DATALOGISK INSTITUT
DET NATURVIDENSKABELIGE FAKULTET
AARHUS UNIVERSITET

# Hovedopgave

***Diplom i Informationsteknologi***
***linien i Softwarekonstruktion***

# Behaviour-driven development

# med JBehave

**Af**
**Jan Duelund & PrebenEriksen**

10-06-2010

Jan Duelund & Preben Eriksen,
studerende

Henrik Bærbak Christensen,
vejleder

**Datalogisk Institut**
Aarhus Universitet
Åbogade 34
8200 Århus N

Tlf.: 89425600
Fax: 89425601
E-mail: cs@cs.au.dk
http://www.cs.au.dk/da

# INDHOLD

# MOTIVATION

Agile udviklingsmetoder er noget som igennem de sidste 10-15 år har vundet kraftigt frem og ikke længere kun er noget man sidder og eksperimenterer med. En af disse metoder er Test Driven Development (TDD). Det er efterhånden blevet en accepteret og god måde at udvikle kvalitetssoftware på, men TDD har dog nogle udfordringer der har vist sig svære at overkomme på en række punkter, en af tingende er at den er meget centreret omkring udviklerne og ikke naturligt tager hånd om kundens/brugerens ønsker.
Hvis man ikke helt ved hvad TDD går ud på kunne man også hurtigt få den ide at det handler om test og ikke om design af software, det skyldes at det sprog man benytter i TDD er meget testfikseret.
Der kunne nævnes flere ting men de gemmes til senere.

Disse udfordringer er der nogle som har tænkt at det må der kunne gøres noget ved. Ud af det er der kommet en "ny" metode som hedder Behavior Driven Development (BDD). Det er en videreudvikling af TDD men som gerne skulle bygge bro i mellem alle stakeholders i et projekt. Man har for det første gjort op med hele test tankegangen og nu i stedet for specificerer opførsel. Denne opførsel er noget som kunden/brugeren er med til at specificere, dvs. brugeren er med til at fortælle hvilken opførsel der skal implementeres og det gøres i et fælles sprog og på en struktureret måde. Fordi der er det fælles sprog skulle bla. udvikler, slutbruger og forretningsanalytiker bedre kunne forstå hinanden når de snakker sammen.

Det vi synes der er interessant er at se hvordan det står til med BDD, det er trods alt en metode som har været her i nogle år. Hvilke forskelle der reelt er ift. TDD og hvilke andre metoder end TDD gør BDD brug af, slipper man nemmere udenom de udfordringer der er med TDD samt hvordan er den med til at understøtte den iterative process?.

# PROBLEMFORMULERING

Udvikling af software har mange forskellige problemstillinger og processer der skal løses og udføres før der kan komme et brugbart produkt ud i den anden ende. Den process der traditionelt har været den sværeste at både forstå og løse i et udviklingsforløb er "hvad skal der bygges og hvordan", dvs. at specificere, designe og teste den konceptuelle model således at den matcher målgruppens krav og forventninger.
BDD har givet os nogle forhåbninger om at vi kan blive hjulpet til at gøre denne process mindre besværlig ved at introducere os for en tankegang og en struktur, som forsøger at holde fokus på hvilken værdi en opførsel eller funktion bringer til systemet, og derved en større forståelse for hvordan forretningsreglerne opererer i systemet.

## Undersøgelser

Vi vil beskrive kerneprincipperne i behaviour-driven development (BDD) og beskrive hvordan teknikken relaterer sig til og arver fra andre agile teknikker herunder test-driven development (TDD) og accept test driven development (ATDD).

Vi vil tage et kig på hvordan BDD påvirker og integrerer de forskellige stakeholders (sponsorer, slutbrugere, business analytikere, udviklere, testere) i et udviklingsforløb.

Vi vil vurdere hvor nemt det er at tilegne sig metoden for de involverede stakeholders og om metoden egner sig bedre i nogen typer projekter end andre.

Udfra en praktisk øvelse vil vi evaluere om teknikken håndterer styring stringent nok til at kunne fastholde et overblik over de samlede mål, uden at miste hverken et tab i produktivitet eller fastlåse folk i en rigid struktur.

Desuden vil vi kigge på JBehave der er et Java framework specifikt konstrueret med BDD for øje, vi vil kigge nærmere på hvordan frameworket kan hjælpe os til at holde fokus på den rytme BDD har stillet op for os.

Til sidst vil vi komme med en aktuel status af BDD, og vil i den sammenhæng forsøge at besvare nogle spørgsmål

- hvor er BDD er på vej hen.
- har BDD nok i sig til at blive en integreret del af den agile værktøjskasse.
- udvikler BDD sig stadig eller er teknikken stagneret.
- hvordan kan BDD komme videre og opnå en anerkendelse som TDD

## Afgrænsning

Alt Java udvikling indholdt i denne rapport er styret efter at opnå overblik og forståelse for hhv. BDD og JBehave, så java applikationen kan være inkomplet på både features og brugbarhed.

Design af userinterface nedprioriteres eller fjernes helt og medtages kun for evt. demonstration.

Vi vil ikke komme detaljeret ind på virkemåden af test teknikker (mocks, stubs osv.) der bruges i forbindelse med udviklingen, men vil blot nævne dem i de sammenhænge vi benytter dem til at producere vores egen implementation.

# METODE

Til at få viden omkring emnet BDD vil vi benytte os af artikler, videocasts, blogs og praktiske eksempler fra primært internettet, desuden vil vi gøre brug af bøger der omhandler BDD eller på anden måde relaterer sig til BDD såfremt disse måtte eksistere. Når vi føler vi har indsamlet nok viden vil vi påbegynde udviklingen af et meget simpelt spil Ludo som ikke nødvendigvis kommer til at indeholde alle de regler der normalt forbindes med Ludo eller på nogen måde vil være spilbart.

Til BDD processen benyttes frameworket JBehave. Vi vil dog ikke udelukke at der kan komme andre værktøjer i spil også, i begge tilfælde vil sproget være Java og værktøjet Eclipse.

Da vi ikke har et "rigtigt" projekt at arbejde på kan vi ikke inddrage nogle stakeholders udover rapportforfatterne. Så til at vurdere om BDD rent faktisk har de påståede fordele overfor stakeholder i forhold til TDD, er vi nødt til at holde os til den tilgængelige litteratur der findes på området samt opfatte os selv som stakeholder i den opgave vi har stillet os selv og derfor er selve opførslen af et spil Ludo sekundært iforhold til at lære og forstå BDD filosofien.

# KONVENTIONER

## Ordforklaring

En del ord har vi valgt at enten bibeholde på engelsk, eller oversætte til engelsk i sammenhænge hvor det virker mere naturligt i den givne kontekst, at benytte den engelske frem for den danske udgave. Nedenstående ord har vi valgt kort at beskrive hvorfor de engelske ord er valgt i stedet for de danske.

### story, scenario, step

Benyttes konsekvent i stedet for de danske oversættelser "historie", "scenarie" og "skridt", eftersom at story, scenario og steps er reserverede ord i JBehave og for at forhindre misforståelser og sikre konsistensen mellem eksempler og det skrevne er brug af disse ord bibeholdt på engelsk.

### feature

Benyttes i stedet for "egenskab" eller "funktion", da ordet "feature" er bredt accepteret i software branchen for at præcisere en egenskab/funktion/samling af opførsel der giver et resultat og har den egenskab at den dækker bredt i forhold til de danske modstykker.

### stakeholder

Er valgt ud fra den betragtning at ordet "interessenter" oftere leder hen på folk med en økonomisk interesse af passiv art end på deciderede personer der deltager aktivt, så vi vælger at benytte os af stakeholder for at holde fokus på at en stakeholder ikke nødvendigvis har nogen økonomisk interesse i systemet.

## Typografi

### Konsol output

```
Hello world...
.
.
.
```

### Kode fragmenter

```
[relativ sti og filnavn hvor fragmentet indgår]

public class CodeFragment extends Infinity {

}
```

**Citater**

> ***A quotation*** *is the repetition of one expression as part of another one, particularly when the quoted expression is well-known or explicitly attributed (as by citation) to its original source, and it is indicated by (punctuated with) quotation marks.*
>
> *Wikipedia*

**Italics font**

Benyttes ved klasse- og metodenavne samt reserverede ord som ingår i teksten uden at være indkapslet i et kode fragment.

# HISTORIEN OM BDD

Dan North som er manden bag definitionen af BDD havde benyttet sig af, og undervist i agile udviklingsmetoder i en årræke som eksempelvis Test Driven Development (TDD) og i den sammenhæng identificerede han flere problemer. Det var de samme problemstillinger han løb ind i igen og igen.

- Hvor skal vi starte?
- Hvad skal vi teste og hvad skal vi ikke?
- Hvor meget skal testes på en gang?
- Hvad skal tests hedde?
- Hvorfor fejler en test?

Alle disse problemstillinger er noget som kun retter sig mod udviklerne, TDD tager ikke hånd om samspillet mellem forretningen og det tekniske.
Desuden som man også kan se på ovenstående punkter så kunne man få den tanke at TDD handler om at teste, TDD handler ikke om at teste, det er kun et positivt bi-produkt af TDD processen.
North begyndte derfor at kigge på hvordan man kunne forbedre hele processen omkring TDD.

## Sigende metodenavn og Agiledox

På et tidspunkt faldt han over et lille utility agiledox. Det den kunne var at tage en testklasse og fjerne ordet test fra klassenavne og metodenavne. Dens output kunne f.eks. se således ud for en testklasse som slår kunder op:
CustomerLookup
- finds customer by id
- fails for duplicate customers
- ...

Ikke umidelbart noget stort men det gør at hvis man har en regel om at metodenavne skal være sigende sætninger og skrevet i forretningsdomænets sprog så kunne det bruges både som en form for dokumentation og det gav pluselig mening at vise det til forretningsfolk, analytikere og testere.

## Fra test til opførsel

Der var dog stadig den ulempe at sproget omkring TDD stadig var for test centreret. Derfor fandt han på at starte hvert metodenavn med should i stedet for test.
Når man navngiver metoderne efter denne metode gør det at man er mere fokuseret på den klasse man nu er i gang med mht. ansvar, samarbejde og delegation.

Mens man udvikler koden så vil test's fejle en gang i mellem. Det kan skyldes tre ting.

- Der er lavet en fejl i koden, ret den..
- Det som testes er flyttet til en andet sted. Flyt testen og ret den evt. hvis det er nødvendigt.
- Det som der testes for er blevet slettet. Slet derfor også testen.

Når man følger en agile metode sker det i sagens natur jævnligt at man bliver klogere og derfor sletter noget funktionalitet som der ikke længere er brug for. Det gør så at testen fejler. Nogle som ikke er så rutineret indenfor TDD kan nogle gange have meget svært ved at slette den test som så ikke længere er nødvendig for der har jo været en grund til at den er lavet.
For at komme helt væk fra test tankegangen ændrer han også så det nu er opførsel man specificerer og ikke test man skriver, det hænger også godt sammen med at skrive should foran alle metoder. En klasses opførsel kan godt ændre sig over tid og det skulle dermed også gøre det nemmere at slette en specificeret opførsel.

Efter at North begyndte at tænke på opførsel i stedet for test mente han det var så stort et fremskridt at han nu kaldte det for Behaviour Driven Development.
I slutningen af 2003 besluttede han sig så for at lave et framework JBehave.
JBehave's sprog skulle dreje sig om at verificere opførsel og understøtte nogle at de ideer han var kommet på.

## Hvad starter men med

Nu var han kommet et stykke af vejen men det manglede stadig noget. Hvilken opførsel begynder man med at specificere? Her fandt han ud af under konstruktionen af JBehave at det var meget nyttigt at finde det vigtigste som systemet endnu ikke kunne. Det gør det nemt at finde ud af hvor man skal starte. Hvis det vigtigste er x så skal skal den nye metode hedde shouldDoX()

## Fælles sprog

I slutningen af 2004 besluttede North også at bruge denne behaviour driven tankegang på selve krav processen også, det krævede dog at de kunne opfinde et konsistent ordforråd/sprog for både testere, udviklere og forretningen. Lykkedes det ville det være et stort skridt i retningen af at eliminere nogle af de misforståelser der ellers var mellem forretningen og teknikken.

I slutningen af 2004 udgav Eric Evans en bog om Domain Driven Design, den beskrev en metode om hvordan man kunne modellere et system ved hjælp af et fælles sprog baseret på forretningen og helt ned på kode niveau.
North indså at det han prøvede at definere var et fælles sprog for selve analyseprocessen. Det viste sig at han havde et godt udgangspunkt, for i det firma han var havde de allerede en "story template", der bruges til at lave en user story og de fungerer samlet som krav til systemet.
Den havde formen:

> As a X
> I want Y
> So that Z

Den fungerer som en overordnet story for hvad systemet skal kunne, problemet er at den ikke er specifik nok til at angive præcist hvad der skal laves og der mangler en form for accept kriterie til story.
Han gav sig i kast med at finde en template som ikke var for stram men på den anden side struktureret nok til at splitte den op i sine dele og uatomatisere dem. Han endte ud med at beskrive accept kriterierne i scenarios. Disse scenarios har formen:

> **Given** some initial context
> **When** an event occurs

**Then** ensure some outcomes

Formen gør at man ved hvornår en story er kodet. Det er den når alle accept kriterier er opfyldt. Ydermere så er disse scenarios også eksekverbare og kan mappes direkte ned i Java kode.

## JBehave

BDD har udviklet sig fra udgangspunktet, efter at specifikke frameworks der er rettet mod BDD er fremkommet, og dette kan ses i kraft af hvordan BDD processen er ændret samtidig med hvordan JBehave har taget flere aspekter af agile teknikker ind i frameworket

JBehave har udviklet sig en del over de seneste par år og der kommer fortsat nye features og forbedringer til. Der vil her komme en sammenfattet oversigt over hvilke features der er kommet i hvilke versioner. Listen er dog ikke komplet da der er fokuseret mest på de ting som er brugt ifm. udviklingen af Ludo spillet, den komplette liste kan findes på www.jbehave.org.

### JBehave 1

Denne version var egentlig mest af alt et tanke eksperiment og et forsøg på at forstå og få afprøvet de nye koncepter omkring BDD. Det var aldrig tænkt som en kommerciel udgave som skulle kunne bruges af virksomheder til udvikling af deres software, den vil derfor heller ikke blive gennemgået men der vil blive startet fra JBhehave 2

### JBehave 2

Denne version var modsat den forrige beregnet til kommercielle projekter, men for at den kunne blive det var det nødvendigt at omskrive hele frameworket, primært fordi man var blevet klogere men i høj grad også pga. nye teknologier, bla. Java 5.

**JBehave 2.0**

Dette er den første version efter JBehave er totalt omskrevet, som følge heraf er der kommet nye features til og andre features er gledet ud ifht. det oprindelige frameworket.

Nye features

- Man kan nu skrive rene tekst scenarios
- Man kan have flere scenarios i en fil
- Man kan overføre parameter fra et scenario
- Man kan konvertere parametre til sine egne objekter

Features som er taget ud

- Integreret mocking. Man skal selv finde et framework og bruge det.
- Unit level framework. I stedet anbefales det at benytte JUnit.
- Swing support. Det er afløst af Tyburn frameworket.

**JBehave 2.1**

Nye features
- Nye annotationer, @BeforeScenario og @AfterScenario som bruges til reset af state i en multi scenario story.
- Support for Ant.

**JBehave 2.2**

Nye features
- Ny annotation, @Aliases så det er muligt at definere multiple aliases til steps.

**JBehave 2.3**

Nye features
- Nyt keyword GivenScenarios, gør at hele scenarios kan bruges som en precondition.
- Nyt keyword Examples, understøtter tabel eksempler i scenarios.
- Support for Named Parameters, skal bruges ifm. examples.

**JBehave 2.4**

Nye features
- Omfattende fil baseret scenario rapporterende framework til forskellige formater(txt, html, xml).
- Candidate steps klasser kan nu laves som POJO's. Dvs. de ikke arver fra Steps.

**JBehave 2.5**

Nye features
- Steps afhængigheder kan nu opsættes i IoC containers (Pico container, Spring og Guice).
- Nye annotationer, @BeforeStory og @AfterStory.

**Fremtidige versioner**

Lige pt. er det JBehave 2.5 som er den seneste men det arbejdes dog allerede på en 3.0. Den er også så langt henne at der er kommet en beta udgave. Der vil også lige blive set på hvad der er nyt i den.

**JBehave 3**

Dette er en videreudvikling af JBehave 2, baseret på de erfaringer man har samlet ved at bruge det i kommercielle projekter. Det generelle indtryk er at designet har holdt vand, dog er der brug for ændringer som kan ødelægge bagud kompatibiliteten, hvorfor man har valgt at udskyde det til JBehave 3.
Ændringerne vil blive beskrevet meget generelt.

Ændringer
- Fokus flyttet fra Scenarios til Story
- Muligt at få kørt flere stories samlet.

De stories man skrev i JBehave 2 er bagudkompatible på nær et punkt. Hvis man har brugt keyword GivenScenarios. Det er nemlig ændret til GivenStories.

På kode niveau er der sket en del ændringer, alle ændringer er dog isoleret til kun at have med opsætning at gøre. Det er pga. skiftet fra Scenarios til Story, alt som før hed noget med Scenarios hedder nu noget med Story. Andre ting har fået tilføjet Story for at tydeliggøre det er ifm. en story det bliver brugt. F.eks. så er klassen Configuration omdøbt til StoryConfiguration. Det betyder så at ens steps klasser kan flyttes direkte fra version 2 til 3.

# PRINCIPPERNE BAG BDD

BDD bygger oven på TDD og man kan derfor godt tolke det som 2. generation af TDD hvor BDD står på skuldrene af giganter og arver en gennemtestet og veldokumenteret teknik. BDD benytter sig også af elementer fra andre agile teknikker og forsøger at samle dem under samme paraply.
BDD har nogle principper der enten adskiller sig fra TDD eller skærper dem, disse principper afspejles i både udtryksform og af teknisk art.

## Filosofien

> ***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*

"Getting the words right" er et vigtig aspekt i BDD tankegangen, BDD vil gerne fjerne sig fra begrebet test, da test-first teknikken i den iterative process mere er et middel til at konstruere et godt design og kode efter best-practice metoder, end at det handler om at konstruere udtømmende tests. Dette er også gældende for TDD men man forsøger at skifte fokus fra begrebet test og i stedet forsøge at dreje folks tankegang over til at fokusere mod en ønsket opførsel, for at sikre at det leverede står i mål med det stakeholder ønsker og har behov for, samt at de opstillede milepæle i projektet overholdes.

## Sprog

> Forretning og teknologi skal benytte sig af samme sprog hvis der refereres til samme ide, der skal ikke foregå nogen oversættelse imellem parterne.

Ideen er hentet fra Domain-driven design hvor man benytter sig af et sprog til at beskrive de processer der indgår i et udviklingsforløb.

> ***Ubiquitous Language,** a language structured around the domain model and used by all team members to connect all the activities of the team with the software.*
>
> *Wikipedia*

Det fælles sprog der benyttes kaldes ”Ubiquitous Language” og er et simpelt sprog der kredser om nogle få ord og har en semantik der er helt klar for alle parter.

> ***A domain model*** *can be the core of a common language for a software project. The model is a set of concepts built up in the heads of people on the project, with terms and relationships that reflect domain insight. These terms and interrelationships provide the semantics of a language that is tailored to the domain while being precise enough for technical development. This is a crucial cord that weaves the model into development activity and binds it with the code.*
>
> *Eric Evans*

Begrundelsen for at benytte et fælles sprog er at alle stakeholders næppe skriver Java kode og at udviklerne ikke skriver automatiserede tests i almindelig dagligdags prosa, så for at sikre at alle parter har samme forståelse af en tekst er det nødvendigt at have defineret et simpelt sprog at udtrykke sig i.

BDD filosofien forudsætter at stakeholders i langt højere grad end i TDD involveres i opbygning og evaluering af softwaren ved aktivt at definere regler, krav og godkendelse i form af forretningslogik oversat til det simplificerede sprog. I TDD foretages der typisk en oversættelse fra en specifikation, eksempelvis en usecase og disse vil af udviklere og testere blive omskrevet til en testcase i et teknisk sprog som benytter eksempelvis JUnit. Dette resulterer i at ikke andre end folk med viden om det tekniske sprog kan konkludere om et kriterie der er opstillet ud fra de opstillede usecases er både accepteret og korrekt fortolket.

## Opførsel

> Udviklingen er styret af forretningen og den værdi en funktion giver til systemet når systemet er sat i drift.

Enhver opførsel et system tilbyder skal være der fordi det konkret tilfører noget forretningsværdi til systemet som helhed, denne indgangsvinkel gør det klart for alle parter at software koster penge og eftersom at penge er den vigtigste faktor i et hvert projekt, vil det virke som en bremse for både udviklere og stakeholder til at kaste sig ud i at bygge teknologi ind i systemet for teknologiens skyld, eller ønske opførsel der ikke er rentabelt i forhold til den pris det vil koste.

> ***Software users*** *tend to ask for a Rolls-Royce solution, when all they really need is a motor-scooter and for software developers trying to build an aircraft carrier when all that the users need is a row-boat.*
>
> *Dan North*

Et spørgsmål der ofte dukker op ved brug af TDD er "hvad og hvor starter man", BDD ligger dette spørgsmål ud til forretningen og her vurderer og prioriterer man på hvilken funktion der er den vigtigste i det forventede system og påbegynder opgaven med at definere og udvikle denne funktion. Denne fremgangsmåde gør at man implicit sikrer sig at princippet YAGNI overholdes, da ethvert stykke kode er krævet enten til at opfylde et krav stillet direkte af forretningen eller krævet af et allerede implementeret krav.

## Metoden

> Udviklingen af applikationen foregår "outside-in" og producerer et output der tilskynder en iterativ process for alle parter.

"outside-in" udvikling fordrer at man kun implementerer funktionalitet der er styret af forretningens krav, den bedste måde at realisere denne fremgangsmåde er at benytte sig af en top-down process og med et output der er nemt aflæselig for kontrol om opførslen af funktionaliteten er overens med kravene. Aflæsningen sker ofte i en form for brugerinterface, men kun hvis et brugerinterface er et vigtig element i systemet ellers vil alternative outputs som er dechifrerbare for alle parter også være acceptable. Et brugerinterface gør at parterne hele tiden kan følge processen i noget de kan forholde sig til, og definere nye specifikationer med udgangspunkt i det fælles sprog der giver ny funktionalitet til applikationen og på den måde får man nedbrudt specifikationerne i mere konkrete krav der driver udviklingen fremad.

# BDD PRAKTISERET MED JBEHAVE

JBehave ver. 2.5 er blevet valgt til at gennemgå processen i BDD, valget er foretaget eftersom at det var det første framework der kom til BDD og det mest omfattende framework der er skrevet i Java. Det der adskiller JBehave frameworket fra de andre frameworks der findes til Java er dens integrerede story-runner som giver mulighed for at specificere og eksekvere tekst-baserede stories indeholdende en række scenarios, dette er essentielt da BDD's design filosofi agiterer for en "outside-in" strategi.

> ***Outside-in*** *development focuses on satisfying the needs of stakeholders. The underlying theory behind outside-in software is that to create successful software, you must have a clear understanding of the goals and motivations of your stakeholders. Your ultimate goal is to produce software that is highly consumable and meets/exceeds the needs of your client.*
>
> *Wikipedia*

## Introduktion af Ludo

For at få indsigt i BDD filosofien og til at demonstrere JBehave frameworket er spillet Ludo blevet valgt som projekt, grunden til dette er at i mangel af interaktion med en eller flere udestående stakeholders ikke kan indgåes en dialog om hvilke features, roller og benefits der skal ingå i systemet. Features, roller og benefits er fast definerede i Ludo i form af et regelsæt der har været gennemprøvet igennem mange år, desuden er problemdomænet allerede godt kendt af udviklerne.

Basis regler for Ludo kan findes på http://wikipedia.org/wiki/Ludo

## Udvælgelse af stories

Formålet for denne øvelse er at evaluere BDD og demonstrere JBehave frameworket, så stakeholder er primært os selv som udviklere og læsere med interesse i emnet. Det er vigtigt at få defineret hvem og hvorfor software skal laves, eksempelvis vil et Ludospil der skulle bruges i en kommerciel sammenhæng eller udelukkende til børneinstitioner have en anden vægtning og prioritering af de features og benefits softwaren skal kunne levere end et Ludo spil der udelukkende eksisterer i kraft af en læringsprocess feks. i forbindelse med indlæring af et nyt programmeringssprog. I det her tilfælde er softwarens slutegenskaber ikke vægtet særligt højt, da det er processen i BDD filosofien der er vigtig og derfor finder man ingen eller meget lidt user interface i det stykke software som bliver udviklet.

På en kort samtale session fremkommer en række stories der beskriver features som har stor værdi for systemet, dvs. at disse features er essentielle for systemet som helhed og de er gode udgangspunkter for at generere nye stories og diskussioner. Det er kun stakeholder der kan afgøre om formålet med at spille Ludospillet er at vinde spillet, ikke at tabe spillet eller et helt tredje kriterie. For at holde en grundlinie i spillet er formålet med denne opgave at kunne spille spillet med

minimum regler fra start til slut med udgangspunkt i at finde en vinder.

### Første release

Derfor vil disse stories dække vores første release af Ludospillet, som så også er dem vi har fundet som minimum for at vores Ludospil kan spilles.

**Desktop user starts game**

> The desktop user executes the game and sees a welcome messages and ask for the number of players (2-4)

**Desktop user assign players to the game**

> The desktop user assigns the number of players that should participate in the game

**The system creates a game board**

> The system sets up a ludo board with homesectors, tiles and goalareas and present it to the players.

**The player enters a ludo piece from home onto the board**

> When the player in turn throws a die and the die shows the value ”6”, then player can enter a piece onto that players home globe on the board, if it is not already occupied by another piece belonging to the player in turn and the player still has pieces not on the board

**The player move a ludo piece around the board**

> When the player in turn throws a die, the player can move a piece already on the board the amount of tiles that the die value shows.

**Player wins the game**

> The first player that have moved all pieces into the goal area wins the game and a message is congratulating the player with the victory

En story er bygget som en rolle + en handling = en benefit, en rolle kan defineres som en hvilken som helst stakeholder i projektet (bruger, afdeling, ledelse, et andet system mv.). At man angiver en specifik rolle hjælper os til at tænke over kravet og hvorfor man skal implementere dette. Ludo spilles af 2-4 spillere og alle spillere har samme præcist samme funktion og mål, men i andre produkter vil man ofte finde forskellige roller der adskiller sig i den måde softwarens anvendes på. Eksempelvis vil en administrativ applikation der håndterer registrering af ordrer kunne forekomme mange forskellige roller (ordreafgiver, ordreindtaster, indkøber, sælger, produktionsafd., statistikmedarbejder osv.) der på forskellig vis ønsker at enten påvirke eller at trække på systemet.

Allerede på nuværende tidspunkt er der fremkommet flere stories hvor der skal en mere detaljeret diskussion igang om hvordan logikken er spundet sammen i den enkelte story. Ingen af disse stories

er medtaget i første release men er placeret i backlog'en for evaluering/prioritering til næste release. Der vil utvivlsomt komme flere stories til som opdages under implementeringen af både første og næste release, disse stories vil blive lagt ned i backlog'en til senere prioritering.

**A players piece knocks home an opponent players piece**

A player can knock home an opponents piece if the player moves to a tile that is occupied by the oppenent piece and that pieces is not placed on a safe tile

**A players piece jumps on startiles**

A player must make a jump to the succeeding star if he move a piece to a star tile unless it is the last star tile on the players path then the piece is moved into goal.

**A players piece is secured on globe tiles**

If a player piece is located on a tile that is either a neutral- or is the players homeglobe then that piece will be secured and cannot be knocked home.

**Desktop user restarts the game**

The desktop user can restart the game when a winner has been found and choose to play another game with new players-

Der kan henvises til Mike Cohns "User Stories Applied: For Agile Software Development" for en introduktion til hvordan processen med at indsamle gode stories fra stakeholders udføres i praksis.

JBehave benytter sig af en template som storien indskrives i, templaten sikrer at man får struktureret sin story og får taget stilling til de tre grundelementer som en story fortæller.

Story template i JBehave
**Story**: [A description of a feature]

**Narrative**:
**In order to** [achieve some value]
**As a** [role]
**I want** [to do something]

Den første story i releasen er ”Desktop user starts game”, og den indsættes i templaten således at den beskriver storien overodnet. Detaljer som storien beskriver undlades i templaten og beskrives senere i form af acceptkriterier.

**Story**: Start the game

**Narrative**:
**In order to** start playing a game of ludo

**As a** desktop user
**I want** to be able to start a  game so I can play with friends

### Hvad har vi gjort indtil nu?

Vi har lavet frembragt en række stories i form af almindelig prosa, som alle relaterer sig til regelsættet i et Ludo spil.

Der er sat fokus på hvilken rolle der opnår den ønskede værdi som en story giver og denne rolle indgår i beskrivelsen af storien.

Stories er blevet prioriteret og der er fundet de stories som ønskes med i den første release, resten af stories er gemt i backlog'en til prioritering i næste release.

JBehaves story template er gennemgået og første story er indsat så den passer ind i templaten.

### Hvad har vi lært?

At stories fremkommer ud fra samtaler og diskussion med stakeholder således at værdien  og rollen i storien fremstår tydeligt.

At der er mange måder at beskrive og forstå en story og derfor er det vigtigt at man får sat en story ind i en template der tydeligt fortæller hvad man vil opnå, hvem der får udbytte af den og hvordan story skal opfyldes.

Vi har fundet ud af at selve story indsamlingen er en iterativ process og at stories kan variere i størrelse og omfang.

Til sidst har vi lært at prioritere stories ind i små korte releases, hvor hver release kan evalueres af stakeholder så der er mindst mulig tab ved en evt. re-specifikation.

## Scenarios og steps

En story beskriver en feature, hvem den vedrører og hvad man får ud af denne feature. Dette er en start for hvad vi skal igang med at lave, men hvordan kan man så sikre sig at man rent faktisk har opfyldt kravene for denne feature? Dette gøres ved at tilføje et acceptkriterie til vores story, et acceptkriterie beskriver hvilke kriterier der skal være opfylt ud fra en given kontekst og er beskrevet i JBehave sammen med selve story i form af scenarios.

Et scenario bygges som en story op vha. en template som beskriver kontekst, handling og en accept. Scenarios indsættes efter story templaten og kan forekomme et kontinuerlig antal gange.

Scenario template i JBehave

**Scenario:** [Scenario description]
**Given** [a context]
**When** [an activity]
**Then** [Outcome]

En story i JBehave kan have mange scenarios, hvor hvert scenario beskriver et kriterie for hvordan man opnår at få den værdi ud som story beskriver. Et scenario beskriver ikke hvad der IKKE skal gøres for at et scenario opfyldes, da scenarios ikke er tests på om bagvedligende logik og struktur kan håndtere en forkert kontekst.

### De første skridt

I JBehave frameworket angives en story i en ren tekstfil og der findes så metoder til at få mappet story til en kompilerbar struktur.

```
[stories/start_the_game]

Story: start the game

Narrative:
In order to play a game of ludo
As a desktop user
I want to be able to start up a game so I can play with friends

Scenario: start up game
Given I am not yet playing
When I start a game
Then I should see 'Welcome to Ludo'
And I should see 'Submit number of players (2-4):'
```

Scenarios består af en række steps der kan bestå af 4 keywords: *Given, When, Then og And*

*Given* repræsenterer den kontekst som scenario skal operere i dvs. den state som vores system er i før vi kan foretage en handling

*When* repræsenterer vores handling der skal udføres for at opnå det ønskede accept kriterie

*Then* repræsenterer vores forventede opførsel

*And* repræsenterer ekstra *Given/When/Then* et kontinuerligt antal gange og kunne ligeså godt skrives med en ekstra *Given/When/Then* i stedet, grunden til at man benytter en *And* er at gøre

læsbarheden på scenario mere klar.

Der er nu oprettet en tekstfil med den første story og det scenario som opfylder kriteriet for at story kan levere den værdi som der er beskrevet.

### Mapning fra tekst til java klasser

JBehave frameworket er lavet således at det gør brug af JUnit frameworket, det har den fordel at alle miljøer der kan eksekvere en JUnit testcase også kan eksekvere et JBehave scenario.

JUnit kan ikke eksekvere en ren tekstfil med stories og scenarios, så der skal foretages en mapning over til en java klasse som JUnit kan eksekvere. JBehave gør dette ved først at der oprettes en klasse der nedarver fra JBehave klassen *Scenario* som benyttes til at mappe selve story tekstfilen med. Det er misvisende at klassen som nedarves hedder scenario eftersom at klassen dækker alle scenarios i en story og derfor vil det have være mere naturligt at kalde klassen for en story

```
[stories/StartTheGame.java]

package stories;

import org.jbehave.scenario.*;

public class StartTheGame extends Scenario {

}
```

Klassens default constructor læser automatisk den story tekstfil der matcher scenario klassens navn. Der er nu en klasse som kan eksekveres i JUnit og eksekveres denne fremkommer flg. output:

```
Narrative:
In order to play a game of ludo
As a desktop user
I want to be able to start up a game so I can play with friends

Scenario: start up game
Given I am not yet playing (PENDING)
When I start a game (PENDING)
Then I should see 'Welcome to Ludo' (PENDING)
And I should see 'Submit number of players (2-4):' (PENDING)
```

Output er ikke fantastisk overskueligt, men ud for hvert step i scenario er tilføjet en besked *(PENDING)* der fortæller at det aktuelle step endnu ikke er defineret, JUnit viser GRØN bar hvilket betyder at man kan oprette alle de udvalgte stories i den aktuelle release med det samme uden at JBehaves story runner fejler.

For at definere et step skal der oprettes en ny klasse der nedarver fra JBehave klassen *Steps. Steps* klassen skal indeholde alle steps der er angivet under hvert scenario i story tekstfilen, i form af metoder der repræsenterer *Given*, *When* og *Then*. JBehave foretager mapningen mellem steppet i den rene story tekstfil og til metoder i *Steps* klassen ved at annotere de enkelte metoder med den tekst der er angivet i steppet i story tekstfilen.

```
[stories/StartTheGameSteps.java]

package stories;
```

```
import org.jbehave.scenario.annotations.*;
import org.jbehave.scenario.steps.Steps;

public StartTheGameSteps extends Steps {

	@Given("I am not yet playing")
	public void NotYetPlaying() {
	}

	@When("I start a game")
	public void StartNewGame() {
	}

	@Then("I should see 'Welcome to Ludo'")
	public void ShowWelcomeMessage() {
	}

	@Then("I should see 'Submit number of players (2-4):'")
	public void ShowNoOfPlayersMessage() {
	}

}
```

For at få scenarioklassen til at kunne eksekvere denne stepsklasse skal stepsklassen leveres med til scenarioklassens constructor.

```
[stories/StartTheGame.java]

package stories;

import org.jbehave.scenario.*;

public class StartTheGame extends Scenario {

	public StartTheGame() {
		super (new StartTheGameSteps());
	}
}
```

Eksekveres scenario klassen i JUnit vil der ikke fremkomme noget output overhovedet og JUnit vil vise GRØN bar. En god praksis er derfor at få scenario til som udgangspunkt at fejle, dette kan gøres ved at indsætte JBehaves pendant til JUnits assert mekanisme og opsætte et udtryk der er usandt. *Ensure* er en statisk klasse der indeholder en metode *ensureThat()*, og den insættes som illustreret nedenfor i stepsklassens *Then* metode.

```
[stories/StartTheGameSteps.java]

@Given("I am not yet playing")
public void NotYetPlaying() {

}

@When("I start a new game")
public void StartNewGame() {

}
```

```
@Then("I should see 'Welcome to Ludo'")
public void ShowWelcomeMessage() {
      ensureThat(false);
}

@Then("I should see 'Submit number of players (2-4)?'")
public void ShowNoOfPlayersMessage() {
      ensureThat(false);
}
```

Grunden til at JBehave frameworket ikke bare benytter sig af assert fra JUnit er at folkene bag JBehave mener at Assert er en dårlig betegnelse der leder over i begrebet test og dermed ikke er i tråd med filosofien om at BDD ikke er test, men specifikation af opførsel.

Metoden *ensureThat()* kan enten validere et boolean udtryk eller validere en værdi op mod en matcher fra Hamcrest frameworket.

Efter at *Step* er instrueret om at fejle vil output fra kørslen af scenarioklassen vise.

```
Narrative:
In order to play a game of ludo
As a desktop user
I want to be able to start the game so I can play with friends

Scenario: start a new game
Given I am not yet playing
When I start a game
Then I should see 'Welcome to Ludo' (FAILED)
And I should see 'Submit number of players (2-4)?' (NOT PERFORMED)
```

De enkelte steps er nu ikke længere markeret med beskeden *(PENDING)* eftersom at der nu er oprettet metoder til disse, i stedet er den første *Then* metode markeret med beskeden *(FAILED)*. Som man kan se i outputtet er den anden *Then* step slet ikke kørt, så den markeres med *(NOT PERFORMED)*. JUnit angiver der er fejl i eksekveringen af scenario klassen og viser RØD bar, som forventet når et step fejler.

### Hvad har vi gjort indtil nu?

Der er beskrevet en række stories og ud fra dem defineret hvilke stories der skal indgå i første release af Ludospillet, derudover er der defineret en række scenarios hvor der opsættes acceptkriterier for hver story således at den opfylder den feature som er beskrevet i storien.

De udvalgte stories med tilhørendene scenarios er oprettet som rene tekstfiler i en form der overholder story templaten *"In order to, As a, I want to"* og efterfølgende scenario templaten *"Given, When, Then"*, derefter er der oprettet en JBehave *Scenario* klasse som matcher story tekst filen.

JBehave *Steps* klassen introduceres, og den gives med til *Scenario* klassen hvor alle steps i en story oprettes med en klasse metode og annoteres med *Given, When, Then* og bruger scenario step beskrivelsen som parameter.

JBehave *Scenario* klassen er blevet eksekveret i JUnit og vi har set den fejle i dens *Then* step.

### Hvad har vi lært?

Hvordan JBehave mapper scenarios og steps fra tekstbaserede stories til java klasser og hvordan

man eksekverer scenarios ved hjælp af JUnit frameworket.

## Beskriv specifikationer med eksempler

På nuværende tidspunkt er der intet applikationskode tilgængeligt til at skrive scenario steps op mod, så derfor skriver man til et imaginært framework som man godt kunne tænke sig var der set fra den kontekst som eksemplet arbejder ud fra.

### Given

*Given I am not yet playing* er første step, her indskrives hvad der skal være kontekst / precondition for scenario, men eftersom at *I am not yet playing* er en kontekst hvor der ikke er noget opsætning der varetages fra applikationen side lader vi *Given* metoden være tom og dermed intet foretager sig. Som vi så tidligere vil en metode der intet foretager sig blive accepteret af JBehave som værende i orden, så det er acceptabelt at have steps der intet foretager sig.

### When

I *When* metoden angives den handling der skal foretages i en given kontekst til at opnå acceptkriteriet i *Then* metoderne. *When I start a game* beskriver den handling der skal foretages og eftersom der stadig intet applikationskode er til rådighed skriver vi til det imaginære framework hvad der virker mest naturligt set udefra. Handling fortæller os at der skal startes et spil så vi opretter i *When* metoden flg. kodelinie

```
[stories/StartTheGameSteps.java]

 @When("I start a game")
public void StartAGame() {
      game.start();
}
```

### Then

*Then* kriteriet lyder *I should see 'Welcome to Ludo'* og *I should see 'Submit number of players (2-4):'*. Problem: Der er ingen steder i scenario eller story der angiver hvad der menes med at 'see'; Der kan antages at der tales om en besked af visuel karakter, men en diskussion med stakeholder er nødvendig for at specificere om denne besked vises på skærmen i et konsolvindue, et html dokument i en browser eller måske på en printer.

Beslutning om ”see” i denne story er at det skal ses i et konsol vindue eller anden form for gui der kan vises på skærmen. For at kunne aflæse et konsol vindue er det nødvendigt at få denne under en eller anden form for kontrol således at output kan aflæses. Et valg til dette kunne være at benytte en testdouble der har den egenskab at kunne simulere et rigtigt objekt, i det her tilfælde et konsolvindue.

Igen starter man med at skrive mod den kode man kunne ønske sig og det vurderes at den simpleste måde at håndtere et konsol output er vha. en liste der indeholder alt hvad der bliver smidt ud til konsol vinduet.

```
[stories/StartTheGameSteps.java]

@Then("I should see 'Welcome to Ludo'")
public void ShowWelcomeMessage() {
```

```
        ensureThat(output.messages(), hasItem(”Welcome to Ludo”));
}

@Then("I should see 'Submit number of players (2-4):'")
public void ShowWelcomeMessage() {
        ensureThat(output.messages(), hasItem(”Submit number of players
(2-4):”));
}
```

Der benyttes en matcher fra Hamcrest frameworket *hasItem* til at evaluere om en given tekst findes i en kollektion af strenge. Første parameter er den aktuelle struktur/værdi der ønskes foretaget en evaluering på og anden parameter er en passende matcher fra Hamcrest frameworket. Hamcrest frameworket stiller en række matchers til rådighed og JBehaves *ensureThat()* metode har fuld support for disse matchers. Hamcrest frameworket brillierer ved sammen med JBehaves *ensureThat()* at gøre en evaluering meget læsbar og giver en god forståelse for hvad det er for en opførsel man ønsker at evaluere.

Linien læses som >> *ensure that output messages has item ”Welcome to Ludo”*

Hamcrest matchers kan også bruges med JUnits assert metode.

Køres storyen rapporteres der tilbage fra compileren at game ikke er kendt, BDD rytmen angiver at man skriver sin eksempler og får strukturen på plads (ydre iteration) og så efterfølgende dykker ned i strukturen for at implementere de enkelte metoder til at kunne levere den ønskede opførsel (indre iteration). På den baggrund skal eksemplet kunne kompileres og der introduceres et *Game* interface med metoden start*()* samt en implementerende klasse *GameImpl*.

*[framework/Game.java]*

```
package framework;

public interface Game {
        public void start();
}
```

Java er et statisk sprog, så der skal tildeles en instans af klassen.

*[stories/StartTheGameSteps.java]*

```
@When("I start a game")
public void StartAGame() {
        Game game = new GameImpl();
        game.start();
}
```

Storyen eksekveres igen og compileren rapporterer fejl på *output* i *Then* metoden.

For at kunne aflæse konsol output introduceres der en *”Test-spy”* der optager alt hvad der sendes til konsollen for verificering af scenario. Interfacet *Output* oprettes med en setter og en getter. Setteren kommer til at hedde *put(String message)* og gemmer en streng i en collection af strenge der er sendt til konsollen, getteren *List<String> messages()* returnerer streng collectionen til verificering.

*[framework/Output.java]*

```
public interface Output {
	public void put(String messages);
	public List<String> messages();
}
```

Implementationen af interfacet vurderes til at være så simpelt at ”Obvious implementation” tages i brug.

```
[stubs/OutputImpl.java]

public class OutputImpl implements Output {

	List<String> messages;

	public OutputImpl() {
		messages = new ArrayList<String>();
	}

	public List<String> messages() {
		return messages;
	}

	public void put(String message) {
		messages.add(message);
	}
}
```

*Game* bliver nødt til at have en instans af *output* med for at kunne skrive til den og det virker fornuftigt at angive dette i forbindelse med instantiering af *game* objektet.

```
[stories/StartTheGameSteps.java]

public class StartTheGameSteps extends Steps {

	Output out;

	@When("I start a game")
	public void StartAGame() {
		out = new OutputImpl();
		Game game = new GameImpl(out);
		game.start();
	}
}
```

Storien køres igen og compileren fortæller at der ikke findes en constructor i GameImpl der tager en Output som parameter, så den indsættes med det samme.

```
[game/GameImpl.java]

public class GameImpl implements Game {

	public GameImpl(Output output){
	}
```

```
        public void start() {
        }
}
```

Storyen eksekveres igen og compilerer korrekt, JBehave rapporterer fejl på den første *Then* og dette afslutter den ydere iteration i scenario afviklingen.

```
Narrative:
In order to start playing a game of ludo
As a desktop user
I want to be able to start up a game so I can play with friends
Scenario: start up game
Given I am not yet playing
When I start a game
Then I should see 'Welcome to Ludo' (FAILED)
And I should see 'Submit number of players (2-4):' (NOT PERFORMED)
```

Næste fase i BDD rytmen er at hoppe fra den ydre iteration ind til den indre iteration hvor der skiftes fokus fra eksempel på interaktion i mellem klasser og til interaktion internt i de enkelte klasser, opgaven er nu at få scenarios *Then* til at levere det korrekte resultat.

### Hvad har vi gjort indtil nu?

Vi har afviklet en cyclus i den ydre iteration og har opnået dette ved hjælp af et eksempel der er defineret med *Given, When, Then*.

Vi har defineret klasse stubs til at kunne hægte den ønskede opførsel op på og vist hvordan disse klassestubs skal bruges for at opnå den ønskede opførsel, dvs. vi har opfundet det interface vi godt kunne tænke os vi havde.

Vi har skridt for skridt arbejdet os igennem den ydre iteration og bibeholdt fokus på præcis det story beskriver.

Vi har benyttet os af en ”test spy” til at opfange beskeder der sendes til et konsol vindue.

### Hvad har vi lært?

Vi lært at beskrive opførsel ”Outside-in” i kraft af et eksempel på hvordan den endnu ikke eksisterende software skal bruges. Dette har hjulpet os til at drive ansvar og opførsel ud af vores story og scenario, som bliver realiseret i kraft af interfaces der servicerer vores eksempel. Vi har gjort det absolut mindste og det simpleste vi kan gøre for at få vores scenario eksempel  til at virke og forsøgt at være så ”clean” som mulig i form a læsbarhed og forståelse. BDD viser med alt tydelighed at teknikken fra TDD med at tage små skridt også virker ved eksempel konstruktion.

Derudover har vi lært at en story ikke er entydig og specielt brugbar til at danne en kontrakt mellem stakeholder og udvikler, en story ligger op til yderligere diskussion og fortolkning og kan ændres flere gange sammen med stakeholder under en iteration, dette vist ved at vi ikke på forhånd ville kunne vide hvordan beskeden visuelt skulle præsenteres.

## Implementering af eksempler

BDD er inde i maven TDD, og det er TDD teknikken der benyttes for at få den ønskede opførsel ud af de interne klasser ud til eksemplerne, således at eksemplerne kan eksekveres i JBehaves story runner uden at fejle.

> *"**The act of writing a unit test** is more an act of design than of verification. It is also more an act of documentation than of verification. The act of writing a unit test closes a remarkable number of feedback loops, the least of which is the one pertaining to verification of function".*
>
> *Robert C. Martin*

### BDD på klasselevel

BDD filosofien anbefaler at man benytter samme teknik som traditionel TDD, dog skal ordlyden skifte fra at fokus ligger på at beskrive tests til at beskrive opførsel. Dette betyder at i stedet for at lave feks. en GameTest class, danner vi en GameBehaviour class.

> ***Test-driven development cycle***
>
> *Quickly add a test*
> *Run all tests and see the new one fails*
> *Make a little change*
> *Run all tests and see them all succeed*
> *Refactor code to remove duplication*
>
> *Kent Beck*

Som i TDD benyttes test-first teknikken, derfor vil der ikke blive skrevet noget produktionskode før der er lavet et fejlende eksempel.

Det som vi som start ønsker at lave er *game.start()*, derfor skal denne opførsel implementeres i kraft af det som man i BDD kredse kalder en behaviour class. En behaviour class er det samme som en ganske almindelig test class, men man har valgt at kalde den en behaviour class for at sikre at folk, for det første ikke spekulerer i test, desuden er der opsat retningslinier for hvordan metoder navngives for at understøtte opførselstankegangen. En metode bør starte med "should" for at holde fokus på at det er opførsel og ikke test vi er interesseret i.

### Start med et eksempel der fejler

Scenario har allerede defineret at *Game* skal implementere en *start()* metode som derefter vil påvirke state i *Output*, så derfor vil *GameBehaviour* klassen beskrive hvordan *Game* opfører sig hvilket så til gengæld vil give en viden om hvordan den indre interaktion imellem *Game* og *Output* klassen vil blive.

Klassen annoteres med JUnits *@Test* og opbygges som nedenstående. Nedenstående eksempel er stort set identisk med det eksempel der blev lavet i scenario, men vi skelner imellem disse to ved at vores scenario er vores acceptance test og den fortæller os hvornår vi er færdige med denne feature, den egenskab og det formål er ikke hvad behaviour klassen skal udtrykke.

```
[behaviour/GameBehavior.java]

public class GameBehaviour {

	@Test
	public void shouldGiveAWelcomeMessage() {
		//Given
		Output out = new OutputImpl();
		Game game = new GameImpl(out);

		//When
		game.start();

		//Then
		ensureThat(out.messages(), hasItem("Welcome to Ludo"));
	}
}
```

JUnit testen køres og giver RØD bar, *start()* skal levere ”Welcome to Ludo” til output for at der kommer en grøn bar og *out.messages()* leverer ingenting

### Få eksemplet til at passere

*OutputImpl* er igen benyttet til at få *game.start()* implementeret og lader game.start() tilføje beskeden ”Welcome to Ludo” ved kald til metoden *put()*.

```
[game/GameImpl.java]

public void start() {
	output.put("Welcome to Ludo");
}
```

JUnit køres på *GameBehaviour* og der er GRØN bar, det betyder at der returneres til ydre iteration og scenario eksekveres igen.

```
Story: Start the game
(stories/start_the_game)
Narrative:
In order to start playing a game of ludo
As a desktop user
I want to be able to start up a game so I can play with friends
Scenario: start up game
Given I am not yet playing
When I start a game
Then I should see 'Welcome to Ludo'
And I should see 'Submit number of players (2-4):' (FAILED)
```

Første *Then* er nu i orden og der skiftes fokus til at få den sidste *Then* til at køre igennem korrekt, så der skiftes endnu engang til den indre iteration og *GameBehaviour* klassen.

Sidste acceptkriterie i scenario er at vise ”Submit number of players (2-4):”, så der oprettes en ny ”test” i behaviour klassen der er stort set identisk med den foregående blot med en anden tekst. Køres denne test på *GameBehaviour* klassen er vi tilbage til RØD bar og kan konkludere at *GameImpl.start()* metoden skal udvides så den leverer den ekstra besked.

```
[game/GameImpl.java]

public void start() {
      output.put("Welcome to Ludo");
      output.put("Submit number of players (2-4):");
}
```

GRØN bar ved kørsel af JUnit på *GameBehaviour* klassen, ved kørsel af scenario i den ydre iteration køers ligeledes igennem og giver GRØN bar, så vi kan konkludere at første story er gennemført i både ydre og indre iterationer.

### Hvad har vi gjort indtil nu?

Vi har ud fra de interfaces der er fremkommet i scenario eksemplerne dannet en behaviour klasse til at drive implementationen af disse interfaces frem i kraft af *GameImpl*.

Vi har lavet en ”test” der fejlede og derefter fået den til at passere JUnit kørslen vha. klassisk TDD fremgangsmåde.

Vi har gjort mindst muligt for at få den ønskede opførsel ud af game.start() ved at proppe teksterne ned i strengcollectionen i *OutputImpl*.

### Hvad har vi lært?

Vi har lært at drive produktionskode ud af de interfaces vi har fundet frem fra vores scenarios vha. en indre iteration som fokuserer på den interaktion der er inde i klasserne.

Vi har lært at skifte mellem ydre og indre iteration indtil ydere iteration har fået acceptkriterierne tilfredstillet.

## Næste feature

Første story er kørt igennem og der tages fat på den næste story ud fra en prioritering der er sat ud fra en naturlig placering i hierakiet og  ud fra hvilken værdi featuren vil tilføre til systemet.

Næste feature der ønskes implementeret er muligheden for at angive antallet af spillere, selve implementationen af denne feature bringer ikke meget nyt frem under som ikke blev beskrevet i forrige afsnit, men der er en afgørende forskel og det er at denne story opererer med flere acceptkriterier.

### Assign Players To The Game

**Desktop user assign players to the game**

> The desktop user assigns the number of players that should participate in the game.

```
[stories/assign_player_to_the_game]

Story: Assign players to the game

Narrative:
In order to decide on how many players that should play ludo
As a desktop user
I want to be able to assign between 2-4 players to the game
```

Kigger man på denne story kan man udlede at der kan vælges i mellem 2 til 4 spillere, dette betyder at story kan have flere forskellige acceptkriterier.

Aktuelle story kan således have tre forskellige valide acceptkriterier: 2 spillere, 3 spillere og 4 spillere.

```
[stories/assign_player_to_the_game]

Scenario: submitting 2 players and show colors
Given 'Submit number of players (2-4):' is shown
When submitting 2 players
Then I should see '2 Player game'
And I should see 'Player 1 is red'
And I should see 'Player 2 is blue'

Scenario: submitting 3 players and show colors
Given 'Submit number of players (2-4):' is shown
When submitting 3 players
Then I should see '3 Player game'
And I should see 'Player 1 is red'
And I should see 'Player 2 is blue'
And I should see 'Player 3 is green'

Scenario: submitting 4 players and show colors
Given 'Submit number of players (2-4):' is shown
When submitting 4 players
Then I should see '4 Player game'
And I should see 'Player 1 is red'
And I should see 'Player 2 is blue'
And I should see 'Player 3 is green'
```

```
And I should see 'Player 4 is yellow'
```

I ovenstående tilfælde er det forholdsvist overskueligt at få implementeret de tre scenarios, men havde der været mulighed for at spille med 2 – 16 spillere i Ludo vil det være et voldsomt footprint at skule have 15 acceptkriterier implementeret i steps klassen, hvor hvert scenario er stort set identisk med det foregående.

JBehave giver os en mulig løsning på dette problem, ved at introducere table examples, denne konstruktion giver os mulighed for at opstille de varierende parametre på tabelform og JBehave vil så injecte disse parametre ind i vores scenario såfremt at disse kan opsættes i et scenario på samme regex. form.

```
Scenario: submitting players and show colors

Given 'Submit number of players (2-4):' is shown
When submitting <players> players
Then I should see '<players> Player game'
And I should see 'Player <number> is <color>'

Examples:
|players|number|color
|2|1|red
|2|2|blue
|3|1|red
|3|2|blue
|3|3|green
|4|1|red
|4|2|blue
|4|3|green
|4|4|yellow
```

De parametre som tabelværdierne skal erstatte omkranses af ”<, >” for at indikere at der skal benyttes en værdi fra tabellen der svarer til navnet på parameteren.

Ovenstående måde at opstille scenarios på gør det langt nemmere at håndtere multiple acceptkriterier der har det til fælles at det kun er parameter værdier der adskiller disse.

De enkelte steps i *Steps* klassen skal så kunne tage en parameter der mappes med parameternavnet og dette kan gøres på forskellige måder. I denne implementation er der valgt det der i JBehave kaldes en ”Named Parameter” og illustreres nedenunder i *When* metoden med annotationen *@Named*.

```
[stories/AssignPlayersToTheGameSteps.java]

@When ("submitting <players> players")
public void ShouldSubmit2Players(@Named("players") int players) {
      game.submitPlayers(players);
}
```

For en mere detaljeret gennemgang af table examples og parameter injections ind i steps klassen henvises der til kapitlet om JBehave teknikaliteter.

### Hvad har vi gjort indtil nu?

Fundet næste feature der skal implementeres og slået scenarios sammen til et scenario med en ”Example table”

### Hvad har vi lært?

Vi har lært at minimere antallet af scenarios og steps metoder når der er en række acceptkriterier hvor kun værdien af en eller flere parametre afgør om scenarios accepteres.

## JBehave teknikaliteter

Når man benytter JBehave stiller det funktionalitet til rådighed som gerne skulle kunne hjælpe med at gøre processen nemmere og mere overskuelig. I dette afsnit vil der blive gennemgået de JBehave features som der er brugt i forbindelse med udviklingen af Ludo.

Nogle af de emner det vil blive gennemgået er:

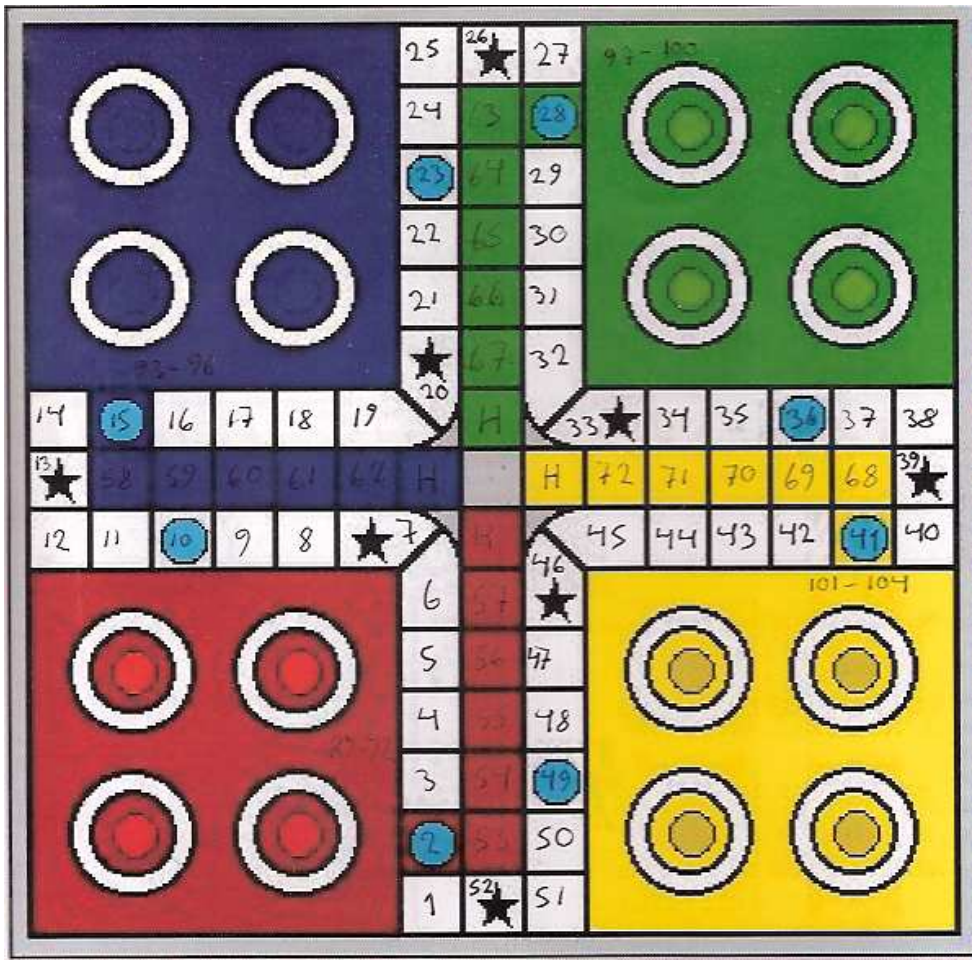
- Table examples
- Parameter injection
- Dependency injection

Derudover vil der også blive gennemgået lidt om mocking og hjælpemetoder som ikke direkte har noget med JBehave at gøre.

### Introducering af brættet

For rent faktisk at kunne spille Ludo kræver det også man har et bræt, det der skal sikres her er at brættet har et bestemt udseende, dvs. har de rigtige felter på de rigtige steder.

Hvert felt på brættet har en position, position 1 er valgt som det neutrale felt lige før den røde globus og efter stjernen ved indgangen til røds center felter. Herefter tælles positionen en op for hvert felt når man går i urets retning. Hjemme felterne, center felterne og mål feltet ligger alle fra position 53 og op efter. Målfelterne er markeret med et H. Det er oprettet 4 pr. farve selv om der på brættet kun er 1 pr. farve.



Da det gælder om at de forskellige felter er de rigtige steder på brættet kunne man godt begynde at skrive nogle scenarios i stil med shouldHaveARedHomeSectorInPos77(), shouldHaveABlueHomeSectorInPos81() osv., det vil dog gøre at der ville komme rigtig mange scenarios hvilket ville gå ud over overskueligheden og efterfølgende opsplitning i flere stories.

Scenarios er grupperet efter tileType hvilket giver i alt disse 6 scenarios i story

GameSetsUpTheBoard:

1. Setting up home tiles
2. Setting up globe tiles
3. Setting up neutral tiles
4. Setting up start tiles
5. Setting up center tiles
6. Setting up goal tiles

I alle disse scenarios bliver der så gjort brug af JBehave's Table Examples.

```
[story/GamesSetsUpTheBoard]

Scenario: setting up a home tiles
Given the use of a standard board
When the game begins
Then a <tileType> with <color> should be in <pos>

Examples:
|tileType|color|pos|
|home|red|89|
|home|red|90|
|home|red|91|
|home|red|92|
|home|blue|93|
...
```

I princippet kunne alle scenarios være samlet i et da de alle har samme regex pattern og dermed også bruger de samme metoder i steps klassen. Grunden til de er delt op er fordi det letter læsbarheden da det nu ikke er et rigtig stort table example men 6 mindre.

### Table examples

Når man sidder og skriver scenarios kan man ofte se at nogle af dem ligner hinanden rigtig meget, det eneste som egentlig ikke er ens er nogle få parametre. Eksemplet ovenfor illustrerer det meget godt, det som ville have varieret her var nede i *Then*.

- Then a neutral tile with color none should be in position 1
- Then a home tile with color red should be in position 89
- Then a home tile with color red should be in position 90
- osv.

I praksis laver man en template for det generelle scenario og så indsætter en parameter som anføres i <parameter>, efter selve scenario template opbygger man så alle de eksempler man har brug for. Dette gøres ved at bruge keyword ”Examples” og så selve tabellen. Den første linie i tabellen er navne på de parametre der indgår adskilt med ”|”. De efterfølgende linier er så de værdier der overføres til parametrene.

Scenario køres en gang for hver række i tabellen, dvs. i eksemplet med home tiles er der 16 rækker og derfor vil scenario blive kørt 16 gange.

I dette eksempel er de alle i *Then* men man kunne godt komme ud for at der både var en parameter i både *Given*, *When* og *Then*.

### Parameter injection

Parameter injection bruges når man skal have felter fra tekst scenario med helt ned i java metoden, nøjagtigt det som er tilfældet når man benytter table examples. JBhave understøtter to former for injection, ”Named parameters” og ”Ordered parameters”.

#### Named parameters

Når man oprettet sin example table, hvordan får man så parametrene med ned i java koden?

Her bruges JBehave's ”Annotated Named Parameters”.

Her er et eksempel på hvordan det ser ud:

```
[story/GameSetsUpTheBoardSteps.java]

@Then("a <tileType> with <color> should be in <pos>")
public void shouldHaveTileTypeWithColorInPos(@Named("tileType") String
      tileType, @Named("color") String color, @Named("pos") int pos) {

      TileType tt = findTileType(tileType);
      Color c = findColor(color);

      ensureThat(existTileInPos(tt, pos, c));
}
```

For at angive der kommer en named parameter skriver man:

@Named(”Parm navn”) type navn

hvor ”Parm navn” henviser til navnet anført i <parm navn>, type = variable typen i java og navnet er parameter navnet i metoden. En fordel ved named parameter er at rækkefølgen i metodens parameter liste er ligegyldig, dvs. de behøver ikke stå i samme rækkefølge som i regex mønsteret.

#### Ordered parameters

Denne metode er noget simplere, men ikke så fleksibel.

Hvis man tager eksemplet fra før og skriver det lidt om så det ser sådan ud:

```
Scenario: setting up a home tiles
Given the use of a standard board
When the game begins
Then a home tile with red color should be in position 77
```

Som det ses er der ikke nogen steder der bliver angivet hvad man ønsker sendt som parameter. Dette flytter man i stedet ind i steps klassen og i dette eksempel under *Then*.

```
@Then("a $tileType tile with $color color should be in position $pos")
public void shouldHaveTileTypeWithColorInPos(String tileType, String
      color, int pos){
            .
            .
}
```

I regex mønsteret er home skiftet ud med $tileType, red med $color og 77 med $pos. Det den så gør er at sende det ord der står på $navn's plads ned til java metoden som en parameter. Det betyder også at her har rækkefølgen en betydning, de skal stå i samme rækkefølge i både regex mønsteret og i metodens parameter liste.

### Enum i example tables

To af de parametre som er med i example tables er Java enum's, det drejer sig om *tileType* og *color.* Det gav den udfordring at det ikke er muligt at konvertere en String til en enum, og giver en fejl når man kører stepsklassen. Løsning var at der blev lavet en hjælpemetode som vha. streng sammenligning, oversætter strengen til den tilsvarende enum.

```
[stories/GameSetsUpTheBoardSteps.java]

private Color findColor(String color) {
      if (color.equals("red"))
            return Color.red;
      if (color.equals("blue"))
            return Color.blue;
      if (color.equals("green"))
            return Color.green;
      if (color.equals("yellow"))
            return Color.yellow;

      return Color.none;
}
```

Ikke det store problem men det havde dog været pænere at kunne skrive *Color.red* som værdi i ens example table.

Der kunne have været valgt en anden strategi; JBehave's parameter converter. JBehave har standard converters for strenge og tal samt tilsvarende liste converters til dem. Det der er brug for her er en converter til typen *Color* og *TileType* og det man kan gøre er at lave sin egen converter ud fra JBehave interfacet *ParameterConverter.* Derefter tilføjer man sin converter i steps klassen og man er klar til at køre. Læs mere om det på www.jbehave.org under parameter converters.

### Hjælpemetoder

Hjælpemetoder bruges når man ønsker at splittet noget op i mere overskuelige dele i en klasse, eller også når man har noget kode som bliver brugt flere forskellige steder i klassen. Hjælpemetoder er ofte med til at skabe et bedre overblik, dog skal man huske at tænke på navngivningen så det er let at se hvad den gør.

I *Steps* klassen er der lavet en hjælpemetode som undersøger om der en en bestemt *Tile* på en bestemt position. Her bruges den til at skabe overblik, og man kan hurtigt se hvad man ønsker at verificere på i *Then.*

```
[stories/GameSetsUpTheBoardSteps.java]

@Then("a <tileType> with <color> should be in <pos>")
public void shouldHaveTileTypeWithColorInPos(@Named("tileType")
      String tileType, @Named("color") String color,
      @Named("pos") int pos){

      TileType tt = findTileType(tileType);
      Color c = findColor(color);

      ensureThat(existTileInPos(tt, pos, c));
}

private Boolean existTileInPos(TileType tt, int pos, Color c) {
      Iterator<Tile> i = game.getBoardTiles();
      while (i.hasNext()) {
            Tile t = (Tile) i.next();
            if (t.getTileType() == tt &&
                t.getPosition() == pos && t.getColor() == c) {
                  return true;
            }
      }
      return false;
}
```

Man skal passe meget på, ikke at få for mange eller for hårdt koblede hjælpemetoder, det har den konsekvens at man ikke umiddelbart kan se hvad der testes på uden at skulle springe frem og tilbage i koden.

### POJO

Hvis man kigger på klassen *GameSetsUpTheBoard* vil man opdage at den ikke nedarver fra JBehave's *Steps* klasse som normalt.

I det første scenario som blev gennemgået blev der brugt nedarvning til at få kørt storien. Der var *DesktopUserStartsGame* som arver fra *Scenario* og *DesktopUserStartsGameSteps* som arver fra *Steps*, begge fra JBehave. JBehave gør det muligt at lave alle sine steps ved hjælp af komposition. Helt generelt er det bedre at bruge komposition frem for nedarvning da det giver en lavere kobling og er derfor også begrundelsen for at dette er muligt i JBehave.

Hvis ens step klasse er en POJO så skal man bruge en stepsfactory til at få lavet sine candidate steps og den eneste forskel fra før er at stepsklassen ikke arver fra *Steps*. Klassen *GameSetsUpTheBoard* arver stadig fra *Scenario* men bruger nu *StepsFactory* i sin constructor til at danne steps med.

```
[stories/GameSetsUpTheBoard.java]

public class GameSetsUpTheBoard extends Scenario {

      public GameSetsUpTheBoard() {
            addSteps(new StepsFactory().createCandidateSteps(
                        new GameSetsUpTheBoardSteps()));
      }
}
```

## Board behaviour

I steps klassen ser man at *Game* skal kunne levere en iterator over tiles retur. Men hvem skal have ansvaret for at holde listen med tiles?. Er det *Game* eller skal det være et helt andet sted?. Det blev valgt, at det ikke var *Game* som skulle tage ansvar for dette,men derimod et nyt interface *Board* som får dette ansvar. Der er tilføjet en tom metode *Game.getBoardTiles()* som returnere en iterator med tiles. *Board* må derfor også skulle returnere en iterator eller selve listen. Den metode tilføjes interfacet med det samme og det bliver valgt at den giver selve listen retur.

Efter som det er opbygning af selve brættet der er interessant lige nu er det *Board* klassen der skal fokuseres på. Det betyder at der oprettes en ny klasse, nemlig *BoardBehaviour.*

### Hvilke eksempler laves

Næste spørgsmål er så hvilke eksempler skal der laves i *Boardbehaviour*? Man kunne begynde at lave eksempler med at der skal være en stjerne i position X osv. men det er jo præcist det samme som er lavet i scenarios så det giver måske ikke så meget mening. I stedet er det valgt disse:

1. `shouldHave4HomeTilesInEachColor()`
2. `shouldHave8StarTiles()`
3. `shouldHave36NeutralTiles()`
4. `shouldHave4NoneColoredGlobeTiles()`
5. `shouldHave1GlobeTileInEachColor()`
6. `shouldHave6CenterTilesInEachColor()`
7. `shouldHave1GoalTile()`
8. `shouldHave93Tiles()`

Alle disse eksempler går udelukkende på antal, dvs. der sikres udelukkende at der er det rigtige antal af de forskellige tiles mens kontrollen af om de så er på de rigtige pladser overlades til scenario. En ting som scenario ikke har med er om der er for mange tiles, det har behaviour klassen til gengæld. Det betyder så, at hvis behavior klassen samt steps klassen får grøn bar er man sikker på der er det rigtige antal, også totalt samt at de er de rigtige steder.

### Bevar fokus

Det som *Board* klassen leverer tilbage til behaviour er en *List* over typen *Tile*, men den er det endnu ikke en implementation af. Hvordan skal man så komme videre uden at miste fokus på *Board* klassen?.

Her er der flere muligheder.

1. Man kan implementere klassen.
2. Man kan mocke sig ud af det

Hvad man vælger afhænger helt af situationen, er det en klasse som samarbejder med en masse andre klasser for at komme frem til det man skal bruge lige her og nu?, eller er det en stor kompleks klasse som vil tage tid at udvikle?, vil det være en fordel at mocke sig ud af det?. Det gør man hurtigt kan komme videre med den klasse man har fokus på lige nu.

Er det derimod en ret simpel implementation, som i tilfældet med *Tile* så laver man den bare med det samme. Mht. tiles er det eneste interessante ved dem lige nu hvilken type de er, hvilke farve har de og hvilken position på brættet har de. Alt dette er noget de fødes med via constructoren. Det eneste der så mangler yderligere er tre get metoder.

### Spillebrik ud på brættet

Nu er brættet bygget op og spillet skal til at begynde, det første man skal forsøge i Ludo når spillet er startet op er, at forsøge at få sine brikker i spil. Dette gøres ved at flytte dem fra sine home sectors og ud på sin homeglobe.

Der er lavet 4 scenarios i story PlayerTriesToGetAPiceOnTheBoard.

1. player gets a piece on the board
2. player has all ready a piece on the globe
3. player throws die again because it wasn't 6
4. player has thrown 3 times without a 6

Hvis man ser på hvad de forskellige indeholder vil man formentligt synes der mangler nogle scenarios, det skyldes at de er gemt til en anden story. Der mangler bla. den hvor der står en brik med anden farve på ens homeglobe. Grunden til den er lagt i en anden story er for ikke at få alt for mange nye ting blandet ind i denne story og man vil få at se senere er der masser af nyt i den som den er.

Når man laver de forskellige stories skal man huske på de også er en form for planlægningsværktøj, dvs. hvis man vælger at lægge stærkt relaterede scenarios i forskellige stories fordi det vil lette udviklingen af en story, er dette helt i orden.

Efter koden i steps klassen er tilføjet kan man allerede se nogle ting som mangler før det kan køres.

Her er en opsummering pr. klasse/interface:

- Nye metoder i *Tile, addPiece(Piece piece)* og *removePiece(), getPiece()*.
- Nyt interface *Piece* med *getColor()* og *getName()*.
- Ny klasse *PieceImpl* da vi opretter en instans af klassen.
- Nyt interface *Die* med *rollDie()*.
- Nye metoder i *Game, rollDie()* og *move(Color color, String name)*.

## Mocking

Nogle gange har man brug for en klasse som der endnu ikke er udviklet. I stedet for så at kaste sig over den klasse og udvikle den helt kan man nøjes med at lave en stand in for den. Dette kan gøres ved at stubbe/mocke den. Når man mocker er det eneste man har brug for et interface og et mocking framework. Til Ludo spillet er der valgt Mockito som er et let anvendeligt framework.

I dette tilfælde er det *Die* vi endnu ikke har udviklet. Selv om den havde været lavet ville det stadig være nødvendigt at stubbe/mocke den for at få den under test kontrol. Når man gør det kalder man det for en test double.

### Mockito

For at få et mock object af *Die* skal man starte med at oprette en instans.

```
Die die = mock(Die.class);
```

Nu er objektet oprettet men den kan stadig ikke gøre noget. En *Die* er ret simpel. Det eneste den kan er *rollDie()*. Det der mangler er så at fortælle instansen hvad den skal returnere når vi kalder den med *rollDie()*.

Det første der er brug for at at slå en 6'er, for at få den til det gør man som nedenstående.

*[stories/PlayerTriesToGetAPieceOnTheBoard.java]*

```
@When("a die is showing a 6")
public void ShouldThrowA6() {
      when(gameData.getDie().rollDie()).thenReturn(6);
      game.rollDie();
      game.move(Color.red, "Rød1");
}
```

Den første linje i metoden er den som fortæller at den skal returnere en 6'er når *rollDie()* kaldes. Den instans af *Die* som der blev mock'ed hentes og der fortælles hvad den skal returnere når den kaldes.

Som vi lige har set med terningen er det nogle gange nødvendig at bruge en anden instans under test, et meget nyttigt redskab her er dependency injektion. Man kan styre det ved at lave det selv hvilket er helt fint i mindre projekter, der er også lavet mange frameworks som kan hjælpe en hvilket er at foretrække i større projekter.

## Dependency injection

Dependency injection også kaldet Inversion of Control er en metode til at få sine klassers afhængigheder af hinanden mindsket, man kan sige at det i alt sin enkelthed går ud på at give klasser sine instanser af klasser som den bruger, på runtime.

Hvorfor et det interessant ifm. specifikation af opførsel? Det er fordi steps klasser ofte har afhængigheder til interfaces i det system det skal verificere.

Hvis man lige kaster et blik tilbage på koden hvor terningen blev sat op til at slå en 6'er. Her

fortæller man jo netop *Game* at den skal bruge en bestemt terning som er et mock objekt og det gøres på runtime. Man har dog koblet steps klassen hårdt op til mock objektet da det bliver oprettet direkte. Det er dog helt ok da det ”kun” er test kode.

Dependency injection har rigtig mange forskellige teknikker at gøre brug af, de vil dog ikke alle blive gennemgået her. Det er heller ikke usædvanligt man gør brug af flere af teknikkerne på samme tid. Det kommer helt an på ens behov. De framework der er lavet har dog som oftest en præference for hvad de foretrækker.

### Constructor injection

En meget brugt og let forståelig metode er constructor injection. Her giver man de enkelte referencer med i objekternes constructor. Det gør så at det hele bliver sat op på starttidspunktet og ikke kan ændres før man genstarte applikationen.

I JBehave er der lidt forskellige måder man kan benytte når man bruger constructor injection.

Den hurtige og enkle er at lave det hele i steps klassen. Her skal der ikke overskrives metoder eller andet. I Ludo spillet ville det se sådan ud:

```
Game game = new GameImpl( new OutputImpl(), new
PlayerTriesToGetAPieceOnTheBoardStub(), new MoveValidatorImpl(), new
DieImpl());
```

Det der gøres her er, at der oprettet en instans af klassen *GameImpl*. I dens constructor giver man den fire instanser med af andre objekter, som *Game* skal bruge. Her er det *OutputImpl*, en test double til *BoardImpl* som kun har to tiles; en red home tile og en red globe tile da det er det eneste der er brug for i dette scenario. Ydermere sendes der også en *MoveValidator* og en *DieImpl* som er de originale, dvs. dem som skal i produktion. Ulempen ved denne metode er at her forurener man klassen lidt, i stedet for kun at specificere opførsel har man nu også noget opsætning.

Det kunne måske være rart at få denne opsætning ud af steps klassen når det er muligt. Det kan gøres ved at man i scenario klassen overskriver en af super klassens metoder og laver to constructors.

Den første constructor bruges kun til at kalde den næste med. Det er dog nødvendigt her da PicoContainer eksemplet arver fra denne klasse og skal bruge constructor nr. 2. Havde det ikke været tilfældet kunne det være samlet i en.

```
[stories/PlayerTriesToGetAPieceOnTheBoard.java]

public class PlayerTriesToGetAPieceOnTheBoard extends Scenario {
      public PlayerTriesToGetAPieceOnTheBoard(){
            this(PlayerTriesToGetAPieceOnTheBoard.class);
      }

      public PlayerTriesToGetAPieceOnTheBoard(
                  Class<? extends RunnableScenario> scenarioClass) {
            useConfiguration(new MostUsefulConfiguration());
```

```
            StepsConfiguration configuration = new
                  StepsConfiguration();
            addSteps(createSteps(configuration));
      }

      protected CandidateSteps[] createSteps(StepsConfiguration
                  configuration) {
            Die die = mock(Die.class);
            return new
                  StepsFactory(configuration).createCandidateSteps(
                  new PlayerTriesToGetAPieceOnTheBoardSteps(
                  new GameImpl(new OutputImpl(),
                  new PlayerTriesToGetAPieceOnTheBoardStub(),
                  new MoveValidatorImpl(), die)));
      }
}
```

I den første constructors kald til den næste angiver man hvilken klasse det er der skal køres på.

Som det kan ses bliver der også brugt injection på selve steps klassen, fordelen ved at flytte opsætningen her ud er, at det også er i denne klasse man sætter andet hvis man ikke er tilfreds med det som JBehave kalder MostUsefulConfiguration.

### Setter injection

Det er en metode hvor man på en klasse eksempelvis har en metode *Game.setDie(Die die)* som sætter *Game*'s instans af *Die* til hvad man gerne vil på et givent tidspunkt. Det giver den fordel at man også efter applikationen er oppe at køre, har mulighed for at skifte terningen ud med en anden. Dette blev gjort til at starte med indtil det blev klart at den kun blev brugt i test øjemed og det giver ikke mening at kunne skifte en terning ud i et spil Ludo efter spillet er startet op. Det blev så efterfølgende rettet så Game for en mock'ed terning med som man så efterfølgende kan sætte op hvad den skal returnere.

### JBehave og dependency frameworks

I den nyeste version af JBehave understøttes tre udbredte frameworks til IoC på steps klasse niveau. Det drejer sig om:

1. PicoContainer
2. Spring
3. Guice

De er kommet med i JBehave efter at nogle udviklere har ytret ønske om det, de har som sådan ikke noget med BDD at gøre men er en måde at få sat sin software korrekt op på i de enkelte steps klasser, dvs. de bruges i forbindelse med verifikationen og ikke i opsætningen af den færdige software.

Den som der er kigget på i forbindelse med Ludo spillet er PicoContainer. Som nævnt før så kan man i mindre projekter sagtens klare sig uden et framework og så selv styre det. Denne kategori må Ludo siges at falde inden for, men for at afprøve metoden og se hvordan den fungerer er det tit en fordel med et lille eksempel. Der er endog folk som mener at kan man klare sig uden et framework er det at foretrække.

Grunden til det lige er PicoContainer der der valgt er at det netop har en præference for konstructor injection hvilket er det som der er brugt i Ludo i forvejen.

**PicoContainer**

Det første man skal have er udvidelsesmodulet Jbehave-Pico som skal med i en classpath. Grunden til at man skal have det med er at det er her PicoStepsFactory er i. Når det er på plads så kan man begynde at oprettet de forskellige klasser.

```
[pico/PicoPlayerTriesToGetAPieceOnTheBoard.java]

public class PicoPlayerTriesToGetAPieceOnTheBoard extends
      PlayerTriesToGetAPieceOnTheBoard {

      public PicoPlayerTriesToGetAPieceOnTheBoard() {
            this(PicoPlayerTriesToGetAPieceOnTheBoard.class);
      }

      public PicoPlayerTriesToGetAPieceOnTheBoard(Class<? extends
            RunnableScenario> scenarioClass) {
            super(scenarioClass);
      }

      protected CandidateSteps[] createSteps(StepsConfiguration
      configuration) {
        PicoContainer parent = createPicoContainer();
        return new PicoStepsFactory(configuration, parent).
            createCandidateSteps();
    }

    private PicoContainer createPicoContainer() {
        MutablePicoContainer parent = new DefaultPicoContainer(
            new Caching().wrap(new ConstructorInjection()));
        parent.as(Characteristics.USE_NAMES).addComponent(
            PlayerTriesToGetAPieceOnTheBoardSteps.class);
        parent.as(Characteristics.USE_NAMES).addComponent(
            GameImpl.class);
        parent.as(Characteristics.USE_NAMES).addComponent(
            PlayerTriesToGetAPieceOnTheBoardStub.class);
        parent.as(Characteristics.USE_NAMES).addComponent(
            MoveValidatorImpl.class);
        Die die = mock(Die.class);
        parent.addComponent(die);
        return parent;
    }
}
```

Der er allerede lavet en klasse som der bare arves fra, det er PlayerTriesToGetAPieceOnTheBoard fra Stories package. Havde man ikke ville kunne køre det som både normalt og som PicoContainer havde det selvfølgelig været samlet i en klasse.

Den har to constructors, en som bare kalder den anden med en klasse. Den anden constructor modtager en klasse som arver fra *RunnableScenario, d*et eneste den gør er at kalde dens super klasse med den klasse. Som det næste overskriver den en af super klassens metoder. det er her det bliver interessant mht. PicoContainer. Her kalder men en privat metode som opretter en PicoContainer og angiver man ønsker at benytte ConstructorInjection hvorefter man så tilføjer de komponenter der er behov for. Som man kan se er det både muligt at tilføje klasser og instanser.

En forudsætning for at alt dette virker er selvfølgelig at man har constructors med de rigtige

argumenter i hver klasse, f.eks. i Ludospillet, her skal *PlayerTriesToGetAPieceOnTheBoardSteps* bruge *Game*. Dvs. den skal have en constructor der tager en instans af *Game* med som parameter.

Det samme gælder for de andre klasser, men det er uafhængigt af om man bruger JBehave PicoContainer eller om man styrer det selv.

**Statemaskine**

Hvis man lige tager et kig på de to sidste scenarios i denne story så vil man se at de omhandler hvad der er tilladt som det næste, dette kunne lede en hen til at det kan være nyttigt med en state maskine.

```
Scenario: player throws die again because it wasn't 6
Given a red piece is located at the red homesector
And no other red pieces on the board
When a die is not showing 6
Then the die is thrown again
```

Her skal det være muligt for rød spiller at slå igen hvis han ikke slog en 6'er første gang, til at styre dette er der valgt en state maskine.

```
[framework/LudoState.java]

public interface LudoState {
      public State getState();
      public void rollDie() throws IllegalActionException;
      public void move(Color color, String name) throws
IllegalActionException;
```

De to eneste states der er brug for i denne story er *rollDie()* og *move()*. Der vil være flere states end dem der er brug for i denne story og for at finde dem kræver det en nøje overvejelse af hvad der skal være tilladt hvornår. Der vil dog ikke blive lavet mere på statemaskinen til Ludospillet.

Havde det være et rigtigt projekt skulle man have haft en samtale med brugeren igen for i det hele taget at finde ud af hvilke states der kunne være samt hvad der var tilladt i det enkelte.

Den måde den er bygget op på i Ludo er vi har *LudoState* interfacet. Så er der lavet en implementation for hver enkelt state hvor de metoder der ikke er tilladt at kalde i en given state bare kaster en exception. De metoder der er tilladte vil så udføre det som *Game* skulle have stået for.

Inden state maskinen var det *Game* som lavede et move eller et kald til den som stod for at flytte brikken. Denne funktionalitet er flyttet ud i *MoveState.move()* og den kalder så *Game* og får en *GameData* retur.

```
[game/MoveState.java]

public class MoveState implements LudoState {

      Game context;
      MoveValidator validator;
      GameData gameData;

      public MoveState(Game context) {
            this.context = context;
            gameData = context.getGameData();
```

```
	}

	@Override
	public State getState() {
		// TODO Auto-generated method stub
		return State.move;
	}

	@Override
	public void move(Color color, String name) throws
IllegalActionException {
		validator = gameData.getValidator();
		List<Tile> l = context.getGameData().getBoard().getTiles();

		validator.move(color, name, l, gameData.getLastRoll());

	}
}
```

Fordi opførslen for move var specificeret og fik grøn bar så var det den der skulle laves først. Refaktorer altid i grøn, det gør man er sikker på det ikke har haft nogle uønskede sideeffekter. Når det er på plads så kan man kigge på de sidste to scenarios.

*GameData* blev også opfundet ifm. statemaskinen. Skulle *MoveState* spørge *Game* om alle de ting den skulle bruge ville de også skulle udstilles i *Game* interfacet, men fordi *Game* også fungerer som en facade ud af til ville alle disse ting også blive udstillet og det er ikke hensigtsmæssigt.

### Behaviour MoveValidator

Man kunne godt vælge at lade *Game* stå for at flytte brikker rundt på brættet, en klasse må dog ikke have for mange ting at have ansvar for. Inden man kan flytte en brik skal der først ses efter om det er et lovligt træk, er det lovligt kan man så flytte brikken. Til at afgøre om et træk er lovligt og så efterfølgende flytte brikken laves en ny klasse *MoveValidator.* Til at vise hvordan denne klasse skal bruges laves en *MoveValidatorBehaviour.* I denne klasse er der 4 tests.

1. `ShouldHaveMovedARedPieceToTheRedGlobe()`
2. `ShouldNotHaveARedPieceOnTheRedHome()`
3. `ShouldNotHaveMovedTheRedPieceFromTheRedGlobe()`
4. `ShouldBeTheSameRedPieceOnTheRedBlobeAsBefore()`

Som man nok kan se på navnene så er der ikke nogen test på noget med state, det er fordi selve *MoveValidator* skal opføre sig ens uanset hvad state er og vil ikke blive kaldt hvis *Game* har den forkerte state men dens opførsel er den samme.

### Behaviour Game

Alt omkring state har noget med *Game* at gøre, derfor skal det også dokumenteres i *GameBehaviour* hvad der er lovligt at kalde hvornår. Når denne story er færdig og giver grøn bar så er man nødt til at tilføje det nye til *Gamebehaviour, d*et vil dog heller ikke blive gjort da det tidligere er besluttet ikke at implementere hele statemaskinen.

#### Hvad har vi gjort indtil nu?

Vi har fået udledt nogle interfaces og implementeret dem i form at et *Board* og en *Tile*. Der er et

bræt som er korrekt opsat og det er muligt at få flyttet en brik ud på ens globe når der slåes en sekser. Slår man ikke en sekser så er det muligt at slå tre gange inden turen går til næste spiller.

Til at nå resultatet er der brugt nogle metoder som JBehave stiller til rådighed samt enkelte som ikke er en del af JBehave men mere en naturlig del af BDD.

### Hvad har vi lært?

Vi har lært hvordan man kan se hvornår der er brug for table examples og hvordan man laver dem. Når man bruger table examples så gør man også brug af parameter injection. Her så vi på de to måder der understøttes af JBehave.

Vi lærte også at bruge Mockito til at mocke et interface og hvordan man fik en bestemt værdi tilbage på et kald til en metode på samme.

Et enme der blev kigget meget på var dependency injection, her fandt vi ud afhvilke typer der var samt hvilke DI frameworks JBehave understøtter.

## Afslutning på første release

Der er stadig to stories der mangler før den første release af Ludospillet er færdiggjort, ”flytning af en spillebrik” og ”en vinder findes”.

### Flytning af en spillebrik

Nu kan man flytte en brik ud på en globe når man slår en sekser. Det der så skal til at ske nu er at man skal kunne flytte en brik rundt på brættet. Det denne story går på er dog udelukkende på når man flytter en brik til et felt hvor der ikke står nogen anden brik.

Til story player_moves_a_piece_on_the_board.scenario er der lavet 7 scenarios og det er:

- `player moves a piece to a neutral tile`
- `player moves a piece to a star tile`
- `player moves a piece to a neutral globe tile`
- `player moves a piece to a globe tile in another color`
- `player moves a piece to a center tile`
- `player moves a piece to a goal tile`
- `player moves a piece to a goal tile from star`

Det eneste der vil blive gennemgået ifm denne story er det nye der er ift. JBehave.

**Extension på en story**

Som det skarpe øje måske har set så ender denne story på ”.scenario”. Indtil nu har der ikke været extension på de stories der er lavet, det er fordi JBehave's default opførsel er, at det skal der ikke være. For at undgå misforståelser om hvad en fil indeholder kan det være en fordel at give dem en extension på. Nogle foretrækker scenarios som vi har valgt da filen indeholder en eller flere scenarios, andre fortrækker ”.story” men pointen er at man kan få det som man vil have det.

Som altid er det i filen som arver fra *Scenario* man skal lave sin opsætning hvis man ikke er tilfreds med *MostUsefullConfiguration*.

```
[stories/PlayerMovesAPieceOnTheBoard.java]

public class PlayerMovesAPieceOnTheBoard extends Scenario {

      public PlayerMovesAPieceOnTheBoard() {
            this(PlayerMovesAPieceOnTheBoard.class);
      }

      public PlayerMovesAPieceOnTheBoard(
                  Class<? extends RunnableScenario> scenarioClass) {

            final ScenarioNameResolver resolver = new
            UnderscoredCamelCaseResolver(".scenario");
            useConfiguration(new MostUsefulConfiguration() {
                  @Override
                  public ScenarioDefiner forDefiningScenarios() {
                        return new ClasspathScenarioDefiner(resolver,
                        new PatternScenarioParser(keywords()));
```

```
                }
            });

            StepsConfiguration configuration = new
                  StepsConfiguration();
            addSteps(createSteps(configuration));
      }

      protected CandidateSteps[] createSteps(StepsConfiguration
            configuration) {
            Die die = mock(Die.class);
            return new
            StepsFactory(configuration).createCandidateSteps(new
            PlayerMovesAPieceOnTheBoardSteps(new GameImpl(null, new
            BoardImpl(), new MoveValidatorImpl(), die)));
      }
}
```

Alt det nye foregår i den sidste constructor. Her laver man en ny ScenarioNameResolver hvor man angiver at man vil have det skal ende på ".scenario". Herefter ændrer man den konfiguration der bruges ved at lave en instans af *MostUseFulConfiguration* og overskriver en af dens metoder. Ud over det er alt som det plejer at være.

I denne story er der også brugt ordered paramertes, det vil ikke blive gennemgået her men man kan se i afsnittet om parameter injection hvor det står beskrevet ud fra et tænkt eksempel.

### Vinderen findes og spillet stopper

Vinder scenario gør brug af en example table til at opsætte alle de positioner hvorfra en vindersituation kan forekomme.

```
[stories/player_wins_the_game]

Scenario: Move last piece from a center tile into goal
Given all <color> pieces but 1 is in the goalarea
And 1 <color> piece is at center tile, <distance> tiles from goal
When a die with value <die> is rolled
And player moves the last <color> piece
Then I should see '<color> player wins the game'

Examples:
|color|distance|die
|red|1|1
|blue|2|2
|red|3|3
|yellow|4|4
|green|5|5
```

Vinderen af spillet er fundet når alle fire brikker er placeret i målfeltet, derfor skal det være muligt at kunne sætte brættet op i en tilstand med tre brikker der allerede er placeret i målfeltet og en brik der er udenfor målfeltet, men i en position hvormed at brikken kan flyttes i mål.

Dette kan bla. foretages ved brug af en mock, implementere en stub ud fra *Board* interfacet, lave en decorator på boardet, valget er at benytte den allerede implementeret *BoardImpl* og nedarve denne

da det forekommer som den hurtigste løsning eftersom at alle felter på brættet således er sat op. Der tilføjes en metode til *EndGameBoardStub* for at sætte en brik explicit på boardet.

For at en vinder situation forekommer skal den sidste brik flyttes fra nuværende position og ind i mål, derfor skal vi have terningen til at slå det korrekte slag og der benyttes en simpel mock af *Die* som returner præcis det slag der skal benyttes for at brikken rykker i mål.

Selve den fysiske flytning af brikken ud fra brikkens placering og terningens værdi foretages af *MoveValidator*, her benyttes den eksisterende implementation *MoveValidatorImpl*, fordi det vurderes at en stub eller mock vil blive kompleks at lave til at kunne håndtere alle examples.

```
[Extracted from stories/PlayerWinsTheGame.java]

// From @Given methods

// Set pieces on specific board tiles
board = new EndGameBoardStub();

((EndGameBoardStub)board).setPieceOnTile(new
PieceImpl(Color.getColor(colorStr), colorStr+"1"), TileType.goal,
Color.getColor(colorStr), 1);

((EndGameBoardStub)board).setPieceOnTile(new
PieceImpl(Color.getColor(colorStr), colorStr+"2"), TileType.goal,
Color.getColor(colorStr), 2);

((EndGameBoardStub)board).setPieceOnTile(new
PieceImpl(Color.getColor(colorStr), colorStr+"3"), TileType.goal,
Color.getColor(colorStr), 3);

Output out = new OutputImpl();

MoveValidator validator = new MoveValidatorImpl();

Die mockedDie = mock(Die.class)
when(mockedDie.rollDie()).thenReturn(die);

game = new GameImpl(out, board, validator, mockedDie);
game.start();
game.submitPlayers(4);


// From @When methods

// Actions needed before a winner can be found
game.rollDie();
game.move(Color.getColor(colorStr), colorStr + "4");


// From @Then method
ensureThat(out.messages(), hasItem(colorStr+" player wins the game"));
```

Selve implementering bliver foretaget i metoden *GameImpl.move* efter move er foretaget, dette resulterer i at *move()* udskriver den ønskede vinder til output.

```
[stories/GameImpl.java]
```

```
public void move(Color color, String name) {
      gameData.getLudoState().move(color, name);
      if (winner(color)) {
            output.put(color + " player wins the game");
      }
}
```

Metodekaldet *winner* er fremkommet efter refaktorering for at gøre move metoden mere læsbar og er en private metode i *GameImpl* klassen, *winner* metoden aflæser om der er fire brikker af den angivne farve på brættets målfelt.

### Hvad har vi gjort indtil nu?

Benyttet en klasse nedarvet fra produktionskoden *BoardImpl* kun til formålet at kunne tilføje en metode der ikke er inkluderet i interfacet.

Benyttet en stub *OutputImlpl* der er benyttet i andre scenarios til at aflæse resultatet.

Benyttet en mock af *Die* interfacet for at kunne kontrollere hvilken værdi terningen skal levere.

Benyttet en implementation af *MoveValidator* interfacet der indgår i produktionskoden.

Fået implementeret vinder opførsel i game implementationen og dermed afsluttet første release af Ludospillet.

### Hvad har vi lært?

At kombinere forskellige typer af test konstruktioner til at få den korrekte opførsel ud af vores eksempel på den nemmeste og mest simple måde.

# ER BDD EN ”SILVER BULLET”?

Frederick P. Brooks argumenterede i midten af 80'erne i artiklen ”No Silver Bullet: Essence and Accidents of Software Engineering” for at der ikke indenfor en dekade vil være en enkelt metode eller teknologi der vil kunne forbedre produktiviteten, pålideligheden og simpliciteten i softwarekonstruktion, i en størrelsesorden der svarer til en fordobling, på samme måde som der er sket i andre produktions/konstruktions relaterede discipliner. I den sammenhæng benyttede han begrebet ”No silverbullet” til den tese han havde opstillet, vi vil bruge samme begreb som indgangsvinkel på vores evaluering af BDD.

## Udfordingerne ved BDD?

Som med alt andet der forsøger at berøre et komplekst område, er der en række udfordringer forbundet med det. BDD er ikke nogen undtagelse og kan på mange måder være en smule svær at overskue, både når man kigger på hvilke problemområder der forsøges afdækket og hvordan man i praksis løser dette.

### Kommunikation

Kommunikation, kommunikation og kommunikation er mantraet i BDD filosofien, hele ideen er at der kommunikeres iterativt med stakeholder og hvis der ønskes klarlægning over en detalje i en story tages kommunikationen med det samme i mellem de parter der berøres. Kommunikation har altid været en vigtig faktor i software udvikling, men selve den måde at kommunikationen foregår på er ukendt for de fleste stakeholders i et projekt der har benyttet sig af traditionelle udviklingsmetoder, eksempelvis vandfaldsmetoden.

Tidligere brugte man mange timer på at diskutere med forskellige personer i en virksomhed der ønskede at få et stykke software produceret, disse møder og samtaler blev foretaget af forretningsanalytikere der samlede ønsker og krav sammen og alt dette endte ud med at forretnings analytikerne sammenfattede en kravspecifikation, der oftere mere lå til grund for en kontrakt mellem kunde og sælger af økonomisk art, end koncist at definere den opførsel man reelt ønskede systemet skulle tilbyde.

Kommunikation er en svær disciplin at mestre, en ting er at mestre disciplinen envejs, men at tale til en gruppe af mennesker med forskellige faglige kompetancer og sociale færdigheder og få disse til at forstå budskabet på samme måde, er en udfordring i sig selv. BDD gør brug af story templates til at opsamle essensen af det diskuterede på en form så alle forstår budskabet på samme måde, og dermed forsøger at skabe en fælles bevisthed om hvilken opførsel der ønskes.

Som to personer med samme baggrund der har skullet evaluere på BDD har vi et begrænset erfaringsgrundlag at vurdere om kommunikationsformen man har valgt er god og intuitiv at bruge for alle parter, men i forhold til at alternativet er, ikke at benytte en template til at beskrive en specifikation er det et klart skridt fremad. Vi vil vurdere nogle persontyper der indgår i et typisk kommercielt produkt og give vores vurdering af BDD ud fra den kontekst de befinder sig i.

### Kunder / brugere

Udfordring for kunden ved at gennemføre et projekt ved hjælp af BDD, er primært at kunde skal investere tiden til at skrive stories og være med til at definere scenarios og acceptance tests sammen

med eksempelvis en tester. Dette er ikke nogen automatisk selvfølgelighed, mange kunder er af den opfattelse at det udelukkende er forretningsanalytikeren der skal definere eksisterende forretning og udtænke behov og logik for det nye system uden aktiv indblanding fra kunden selv. Udover definering af stories har kunden til opgave at finde egnede brugere der skal medvirke i projektet, at finde de brugere der repræsenterer en bestemt rolle i det projekt der skal laves er en proces man umiddelbart vil tro er ukompliceret, men dette kan sagtens være tilfældet at det ikke er. Ofte vil en kunde finde brugere der rent faktisk ikke er deciderede brugere af et system, og i stedet indsætte domæne eksperter for at afdække en bestemt brugerrolle, dette gøres både bevist og ubevist fra kundens side, måske har kunden ikke tiltro til at en bestemt bruger kan give noget til systemet som giver værdi for systemet, måske sætter man afd. ledere ind i stedet for den specifikke brugere i en tro på at man sikrer kvaliteten i softwaren, af andre grunde kan eksempelvis være sikkerhedsårsager. Domæne eksperter kan være gode nok at bruge til overordnede sammenhænge mellem delsystemer, men vil ofte have en meget anderledes tilgang og forståelse af systemet som en ”ikke domæne ekspert” og derfor er det vigtigt at få de brugere med i processen der rent faktisk skal bruge systemet. Kunden skal også have et grundlag for at kunne forstå og deltage i processen med at definere forretningsopførsel og har kunden ikke været involveret i et projekt der benyttede sig af agile tekniker her i blandt indsamling og kreering af story og acceptancetests, brug af ”Ubiquitous language” som benyttes af BDD skal kunden sørge for at afsætte den fornødne tid til undervisning af brugerne i den kommunikationsform der benyttes i BDD. Kunden har sammen med udviklerholdet ansvar for at prioritere de stories der skal laves ud fra den værdi storien giver til kunden.

Vi har ikke opnået nogle praktiske erfaringer med kunder og brugere igennem vores evaluering og kan derfor kun gisne om hvordan et forløb med denne kommunikation vil forme sig, en ting er vi dog enige om og det er at kunden og brugerne skal være motiveret og villig til at bevæge sig en smule ind på ukendt terræn og mødes midtvejs med testere, analytikere og udviklerer. Gevinsten er forhåbentligt et bedre system, tilfredse kunder og topmotiverede brugere, men det må afdækkes i en anden sammenhæng.

### Forretningsanalytikere

Der kræves af forretningsanalytikerne at de har stor erfaring i at trække essensen ud af en samtale og mappe dem ned i en story template uden at miste forretningsopførsel, og bibeholde den simplicitet der skal være i en story-template for at denne ikke  misforståes af andre der er involveret i projektet. Forretningsanalytikeren skal kunne tale med flere forskellige typer af brugere fra slutbrugeren uden nogen it erfaring over brugere med stor domæne viden til management med primært økonomiske og tidsmæssige interesser og sørge for at den ene gruppe ikke obstruerer den anden gruppe, eksempelvis ved at en afdelingsleder med stor domæne viden sidder med til story-writing eller diskussion om en feature der udelukkende omhandler en bestemt bruger, grunden til dette er at brugeren kan blive intimideret af sin overordnede og ikke selv udtrykke det behov der er vigtigt at få frem i storien og derfor vil det blive den overordnet der vil definere behovet.

Vi har selv ageret forretningsanayltikere i forbindelse med udvikling af vores Ludospil, der er blevet skrevet stories ud fra et eksisterende regelsæt fra et brætbaseret Ludo og der er skrevet stories der håndterer opførsel der ikke er beskrevet i det officielle regelsæt eksempelvis story om opstart af spillet. Vi syntes at måden at nedskrive og samle forretningsregler i stories på ”In order to... As a... I want...” er langt bedre end hvad vi har benyttet os af hidtil, metoden er simpel og man kan bruge tiden på at definere sine forretningsregler godt og sigende istedet for at bruge tiden på at udfylde alen lange  formularer som eksempelvis ”IEEE 830 software requirements” som er så omfangsrig at man risikerer at tabe fokus ved at læse sig igennem yderst detaljerede krav. Et andet alternativ er at

forretningsanalytikeren benytter sig af usecase templates men vi syntes at stories virker mere smidige, gør det nemmere at ”refaktorere” med det samme når man har diskussionen med kunden, desuden er indlæringskurven ved brug af usecases langt højere og bliver derfor ofte brugt forkert eller er decideret mangelfuld. Stories er stadig svære at definere, men det er ikke selve formen og teknikken som der benyttes i BDD der gør dette svært, det er selve processen i at udtrække regler fra et domæne og formulere dette i konkrete eksempler og det er ens i alle former for kravdefinering.

### Testere

I et mere traditionel udviklingsforløb vil testholdet modtage produktionskoden fra udviklerne når den er færdig og derefter skrives der testcases til koden. BDD filosofien vil have testholdet, såfremt et sådan er tilknyttet projektet, med i processen fra starten af og testholdets opgave er at hjælpe systemets brugere med at skrive scenarios og acceptance tests, det betyder ikke, at testere ikke skal lave testcases på systemet efter at udviklerne er færdige med koden men at de indirekte får indvirkning på hvordan systemet vil interagere og kan testes. Vi har skrevet scenarios og acceptance tests til Ludospillet ud fra en testers vinkel dvs. at prøve at holde fast i test kasketten, det har været en smule svært at adskille definering af krav og storyopbygning (bruger, analytiker), oprettelse af scenarios og accept kriterier (tester) fra hinanden når det er samme person der optræder som alle roller, men kan sagtens se hvad de enkelte roller i forløbet bibringer med. Det er jo også en del af tankegangen i BDD at tingene hører sammen i form af at alle parterne diskuterer story, scenarios, model osv. iterativt og vi har gjort det iterativt på den måde at vi har ved oprettelsen af scenarios og accepttests har skærpet eller løsnet storien for at være enten mere specifik eller mere generel end hvad vi i første omgang havde kommet frem til.

### Udviklere

Udfordingerne for udviklere der ikke har erfaring fra agile arbejdsmetoder er mange, herunder er det hele den iterative tankegang i alt hvad der foretages fra indsamling af forretningslogik vha. stories og acceptance tests, delleveringer af software i form af små korte releases, selve processen i implementeringen ved brug af test-first teknikken. Udviklere der skal benytte sig af BDD og ikke har erfaring med TDD har alle de samme udfordringer foran sig som udviklere der skal til at arbejde ud fra en TDD metode, forskellen er udelukkende i hvor man starter og i hvilken rækkefølge man implementerer de ønskede features. Rækkefølgen er en beslutning som ikke vælges af udvikleren men af brugeren ud fra en prioritering om den værdi en feature giver til systemet som helhed. Har udviklerholdet erfaring med brug af TDD og evt. andre agile metoder skal man for at adoptere BDD filosofien primært bruge kræfter på at vænne sig til at foretage udviklingen outside-in samt at beskrive denne outside-in opførsel ved hjælp af scenarios i stedet for de klassiske testcases. Dette kan lyde som en overkommelig transition, men de erfaringer vi har gjort os med dette projekt er at det er svært at springe fra en scenario iteration der kan være ret så omfattende og involvere mange forskellige entiteter og så ind til den indre iteration hvor den del af systemet der skal bearbejdes for at få den ønskede opførsel ved hjælp af test-first metoden implementeres. Det er svært at tilføje en ydre rytme uden at miste fokus. Alt hvad der foretages i indre iteration skal tilfredstille den ydre iteration og derfor skal man reelt have fokus 2 steder, på det overordnede scenario og på den aktuelle tilføjelse af opførsel på klasse/metode niveau. En af de problemer som viser sig sværest at håndtere i forhold til TDD er scenario, hvor meget skal man isolere væk fra scenario i form af mocks, stubs osv. og hvor meget af eksisterende produktionskode kan man bruge til at hjælpe scenario til at levere den ønskede opførsel uden at scenario bliver en decideret integrationstest. I TDD har man en unit under test og isolerer alt væk fra denne, dette er en håndgribelig størrelse at forstå, men et scenario er ikke en unit under test men nærmere en behaviour under test og for at en overordnet opførsel skal opfyldes kan der sagtens indgå mange interfaces der skal levere for at en

opførsel er komplet og det kan give overkomplekse mocking strukturer for at få opsat en betingelse for at scenario kan opfyldes og så forsvinder ideen med at et scenario er et eksempel på hvordan systemet bruges, da selve operationen for opfyldelsen af scenario er minimal i forhold til opsætning. Man kan gemme opsætning af state i mocks og stubs i eksempelvis XML filer, men det gør at overblikket forsvinder, der skal laves loadere som ikke er interessant for andet end scenario og refaktorering besværliggøres voldsomt. I TDD er beskeden at hvis en test bliver overkompliceret i forhold til hvad der skal testes udelades testen og der foretages et review af UUT i stedet. Det er straks mere kompliceret med en story da den decideret specificerer et krav til systemet og derfor kan man ikke undlade at lave denne da det vil medføre at kravet ikke er repræsenteret som en specifikation af systemet. En løsning kunne være at oprette story på normal vis og lave en steps klasse der beskriver i klar tekst under hver metode *Given, When, Then* hvad der skal foretages kan man bibeholde en form for specifikation sammen med alle andre specifikationer uden at skulle lave opførsel i systemet som ikke var dokumenteret.

## Værdi for stakeholder

BDD er ikke en metode der kun tilgodeser udviklere og hjælper dem til at få arbejdet struktureret så der kan produceres pålideligt, fleksibelt og testbart software til tiden. Det er alle personer der er involveret i produktudviklingen på en eller anden måde der får noget ud af et projekt der er kørt ved hjælp af BDD processen, og vi opstiller en række eksempler hvorpå en defineret gruppe kan få værdi ud af BDD.

### Eksterne stakeholdere

Slutbrugere vil få et samhørighed med det stykke software der udvikles, de vil få en oplevelse af at de har været med til at definere, rådgive og levere viden om hvordan systemet skal opføre sig og se ud.

Afdelingsledere vil få en gruppe af brugere der fra starten af er motiveret til at benytte sig af den nye software og at slutbrugerne i mindre grad har en indlæringskurve der forhindrer dem i at opfylde deres job effektivt, ledere vil bistå med viden om domænet og de interaktive handlinger der foretages imellem jobfunktionerne i en given afdelingen og imellem andre afdelinger.

Management vil få mere kontrol og overblik over økonomien i projektet eftersom at man præcist ved hvad der laves og hvad der færdiggøres i små korte iterationer som kan evalueres og gennemgåes, så chancen for en deadline og budget overskridelse er minimeret kraftigt i forhold til en traditionel upfront kravspecifikation og designdokument forretning.

### Interne stakeholdere:

Forretningsanalytikere vil have et værktøj til at definere forretningsregler sammen med eksterne og interne stakeholders således at misforståelser kan minimeres. En anden fordel er at man driver specifikationen frem skridt for skridt og får diskuteret alle de features der er fremkommet, får disse features skrevet ned på en koncis og formel måde således at de ikke går tabt.

Udviklere vil få en større tillid til at de får bygget softwaren til det formål det skal bruges og at man ikke forfalder sig til at kode tekniske konstruktioner der ikke giver noget værdi til systemet. Udviklerne vil have alle de samme fordele som TDD giver og kan derfor benytte sig af allerede eksisterende kompetencer som man må have oparbejdet i brug af TDD.

Testere er med fra første dag og skal ved udarbejdelse af scenarios definere acceptkriterier sammen med brugerne, dette betyder at en tester indirekte kan influere designet af systemet i kraft af at definere acceptkriterier og tests, således at alle dele er nemme af få under test. Er systemet nemt at få under test fra starten af hjælper man testeren til at lave testcases på systemet uden at der skal ske en masse refaktorering af designet for at få en unit under test og giver i sidste ende et mere fleksibelt og modificerbart system.

Projektledere får et overblik over processen, softwarens kvalitet og er i en situation hvor der kan meldes til andre stakeholders om projektets status, ressourcer og eventuelle problemer der måtte opstå allerede på et tidligt tidspunkt i projekt forløbet.

Management har det overordnede ansvar for at projektet overholder budgettet og deadline, til dette har man via sine projektledere en nem og hurtig måde at få overblik over systemet. Eftersom at kunden også deltager i processen og de fastsatte iterationsmøder, er en stor del af de møder der udelukkende er baseret på projektstatus og ansvarspålæggelse af en ikke defineret feature eller en feature der er misforstået, forsvundet. Set i et økonomisk perspektiv har management fået et redskab til at kunne værdisætte et projekt overfor bestyrelse og revision uden at skulle gætte sig frem til aktuel værdi da man kan sætte point el. procent på hver story og derved vide hvor langt man er fra at have færdiggjort systemet.

Gøres BDD processen rigtigt giver det goodwill overfor kunden og i markedet generelt, at man er en professionel forretning der leverer til tiden og leverer det ønskede til den aftalte pris. En vigtig værdi for en virksomhed er at medarbejderne ikke er stressede og utilfredse, men i stedet er glade og motiveret og BDD kan hjælpe medarbejdere med at kunne strukturere og organisere sit arbejde og kan ved samarbejde med både eksterne og interne stakeholders få glæde og selvtillid af successfulde delresultater der inspirerer til at få næste delresultat i hus.

## Er det praksis muligt at indføre?

Hvis f.eks. IT-afdelingen i en virksomhed overvejere at benytte BDD og tror det vil kunne hjælpe dem med at udvikle bedre software, kan de så bare indføre denne filosofi uden at indrage resten af organisationen?. For at indføringen af BDD skal blive en succes kræver det at alle i hele organisationen kan se formålet med skiftet og hvilket udbytte der efterfølgende kan høstes.
Der vil her blive beskrevet hvad der skal være opfyldt for at det vil kunne blive en succes. Herefter vil erfaringer fra et projekt i den finansielle verden blive gennemgået. Hvad virkede godt og hvad kan gøres bedre.

### Forudsætninger for BDD

I et traditionel udviklingsforløb behøver en bruger og udvikler ikke nødvendigvis at tale sammen. Her vil man bare få udleveret et designdokument som man så skal forholde sig til, findes der fejl i det, gives det blot tilbage til den man fik det af og så er det vedkommendes problem at få det rettet.

Man kan så sige hvorfor så vige fra dette når det er så simpelt. Nok er det simpelt men det har også store ulemper med sig. Når noget skifter hænder i denne process er der sket en eller anden form for oversættelse. Her er der så mulighed for misforståelser med deraf følgende fejl i systemet. Jo senere fejlen opdages jo mere koster det også at rette den. Og det er præcis denne manglende kommunikation i mellem de forskellige stakeholders at BDD forsøger at løse.

Ved BDD kræves der at udvikleren taler med dem som skal bruge systemet, så her er kommunikationen mellem alle stakeholders meget vigtig. Det betyder så også at alle stakeholders skal være indstillet på at gå ind i dette samarbejde. Det er også vigtigt at core stakeholders sidder forholdsvis samlet eller som minimum er til rådighed, det nytter ikke noget man ikke kan få fat i en bruger/forretningsanalytiker før om end uge hvis man har nogle uddybende spørgsmål til en story.

Ledelsen skal i høj grad også bakke op om alt dette da det jo betyder en omorganisering af ens projekter. Førhen skulle en tester bare være til rådighed når koden var kodet færdig og skulle testes, nu er testeren med helt fra starten når accept kriterierne skal fastlægges på de enkelte scenarios. Brugerne af systemet er også med hele vejen igennem nu og ikke kun i starten og afslutningen.

Når man skal overbevise de enkelte stakeholders om at BDD er vejen frem så er det vigtigt at gøre dem klart hvad de får ud af det, dvs. man skal prøve at sætte sig i deres stol når man forklarer hvad det hele går ud.

## BDD i den virkelige verden

Mauro Talevi som er udvikler på Jbehave frameworket, har været med i et stort projekt i en global investeringsbank hvor de skulle bruge BDD for første gang *[se videocast med Mauro Talevi]*. Der vil her blive beskrevet nogle af de erfaringer de fik i forbindelse med dette projekt. Der vil ikke blive beskrevet hvad de udviklede men der vil udelukkende blive kigget på hvad der virkede godt og hvad der kunne gøres bedre i selve processen.

## Hvad fungerede godt

Ligesom BDD foreskriver det så brugte de stories og scenarios til at kommunikere opførsel med og selv om det var første gang forretningen skulle arbejde ud fra denne metode var de motiverede til at afprøve det. Kommunikationen mellem udviklerne og forretningen fungerede rigtig godt, det gavnede at forretningen kunne se de havde indflydelse på processen hele vejen igennem. Forretningen var også meget begejstrede for at input og output var meget mere synlig end hvad var de var vant til.

Fordi det hele var mere synlig steg tilliden også, de kunne se direkte på scenarios hvad der skulle ske i den iteration der var i gang. Til at starte med lavede man scenarios meget detaljeret, netop for at opbygge denne tillid, dvs. at på en 14 dages iteration havde man kun en til to scenarios med. At lave meget detaljerede scenarios til at starte med gjorde at både udviklere og forretningen følte sig tryg ved at man kom hele vejen rundt og fik det hele med. Efterhånden som tilliden steg tog man et skridt tilbage i detaljeringsgraden på scenarios, dog ikke mere end man var sikker på at bibeholde den tillid som man havde opbygget.

Selv om BDD minimerer det så slipper man aldrig af med at forretningen siger ”det troede vi var lavet”, det er mennesker man har med at gøre så misforståelser vil altid opstå.

I og med man har sine scenarios som udgør ens regressions test suite så kan man tillade sig at være lidt aggressiv i sin refaktorering. Kunne man ikke det ville det være forbundet med stor risiko at ændre noget af den underliggende model fra iteration til iteration.

## Hvad skal man være opmærksom på?

Der er også nogle ting man skal være opmærksom på, nogle af de ting er en konsekvens af noget af det som fungerede godt. Disse ting vil vi komme med en lille beskrivelse af.

**Store datamængder**

Når man når op i lidt større systemer så får man store data mængder at holde styr på, det er en

konsekvens af at alt in- og out-put bliver synligt. Derfor er det også vigtigt man kun verificerer det som der er brug for og ikke mere, omvendt skal man også være sikker på at få det hele med. Man kan også komme ud for at når man starter med BDD, så specificerer man meget detaljeret. Efter man så har fået lidt erfaring med det opdager man så at man skal slette noget af det igen, man skal bare være sikker på det ikke går ud over ens tillid til at systemet gør som det skal.

**Hvad skal Then være**

At finde *Then* kan også være svært. Hvad er det der er vigtigt, er det et enkelt felt eller er det staten på et objekt. Er det staten på et helt objekt så er det måske ikke hensigtsmæssigt at have alle data direkte i scenario men der imod ligge det ud i f.eks. en xml fil, alternativt kunne man have data på de vigtigste felter direkte i scenario og resten i en fil. Begge metoder betyder at det kan blive svære at læse et scenario eftersom at man ikke kan aflæse precondition direkte, men hvis opsætning er så kompleks at man mister overblik over det væsentlige; at specificere opførsel og definere acceptjriteriet, kan det være nødvendigt at isolere opsætningen.

**Ikke altid ”Driven”**

Selv om BDD foreskriver at core stakeholders skal sidde sammen eller i hvert fald stå til rådighed så er det ikke altid virkeligheden. Nogle gange sker der bare ting så forretningen ikke lige er til rådighed, her er man så nødt til at tilpasse sig lidt fordi man er nødt til at komme videreMan må så klare sig med den viden man har eller det scenario som er skrevet men ikke helt er gennemarbejdet og så må man efterfølgende få verificeret at accept kriterierne er korrekte. Det er dog uhyre vigtigt man får gjort det så man i sidste ende står tilbage med nogle scenarios som forretningen har sagt god for.

Når dette opstår så er det meget vigtigt at man ikke begynder at skyde skylden på hinanden, det vil være alt ødelægende for den kommunikation som er så vigtig. Det man som udvikler gør er at forklare forretningen at det kommer med i næste iteration og så må man tage den derfra, det gør også at de føler der bliver lyttet til dem, de bliver taget alvorlig og de rent faktisk har indflydelse på hvordan slutproduktet bliver.

**Scenarios er kode**

Det er vigtigt at scenarios betragtes som ganske almindelig kode med de samme regler. Hvis man har noget dubleret kode så kan det hele hurtigt blive meget uoverskueligt. Har et scenario alt til fælles med et andet så gør den afhængig af den i stedet for at dublere den. En anden ting man skal passe på med det er copy paste, hvis det scenario man skal i gang med minder lidt om et andet så bliver man let fristet. Meget ofte resulterer det i at ting bliver dubleret, her skal man udtække det som er ens da det bliver meget lettere at vedligeholde og forstå.

**Genbesøg gamle scenarios**

Efter at de havde været i gang i et års tid besluttede de lige at se nogle af de gamle scenarios igennem. Her fandt de ud af at ikke alle data var i sync med projektet. Det var ikke noget som gjorde det fejlede men det kunne være noget som var unødvendig at verificere mere. Det kunne også være data kunne reduceres uden det gik ud over dækningen, man er dog også nødt til at inddrage forretningen og det kan godt være lidt svært at argumentere for at det er nødvendigt at gøre. Det koster noget tid men resulterer også i at der ikke er så meget der skal køres igennem for at verificere ting og fordi der ikke længere er så mange data er det også nemmere at forstå, så det er meget vigtigt at holde sine scenarios ajour med virkeligheden.

**Vær forberedt på diskussioner**

Det er to forskellige verdener som støder sammen her, forretningen som typisk ikke er tekniske og udviklerne som i høj grad er det, dette forekommer ofte også inden for udvikler gruppen. Man skal

være opmærksom på at forskellen mellem de to grupper ikke udmønter sig i at en af grupperne ikke føler sig veltilpas i samarbejdet, det gælder om at finde et passende abstraktions niveau på scenarios, eksempelvis vil forretningen typisk være interesseret i features men de er tit for store til at skrive en story på. Udviklerne derimod vil gerne have det på et niveau hvor det føler det giver hurtig fremdrift, så det gælder om at finde en balancegang mellem forretningen, udviklerne og BDD.

**Resultatet**

Både udviklerne og forretningen synes det var en god måde at køre projektet på, de mødte selvfølgelige nogle udfordringer undervejs men når man bare var klar over de var der så var det til at håndtere. Kommunikation mellem stakeholders er alfa omega for om hele filosofien bag BDD kommer til at virke, i dette tilfælde virkede det rigtig godt. Er kommunikationen ikke på plads vil det fejle, fordelen er så at man relativt hurtigt finder ud af det når man skal vise hvad der er lavet i iterationen og så har man muligheden for at rette op på miseren.

## Giver BDD mening i alle former for projekter?

Er BDD en egnet fremgangsmåde i udviklingen af alle typer software? Intet taler for at man ikke kan benytte BDD eftersom alt software har en eller anden form for opførsel som man kan verificere og giver værdi til en eller flere personer. Der er dog en række problemer forbundet med at bruge teknikken på forskellige typer software projekter. Vi vil her prøve at komme med vores bud på hvor det er fornuftigt at bruge samt hvor vi mener der skal bruges andre metoder.

Til at understøtte vurderingen vil der blive taget udgangspunkt i et par eksempler. Det er lidt nemmere at forholde sig til end en generel snak.
Hvert eksempel vil vi prøve at holde op mod nogle af punkterne i Dan North's definition af BDD.

> ***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 output, resulting in the delivery of working, testet software that matters.*
>
> *Dan North*

### Specifik It-system

Her er et projekt som skal lave et system hvor man ved hvem de rigtige brugere er og de er også i den organisation som systemet udvikles for. Ud over brugerne er der også andre stakeholders som har krav til systemet, dette kan eksempelvis være aspekter omkring sikkerhed, lovgivning, virksomhedens regelsæt osv.
Brugerne ved måske ikke helt præcis hvad det er de ønsker men de har dog en ide om hvilket output man ønsker ud fra input.

**Outside in og pull based**
Brugeren er med til at definere hvad systemet skal kunne i kraft af de stories og scenarios der skrives, ud fra en outside-in og pull based strategi. Det sikrer at systemet kun kommer til at kunne det det skal, hverken mere eller mindre. Det kan selvfølgelige være man sorterer nogle features fra pga. tidsmæssige eller økonomiske overvejelser men så er det overlagt man gør det.
Det skulle også gerne være her at brugerne kommer i gennem en læringsproces hvor de finder ud af

helt præcist hvad de selv har af ønsker og behov og hvordan systemet skal afdække dette.

**Multiple stakeholders**
I et projekt er der 2 typer af stakeholders, core- og incidential -stakeholders.
Core stakeholder er dem hvis vision man skal implementere. Det betyder at de er faste uanset hvordan man hvordan man vælger at lave det. Incidential stakeholders er nogle som variere afhængig af hvilken løsning man vælger.
Alle disse stakeholders skal der tages hensyn til og det er bestemt også en af BDD's fokusområder. Det som er uhyre vigtigt er at der er en god kommunikation i mellem alle parter og alle bliver hørt og lyttet til.

**High automation og working testet software**
Alle de krav som de forskellige stakeholders kommer frem til bliver alle verificeret automatisk via scenarios, det gør at når stakeholder bliver klogere og kommer med ændringer til disse krav eller kommer med nye afledte krav, kan man hurtigt afsløre om disse tilføjelser har nogle uønskede effekter på de øvrige krav.

Sammenfattet så vil BDD passe rigtig godt ind. Her indgår alle de elementer som BDD forsøger at sætte fokus på.

### Standard It-system

Dette projekt minder egentlig meget om det foregående, dog med den ændring at det er en salgs organisation som ikke har adgang til de rigtige brugere, der er også her forskellige stakeholders men igen inden for deres egen organisation og ikke fra brugernes.

**Outside in og pull based**
Her er det organisationens egne forretningsfolk som definerer via stories hvad systemet skal kunne. De har ikke adgang til nogle konkrete brugere og må derfor vha. egen domæne kendskab, brug af user proxies, interviews med potentielle kunder og brugere oa. udlede hvad der kunne være brug for og dermed opstille nogle krav til systemet. Fordi systemet skal kunne sælges til en given gruppe så er det ofte nødt til at være mere generelt, det betyder at man ikke kan være sikker på at det man får udviklet vil blive anvendt af alle som køber systemet. De enkelte inden for samme gruppe har sikkert heller ikke samme krav til systemet og derfor kan der senere komme tilretninger alt efter hvilken kunde man får solgt det til.

**Multiple stakeholders**
Også her er der flere stakeholders, men igen ingen fra kundernes organisation. Det kan sagtens være at kundens sikkerheds afdeling stiller nogle yderligere krav når systemet sælges, dette behøves ikke at have noget med systemets interne sikkerhed at gøre men mere hvordan systemet skal passe ind i kundens sikkerheds strategi.

**High automation og working testet software**
Det system som er udviklet vil stadig være automatisk verificeret og virke ud fra hvordan vores organisation har defineret det skal virke, dog kan der komme yderligere krav fra kunden.

BDD vil godt kunne bruges her også, vi er dog begyndt af fjerne os lidt fra hvad der er optimalt for tanken bag det. Usikkerheden om hvad der helt nøjagtigt er brug for er der stadig men ingen brugere at spørge til råds, man er nødt til at komme med mere eller mindre kvalificerede gæt på funktionaliteten. Hele verifikationen af systemets opførsel er der stadig og gør at man kan lave

special tilretninger for kunder uden at ødelægge eksisterende funktionalitet.

### Teknisk It-system

Her skal lavet et system hvor man har helt styr på hvad det skal kunne. Der er ingen eller meget lidt bruger påvirkning, det kunne f.eks. være noget driver eller sikkerheds software. Her er der ikke meget usikkerhed om hvad der skal laves og heller ikke rigtig nogen brugere at spørge.

**Outside in og pull based**
Under denne process er det meningen man skal blive klogere på hvad systemet skal kunne. Det er der ingen tvivl om i dette tilfælde. Derfor vil det bare være at lægge et ekstra lag på hvis man insistere på at gøre det.

**Multiple stakeholders**
Der kan godt være flere stakeholders men igen så er der ingen grund til at snakke med dem om hvad de har af ønsker, det er der jo allerede helt styr på.

**High automation og working testet software**
Det vil man selvfølgelig stadig opnå hvis man bruger BDD, det er der bare også andre metoder som sikrer uden at der er alt det overhead med som BDD tilføjer.

BDD vil ikke være det naturlige valg her, det vil ganske enkelt være at skyde gråspurve med kanoner. En af de helt store ting i BDD er kommunikation mellem alle de involverede stakeholders, det er der ikke brug for her, og det som denne software skal løse er ikke noget en bruger nødvendigvis er særlig interesseret i, det er bare en forudsætning at det virker.

### Opsummering

De steder det kunne være en løsning at bruge BDD er der hvor der er en hvis form for usikkerhed om hvad der egentlig skal laves. For brugerne giver det også mest mening hvis det fra deres stol er ”processing software”, dvs. de kan ud fra et givent input fortælle hvad outputtet skal være.
Ud fra de tre eksempler kan man også se at jo længere væk fra brugeren man kommer jo mindre godt passer BDD ind. Kommer man over i de rent tekniske systemer som f.eks. sikkerhedssoftware, drivere og den slags så er det ikke BDD der er ens løsning eftersom at BDD ikke er specielt velegnet til at definere non-functionelle krav via en outside-in tilgang.
Dog kan der sagtens komme krav til et system fra sikkerhedsafdelingen som et system skal overholde og det skal der selvfølgelig laves en story på.

## BDD anno 2010 og fremefter

BDD har de senere år manifesteret sig mere og mere i udviklerkredse, men specielt i ruby miljøet hvor Dave Astel der er udvikler på RSPEC frameworket nok er den person der oftest har slået på tromme for at BDD skal være defacto standard for måden at producere software på og Aslak Hellesøy, skaberen af værktøjet Cucumber, som er en pendent til Jbehaves storyrunner har også en stor andel i at BDD er ved at få et solidt fodfæste i ruby miljøet. I andre miljøer som eksempelvis Java, hvor det hele startede i kraft af at Dan North i 2003 byggede JBehave frameworket som erstatning for JUnit, har projektet længe været på standby og først i 2008 fået implementeret mulighed for at eksekvere scenarios skrevet i et naturlig sprog, inden da var det svært at overbevise erfarne TDD'ere om hvad værdien af BDD er, da den generelle holdning blandt disse var at BDD blot er en skarpere definering af hvad TDD i virkeligheden er.

BDD er så småt også begyndt at vinde indpas i Java verdenen, der er fremkommet en rimelig stigning af blog indlæg, kode eksempler og artikler indenfor de seneste par år med udgangspunkt i Java og JBehave, samtidigt har JBehave projektet fået fart og implementeret nye features ind i frameworket der er med til at understøtte BDD filosofien. I .net verdenen er BDD ikke så udbredt en metode, men der er så småt begyndt at dukke værktøjer og frameworks op, mange ad dem lider af at være upolerede og har kun byttet ordvalget ud ifht. hvad der benyttes i xUnit, det giver ikke voldsomt meget hjælp til at gennemføre en BDD proces. Reglen er ofte i .net verdenen at Microsoft skal implementere support for BDD direkte ind i deres udviklingsværktøjer før at det kommer fra græsrods niveau op til at være mainstream.

BDD er ikke en ny revolutionerede metode til at udvikle software på, BDD har plukket fra andre agile metoder og forsøgt at binde disse sammen på en måde der gør transitionen i mellem indsamling af specs, udvikling og acceptkriterier/tests mere naturlig og konsistent. BDD filosofien i sin enkelthed forsøger at samle eksisterende best-practice metoder og være selve udgangspunktet for alle faser af et udviklingsforløb, så derfor er BDD stærkt at kunne mestre eftersom at hele holdet deler samme arbejdsmetodik og forståelse. Det ultimative mål for BDD frameworks vil være at kunne eksekvere stories og scenarios og tests i et naturligt sprog, direkte uden nogen form for mapning til interne klasser. Dette er selvfølgelig ikke på nuværende tidspunkt realistisk med de statiske og dynamiske programmeringssprog som vi kender i dag.

### BDD som en anerkendt disciplin

TDD, som har ca. 4 år mere på bagen end BDD er i dag en anerkendt og udbredt metode at udvikle software efter, spørgsmålet er så hvorfor BDD ikke har fået større udbredelse?. Der kan være forskellige grunde til at BDD ikke er vokset ud af skyggen på TDD og gjort opmærksom på sig selv i eget regi. Vi har forsøgt at sammenfatte en række kardinalpunkter som vi mener enten har eller er årsagen til at BDD ikke er udviklet i samme tempo som TDD.

**Information**

Efter 4-5 mdr. har vi oplevet at gode artikler og use cases omkring brugen og forståelsen af BDD er yderst begrænsede, mange mennesker har skrevet forum indlæg og blogs og skriver om egne erfaringer baseret på meget simplificeret problemstillinger, problemet er ofte at man udtager delelementer fra BDD og forsøger at diskutere disse ud fra en tro at læseren er hjemmevant i test-driven development og man har svært ved at konkludere om teknikken kan holde vand på flere abstraktionsniveauer, samt hvordan man benytter denne på hhv. funktionelle og non-funktionelle krav. Vi har ligeledes oplevet at folk har forskellige tolkninger af hvad og hvordan BDD skal bruges, det kan være forvirrende når man ikke har et udgangspunkt at tolke ud fra ud over nogle ideer og betragtninger fra Dan North. Skal BDD udbredes skal der laves artikler der omhandler hvordan BDD kan bruges i praksis, og de skal være så omfattende at man kan komme længere ned end blot kradse i lakken. Artikler der beskriver BDD i forskellige sammenhænge kan i dag tælles på 1 hånd, og BDD har brug for mange flere artikler der går i dybden med emnet, og helst skrevet af folk der kan formidle viden og er anerkendt for at være dygtige og innovative, som Kent Beck eksempelvis har været det med Extreme Programming og TDD. Folk lytter og giver en metode chancen for at vise sit værd hvis den er anbefalet af mennesker hvor man har stor tiltro til deres faglige evner. BDD mangler i den grad også et bagkatalog af litteratur i bogform, en søgning juni 2010 viser at der overhovedet ingen bøger er skrevet om emnet BDD, der er dog en bog undervejs med udgangspunkt i ruby. Skal en metode have en chance for at få en plads i den agile

værktøjskasse skal den være veldokumenteret i såvel teori og i praksis, det er svært at overbevise folk om at en metode der har eksisteret i 7 år har nogen silver bullet effekt når der måske er 5 kortere artikler og 0 bøger til at dokumentere metoden, resten skal stykkes sammen af dårligt tilgængelige websider og  blog indlæg hvoraf nogle er af tvivlsom kvalitet.

**Uddannelse**

Undervisning i brug af TDD bliver i vid udstrækning benyttet i IT uddannelser der er målrettet softwareudvikling, dette gør at eleven som må formodes at være den næste generation af udviklere og derfor er med til at sætte en standard for hvordan software udvikling skal foregå i fremtiden. TDD har formået at få en plads i undervisningen hvilket giver en selvforstærkende effekt ved at man nu har fået en blåstempling som værende en metode der kan vise resultater både i teori og i praksis, denne blåstempling er et incitament for enhver it-afdeling el. software hus at undersøge hvilke fordele man kan få ved at bruge TDD til at skrive software med. BDD bliver nødt til på samme måde at komme ind med modermælken og lære nye udviklere hvordan man skal anskue et udviklingsforløb ved at tænke på opførsel i en outside-in strategi. En af Dan Norths begrundelser for at definere principperne for BDD var, at udviklere havde svært ved at håndtere begrebet test og ikke formåede at skrive software ud fra test-first teknikken og efterflg. forstå at det de rent faktisk skrev var eksempler og specifikationer på det stykke software som de var i gang med at udvikle, så derfor giver det i BDD sammenhæng ikke mening at man skal skoles i brug af TDD som en selvstændig disciplin selvom teknikken og målet er den samme, for derefter at skulle rewire hjernen til at kunne anskue problemet fra en lidt anden vinkel. Som vi ser det skal BDD ind i undervisningen og indlejre TDD som en del af BDD og fordi vi mener det skal indlejre TDD, er at det ikke giver mening at modtage undervisning i begge discipliner da man kan sige at TDD er et subsæt af BDD og at BDD altid vil blive betragtet som værende en afart af TDD hvis begge skulle leve side om side i undervisningen som særskilte discipliner.

**Værktøjer**

En af TDDs store force er den konsensus der ligger i de forskellige sprog om hvordan tests skal skrives og implementeres. xUnit er nemt at forstå og bruge, der er ikke et enormt opsætningsarbejde for at få en test til at virke. Det simple framework er implementeret på stort set alle sprog og alle implementereringerne virker stort set på samme måde hvilket betyder at man ikke skal kende til den enkelte implementation for at vide præcist hvordan værktøjet virker. Den simpelhed og gennemskuelighed der er over xUnit frameworket gør at man ikke skal bruge tid på at manøvrere i frameworket, men udelukkende bruge tid på at skrive gode tests der driver designet frem. BDD mangler et værktøj med denne klare linie i hvordan og hvad det skal kunne foretage. JBehave er det første framework der er blevet skrevet til BDD og der findes dog også en .net variant NBehave som dog ikke har alle de features implementeret som der er til rådighed i JBehave. Dan North skrev også RBehave som er til ruby, men har valgt at forene RBehaves story module med RSPEC således at RBehave ikke længere er et aktivt produkt. I andre sprog er der overhovedet ikke noget framework eller værktøj der understøtter BDD og eksekverbare stories, disse sprog må så lave en form for kunstgreb i xUnit for at få en tilnærmende effekt der minder om BDD og det er let at forfalde til at lave ”tests” igen hvis værktøjerne modarbejder eller forurener ens tankegang med begreber og konstruktioner der ikke hører til i BDD. BDD bør have et veldefineret framework ala xUnit som kan hjælpe med at holde fokus på BDD processen, hvor frameworket skal være så simpelt at det er nemt at portere til andre programmeringssprog og platforme, som det er i dag arbejder man på forskellige løsningsmodeller i de forskellige programmeringssprog og selv i samme sprog har man forskellige frameworks og værktøjer der gør tingene på sin egen måde og betrager BDD på forskellige abstraktionsniveauer. JBehave fokuserer på et outside-in view ved hjælp af stories og de

tilhørende scenarios. JDave et andet java BDD framework opererer mere som xUnit gør det og inddrager overhovedet ikke begrebet stories og scenarios.

# KONKLUSION

Hovedspørgsmålet som vi har ønsket besvaret med denne rapport er at vurdere om BDD giver os en samlet udviklingsteknik der hjælper os med at forstå, specificere og implementere et projekt ud fra stakeholders ønsker og krav. Derunder har vi i problemformuleringen opstillet en række punkter der samlet skal gives os den fornødne viden til at vurdere om dette mål er løst på en måde der gør forløbet sammenhængende for alle stakeholders i projektet.

Vi vil kort redegøre for vores egen erfaringer og tilgang til BDD og hvilke problemer vi har stødt ind i. Vi havde intet forudgående kendskab til BDD inden vi kastede os ud i at evaluere denne teknik, men fandt hurtigt ud af at BDD primært er et udpluk af best practise metoder fra andre agile metoder der på hvert sit område af udviklingsprocessen bidrager til den samlede løsning.

Vi startede med at opsamle viden om emnet BDD og fik som BDD foreskrev konstrueret en række overordnede stories som dækkede de mest basale regler i Ludo. Scenarios og acceptancekriterier blev defineret til de enkelte stories, men defineringen af scenarios endte ud med at vi tænkte for meget over hvordan scenarios skulle implementeres inden vi havde skrevet en eneste linies kode, og scenariernes detaljegrad var hæmmende for vores evne til at skrive scenarios til de øvrige stories, da man ubevist blev fanget i at tage de foregående stories scenarios og den måde de var tænkt implementeret med i efterfølgende stories. Resultatet blev at stories, scenarios og acceptkriterier blev for afhængige af hinanden og vi indirekte havde lavet alt for meget upfront design. Vi havde mistet fokus på den feature som scenario beskrev da vi fra første færd havde for mange interfaces som vi skulle tage stilling til udsprunget fra vores definering af stories og scenarios, så vi besluttede at lave en ”do-over” og kiggede på vores scenarios igen og omdefinerede disse særskilt uden at tage hensyn til hvad de øvrige stories foreskrev. Efter vi startede forfra gik det langt bedre med at få gang i processen, dog blev scenarios stadig refaktoreret og ændret i kraft af at vi blev klogere på hvordan systemet groede sig frem og hvad det skulle løse. Dette er fuldstændigt i tråd med BDD tankegangen at features bliver skarpere defineret ved en iterativ kommunikation mellem alle stakeholders.

Vi har i forløbet konstateret at få defineret stories og scenarios er den langt sværeste del i processen, og kræver en god portion erfaring i at lave dem i en størrelse der er håndgribelig uden at være alt for detaljeret. Det er svært at sikre sig at en story har den størrelse den skal have og der er ikke nogen entydig regel at forholde sig til, eneste holdepunkt er at en story skal kunne dækkes ved max. 5-7 scenarios, men så risikerer man at skubbe problemet ud til at et scenario bliver for omfattende og svært at lave et eksempel på uden at inddrage for meget interaktion for at få den ønskede opførsel. Man kommer helt sikkert ud i en masse overvejelser om hvornår en story er for stor og skal splitte den op i delhistorier der sammenlagt giver samme opførsel og her er den virkelige udfordring ved at benytte sig af BDD.

*Vi vil tage et kig på hvordan BDD påvirker og integrerer de forskellige stakeholders i et udviklingsforløb.*
Kigger man på BDD som et kommunikationsmiddel er det en glimrende måde at få indraget alle stakeholders i en struktur der kan forståes og diskuteres ud fra, i modsætning til en normal ustruktureret nedskrivning af en samtale. Desuden har stories og scenarios den egenskab at når man senere hen skal tolke det nedskrevne ikke i samme grad risikerer at der opstår misforståelser i form af manglende kontekst som ikke er påtvunget i en normal prosaform.

*Vi vil vurdere hvor nemt det er at tilegne sig metoden for de involverede stakeholders.*
Metoden med at læse og forstå stories, scenarios og acceptkriterier vil vi vurdere som værende nemt, det er straks værre at definere disse og det kræves at brugere, forretningsanalytikere der hjælper med at definere disse stories, har en grundig forståelse for hvad essensen i den story man forsøger at definere er, hvilke elementer der er relevante for at forstå hvilken opførsel man ønsker at få frem (rolle, handling, opførsel). Efter vores egne oplevelser med at definere disse stories, kombineret med personlige erfaringer med forskellige typer af stakeholders, kan dette efter vores overbevisning være svært at få alle stakeholders committet til at afgive den fornødne tid og indsats til at processen bliver optimal.

*Egner metoden sig bedre i nogen typer projekter end andre*.
BDD har sin helt klare force hvis følgende forudsætninger er til stede i et givent projekt, specifikationen er uklar eller mangelfuld og hvis der er en eller anden form for interaktion med stakeholders med anden funktion end konstruktionen af softwaren. Den vigtigste forudsætning er dog at systemet skal være af typen ”processing software” og have en observerbar opførsel da det vil være yderst vanskeligt at opstille stories og scenarios på non-functional krav der ikke har en observerbar opførsel set fra stakeholders side.

*Udfra en praktisk øvelse vil vi evaluere om teknikken håndterer styring stringent nok til at kunne fastholde et overblik over de samlede mål, uden at miste hverken et tab i produktivitet eller fastlåse folk i en rigid struktur.*
Efter vores arbejde med Ludo spillet har vi haft et godt overblik over forløbet og en konstant ide om status på den opførsel der skulle implementeres i henhold til de features vi fastsatte skulle være inkluderet i vores release. Opdelingen i de enkelte stories giver os en række naturlige milepæle der giver tilfredsstillelse at nå, desuden har vi hele tiden styr på slutpælen og kan se afstanden mindskes dag for dag. Prisen på den korte bane er tab i produktivitet da man holder styr på både et scenario harness og et normalt testharness, begge skal vedligeholdes og refaktoreres til at reflektere produktionskoden, så umiddelbart vil vi mene at der i kraft af dette vil være en smule længere udviklingsforløb. Det længere udviklingsforløb kan indhentes i form af færre misforståelser og ,indfangning af de misforståelser der måtte opstå på et langt tidligere tidspunkt. BDD processen er åben for personlig fortolkning og gør metoden fleksibel til at indlejre andre agile metoder uden at miste filosofien og strukturen i BDD, hvilket man har set efter at man har indført hele story/scenario/acceptance som et bærende element i processen uden at det ændrer på det oprindelige BDD budskab. BDD kan dog gå hen og blive en smule rigid og fastlåst når den hænges op på et BDD framework, eftersom at man så er afhængig af hvordan det enkelte framework understøtter og fortolker BDD processen.

*Desuden vil vi kigge på JBehave der er et Java framework specifikt konstrueret med BDD for øje, vi vil kigge nærmere på hvordan frameworket kan hjælpe os til at holde fokus på den rytme BDD har stillet op for os.*
JBehave giver os mulighed for at kunne eksekvere stories med tilhørende scenarios og acceptkriterier der direkte er skrevet af kunden, når der er foretaget en mapning til en Java klasse. Det bevirker at vi har specifikationerne direkte sammen med kildetekst og ikke som et eksternt dokument i en sagsmappe der ikke har en direkte kobling med den stump kode som definerer

opførslen. På den måde vil javaklassen der udfører den opførsel der er opstillet i scenario altid være identisk med den i textscenario, havde den lagt eksternt vil disse kunne leve deres eget liv og ikke være synkroniseret med hinanden.
JBehave stiller 2 Java klasser til rådighed til at repræsentere hhv. scenarios og steps fra en story, dette bevirker at hver story har sin egen separate opdeling og hjælper med at organisere opførsel i overskuelige enheder, dette kan lyde som en lidt ligegyldig egenskab, men vi har læst testklasser der indeholder ufattelig mange tests der intet har med hinanden at gøre og derved mistes der overblik og man riskerer at lave dobbelte tests hele tiden. JBehave hjælper også med at minimere antallet af scenarios ved at stille example tables til rådighed, men vi mener man skal passe på med at benytte disse medmindre at der er en meget god grund til dette, man kan sammenligne det med at man i en testcase ikke vil teste på alle værdier fra 0-255 hvis der ikke er nogen forskel på eksempelvis 7 og 85. Brugen af exampletables gør storien mere kompleks at læse og forstå og steps klasserne i java bliver også svære at aflæse da man får injected parametre i stedet for at kunne læse variabel tildeling direkte i metoden. For at forbedre BDD processen i JBehave kunne vi godt have benyttet os af en bedre måde at ændre i stories/scenarios osv., den virker tung og besværlig eftersom den er 100% manuel og er derfor en kilde til stor frustration når man sidder med et scenario der pludseligt ikke kan køre fordi man selv manuelt skal synkronisere textscenario og javaklassens metode annoterering. JBehave har altså scenario og acceptkriterier stående to steder og strider imod TDD rytmen om at fjerne dubleret kode og eftersom at textstories er en del af kode/testbasen er dette noget man bør have fokus på at forbedre i fremtidige versioner af JBehave.

Lige nu virker BDD som en metode udviklere er begyndt at interessere sig en smule for, men ikke har mod til at kaste ud i et større kommercielt projekt. Vi har nogle formodninger om at det falder tilbage på det faktum at man skal involvere alle stakeholders før at BDD giver nogen værdi og at stakeholders enten ikke er modne til at adoptere eller at udviklerne ikke har nok overblik over hvad det vil kræve af tid og erfaring at sætte i søen med et positivt udfald. Vi vil anbefale at man adopterer BDD internt på projekter der ikke involverer eksterne stakeholders t. på et inhouse projekt hvor stakeholders er let tilgængelige og er med på at dette er en læringsprocess hvor det primære er at få værdifuld erfaring i en ny måde at udvikle software på. Dette gælder i princippet alle former for agile metoder, de kræver noget af de involverede og ændrer hele måden at arbejde og tænke på i forhold til vandfaldsmodellen som måske er nemmere at forstå konceptet i men har vist sig at være umuligt at gennemføre i praksis uden der opstår store problemer.

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”*.

# REFERENCER

## Webressourcer

www.jbehave.org (Official JBehave website)

www.behaviour-driven.org/ (Official BDD website)

blog.dannorth.net/introducing-bdd (Dan North)

www.ryangreenhall.com/articles/bdd-by-example.html (Ryan Greenhall)

## Videocast

www.infoq.com/presentations/bdd-dan-north (Dan North)

www.skillsmatter.com/podcast/agile-testing/introduction-to-behaviour-driven-development (Elisabeth Keogh)

www.skillsmatter.com/podcast/agile-testing/a-year-of-misbehaving-a-retrospective-on-using-bdd-in-a-commercial-enterprise (Mauro Talevi)

## Artikler

Mocks aren't stubs (Martin Fowler)

No Silver Bullet: Essence and Accidents of Software Engineering (Frederick P. Brooks)

## Bøger

The RSpec Book (David Chelimsky et al.)

User Stories Applied (Mike Cohn)

Test driven development (Kent Beck)