det vil sige at hvis vi

ændre i vores feature beskrivelse eller i et scenarie, så bliver test koden også opdateret. Testkoden leder så - via reflection - efter nogle metoder, den kan eksekvere. Disse metoder, som tilsvare de enkelte steps i et scenarie, defineres i en „Step Definition File“.

4.3.3 SpecFlow Step Definition filer

En „Step Definition File“ er en almindelig **cs**-fil. Templaten til filen giver en række eksempel metoder der er markeret med attributer, som tilsvare de **Gherkin**-steps, som er i templaten for feature filer. „Step Definition File“ templaten er mest anvendelig som eksempel kode, når man er begynder med SpecFlow. For de steps som er defineret i templaten passer jo ikke til det projekt, der skal bygges, så det første man gør er at slette metoderne.

For at få de steps, der er defineret i scenarierne i vores egen feature fil, så skal vi have defineret vores egne steps. Vi kunne selvfølgelig selv skrive metoderne og det attributer, som binder metoderne til scenariernes Gherkin-steps, men i stedet bruger vi NUnit til at fortælle os, hvad vi skal bruge.

Vi afvikler ganske enkelt de test, som SpecFlow har genereret for os. Output bliver så en meddelse om at vi mangler step definition samt hvilken metode, der var forventet [2]! Vi kan nu tage metoden med bindingsattributen og kopiere over i step definitions filen. Selve testen, som er indholdet af metoderne i step definitions filen, skrives som en normal unit test, men vi har fået mappet vores steps med NUnit-koden bag vores scenarier på en simpel måde - og minimeret risikoen for tastefejl.

```
TaskHandoutFeature.Get5ClosestTask : Ignored

Given a technician enters a location
-> No matching step definition found for the step. Use the following code to create one:
    [Binding]
    public class StepDefinitions
    {
        [Given(@"a technician enters a location")]
        public void GivenATechnicianEntersALocation()
        {
            ScenarioContext.Current.Pending();
        }
    }
```

Figur 2: Manglende step definitionerne

Hvis vi nu afvikler vores tests, så vil vi naturligvis få fejl på hele striben, ganske som vi ville have fået med en almindelig TDD tilgang.

Til forskel fra TDD har vi dog en række tests at starte med. Vi har en historie at udfylde, en template, et „starting point“, som fortalere for BDD mener er et af de elementer, der mangler i TDD. [Nor06, „Introducing BDD“]

4.3.4 Making tests smarter

Nu er det ikke sikkert at alle vil have de 5 nærmeste opgaver fra listen, som vi ellers har set på ovenfor. Det kunne også være at man ønskede et scenarie hvor man angav antal kilometer (eller andet) til x-antal opgaver. Derfor bliver det tidligere scenarie en anelse statisk og for kunstigt til måske reelt at kunne anvendes. Men SpecFlow har faktisk en måde til at omgå dette problem på. I metodens beskrivelse attribut kan man anvende **regulære udtryk** (herefter RegExp), dette giver os mulighed for at sende parametre med ind til testen som det fremgår af figur [3]

```
[Then(@"the result should be a list of the (.*) closest assignments")]
public void ThenTheResultShouldBeAListOfTheXClosestAssignments(int x)
{
    var taskList = _taskHandler.SortByDistance(_taskHandler.GetTasks(1000, (x+1))).ToArray();
    var secondNearest = taskList[x-2];
    var nearest = taskList[x-1];
    var furthest = taskList[x];
    Assert.IsTrue(secondNearest.Location.Distance(nearest.Location) < nearest.Location.Distance
}
```

Figur 3: RegExp i metodens beskrivelses attribut

Dette gør os i stand til at teste flere scenarier med samme test body uden at ændre i test koden, når først metoden er implementeret med parametre. Det er muligt at skrive mere avancerede RegExp, men for nemhedens skyld anvendes her et wildcard (punktum), der betyder at alle karaktere accepteres, efterfulgt af Kleene-stjerne for vilkårligt antal.

Gherkin syntaksen tillader os ikke at sende avancerede objekter til vores test kode. Gherkin er rent tekstbaseret og uafhængig af vores implementations sprog. Det betyder dog ikke at vi ikke kan sende mere komplekse strukturer over til testkoden, men vi er nødsaget til at stille det op på tabelform (se fig. [5]). De praktiske eksempler kommer vi til senere.

4.4 TDD - Test driven development

Test driven development er som navnet antyder udvikling baseret på test. Testing med TDD udføres som Kent Beck i sin bog Test-Driven Development by Example [Bec03] i en 5-trins metode.

```
1 Skriv test tidligt (uden for meget foranalyse)
2 Kør alle tests og se de nye fejle
3 Lav en lille ændring (implementer så meget ny kode at den nye
      test ikke fejler mere)
4 Kør alle test igen og se dem virke
5 Refaktoriser (for at fjerne duplikeret kode og lign.)
```

Test-Driven Development er opfundet af Kent Beck og Ward Cunningham udfra mantraet om at software er produceret ved at skrive kode. Det handler altså i bund og grund om hurtigst muligt at få produceret noget software som kunden kan forholde sig til i stedet for en masse diagramer og analyser, som de færreste kunder alligevel egentlig forstår. Det er nemmere for en kunde at forholde sig til noget kode, som kan afvikles og giver værdi i en tidlig fase.

Problemer i software opstår ikke i diagrammer eller use-cases, de opstår i koden. Forståelsen for hvordan systemet skal virke er væsentligt nemmere at komme til enighed om, når der faktisk er et system fremfor en masse analyse resultater.

Når man implementerer små dele af systemet løbende, som det foreskrives med TDD, så er koden hele tiden i frisk erindring hos udvilkerne, og kunden kan løbende komme med korrektioner til sine krav. I en klassisk vandfaldsmodel, hvor man analysere hele systemet igennem før man begynder på at kode, vil det ofte være forbundet med store omkostninger at lave rettelser sent i udviklingsprocessen, hvor kunden rent faktisk ser, hvad systemet kan kan.

Ændringer vil betyde en ny analyse fase og en ny udviklingsfase. Det gør det også med TDD, men da analysen måske blot tager 5-10min og kodefasen et par timer, så er der begrænsede omkostninger i forhold til den klassiske model, hvor det sagtens kan tage uger eller måneder.

En anden fordel er at kunden undervejs kan justere sit budget eller behov. Mange kunder ønsker en Ferrari, men har kun råd til en Toyota. Med en agil udviklingsmodel som TDD kan kunden løbende justere sine behov. Det kan måske godt værre at motoren ikke behøver være så stor, men at forlygterne skal være 10.000lux[1] Med andre ord så bliver det nemmere for kunden at gennemskue og prioritere hvad

[1] 1 lux er belysningen af en flade med arealet 1 kvadratmeter, der rammes af lysstrømmen 1 lumen (Wikipedia)

der er „nice-to-have“ og hvad er „need-to-have“. Det kan være meget svært at vide i starten af processen, særligt hvis man kun har en ide om, hvad man kunne tænke sig af systemet.

Endelig handler TDD om at holde fokus. Hvis ikke man holder fokus kan man ende op med et virvar af løse ender, da der ikke er en masse analyse og diagrammer at binde koden op på. Men hvordan sørger man så for at holde fokus? *Fake-it till You make it* - en måde et gøre det på er som nævnt i Reliable and Flexible software [Chr07, p. 20]: „når du har skrevet en test som fejler, så returner en konstant. Når testen køre så transformer den gradvist“.

Erstatningen med en konstant er den mest simlpe form for „fakes“. Det er ikke altid så simpelt, som at erstatte en null-værdi med en integer. Der findes mange typer „fakes“, „stubs“, „test spies“ eller hvad man vælger at kalde dem. Hver i sær har de forskellige egenskaber, men de har det tilfælles at de hjælper udviklerne til at se bort fra den egentlige implementation af en klasse. Der findes også frameworks til at understøtte „falske objekter“ bl.a. „Moq“[WC] som vil blive anvendt i projektet her.

Når man nu ikke bruger særlig tid på analysere og lave design, laver vi så bare **spaghettikode**? Nej (selvfølgelig ikke); dels laver vi noget design og tænker os stadig om. Det handler bare om at holde sig til små steps og få implementeret det hurtigt. Kent Beck mener at kompetente udviklere automatisk vil ramme de rigtige desing mønstre via trianguleringen. En meget pasende beskrivelse på triangulering findes i Reliable and Flexible software explained [Chr07, p. 21] Det er netop en del af den måde flyvemaskiner bruger til at navigere på.

For at holde styr på hvad vi skal have implementeret, så anvender vi en simpel testliste. En testliste kan være en ganske simpel punktopstillet liste med features i systemet. Lidt á la det man finder i BDD's scenarier.

Der er en masse mere interessant materiale omkring TDD, men det er udenfor scopet af denne rapport.

5 Analyse og Resultater

Når man har arbejdet en lille smule med TDD virker BDD som en kærkommen måde at få struktureret sit arbejde på. Det er en metode, som nærmest forudsætter at man har erfaring med TDD. I al fald hvis man vælger at arbejde med SpecFlow.

SpecFlow hjælper med at få løst de problemer, som Dan North lægger til grund for ideen til BDD. Man skriver tingene op på en måde, som sørger for at holde fokus på features i systemet. Features og scenarier kan skrives af andre end udviklere, men det kan test lister i virkeligheden også.

Fordelen ved at skrive det i SpecFlow med **Gherkin**-syntaksen, er at man sikre kæden mellem test cases og test beskrivelser, da de via SpecFlow er linket sammen og testene ganske enkelt fejler, hvis ikke de begge bliver opdateret i forbindelse med ændringer.

Gherkin syntaksen er så simpelt at starte med, at man hurtigt kan komme igang. SpecFlow er samtidig utrolig nemt at installere og så er der „kun“ nUnit tilbage før man er igang med BDD. Såfremt man er bekendt med TDD! Det er værd at bemærke at BDD ikke som sådan er en erstatning for TDD, men mere er en anden vinkel at anskue tingene på.

5.1 Dokumentation

Den litteratur, som jeg har fundet og anvendt om BDD, har jeg på mange områder fundet overfladisk. Det virker som om BDD i manges øjne er en genvej til lykken. Man bliver efterladt med indtrykket af at TDD udvikling er for besværligt og omstændigt, mens BDD klarer det hele meget nemmere.

Jeg havde nok forventet at finde en noget mere grundig beskrivelse eller en reference til en bog eller lignende på Dan Norths website [Nor06]. Men ingen steder finder jeg noget som bare til nærmelsesvis kommer i nærheden af at være lige så beskrivende som litteraturen om TDD. Både Kent Becks bog [Bec03] og Henrik Bærbak Christensens [Chr07] kommer meget breddere rundt om den metode de hver især beskriver.

Baseret på litteraturen er BDD altså en overfladisk eller simplificeret udgave/gren af TDD. Jeg vil da også senere i denne rapport bruge litteratur omkring TDD til at vurdere forskellige egenskaber ved BDD

for at kunne sammenligne metoderne, eller for at kunne forholde mig til om BDD kan anvendes i en given kontekst.

Opsummering

Dokumentation	TDD	BDD
Mængde	+	-
Kvalitet	+	-

5.2 Fællestræk

Der er en lang række egenskaber som BDD og TDD har tilfælles, og der er da heller ingen af de artikler, som jeg har læst, der udråber BDD som en ny teknologi. Selv opfinder Dan North påpeger at det blot er en nu måde at tænke de samme tanker på.

Både BDD og TDD er enige om at automatiseret test af software er en god ide. Begge metoder ønsker at sikre mere pålidelig software med et minimum af defekter. Samtidig er temaet om fleksibilitet gennemgående og ideen om at det er software og der virker og kører, som giver værdi for kunden. Disse fakta er jo på ingen måde overraskende, da BDD som Andrew Glover[Glo] meget præcist beskriver: „It's just an evolutionary offshoot of TDD in which the word *test* is replaced by the word *should*."

Derfor er det heller ikke overraskende at begge metoder anvender triangulering til at nå det endelige mål. Men det er faktisk en ting, som man skal holde lidt øje med, når man tager fat på BDD. Både en samtale med en kollega og mine egne eksperimenter viste at man let kommer over på et spor, hvor man forsøger at implementere det scenarie, der er beskrevet uden at bruge triangulering.

Det kan naturligvis ikke ligges udviklingsmetoden til last, men ikke desto mindre finder jeg det interessant at den kollega, som jeg snakkede med om mente at BDD manglede helt trianguleringsdelen. Dette har dog siden vist sig ikke at være tilfældet.

Vores samtale fandt sted før jeg begyndte mine eksperimenter og kun havde læst ganske lidt på teorien. Vi var på daværende tidspunkt enige om, at det virkede som en - efter vores opfattelse - mangel i metoden, som netop mindskede pålideligheden i softwaren, ganske som en flyvemaskine med kun et pejlepunkt formentlig heller ikke flyver ligeså præcist som et der anvender triangulering.

Et andet koncept som man er enige om hos både BDD og TDD folk er at testen skrives først. Men som Glover også nævner i sin artikel, så ser mange det som en taktil handling at lave tests, og det konceptuelle framework ved ideen vil af nogle udviklere opfattes som en selvmodsigelsen, da de anser tests for noget konkret istedet for en abstrakt aktivitet. For at imødekomme dette problem med forståelsen, så vælger man i BDD at „omdøbe“ *test* til *skulle*. Ideen med at bruge ordet *skulle* fremfor *test* er at det bliver mere sprogligt flydende.

Der er ingen tvivl om at formulering i BDD's scenarier er mere beskrivende rent sprogligt end en test liste fra TDD. Det ses blandt andet i Glovers påstand om at det er nærmere måden vi taler på at sige: „shouldThrowExceptionUponNullPush“ end „TestThrowsExceptionUponNullPush“.

Jeg mener man er nødt til at se på hele scenariet, hvis man egentlig skal have nogen glæde ud af den sproglige del af BDD. Langt de fleste mennesker vil formentlig være i stand til at abstrahere fra om *test* i det foregående eksempel er sådan, som man ville tale og alligevel forstå hvad testen handler om. Hvis der er en fordel i det sproglige kommer det først til udtryk, når man sætter et helt scenarier sammen.

Det er klart at efter som BDD udspringer af TDD, så har de to metoder flere fællestræk end dem jeg har valgt at nævne her. Jeg har valgt de punkter ud, som jeg anser for at være de mest væsentlige. Det er de punkter hvor metoderne ligner hinanden, men alligevel er forskellige i anskuelse eller ordvalg.

Opsummering

Fællestræk	TDD	BDD
Fleksibilitet	+	+
Lang foranalyse	-	-
Test før kode	+	+
„Should“	-	+
„Test“	+	-

5.3 Design Patterns

Et af de punkter, som skiller de to metoder ad - i følge Dan North, at man med TDD ikke får anvendt de rigtige design patterns. Når man ikke har et samlet billede af, hvordan en bestemt del af systemet skal opfører sig.

Kent Beck mener derimod at udviklere, efterhånden som de får lidt erfaring med TDD, ganske enkelt vil implementere de rigtige design mønstre på grund af deres erfaring med systemudvikling. Det vil være en naturlig del af refaktoreringsdelen i TDD, at man indbygger de mønstre som passer ind i programmet.

Jeg tvivler ikke på at det for personer - som Kent Beck, vil komme som en naturlig del af udviklingsforløbet, men de elever/studerende, som gav Dan North inspiration til BDD, er næppe så erfarende med hvilke mønstre, der passer bedst til en given situation. Så på den baggrund vil jeg nok hælde til at være enig med Dan North. Beck's holdning til at møsntrene er et naturligt trin kommer måske også af hans definition på design mønstre:

> Et design mønster er en særlig prosa form for optagelse af design information, således at konstruktioner, der har fungeret godt i fortiden kan anvendes igen i lignende situationer i fremtiden [Chr07]

Betyder det så at man skal være en super erfaren og knald dygtig udvikler for at kunne anvende TDD? Det vil naturligvis altid være en fordel at være teoretisk dygtig, men selvom TDD foreskriver et minimum af analyse, så betyder det jo ikke at man ikke må stoppe op og tænke sig om. Dette illustreres meget fint i Reliable and Flexible software by Example [Chr07] kapitel 3.2.

HBC lister mulighederne op for de valg man kan tage, når man eksempelvis konfronteres med krav om ny flexibilitet.

```
1 Source-code-copy: Kopier den eksisterende kode og modificer den
2 Parametresiret: Brug en parameter til at fortælle hvilken
      version, der skal bruges
3 Polymorphi: Fælles egenskaber i superklassen og særtræk i
      subklasserne
4 Compositionelt design: Uddeleger ansvaret til specialisterne.
      Placer det nye krav i en ny klasse ("'Let someone else do
      the dirty work"')
```

HBC har en grundig gennemgang af for og ulemper ved de forskellige muligheder, så jeg vil ikke grave dybere i det her, men jeg har blot taget det med, for at illustrere at man også med TDD må stoppe og tænke over hvad der er den bedste vej fremad.

Opsummering

Design patterns	TDD	BDD
Tænkes ind fra starten	-	+
Kommer med erfaring	+	-

5.4 De tomme tests

En erfaring som arbejdet med BDD og SpecFlow har vidst er at man reelt kan komme op med en række ubrugelige tests. Tests hvor der ikke er noget at teste for, som eksempelvis denne:

```
1       [Given(@"a task has been reported")]
2       public void GivenATaskHasBeenReported()
3       {
4          //TODO: See if the statement below can be turned in
               to a resonable test:
5          //Now for this test is to give any meaning I would
               have like to replace "task" with a regex and
               validate that the task was instanceof...
6       }
```

Et lignende eksempel er, når man gerne vil assert'e på en property på et bestemt objekt:

```
1 And I am an administrator
```

Det jeg i virkelighed gerne vil have her er et `User`-objekt, som jeg kan lave en assert på der hedder noget á la `Assert.AreEqual(UserType.Administrator, myUserObject.UserType)`.

Jeg kunne naturligvis skrive objektet på tabelform og mappe tabellen i test casen, men det giver bare ikke rigtig den ønskede værdi i testen. Da jeg jo i virkeligheden ligeså godt kunne lave et nyt objekt og sætte `UserType`-property'en udfra strengen „administrator"

Opsummering

Tomme test	TDD	BDD
Indeholder test kode uden antagelser	-	+
Automatisk genererede test	-	+
Automatisk genereret test kode	-	-

5.5 Intuitivt med Gherkin

En af ideerne med BDD er at specifikationer oversættes direkte til test uden egentlig at være behandlet af udviklere og oversat til testkode fra use-case eller lignende. Derfor fik jeg hjælp fra et par kollegaer, som ganske vidst er habile systemudviklere, men ikke har kendskab til hverken **Gherkin** eller BDD i anden form, til at skrive et par af de scenarier og features, som systemet skulle indeholde. Ville jeg kunne bruge SpecFlow direkte på det de kom med? Jeg havde lynhurtigt introduceret dem for den allermest basale syntaks i Gherkin, så det de kom med var formateret rigtigt.

Årsagen til at det kun var den mest grundlæggende **Gherkin** syntaks, var at en af de problemer/udfordringer som vi oftest løber ind i vores virksomhed er, at når tingene tager for lang tid så vender vi tilbage til det vi plejer. Derudover forestiller jeg mig at marketing personale hurtigt vil stille sig på bagbenene, hvis man forsøger at lære dem alt for meget om, hvordan de skal skrive testbeskrivelserne. Så hvis BDD skal have en chance for at virke i et bredere forum end kun blandt udviklere, som i forvejen kan skrive koden, så anser jeg det for vigtigt at det ikke kræver alt for megen undervisning for at få folk igang. Så efter en lille times snak om hvordan man skulle skrive features og scenarier, overlod jeg resten til mine 2 kollegaer.

Det skulle vise sig at syntaksen ikke havde voldt dem nogen problemer. Ideen om at „hvis jeg er <x>, så når jeg gør <y>, skal det medfører <z>“ havde været en meget intuitiv måde for udviklere at tænke på. De var begge enige om at det i virkeligheden minde meget om at skrive pseudokode bare med nogle faste nøgleord, som vi alligevel ofte vil anvende til pseudokode for løkker og betingelser.

De scenarier, som mine kollegaer havde skrevet kunne hurtigt overføres til „Feature filer“ og dernæst videre til „Step Definitions“. Så alt så ud til at være et mirakel indenfor systemudvikling og testing. Men - når noget virker for godt til at være sandt, så er det som regel heller ikke sandt.

Da jeg begyndte at formulere tests udfra de beskrivelser jeg havde fået, så viste det sig at det hele ikke var helt så nemt.

For eksempel så optrådte der mange steder objekter, der kun foreskrev typen af objektet, men ikke nogen egenskaber som man kunne lave brugbare test udfra.

```
15  Scenario: Approved task
16      Given the task has the state approved
17      And it is one of the 5 nearest task
18      Then it should be present
19      When I click get 5 nearest task
```

Figur 4: Scenarie direkte fra skribent

I figur [4] vildet eksempelvis ikke være særlig brugbart at vide om et objekt af typen „Task“ har en attribut eller tilstand, der hedder „Approved“. Hvis man ikke ved hvad „approved“ dækker over. Er det en `enum` eller blot en bolsk værdi? I et system med flere hundrede „Task“-objekter, er det ikke særlig brugbart, hvis man ikke ved, hvilken task der er. Derfor er nedenstående eksempel ikke videre brugbart:

```
Given the task has the state approved
```

Faktisk har eksemplet to uhensigtsmæssigheder. Den første er selvfølgelig den med det ubestemmelige objekt. Den er er at ingen ved om „state approved“ oversættes til en bolsk værdi! Det er måske nok det mest oplagte for os der kender til systemet, men der kunne refereres til en bestemt tilstand i systemet. Derfor skal man i virkeligheden allerede her have en afklaring mellem udvilkerne og de personer, som skriver testene. For ud af ordene er det ikke klart hvad der menes.

En mere nøjagtig beskrivelse kunne være som følger:

```
Given the property approved on the task with id #7 is true
```

Her er det klart at vi snakker om en egenskab på et „Task“-objekt med id nummer 7, der skal være sat til „true“.

Det betyder altså at vi er nødt til at instruere testskriverne lidt mere nøjagtigt om hvordan en test eller et scenarie skal formuleres, hvis vi skal kunne lave asserts på det.

Det er væsentligt at man i sine scenarier overvejer hvilken adfærd, man ønsker at afdække. Figur [4] kan godt ses i et bredere scope. I det givne scenarie kan man naturligvis argumentere for at alle tasks, som kan søges frem skal være „approved“ og at id'et derfor ikke er så vigtigt. Men er den del af scenariet så i virkeligheden med til at sløre

hvad der er vigtigt i testen? Eller er den med til at give en bredere forståelse af scenariet som en slags use-case? I det givne eksempel vil det nok være det sidste, men det betyder jo, at man er nødt til at tænke over hvilke konsekvenser, det kan have om man:

a) Tager objektet med i scenariet

b) Lader det være ubestemt

c) Forsøger at lave assert på egenskaber ved objektet

Opsummering

Intuitivt med Gherkin	TDD	BDD
Use-cases kræver syntaks	-	+
Use-cases skal forholde sig til objekter/strukture	-	+
Direkte use-case konvertering til kode	-	+

5.6 SpecFlow og avancerede objekter

Så længe man arbejder med objekter, der er bygget af simple typer, så virker SpecFlow smart og simpelt. Dette bliver dog pillet noget ned, når man vil til at anvende mere avancerede typer af objekter.

I starten af forsøgene opererede jeg med et `User` objekt med tre string properties:`UserName`, `UserEmail` og `UserType`. I figur [5] er bruger objektet delt op i tabelform, så alle properties er udstillet i scenariet. Hvis man anvender SpecFlow Assist Helpers [D], så kan via **extension methods** mappe tabeldata til objekter.

```
6   @ApproveTask
7   Scenario: Approve a task
8       Given a task has been reported
9       And I am a user of the following type
10      | userName | userEmail | userType      |
11      | Any      | Any       | Administrator |
12      When I checke approved
13      Then the task is marked as approved in the TaskList
```

Figur 5: Scenarie med objekt på tabelform

Det fungere fornuftigt så længe vi har med simple typer at gøre. Men hvad sker der, hvis vi vælger at implementere `UserType` som en **`enum`**? Så kan SpecFlow Assist Helper ikke længere hjælpe. Nu bliver vi nødt til at lave mapningen manuelt udfra streng værdien i tabellen fra scenariet. Det betyder imidlertid en øget kompleksitet i test koden,

da man enten øger mængden af kode i selve testen med betingelser og løkker, eller man begynder at introducere hjælpefunktioner for at håndtere mapningen fra streng til objekt.

Både det faktum at objekter i scenariet skal skrives på tabelform [5], og problemet med mapning i testen gør at SpecFlow kan blive noget trættende at arbejde med. Mapningen behøver dog kun foregå på de properties på objektet som man ønsker at bruge. Figure 5 viser en tabel med tre properties fra `User`-klassen. Egentlig skal vi kun bruge `UserType`, så `userName` og `userEmail` er i virkeligheden overflødig i denne kontekst. Det hjælper lidt for de personer, som skal skrive scenarierne. Hvis man krævede at alle properties skulle mappes, var de jo nødsaget til at have en stor indsigt i domænet og objekternes struktur. Et issue som dog består - i forhold til kendskab til objekt strukturen - er at for at lave en fornuftig mapning, så skal kolonnenavnene - i tabellen fra scenariet - gerne stemme med de properties, som findes på det objektet den skal mappes til.

Det er naturligvis en fordel, at de personer som definerer hvilke funktioneret system skal índeholde kender til domænet. Men der er stor forskel på at de kender til forretningsdomænet og til at de kan gennemskue (og overskue), hvordan udviklerne mapper forretningsbegreber til kode. Når man fastholder principperne om at skrive tests først og kode bagefter, så betyder det „testskriverne“ definere mapningen af objekter. Det medfører endnu en et problem omkring komplekse objekter. Vi er alle i vores hverdag vant til „objekter“ som indeholder andre „objekter“. I projektet her vil det for eksempel være nærliggende at en bruger har en opgave. En opgave kan så igen have en lokation. Med **Gherkin** syntaks vil dette scenarie være meget komplekst at beskrive. Eksemplet i figur [6] indeholder kun et „task“-objekt, men giver alligevel et billede af en forholdsvis bred struktur, som skal mappes i testen! Som det fremgår af testkoden i figur [7], så er det langt fra en triviel opgave. Det er absolut et punkt, hvor der er mulighed for at lave fejl i testkoden.

Hvis vi ser på muligheden for at understøtte et lignende scenarie med TDD [8], så vil vi have beskrivelse i et dokument (eller kommentar) og altså afkoblet fra kode og syntaks. Det betyder naturligvis, at vi ikke får hjælp af et værktøj til at håndtere opdateringer. Omvendt er vi også fri til at anvende vores objekter i tests på samme måde som de vil være anvendt i produktionskoden. Eksemplet i figur [8] er meget forsimplet og blot med til at illustrere uafhængigheden mellem beskrivelse og kode, plus at der ikke skal foretages nogen mapning af objekterne, da de også er uafhængige af beskrivelsen.

```
Scenario: Technician has task on location
    Given I am a Technician "bob@nowhere.com"
    And I have task no.7
    Then the following datastructure should exist
    | User.id         | UserName         | Task.id | Approved | Location.id | Latitude          | Longitude          |
    | bob@nowhere.com | Bob Lee Swaggart |         |          |             |                   |                    |
    |                 |                  | 7       | True     |             |                   |                    |
    |                 |                  |         |          | 12          | 55.411173226724891 | 11.351395325062256 |
```

Figur 6: Complex object table

```
[Then(@"the following datastructure should exist")]
public void ThenTheFollowingDatastructureShouldExist(Table table)
{
    List<ITask> tasks = (from tableRow in table.Rows
                let user = new User
                            {
                                UserEmail = tableRow["User.Id"], UserName = tableRow["UserName"]
                            }
                select new Task(int.Parse(tableRow["Task.Id"]), new Location()
                                                                    {
                Longitude = double.Parse(tableRow["Longitude"]), Latitude = double.Parse(tableRow["Latitude"])
            }) {Owner = user}).Cast<ITask>().ToList();

    //Select single hence there can be only one task with that id
    ITask assertionTask = tasks.Single(x => x.Id == 7);
    Assert.IsNotNull(assertionTask);
    Assert.AreEqual("bob@nowhere.com", assertionTask.Owner.UserEmail);
    Assert.AreEqual(55.411173226724891, assertionTask.Location.Latitude);
    Assert.AreEqual(11.351395325062256, assertionTask.Location.Longitude);
}
```

Figur 7: Complex object table mapning

```
/*
 * Task no. 7 with longitude 11.351395325062256 and latitude 55.411173226724891
 * has the owner with email address "bob@nowhere.com
 */

[Test]
public void Task7IsOwnedByBob()
{
    //TODO: Task should be loaded from repository
    var task = new Task(12, new Location()
    {
        Latitude = 55.411173226724891,
        Longitude = 11.351395325062256
    })
                {
                    Owner = new User("Bob Lee Swaggart", "bob@nowhere.com")
                };

    Assert.AreEqual("bob@nowhere.com", task.Owner.UserEmail);
}
```

Figur 8: Complex object med nUnit

Opsummering

Avancerede objekter	TDD	BDD
Kræver særlig test kode	-	+
Skal mappes før test	-	+

5.7 SpecFlow værktøjsudfordringer

I min vurdering er det største problem ved BDD ikke i tanken. Problemet opstår, når vi skal kommunikere med computeren. Når vi laver use-cases eller test lister, så anvender vi tit objekter i ubestemt form. Det kan vi godt oversætte til **Gherkin** synktaks og videre til SpecFlow. I al fald så længe vi ikke skal lave antagelser omkring dem. Men hvis vi begynder at skulle lave antagelser på objekterne møder vi hurtigt problemer.

Vi kan eksempelvis tage følgende eksempel på et punkt fra en test liste: `- en opgave skal have status "'approved"' før den kan lukkes`

Det ville kunne give en test, som den i figur [9]

```
31      [Test]
32      public void ATaskMustBeApprovedBeforeFinished()
33      {
34          var task = _taskHandler.GetTasks(int.MaxValue, 1, null).Single();
35          if(task.Approved)
36          {
37              task.Finished = true;
38          }
39          Assert.IsTrue(task.Finished);
40      }
```

Figur 9: Simpel test på objekter

Umiddelbart kunne det oversættes til **Gherkin** [10]:

```
18  When I want to close a task
19  Then the task should be approved
```

Figur 10: Gherkin sample

Ved første øjekast ser det jo fint ud. I figur [10] fremgår det at vi snakker om en „task“ og noget om „approved“, så det er jo magen til det, som står i testlisten ovenfor. Men nu skal det så oversættes til en SpecFlow Step Definition fil. Med det nævnte eksempel går det også

fint. Der bliver ganske vidst to tests ud af det, og det er her synes jeg vi ser en svaghed. For når dette simple scenarie (som faktisk kun er et udsnit) resulterer i to test, så eksploderer mængden af test kode lynhurtigt.

En af de fordele som nævnes i SpecFlows dokumentation[Spe] er, at man kan genanvende sine tests ved at bruge regulære udtryk i metode-attributterne som wildcards til variabler. Men i den mængde af testkode som genereres, så bliver det hurtigt meget vanskeligt at overskue, hvornår det er praktisk at gøre det. Derudover er det i strid med TDD princippet om evident test og evident data [Bec03, p.130-131]. Fordi man ikke kan læse ud af testen hvad den tester for.

Jeg har i mine forsøg, som med vilje er holdt simple, forsøgt at bruge disse wildcards, som beskrevet i dokumentationen. I nogle tilfælde var det rart at man ikke skulle lave nye tests, som lignede eksisterende til forveksling. Andre gange tog jeg mig selv i at gå tilbage og rette mine scenarier til så de passede til de mere generiske tests. Nu er det selvfølgelig tilladt at blive klogere undervejs i en udviklingsproces, og særligt i en opgave som denne er det en del af læringsprocessen. Der var da også to forskellige grunde til, at jeg gjorde det.

For det første blev jeg klogere på hvordan man kunne opbygge et scenarie og kunne derfor forbedre det eksisterende. Den anden grund var, at når jeg brugte de regulærer udtryk, så passede de eksisterende scenarier ikke. Men nu er det ikke meningen, at man skal rette i scenarierne på den måde, så det skråplan fik jeg hurtigt stoppet igen. Dog havde jeg nu lært, at man nøje må overveje, om der er gavn af en generisk test.

Opsummering

Værktøjsudfordringer	TDD	BDD
Generiske tests	-	+
Mulighed for input parametre	-	+

5.8 Udvikling med TDD og nUnit

Der findes forskellige værktøjer til at lave automatiserede tests. I denne opgave har jeg valgt at anvende nUnit, som er pendant til jUnit, der anvendes af HBC i hans bog [Chr07]. I nyere udgaver kommer MS Visual Studio ganske vidst med MS Test Suite. Det fungere på nogenlunde samme måde som nUnit, men da jeg føler mig mere fortrolig med nUnit, og bedre kan lide den feedback jeg kan få herfra, så

har jeg valgt at blive ved med at anvende det. Derudover gør det også min sammenligning af teknologierne lidt nemmere, da SpecFlow også anvender nUnit.

Som foreskrevet i teorien starter vi med en test liste over de features, som lige ligger nærmest, når vi tænker på hvad systemet skal kunne.

```
1 Der skal kunne hentes en liste med de 5 nærmeste opgaver
2 Der skal kunne finde en liste med åbne opgaver inden for 10km
3 Der skal kun oprettes en opgave
4 En opgave skal have en lokation
5 Opgaver kan kun godkendes af administratorer
```

Præcis som med SpecFlow, så skrives testen først. Der er ikke nogen værktøjer at tage hensyn til, så navngivningen på testen er mere fri. Det er dog god kutyme at navngive sine test, så de er beskrivende for testen, hvilket ofte medføre lange metodenavne, der normalvis ikke er best practice.

Best practice og erfaring er nogle af grund elementerne i TDD. Hvilket netop er grunden til at BDD overhovedet er kommet frem. For det kan være en tung og noget træg indlæringsproces at komme i gang med TDD. Kent Beck ligger da heller ikke skjul på at rutine og erfaring er vigtigt for at opnå gode resultater med test-driven development. For eksempel var vi i min gruppe på PaSOOS-kurset i begyndelsen bekymrede over den manglende analyse, som ellers kan hjælpe med at afdække hvilke design mønstre vil fungere bedst. Kent Beck siger dog at det ikke er et problem og man med rutine, gennem refaktoriseringen, vil ende op med at vælge de mest fornuftige mønstre.

En test som `ApproveTaskAsAdmin` giver naturligt anledning til at udtænke en ny test. For hvad så den anden vej rundt? Så skal der naturligvis være en test, som afviser en godkendelse fra en hver anden bruger. Hvilket giver anledning til en ændring i test listen. Jeg føjer simpelthen en ny test til `ApproveTaskAsNonAdmin` [11].

Det forekommer mig, at netop denne test nemt kunne være misset med BDD. Jeg mener ikke, at det er nær så intuitivt, at skrive et scenarier, som tager den negative sti. Det baserer jeg primært på min egen fornemmelse med BDD, men også på at eksemplerne ikke findes i den litteratur, jeg har været igennem.

```
44          [Test]
45          public void ApproveTaskAsNonAdmin()
46          {
47              var mUser = new Moq.Mock<IUser>();
48              mUser.Setup(x => x.UserType).Returns(UserType.Technician);
49
50              var taskHandler = new TaskHandler();
51
52              Assert.Throws<AccessViolationException>(
53                  () => taskHandler.ApproveTask(_taskContext.MockedTask,
54                                                _taskContext.DataAccessor,
55                                                mUser.Object)
56                                                );
57          }
```

Figur 11: Exception test

I forbindelse med PaSOOS kurset var vi også igennem både white-box og black-box testing samt ækvivalensklasser. Netop for at kunne afdække hvilke test, som var nødvendige. I de beskrivelser jeg har læst omkring BDD, der nævnes ingen af dem! Der nævnes blot, at det med TDD kan være svært at vide, hvornår det er nok. Derfor er scenarierne gode til at afgrænse det. Jeg mener TDD fokuserer på at sikre reliabilty i koden, hvor BDD går på at opnå accept af at funktionaliteten er tilgængelig, selvom den måske ikke er pålidelig.

Opsummering

TDD udvikling	TDD	BDD
Fokus på simple test	+	-
Adskilte test	+	-
Minimeret risiko for ripple effect	+	-
Fokus på systematisk testing	+	-

5.9 Udfordringer med BDD

En af mine indledende hypoteser var at det ville være værktøjet, som spændte ben for metoden. Forstået på den måde at det ville være opsætning og konfiguration af værktøjet - altså SpecFlow - som ville være dræbende for BDD metoden. Det har dog, som det fremgår af afsnit [4.3], vist sig at være overraskende simpelt at komme igang. Download og installation forløb gnidningsfrit, og for en udvikler, som jeg selv, der er bekendt med at arbejde med pakker (som .jar fra Java eller **.dll** fra .Net), så var det intet problem at få refereret de rigtige libraries. Derefter var man faktisk igang. Derfor kunne jeg hurtigt forkaste denne hypotese, da det i alt tog under 15min. at komme igang.

Da jeg i mine forsøg med BDD - eller måske rettere SpecFlow og **Gherkin** - flere gange mødte problemer, som ikke var nævnt i nogle af de instruktioner og artikler, jeg har gennemgået, syntes jeg det ville være interessant at se på hvor det i virkeligheden gør ondt på BDD som metode.

Selve tanken om at udnytte use-cases - eller scenarier for nu at blive i BDD terminologien, virker fornuftig. Når der alligevel laver use-cases, virker det fornuftigt at udnytte dem til at specificere vores test ud fra. Problemet opstår bare i det øjeblik, at vi ikke længere følger en formel model. Når vi overlader systemudviklingen til det, vi lige kan tænke os frem til, så er der efter min vurdering alt for stor risiko for at overse noget.

Men er det så risikoen for at der kommer til at mangle noget, som er den vigtigste svaghed ved BDD? Ikke efter min mening. Som i så mange andre sammenhæng omkring softwareudvikling, så er kommunikation en vigtig faktor. Hvis de forskellige interessenter er i stand til at få afklaret alle krav til systemet - eksempelvis gennem nogle views [HBCH04], så vil man formentlig kunne dække de fleste huller.

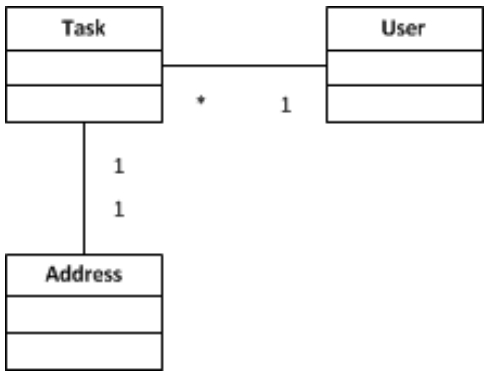
Det betyder naturligvis, at man ikke kan overlade al scenarie skriveriet til marketing folk, og at alt ikke kan klares med SpecFlow. Men der er jo ingen forbud i BDD mod at have flere arbejdsområder til at samarbejde omkring udarbejdelsen af scenarierne.

Det er ved mange lejligheder bevist at automatiseret test er med til at minimere mængden af fejl i software. Så derfor vil jeg undlade at gå ind i en dybere analyse af det her.

Jeg ser det helt klart som et væsentligt problem med BDD - i al fald med SpecFlow, at der genereres så meget kode, som ikke rigtig gør noget andet end at linke scenariet og testen samme (se [5.4]). Det er også i strid med Kent Becks mantra om „Clean code that works“. Bevares Beck taler om produktionskode, men testkoden er jo ikke meget værd, hvis den er komplet uoverskuelig.

Et andet punkt som jeg anser for at problematisk er de avancerede objekter - de objekter som indeholder andre objekter. De er meget vanskelige at beskrive med **Gherkin**. Det bliver så ikke nemmere, hvis objekterne er mere komplicerede og består af flere niveauer (se figur [12]). Det kan lade sig gøre med „nestede“ tabeller, men det gør

scenariet uoverskueligt og komplekst. Det skal dog i den forbindelse nævnes, at jeg ikke har fundet brug for at lave så komplicerede scenarier i mine forsøg, for at opnå hvad jeg ønskede. Men alligevel vurdere jeg at det kan være problematisk at arbejde med komplekse objekter (se 5.6)



Figur 12: Simple component & connector diagram

Det virker smart at man kan lave generiske tests ved at parametrisere sine test cases. Der er dog en vis risiko for at øger kompleksiteten i selv testen og dermed kan medføre, at der opstår bugs/fejl i test koden. På det område må TDD siges at have en fordel, da man her tester en fuktionalitet i en test og ikke genanvender den med andre input.det kan selvfølgelig forsvares, hvis man tænker på at genbruge en test til at teste en given funktionalitet med grænseværdier fra en ækvivalensklasse, men det vil ikke fremgå af testen, som det typisk vil i en TDD test. Det bør så fremgå af scenariet, men jeg synes man skal være påpasselig med det. For når det gælder test cases, så handler det om at have fokus på *KISS*-princippet - *Keep It Simple Stupid*.

Opsummering

Udfordringer	TDD	BDD
Opsætning	-	-
Maintainability	+	-
Dårlig understøttelse af komplekse strukture	-	+

5.10 TDD efter BDD

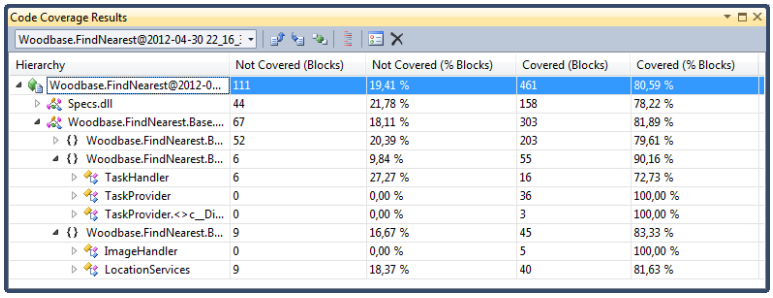
En af de ting som jeg ønskede at undersøge i denne opgave var om BDD havde nogle huller, som ville være blevet lukket, hvis man havde anvendt TDD til udviklingen. Min hypotese var at BDD ville dække koden ligeså som TDD. Det har naturligt givet mig nogle udfordringer at være alene om opgaven i denne situation. Da jeg jo uanset hvor jeg startede med at lave test selv ville skulle lave både TDD test og BDD test. Frygten var at det ville være svært at adskille de to metoder, således at man tog for mange test med på grund af hukommelse eller at man undervejs i udviklingsprocessen foretog nogle valg, som man fandt smartest i et givent scenarie.

Af praktiske årsag valgte jeg at starte med BDD, da jeg forventede at skulle bruge mest tid på denne metode, da det var nyt stof for mig. For at dæmme nogenlunde op for min bekymring om at lade mine TDD test blive præget af hukommelse, så valgte jeg at vente et par uger med at starte på TDD udviklingen.

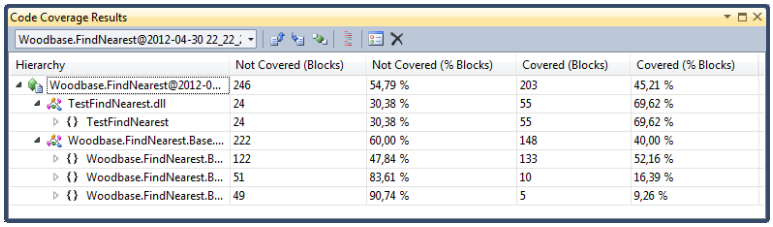
Min udvikling med TDD forsøgte jeg at starte på så blankt „papir" som muligt. Men da jeg i virkeligheden ønskede at teste eksisterende kode, så var jeg nødsaget til at skelle til min navngivning fra BDD. Selve processen med TDD er beskrevet i afsnit [5.8]

For at vurdere om den ene metode var bedre end den anden måtte jeg have nogle parametre at måle på. En parameter som jeg kunne måle på var code coverage. Hypotesen var at TDD ville have markant høje dækning af koden end BDD. Men noget overraskende, så viste det sig at BDD faktisk havde en ret høj grad af coverage, som det fremgår af figuren [13]. Det er også her at årsagen til den forholdsvis lave grad af coverage i figur [14] skal findes. For da jeg så hvor høje procenterne var for BDD valgte jeg at fokusere på at søge efter defekter i stedet.

Kunne jeg med TDD finde fejl i min kode, som jeg ikke havde lokaliseret med BDD? Og måske endnu mere interessant - kunne det påvises, at det var på grund af BDD, at fejlen var blevet overset? Min hypotese var at den nok ville blive svært. For at få de mest ensartede resultater, så var jeg nødsaget til at have nogle ensartede test. Derfor tog jeg udgangspunkt i en test fra mit BDD projekt, som jeg mente virkede: `public void ThenTheResultShouldBeAListOfTheXClosestAssignments(int x)` [15], som - overført til TDD projektet - kom til at se ud som figur [16]. Faktisk er nUnit testen en kopi af SpecFlow step'et som har fået



Figur 13: BDD Code coverage



Figur 14: TDD Code coverage

fjernet den variable del.

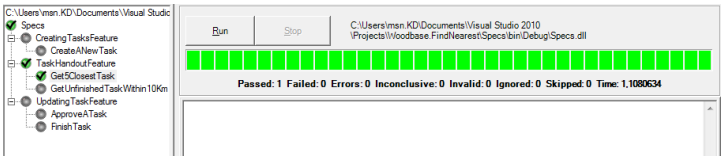
```
36  [Then(@"the result should be a list of the (.*) closest assignments")]
37  public void ThenTheResultShouldBeAListOfTheXClosestAssignments(int x)
38  {
39      const int intmax = int.MaxValue;
40      var tasks = _taskHandler.SortByDistance(
41                  _taskHandler.GetTasks(intmax, (x + 1), true));
42      var taskList = tasks.ToArray();
43      var secondNearest = taskList[x-2];
44      var nearest = taskList[x-1];
45      var furthest = taskList[x];
46      Assert.IsTrue(secondNearest.Location.Distance(nearest.Location) <
47                    nearest.Location.Distance(furthest.Location)
48                    && nearest.Location.Distance(nearest.Location) == 0);
49  }
```

Figur 15: Step definition fra SpecFlow

```
23  [Test]
24  public void Get5NearestTask()
25  {
26      var tasks = _taskHandler.SortByDistance(_taskHandler.GetTasks(int.MaxValue, 6, true));
27      var taskList = tasks.ToArray();
28      var secondNearest = taskList[3];
29      var nearest = taskList[4];
30      var furthest = taskList[5];
31      Assert.IsTrue(secondNearest.Location.Distance(nearest.Location) <
32                    nearest.Location.Distance(furthest.Location)
33                    && nearest.Location.Distance(nearest.Location) == 0);
34  }
```

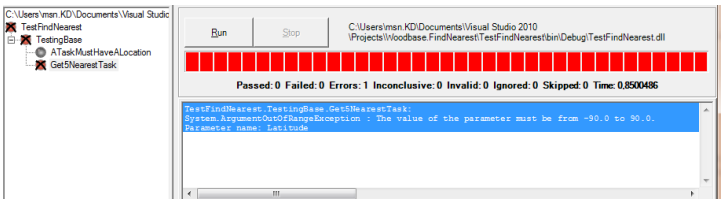
Figur 16: Test fra nUnit test projekt

Hvis der skulle være en defekt ville jeg forvente at finde den i en del af koden, som hang sammen med den variable del af SpecFlow testen. Men egentlig forventede jeg ikke at finde nogen problemer, for SpecFlow testen gav resultatet grønt - og dermed ingen fejl, når den blev kørt med nUnit [17]



Figur 17: Resultat af test fra BDD projekt

Derfor var det en anelse overraskende at resultatet fra TDD projektet blev rødt og vidste fejl i koden. Nu var der jo ændret noget, da testen blev flyttet fra BDD projekt til TDD projektet, så måske lå fejlen som antaget i den variable del. Men et hurtigt kig på fejlmeddelsen fra nUnit i figur [18] viser, at det faktisk er en helt anden fejl.



Figur 18: Resultat af test fra TDD projekt

TDD testen fejler på en `ArgumentOutOfRangeException`, som ikke blev ramt med mine BDD test. Men det er den samme kode, de test eksercerer. Så var det nu lykkedes at finde et eksempel på, at BDD tests ikke var lige så gode som TDD? Dog ikke, af en eller anden grund fortolker de to projekter datatypen **`double`** forskelligt. Men det kan ikke næppe udlægges som en fejl i metoden.

Jeg har ikke fundet nogle defekter eller huller i mit lille projekt, som jeg kan udlede skyldes at udviklingsmetoden har været BDD fra starten. Det er dog fortsat min vurdering at BDD i sin rene form, som beskrevet på Dan Norths blog [**??**] på større projekter vil misse mange af grænse tilfældende og oftest ramme „happy path". Da der i beskrivelsen ikke er noget, som sikre at det skal være anderledes.

Opsummering

Sammenligning	TDD	BDD
Efterlader oplagte huller	-	-
Finder huller i modparten	-	-
Fornuftig code-coverage	+	+

5.11 Tidsforbrug - sparet eller øget

En vigtig parameter i software udvikling er tid. Tid er penge, og kunderne ønsker ofte så meget software som muligt for så lidt midler som muligt. Det er blandt andet end af bevæggrundene i TDD for starte udviklingen, så hurtigt som muligt. Dels kan man løbende levere kørende software til kunden, hvis man ønsker det,ved f.eks. at anvende **SCRUM**. Dette betyder at kunden kan få gavn af systemet fra en tidlig fase, hvor det måske kun er nogle kernedele, som fungerer.

Det kan give kunden en følelse at at få udbytte af sin investering hurtig, og måske derved blive inspireret til flere features og se fordelen af at investere i mere funktionalitet. Men kan BDD så øge produktiviteten i forhold til TDD?

Da BDD som bekendt er en gren af TDD, så ser begge metoder ens på gavnen af løbende afleveringer. Dermed er svaret på ovenstående spørgsmål altså nej. Man kan dog hævde at, hvis Dan North har ret i at man med BDD opnår en mere fokuseret udvikling, og ved præcis hvad der skal laves, og hvornår nok er nok; og samtidig antager, at man skal bruge tid på at få styr på disse dele med TDD, så må der være noget tid at spare der.

Tidsbesparelsen må i så fald være betinget af udviklernes erfaring med metoden, da rutinerede TDD udviklere formentlig ikke vil tabe nær så meget tid på at „strejfe“, som de studerende, der gav Dan North ideen med BDD.

Omvendt kan man argumenterer for at nødvendigheden af at skulle formulere features og scenarier i en bestemt syntaks, er et fordyrende element. En use-case, eksempelvis udfra den skabelon man kan se i Larmans bog [Lar04], kan skrives med almindeligt tale sprog og følger blot en opskrift for hvilke elementer, der skal være med (Se figur [19]).

Use Case UC1: Process Sale

Primary Actor: Cashier
Stakeholders and Interests:
- Cashier: Wants accurate, fast entry, and no payment errors, as cash drawer shortages are deducted from his/her salary.
- Salesperson: Wants sales commissions updated.
- Customer: Wants purchase and fast service with minimal effort. Wants proof of purchase to support returns.
- Company: Wants to accurately record transactions and satisfy customer interests. Wants to ensure that Payment Authorization Service payment receivables are recorded. Wants some fault tolerance to allow sales capture even if server components (e.g., remote credit validation) are unavailable. Wants automatic and fast update of accounting and inventory.
- Government Tax Agencies: Want to collect tax from every sale. May be multiple agencies, such as national, state, and county.
- Payment Authorization Service: Wants to receive digital authorization requests in the correct format and protocol. Wants to accurately account for their payables to the store.

Preconditions: Cashier is identified and authenticated.
Success Guarantee (Postconditions): Sale is saved. Tax is correctly calculated. Accounting and Inventory are updated. Commissions recorded. Receipt is generated. Payment authorization approvals are recorded.

Main Success Scenario (or Basic Flow):
1. Customer arrives at POS checkout with goods and/or services to purchase.
2. Cashier starts a new sale.
3. Cashier enters item identifier.
4. System records sale line item and presents item description, price, and running total. Price calculated from a set of price rules.

Cashier repeats steps 3-4 until indicates done.

Figur 19: Eksempel på use-case (klippet ned fra [Lar04, p. 50-53])

Grundet den sproglige frihed må det alt andet lige være hurtigere at skrive use-cases end BDD scenarier uanset om det så er til JBehave (som nævnt i Behaviour-driven development med JBehave [DE10]) eller til SpecFlow med Gherkin syntaks.

Samlet set er der altså flere ting, som spiller ind på om der kan spares tid med BDD, her i blandt udviklernes erfaring og om testskriverne kender til **Gherkin** (for SpecFlow). Men opsumeret så må tidsbesparelsen alt andet lige være begrænset og ikke et argument for at vælge BDD fremfor TDD.

Opsummering

Tidsforbrug	TDD	BDD
Krævende på use-cases/scenarier	-	+
Mindske ved erfaring	+	+

5.12 Udvidelse af krav/fleksibel software

En af de egenskaber, som jeg værdsætter ved TDD, er fleksibiliteten. Muligheden for hurtigt at tilpasse sig ny eller ændrede krav. Henrik Bærbak Christensen har afsat en betydelig del af sin bog [Chr07] til netop at vise og forklare hvordan man sikre at kunne understøtte denne egenskab. Da jeg ikke har fundet nogen særlig litteratur om dette til BDD, vil jeg anvende HBCs beskrivelse til at vurdere, hvor vidt jeg mener, at kunne overfører det direkte til BDD.

Kapitel 13 i Reliable and flexible software explained [Chr07, p. 176-188] omhandler „Principper for fleksibelt design". Jeg syntes det kunne være interessant at se på disse principper, i forhold til den erfaring jeg har fået med BDD.

3-1-2 design [Chr07, p.176]
3. Indentificer variabilitetspunkter
1. Programmer mod et interface - ikke en implementation
2. Favoriser kompositioner over nedarvning

Bagrunden for den lidt besynderlige rækkefølge for design principperne i boksen[5.12] skal findes i den praktiske anvendelse af de principper, som er beskrevet i *Design Patterns - Elements of Object-Oriented design* af Gamma et al. [Chr07, p. 176]. Kan disse principper anvendes i en BDD kontekst?

Der er ingen problemer i at anvende design principperne i en BDD process. Det handler i virkeligheden ikke så meget om metoden, som det gør om udviklernes kendskab til design principper for fleksibel software. HBC illustrerer i sin bog forskellige måder til at imødekomme udvidelse af funktionaliteten i et stykke software. I kapitel

3 [Chr07, p. 49-87] ser vi således på nogle forskellige fremgangsmåder: *Source code copy*, *Parameteriseret*, *Nedarvning* og *kompositionelt* og deres fordele og ulemper.

Det er for så vidt ligegyldigt hvilken udviklingsmetode vi anvender. Principperne kan anvendes både ved TDD, BDD og også ved en mere klassisk metode, som beskrevet af Lars Mathiassen i *OOA&D* [Mat01]. Alligevel synes jeg det kunne være interessant, at se på om jeg mener metoden „opfordre“ til at man følger design principperne for fleksibelt design. Jeg vil ikke komme så meget ind på *OOA&D* i denne opgave, men lige påpege, at den klassiske vandfaldsmodel, der der anvendes i min opfattelse eksempelvis ikke understøtter design princip 3 (i 3-1-2) særlig godt. Da man jo primært kigger på hvad kundens nuværende krav er. De mere agile metoder er noget mere åbne for det fra „naturens“ side. (Se eksempelvis [5.12.3])

5.12.1 1. Programmer mod et interface

HBC beskriver i sin bog [Chr07] at abstraktioner er en central del af moderne software design, og BDD må betragtes som en moderne metode. Det kan derfor undre, at der ikke er gjort noget for at beskrive princippet i metoden. Faktisk mener jeg at man skal være mere omhyggelig for at anvende princippet i BDD end med TDD, da scenarierne kan have en tendens til at beskrive detaljer, som ikke ville være i en test liste.

BDD gør i sig selv ikke noget for at fremhæve dette princip, men der er heller ikke noget til hinder for, at vi kan bruge det. Så hvis udviklerne er bekendte med princippet, kan det sagtens anvendes i BDD kontekst.

5.12.2 2. Favoriser kompositioner over nedarvning

Da BDD er så nært beslægtet med TDD, så ser jeg ingen forhindringer i at vælge et kompositionelt design fremfor eksempelvis nedarvning. Faktisk er det et princip, som jeg jævnligt bruger, selvom jeg ikke anvender TDD eller BDD. Jeg hørte på PaSOOS kurset en sætning, som er blevet en slags mantra for mig: „Overlad arbejdet til specialisterne“.

5.12.3 3. Indentificer variabilitetspunkter

En vigtig del i moderne software udvikling er evnen til hurtigt at kunne omstille til nye krav. Nogen gange fordi kundens budget eller krav

ændres, andre gange er det muligheden for at ændre i delafleveringer, så del A og C kommer i produktion før del B. Både TDD og BDD kan understøtte dette uden synderligt besvær. Med TDD er det et spørgsmål om at ændre i testlisten. For BDDs vedkommende er det et spørgsmål om at ændre i scenarierne eller rækkefølgen for deres implementation.

I TDD går vi efter at implementere et punkt fra testlisten ad gangen, og BDD er ikke meget anderledes, da vi går efter at implementere et scenarie adgangen. Men ideen er at det gerne skulle være ligegyldigt hvilken rækkefølge vi gør det i, og at vi gerne skulle kunne gøre det uden at påvirke den eksisterende kode.

Opsummering

Fleksibilitet	TDD	BDD
Programmer mod et interface (oplagt)	+	-
Favoriser kompositioner over nedarvning (oplagt)	+	+
Indentificer variabilitetspunkter (oplagt)	+	+

5.13 Uddybende og dækkende test

Behaviour Driven Development hjælper helt klart til med til at holde fokus, når man har nedskrevet sit scenarie. Man får hurtigt formuleret en hel række test cases (såfremt man kender syntaksen til at skrive tests med). Det efterlader ved første øjekast én med følelsen af, at man har fat i et super fedt værktøj, hvilket sagtens kan være sandt. Men BDD er ikke en mirakelmetode, som fratager udviklere behovet for at tænke.

Det er min helt klare opfattelse, at man meget hurtigt ender op med en masse test cases, der udelukkende tester „the happy path“. Med „happy path“ tænker jeg på det, som får programet til at køre. Fokus på de ting, som systemet skal kunne. Men hvad med de elementer, som sikre at vores system kører stabilt? Det er selvfølgelig muligt også at komme op med alle disse scenarier også, som i eksemplet nedenfor (figur [20]):

Scenarier som figur[20] kan formuleres af alle som kender syntaksen. Men det efterlader mig med et spørgsmål, som det netop er noget af det BDD skulle hjælpe med: „Hvornår er nok nok?“

```
15  Scenario: Request negative number of tasks
16      Given I'm a technician
17      And I request -5 task
18      When I press GetTasks
19      Then I should recieve the message Invalid request! Amount must be a positive integer
20
```

Figur 20: Fejl scenarie

Det fik mig til at overveje, hvorfor jeg fik den følelse, og hvorfor endte jeg op med at søge „happy path“? En del af forklaringen ligger naturligvis i min forholdsvis begrænsede erfaring med BDD, men en anden og meget tungt vejende del ligger i den litteratur, som jeg har læst om BDD.

Hverken SpecFlows wiki sider [con], Dan Norths blog [Nor06], Behaviour-Driven.org [Bdd10] eller Wikipedia [wika] nævner noget om hvordan man sikre reliability med BDD. Derfor er det knap så overraskende, at man til en start vælger at fokusere på de områder, der giver øget funktionalitet fremfor områder, som giver mere reliability.

Næste spørgsmål, der så melder sig, er så om man ikke kan anvende BDD til at lave pålidelig software? Naturligvis kan man det, men det er ikke umiddelbart noget, som ligger i metodens natur at fokusere på eller understøtte. Men hvordan gør man så? Mit bud går på at søge inspiration hos TDD endnu engang.

Som nævnt tidligere, så foreskriver begge metoder en „test-first“ tilgang. HBC har i sin bog [Chr07, p. 40-41] en liste med møsntre i TDD processen. Denne liste kan sagtens anvendes i BDD:

- Test
- Test først
- Test liste (scenarie i BDD)
- Assert/påstand først

- Fake-it
- Triangulering
- Åbenlys implementation

- Isolerede test
- Tydelige test

- Test data
- Indlysende data

- Pauser

Det tredje og næst nederste punkt i listen [5.13] er i mine øjne særligt interessant i forhold til BDD. For mens det i den undervisning jeg har fulgt om TDD har været en naturlig del at lave ækvivalensklasser til at afdække hvilke test vi har brug for - altså „hvornår nok er nok“. Det er min opfattelse, at hvis man udelukkende anvender de scenarier man kommer op med, så vil man i mange tilfælde måske nok producere fornuftig software, men den vil være knap så pålidelig, som hvis den var udviklet med TDD og mere systematisk test med blandt andet ækvivalensklasser og grænsetilfældeanalyse.

Grænsetilfældeanalyse er endnu en ting, som ikke er nævnt i nogle af artiklerne om BDD. Som det fremgår i Myers bog [Mye04, ch. 4, p.43-91] skal man sikre sig at man tester grænsetilfældende af, da det hyppigt er der fejlene opstår. Derfor er det i disse områder vi finde det bedste og mest indlysende test data.

Problemet med ækvivalensklasser, systematisk testing og grænsetilfældeanalyse er at det laves af folk, som kender metoderne. Derved

bygger vi yderligere krav på til vores test forfattere og deres egenskaber. Vi kan næppe forvente at virksomhedens marketingsafdelinger kommer til at lave ækvivalensklasser, så vi kan altså ikke overlade det til dem at skrive alle vores scenarier, hvis vi ønsker at øge pålideligheden på denne facon.

Faktisk mener jeg at gevinsten ved scenarier frem for use-cases svinder markant, når man tager ovenstående med ind i billedet. Da det sender en stor del af arbejdet tilbage til udviklingsafdelingen, og underminerer ideen om at BDD fortæller, hvad du skal teste, og hvad du ikke skal. For med systematisk testing kan man finde de samme test på en mere præcis måde, selvom det måske nok tager længere tid.

Opsummering

Uddybende og dækkende	TDD	BDD
Systematisk testing	+	-
Evident data	+	-
Test-first	+	+
Foreskriver grænsetilfældeanalyse	+	-

5.14 Fordele ved BDD

Hvis ikke man mente der var nogen fordele ved BDD, så havde Dan North nok opgivet den med det samme. Jeg kan da også se et par punkter, som er vældig fornuftige.

Det punkt som alle fremhæver ved BDD er at det hjælper med at holde fokus. Jeg må sige, at jeg synes det med at have et scenarie at udvikle efter, virker praktisk og nemt. Min erfaring med TDD strækker sig da heller ikke til meget mere end PaSOOS kurset, så derfor kan jeg nemt identificere mig med de studerende, som gav Dan North ideen. Det kan virker meget uoverskueligt blot at starte med en testliste. Derfor er rettesnoren i scenariet en rar livline.

Jeg synes også det er en god ting, at testlisten, som så er de features, der er beskrevet i projektet, hænger sammen med testkoden. Hvis du retter i beskrivelse af dit scenarier, så vil testen også fejle. Dine krav har jo ændret sig, så testen er ikke længere valid.

Opsummering

Fordele	TDD	BDD
Test-liste synkroniseret med koden	-	+
Overskueligt for nybegyndere	-	+

5.15 BDD som alternativ?

Hvorfor så overhovedet anvende BDD, hvis det ikke er ligeså godt som TDD? Jeg har tidligere været inde på en vurdering af, om der kunne spares tid ved at anvende BDD i stedet for TDD [5.11]. I det følgende vil jeg se på hvilke andre faktorer, som kan gøre BDD attraktivt som alternativ til TDD.

> While test-first programming works for some people, it doesn't work for everyone. For every application developer who avidly embraces TDD, there are many others who actively resist it. [Glo]

En god pointe til hvorfor man skal bruge BDD findes i boksen ovenfor [5.15]. Jeg erindrer fra den afsluttende opgave på fagpakken „Programmering af store objekt orienterede systemer (PaSOOS)“, at det netop det med først at skrive testen, var en af de udfordringer, som min gruppe sloges med. Om end vi i gruppen var bekendte med både unit tests og objektorienterede sprog, så var omstillingen fra at teste eksisterende kode til at teste - om man så må sige - kommende kode vanskellig. Selvom man med BDD også skal skrive tests først, så kan flowet i scenariebeskrivelserne give en mere glidende over gang. Da man nemt får en fornemmelse af at kode meget tæt på use-casen, hvilket gør „pillen“ nemmere at sluge.

Fra samme artikel [Glo] som boks [5.15] kommer følgende beskrivelse af hvorfor man ikke bruger TDD:

> Two common reasons for not doing TDD are "There's not enough time for testing"and "The code is too complex and hard to test."Another hurdle in test-first programming is the test-first concept itself. Many of us view testing as a tactile activity, more concrete than abstract. Experience tells us that it isn't possible to test something that doesn't already exist. For some developers, given this conceptual framework, the idea of testing first is an oxymoron. [Glo]

Lad os lige skille den [5.15] lidt ad!

1. "There's not enough time for testing": En klassiker! Det er et nemt argument, som af flere grunde kan gennemhulles. For hvorfor er der ikke tid nok? En af de grunde jeg oftest har set er at udviklingen - uden hverken BDD eller TDD - har taget længere tid end forventet og derfor flytter projektledere timer fra test til udvikling.

En anden grund er at tiden til at lave tests øjensynligt er sværere at sælge end reel udviklingstid. Kunden forventer jo at få et produkt der virker, så hvorfor skal de betale for, at vi skriver test cases? Det er der selvfølgelig en masse argumenter for, men erfaringen fortæller mig, at det tilsyneladende er svært for sælgerne at trænge igennem med det. Det kan selvfølgelig også være at sælgerne er for dårlige, eller at kundens budget er for lille til deres ønsker, men sælgeren gerne vil have opgaven hjem og derfor ikke forsøger hårdt nok at sælge det ind.

Ingen af ovenstående grunde har dog nogen umiddelbar sammenhæng med BDD eller TDD. Slet ikke hvis vi ser på hvad Kent Beck mener om sagen. I følge Beck tager det nemlig ikke længere tid at udvikle med TDD end med en mere klassisk tilgang med koden før testen. Alligevel bliver det brugt som argument for ikke at bruge TDD. Årsagen skal nok også findes i citatet [5.15] *Many of us view testing as a tactile activity*. Ganske som jeg beskrev vores opgave fra PaSOOS-kurset, så kan tankegangen være svær at omstille sig til, og det vil da for langt de fleste også kræve ekstra tid i starten at bruge TDD.

Kan BDD så afhjælpe denne modstand mod test drevet udvikling? Næppe, for med BDD skal vi have lavet både scenarier og tests. Set udfra mit perspektiv på virkeligheden, så forudser jeg også en del modstand fra marketingsfolk, hvis vi beder dem om at komme med kravspec eller use-cases formuleret på **Gherkin** syntaks. „Det har vi ikke tid til" eller „Det må være op tid udviklerne at lave tests". Det vil naturligvis kunne løses med en ledelsesmæssig beslutning om at de skal, men jeg tvivler på at det fjerner kommentaren: „There's not enough time for testing".

2. „The code is too complex and hard to test." Så er den godt nok gal. For det første ville situationen jo aldrig opstå, hvis den fra starten var udviklet med TDD. For det andet så må systemet have en utrolig dårlig maintainability, hvis det er for komplekst til at skrive test til det, hvem tør så rette i koden? Endelig, hvis jeg omsætter det til min egen hverdag, så lyder det som en dårlig undskyldning for at have lavet noget kode for længe siden, som ikke er dokumenteret, og som man nu ikke rigtig ved hvordan hænger sammen.

TDD ville som sagt kunne have afhjulpet situationen fra starten, men kunne der have været gavn af BDD? Det tror jeg faktisk godt der kunne. Hvis man kan beskrive den ønskede funktionalitet med et scenarie (eller flere), så kan man i den givne situation anvende BDD som acceptance test. Der kan korrekt argumenteres for, at man ikke ender med at have en række udtømmende tests, med mindre man laver eksempelvis noget blackbox testing. Men BDD kunne blot ved at teste „happy path“ øge maintainability en lille smule, da vi i så fald ville have tests der fejlede, hvis vi introducerede fejl.

Set i forhold til min virkelighed, som en del af udviklingsteamet hos Avior A/S, hvor vi på nuværende tidspunkt i mange projekter opererer med manuelle test, og ikke har fokus på at skulle anvende TDD, selvom vi gerne anser os selv for at være agile. Ville BDD kunne være et alternativ for os som udviklingshus? Svaret må være et tøvende ja. Anvendelsen af BDD vil ikke kunne sættes i stedet for TDD. Vi vil have gavn af at anvende BDD og scenarierne som acceptance tests til at dokumentere overfor vore kunder, at vores software opfylder de krav, som de har stillet.

Vi kan altså ikke sige, at vi kan erstatte TDD med BDD. Dels fordi vi aldrig har brugt TDD, men også fordi vi primært vil sigte efter at opnå kundens accept af at softwaren virker som aftalt, og ikke hvor robust/reliable den er. Det må dog anses for at være bedre end manuelle eller ustrukturerede unit tests.

6 Relateret arbejde

6.1 Behaviour-driven development med JBehave [DE10]

Jan Duelund og Preben Eriksen havde forud for opgaven ingen erfaring med behaviour driven development. Deres opgave omhandler udviklingen af et Ludo spil. Scenarierne bestod af reglerne for Ludo, men en af de erfaringer, som de gjorde, i den forbindelse, var at de brugte meget lang tid på at formulere scenarierne, før de kom igang med at producere kode.

Ydermere oplevede Jan Duelund og Preben Eriksen at detaljegraden i deres scenarier blev for stor, blandt andet fordi de havde vanskeligheder med at adskille scenarierne og fik kædet dem sammen. Resultatet af denne sammenkædning blev at de ubevidst endte med at lave for meget upfront design. Med deres egne ord: „vi havde mistet fokus på den feature som scenario beskrev".

På baggrund af disse erfaringer besluttede gruppen at lave et „do-over". Under denne refaktorisering af deres scenarier oplevede de, at de blev skarpere på at lave scenarier og fik nemmere ved at følge filosofien bag BDD. Konklusionen på disse erfaringer blev at udformningen af scenarier er den klart sværeste del af BDD processen.

Jan Duelund og Preben Eriksen mener også at BDD som kommunikationsværktøj er kan være med til at mindske risikoen for misforståelser, når scenariet skal implementeres, da der skal overholdes en helt bestemt struktur.

Det vurderes i opgaven også som simpelt at læse og forstå stories, scenarios og acceptkriterier, men udformningen af dem er en langt vanskeligere opgave. Det vil kræve tid fra de forskellige stakeholders at få udarbejdet fornuftige og brugbare scenarier, og det kan være svært at overbevise de nødvendige stakeholders om at afsætte den fornødne tid til det.

Gruppen vurderer at BDD har sin force, hvis: „specifikationen er uklar eller mangelfuld, og hvis der er en eller anden form for interaktion med stakeholders med anden funktion end konstruktionen af softwaren." Modsat vil det være vanskeligt at opstille scenarier for non-funktionelle krav, da disse ikke har en adfærd, som kan observeres af stakeholders.

BDD har hjulpet gruppen til at have et løbende overblik over udviklingsforløbet. Scenarierne har fungeret som naturlige milepæle i projektet. Risikoen ligger i at BDD hænges op på et bestemt framework som JBehave, hvorved man bliver afhængig af hvordan frameworket understøtter og fortolker processen.

En fordel ved JBehave er den direkte mapning af scenarier direkte til Java-kode. Så man specifikationen koblet med kildekoden og ikke har den liggende i en ekstern sagsmappe.

Anbefalingen fra Jan Duelund og Preben Eriksen er at man adoptere BDD på interne projekter, så stakeholders er let tilgængelige. Baseret på antagelsen om at det kan være svært, at få alle stakeholders overbevist om at afsætte den fornødne tid, hvilket vil være en „show stopper“ for processen.

BDD bør være en integreret del af en udviklers måde at arbejde på. Selvom man ikke gennemfører BDD processen helt ud i yderste led, giver BDD en god måde at sikre og minde os om at vi skal lave software til dem der skal benytte den og derfor er det vigtigt at systemet gror i den rigtige retning i mod "...the delivery of working, tested software that matters". [DE10]

6.2 Test driven development [Mør05]

Troels Jegbjærg Mørch arbejdede i 2005 med TDD i forbindelse med sin hovedopgave på diplomuddannelsen ved Århus Universitet. Udgangspunktet var at undersøge om TDD gav:

```
1 Færre fejl.
2 Mindre debugging.
3 Mere selvtillid hos udviklerne.
4 Et bedre design.
5 Højere produktivitet.
```

Konklusionen blev at TDD rent faktisk gav:

```
1 Højere kvalitet i koden.
2 Færre fejl i koden.
3 Mindre debugging.
4 Øget selvtillid hos udviklerne.
5 Samme eller længere udviklingstid.
6 Samme eller højere produktivitet.
```

Et resultat som understøtter Kent Becks hypotese. Men Troels Jegbjærg Mørch påpeger også problemområder i processen.

```
1 Det var svært for udviklerne at vende sig til processen.
2 Den øgede produktivitet skal ses over hele projektets levetid.
    Da nogle af styrkerne ved TDD netop skulle være
    modifiability og extensability, når testene kan bruges til
    at vise at eksisterende kode ikke ødelægges.
```

Troels Jegbjærg Mørch mener at de mange test i TDD, kan opfattes som kørende dokumentation på systemet.

Der er brugt flere forskellige værktøjer i Troels Jegbjærg Mørchs projekt, men som det antages i konklusionen, så har Visual Studio sidenhen udviklet sig til at omfatte mange af dem i et. Det er derfor med VS2010 muligt at kører et helt TDD projekt udelukkende i Visual Studio.

7 Konklusion

Kan man ved brug af SpecFlow opnå en situation, hvor kunden reelt kan formulere test scenarier gennem deres egne brugsmønstre? Nej, gennem mit arbejde med denne opgave er jeg helt klart af den overbevisning, at man ikke kan lade kunden skrive alle test scenarierne selv.

Jeg mener at det overhead, der er på at lære en bestemt syntaks som **Gherkin**, for at kunne formulere scenarier, er for stort i forhold til den gavnlige virkning. Dels fordi jeg, som beskrevet flere steder i opgaven, mener, at man vil søge „the happy path“, og stadig have brug for at udviklere eller arkitekter til at lave de finmaskede test cases. Men også fordi det er min vurdering, at der vil være for mange fejl/problemer med de scenarier, der vil komme fra kunden.

Problemer som uklare objektbeskrivelse, som dem der for eksempel de tidligere omtalte „avancerede objekter“ [5.6], men også generelt mangelfulde scenarier, som vil kræve yderligere afklaring. Man kan f.eks. forestille sig en beskrivelse af at „`En opgave skal godkendes før den kan leveres til en teknikker`“, men hvem må godkende opgaver? Må alle eller må kun nogle bestemte administratorer eller folk med særlig ophøjede rettigheder?

De samme mangler kunne naturligvis forekomme i enhver anden overlevering af krav i form af use-cases eller lignende. Forskellen er blot at der ikke vil være overhead på at overholde en bestemt syntaks, men måske blot en skabelon at skrive i for at formaliserer det.

Der til kommer, at vi i første omgang skal have, de personer, som skulle skrive scenarierne, til at acceptere den nye arbejdsgang. Rigtig mange mennesker har det godt med at gøre „som de plejer“. Det er trygt og sikkert, og det kender vi.

Hvor stort er overhead på konfiguration af SpecFlow kontra det selv at skrive test cases? Det var i første omgang her, jeg frygtede at løbe ind i problemer. Inden jeg påbegyndte opgaven, havde jeg en forestilling om at BDD (uanset værktøj) ville være et konfigurationshelvede. Men her er jeg blevet positivt overrasket. SpecFlow kræver utrolig lidt konfiguration og integrationen med VS2010 er meget gennemført. Faktisk fungerer det så godt, at det på trods af det `code-blob`, som jeg mener BDD laver, er til at navigere rundt i. Der er mange smarte navigationsmuligheder integreret i kontekstmenuen

fra de forskellige vinduer i VS2010. Så konfigurations overhead er der ikke meget af.

Er BDD et fornuftigt alternativ til TDD? Efter i dette projekt at have haft lejlighed til at sammenligne de to metoder, så må jeg konkludere at BDD ikke er et fornuftigt alternativ til TDD. Der er for meget unødvendig „støj". Man kommer ganske enkelt for langt væk fra „Clean code that works". Jeg bryder mig ikke om at der laves „tomme tests" [5.4], som man er nødt til at forholde sig til alligevel, og vurdere hvad de skal indholde, om der skal laver antagelser på dem eller ej.

Hvis BDD skal være lige så grundigt som TDD. Så skal der stadig laves black-box testing og lignende, så her finder man heller ingen gevinst. Tilføjet det fordyrende led med at man er nødsaget til at oplære folk i at skrive scenarier, så konkluderer jeg simpelthen ikke at den gavnlige effekt er tilstede i tilstrækkelig grad til at BDD er indsatsen værd.

Kan BDD først bruges senere, da det operere på et bredere scope? BDD behøver ikke nødvendigvis operere på et bredere scope. Alene antagelsen mener jeg viser, at BDD i den gængse opfattelse vil være mere overfladisk end TDD. Hvis man ser bort fra de fordyrende led og `code-blob`, så er der ikke noget i vejen for at kunne anvende BDD fra starten af et projekt.

Jeg vil ikke kunne anbefale BDD som erstatning for TDD. Hvis man kun ønsker at teste at koden levere nogle givne krav, så kan BDD være godt nok. Men så vil der også være tale om acceptance test, og netop testing af de dele af systemet, der er aftalt som „happy path". Fordelen her vil jo være at der er en direkte paralel mellem beskrivelsen af funtionaliteten og den kode som tester den. Men en erstatning for TDD er det ikke.

Appendices

A MS Visual Studio 2010

Visual Studio 2010 er et powerfuldt IDE, der sikrer kvalitetskode gennem hele applikationens livscyklus - fra design til implementering. Uanset om du udvikler applikationer for SharePoint, webben, Windows, Windows Phone eller andet, er Visual Studio den ultimative all-in-one løsning. (Kilde: http://www.microsoft.com/visualstudio/da-dk)

Visual Studio kan udvides med diverse pakker fra 3. part som for eksempel SpecFlow og SpecRun til udvikling med BDD. Der findes også pakker som kan understøtte TDD udvikling som for eksempel TestDriven.Net eller ReSharper. ReSharper kan yderligere hjælpe til med at overholde forskellige standarder for formatering, implementering af metoder fra interfaces og meget mere.

B SpecFlow

SpecFlow er et tredje parts værktøj, som kan integreres med Visual Studio 2010. SpecFlow forsøger at skabe en kommunikations bro mellem domæne eksperter og udviklere ved at binde læsbare adfærd specifikationer med den underliggende implementering.

Visionen er at give en pragmatisk og gnidningsløs tilgang til Acceptance Test Driven Development og Behavior Driven Development for moderne .NET-projekter.

SpecFlow er open source og distribueret under BSD License. SpecFlow er hosted på GitHub.

(Kilde: http://www.specflow.org/)

C NUnit

NUnit er et unit-test framework til alle .Net sprog. Oprindeligt porteret fra JUnit, den nuværende produktion udgivelse, version 2.6, er den syvende store udgivelse af denne xUnit baserede testværktøj til Microsoft. NET. Det er skrevet udelukkende i C# og er blevet helt nydesignet for at drage fordel af de mange funktioner i .NET sprog, for eksempel brugerdefinerede attributter og andre reflection egenskaber. NUnit bringer xUnit til alle .NET sprog.

(Kilde: http://www.nunit.org/)

D SpecFlow Assist Helpers

https://github.com/techtalk/SpecFlow/wiki/SpecFlow-Assist-Helpers

SpecFlow assist helpers er en række extension methods, som løser den noget langsommelige og trættende opgave med at mappe tabeldata fra scenarier til objektdata i de egentlige test cases.

Eksempel:

Klassisk løsning:

```
1 public void GivenImaAdministrator(Table userData)
2       {
3          var user = userData.Rows.Select(x => new
               User(x["userName"], x["userEmail"],
               x["userType"])).First();
4          Assert.AreEqual("‘Administrator"', user.UserType);
5       }
```

SpecFlow assist løsning:

```
1 public void GivenImaAdministrator(Table userData)
2       {
3          var user = userData.CreateInstance<User>();
4          Assert.AreEqual("‘Administrator"', user.UserType);
5       }
```

SpecFlow assist helper mapper kræver en tom constructor og kan så mappe public properties udfra headers på tabellen

Litteratur

[Bdd10] BddWiki. Behaviour-driven development. Website, September 2010. http://behaviour-driven.org. URL: http://behaviour-driven.org.

[Bec03] Kent Beck. *Test-driven development by example*. Addison-Wesley, 2003.

[Beh] Behat.org. Writing features. website. URL: http://docs.behat.org/guides/1.gherkin.html.

[Chr07] H.B. Christensen. *Reliable and flexible software*. Imhotep, short course preprint edition, 2007.

[con] Various contributors. Techtalk/specflow. website. URL: https://github.com/techtalk/SpecFlow/wiki/Documentation.

[DE10] Jan Duelund and Preben Eriksen. Behaviour-driven development med jbehave. Master's thesis, Århus Universitetet, 2010. URL: http://cs.au.dk/fileadmin/site_files/cs/public_study_admin/Jan_Preben.pdf.

[Glo] Andrew Glover. In pursuit of code quality: Adventures in behavior-driven development. website. URL: http://www.ibm.com/developerworks/java/library/j-cq09187/index.html.

[H] Henrik W H. C# 3.0: Extension methods. website. URL: http://blogs.msdn.com/b/henrikwh/archive/2007/09/30/c-3-0-extension-methods.aspx.

[HBCH04] Aino Corry Henrik Bærbak Christensen and Klaus Marius Hansen. An approach to software architecture description using uml. 2004.

[Lar04] Craig Larman. *Applying UML and Patterns*. Prentice Hall, 2004.

[Mat01] Lars Mathiassen. *Objektorienteret analyse og design*. Forlaget Marko, 2001.

[Mør05] Troels Jegbjærg Mørch. Test driven development. Master's thesis, Århus Universitetet, 2005. URL: http://www.daimi.au.dk/fileadmin/images/common/secretaries/M_rchTJ.pdf.

[MT] Morten Mejlsing and Casper Thomsen. Spaghettikode. website. URL: http://www.version2.dk/leksikon/Spaghetti-kode.

[Mye04] Glenford J Myers. *The art of software testing 2nd edition*. John Wiley & Sons Inc., Hoboken, New Jersey, 2nd edition, 2004. Tom Badgett and Todd Thomas with Corey Sandler.

[Nor06] Dan North. Introducing bdd. website, 2006. URL: http://dannorth.net/introducing-bdd/.

[Spe] SpecFlow. Specflow - pragmatic bdd for .net. website. URL: http://www.specflow.org.

[WC] Juan Wajnerman and Daniel Cazzulino. Moq. website. URL: http://code.google.com/p/moq/.

[wika] wikipedia. Behavior driven development. website. URL: http://en.wikipedia.org/wiki/Behavior_Driven_Development.

[wikb] wikipedia. Regulære udtryk. website. URL: http://da.wikipedia.org/wiki/Regul%C3%A6re_udtryk.

[wikc] wikipedia. Scrum. website. URL: http://da.wikipedia.org/wiki/Scrum.