# Indholdsfortegnelse





Af Nick Scott Sylvester, Kennie Helmbo og Sofus Albertsen

# Forord

Dette bachelorprojekt er udarbejdet af 3 studerende på uddannelsen Professionsbachelor i Softwareudvikling ved Erhvervsakademiet København Nord.

Denne rapport er udarbejdet i perioden fra d. 1 maj, 2012 til d. 8. juni, 2012. Den tager udgangspunkt i det forløb vi har været igennem i firmaet Semaphor i perioden fra d. 23 januar, 2012 til d. 30 april, 2012.

Rapporten skal læses med det for øje, at næsten alle værktøjer og teknologier har været os helt ukendte før projektets start. Ydermere har vi løst regnet brugt halvdelen af vores tid på rent faktisk at lave det udviklingsprojekt, vi fra starten havde fået stillet. Derfor kan vi ikke, med den korte tid vi har haft, sige at vores rapport er udtømmende, ej heller komplet i forhold til gennemgangen og udvælgelse af de værktøjer, som er beskrevet.

Vi vil gerne takke Semaphor for opgaven, vejledningen og samarbejdet. Ligeledes vil vi gerne takke vores vejleder, Lars Bogetoft, for konstruktiv kritik og vejledning.

Forfattermatrikset, der beskriver fordelingen af hoved- og bi-forfattere på hver hoved- og biafsnit, ligger som bilag 3 på cd'en.

# Indledning

I alle udviklingsprojekter benytter man en samling af værktøjer, metodikker og standarder. Det kan være disse er bestemt på organisationsniveauet eller skal bestemmes for det individuelle projekt. Uanset hvad, skal denne samling udarbejdes på et tidspunkt. I den proces bliver der investeret tid og penge. Af den grund har man ofte ikke interesse i at bruge endnu flere ressourcer på at offentliggøre de oplevelser, man har haft, og den viden man har tilegnet sig. Andre organisationer, der skal i gang med et udviklingsprojekt, har også brug for at samle en række redskaber, men bliver nødt til igen at starte fra bunden med at ”opfinde den dybe tallerken”.

Vi har i forbindelse med vores eget udviklingsprojekt, haft samme udfordring. Vi havde ingen prædefinerede redskaber og måtte derfor også starte fra bunden. Vores fokus vil være på en række udviklingsværktøjer, som vi har valgt at kalde ”toolbox”. Vi vil igennem rapporten prøve at besvare spørgsmålet: "Hvordan kan man sammensætte relevante værktøjer for at skabe et udviklingsmiljø, der er så tilpas generel, at det kan bruges af alle der udvikler indenfor Java?".

## *Problemformulering*

- Hvordan kan man lave et udviklingsmiljø, som kan understøtte små projekter, men som samtidig kan opskaleres, hvis behovet opstår?
- Hvilke typer af værktøjer har man som udvikler brug for, når man søger at få mest mulig støtte i udviklingsarbejdet?
- Hvilke kriterier er vigtige, når man skal udvælge værktøjer?
- Hvordan kan et samarbejdsværktøj, helt konkret Rational Team Concert, hjælpe i udviklingen af et produkt?
- Hvordan kan et projekt benytte test så det hjælper udviklingen og kodekvaliteten, uden samtidig at sinke udviklingen generelt?

## *Formål*

Ethvert udviklingsprojekt har en samling af værktøjer, og disse værktøjer kan hjælpe med bl.a. kvalitetssikring, samarbejde, produktivitet og automatisering. Vi vil prøve at besvare de førnævnte problemstillinger ved at beskrive de oplevelser, vi har haft med at installere, indstille og bruge en række værktøjer. Det er vores forhåbning, at vores toolbox vil kunne hjælpe små virksomheder med at få etableret et udviklingsmiljø, som kan vokse sammen med firmaet, hvis firmaet opskalerer sine aktiviteter.

**Det konkrete problem** kan udtrykkes således; en lille udviklingsvirksomhed udvikler et system, der er så komplekst, at udviklingen ville sande til uden ordentlige værktøjer til at hjælpe udviklerne.

**Det overordnede problem** kan beskrives således: Udvikling er kompleks og fejl er uundgåeligt. Samtidig er der et konstant pres på at få afleveret funktionalitet. Ved hele tiden at skulle producere, får man ikke tiden til at finde og rette fejl. Ved ikke at rette fejl, er der ikke værdi i den funktionalitet, som man leverer. Samtidig forøges risikoen for at tabe overblikket, jo flere versioner af ens kode der er i produktion. For hvordan ved man hvilke versioner, der er ramt af en bestemt fejl, og hvordan kan man sikre sig, at fejlen bliver rettet i alle de implicerede versioner?

For at undgå denne problemstilling skal man afsætte tid og penge til at oprette et passende udviklingsmiljø, eller opgradere det eksisterende. Dette bliver omkostningsfyldt, hvis det skal gøres fra bunden.

**Den overordnede løsning** på problemet kan beskrives sådan: Hav metoder og værktøjer til at understøtte udviklingsarbejdet. Hav den rigtige samling, for hvert værktøj repræsenterer en udgift, idet man skal sætte sig ind i det, og bruge det, foruden evt. licensomkostninger. Ved at have den rette støtte kan man mindske den risiko der ligger i at man, når man udvikler store løsninger, ikke taber overblik over forløbet.

Vi vil gennem denne rapport præsentere en **konkret løsning** på problemstillingen.

Inden vi gik i gang med at udarbejde vores toolbox havde vi en ide om hvilke projektområder vi ville fokusere på og hvilket aspekt de hver især understøtter. De følgende tre punkter er vigtig for ethvert projekt, uanset størrelse.

**Test** er med til at højne kodekvalitet sikre kvaliteten af ens kode. Ved at lave gode test, har man en større sikkerhed for, at ens system opfører sig, som det skal.

**Projektstyring** er en vigtig aktivitet, når man har deadlines, der skal overholdes. Ved hele tiden at have overblik over projektets tilstand, kan man hurtig identificere og rette op på de steder, som sinker projektet.

**Konfigurationsstyring** hjælper med at integrere og styre kodebasen og andre artefakter. Ved brug af konfigurationsstyring forbedrer man projektets versionshåndtering. Derved kan man nemmere identificere og udbedre fejl på tværs af versioner.

Vi vil beskrive opbygningen af vores toolbox ud fra disse tre fokusområder. Derved vælger vi ikke at beskæftige os med udviklingsmetodikker, de menneskelige aspekter af projektledelse, organisationsopbygning, eller hvordan man bedst optimerer de omgivelser man sidder i. Vi prøver derimod at beskrive hvilke steder en udvikler kan have størst gavn af at blive understøttet, og ud fra de kriterier vi oplister, at give konkrete forslag til værktøjer, der kan opfylde disse krav.

## *Rapportens struktur*

Vi vil for kontekstens skyld beskrive den udviklingsopgave som lå til grund for vores opbygning af udviklingsmiljøet. Her beskriver vi opgavens krav, herefter gennemgås implementeringen og afslutningsvis forklare hvordan fokusset for vores bacheloropgave ændrede sig fra et udviklingsprojekt til et udviklings**miljø**projekt.

Derefter præsenterer vi de områder som generelt kendetegner et udviklingsforløb, og vi klargør hvilke vi vil beskæftige os med. Vi vil kigge på disse områder med tilhørende aktiviteter og

beskrive de konkrete værktøjer, vi har valgt at bruge under aktiviteterne. Samtidig vil vi beskrive kriterierne, som har været baggrund for de valg.

I det sidste kapitel beskriver vi den opsætningsproces, vi har haft i løbet af projektet og de konkrete samspilsproblemer værktøjerne imellem, som vi er stødt ind i. Vi vil også diskutere, hvordan vi tror vores slutopsætning vil klare en opskalering, hvis den skulle tages i brug på projekter af stigende størrelser.

Til sidst i rapporten vil vi komme med en konklusion som samler op på hele rapporten og besvarer vores problemformulering med afsæt i rapporten.

## *Læsevejledning*

Rapporten er opbygget så den kan læses kronologisk. Vi har bestræbt os på at gøre kapitlet ”Opsætning” uafhængig af rapporten, da denne del er vores bidrag til andre, som er interesserede i at lave en opsætning baseret på vores toolbox.

Rapporten er skrevet med den forudsætning, at læseren har et fagligt teknisk niveau svarende til en 3. semester studerende på professionsbacheloren i softwareudvikling. Der er så vidt muligt forsøgt at henvise til forklaringer eller definitioner på udtryk som ikke er blevet berørt centralt i undervisningen.

Vi har bestræbt os på at gøre rapporten så læsevenlig som muligt. Dog må vi advare om, at der er mange tekniske begreber og komplicerede funktioner i spil, så det kræver tålmodighed at komme igennem kapitlerne. Der er komplicerede fagområder i spil som kan være vanskelige at gøre letlæselige og mange termer som kan være svære at forstå. Fordi vi beskriver konkrete løsninger med konkrete værktøjer, bliver der også brugt termer, der er specifikke for disse værktøjer. Vi vil bestræbe os på at forklare disse begreber, når de opstår[1].

Referencer til litteratur i form af bøger bliver med begyndelsesbogstav for hvert ord i den primære titel. Eksempelvis bliver en reference til bogen ”Foundations of Software Testing” markeret [FoST].

Internetreferencer bliver refereret med deres kronologiske forekomst i rapporten. URLen: http://www.sa.dk/content/dk/for_statslige_myndigheder/aflevering/it-systemer/aflevering_efter_bekendtgorelse_nr_1007 er den første forekomst af link i rapporten, og bliver derved navngivet [1].

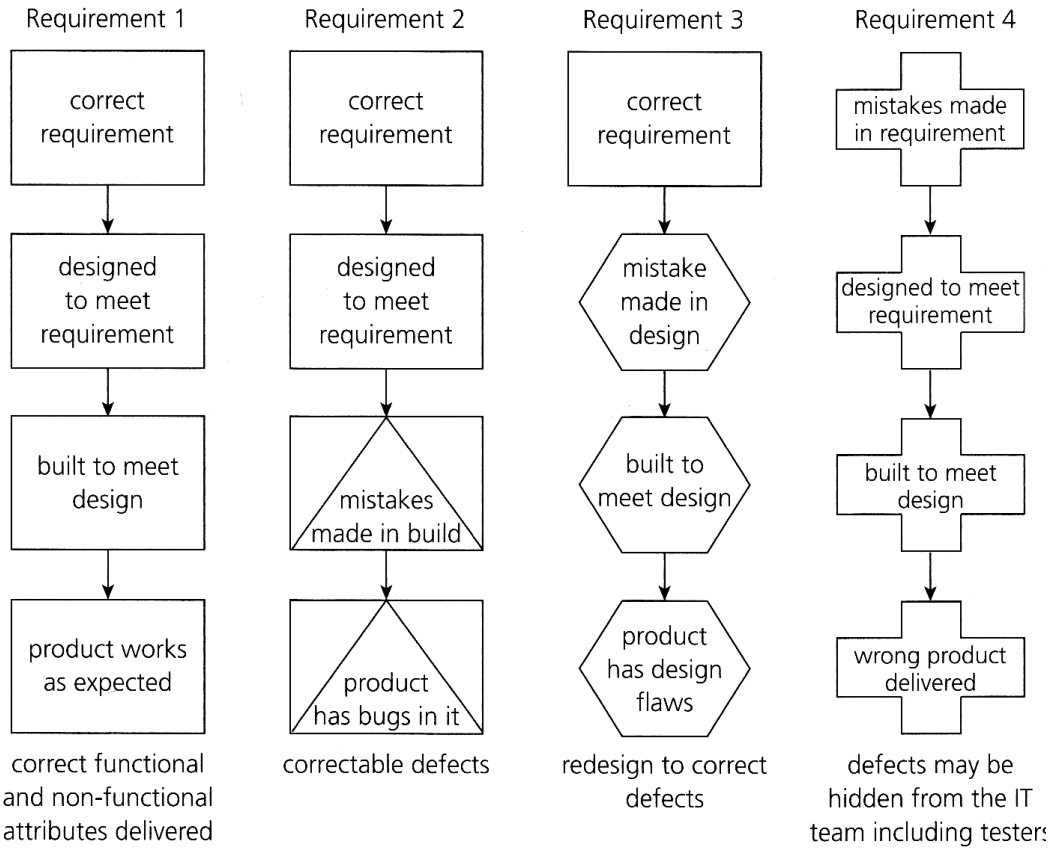
Alle de figurer som optræder i denne rapport, kan findes på CD'en i mappen "~/Figurer", hvor ”~” er roden på mediet. Filerne er navngivet i forhold til deres placering i rapporten. For eksempel hedder det første billed i rapporten "Figur 1. filstruktur forklaret.png". Bilag, kopi af web sider og source koden ligger i henholdsvis:

- "~/Bilag"
- "~/Web Pages"
- "~/Source som Eclipse projekter"

1 Hvis der er specifikke begreber indenfor RTC som ikke er beskrevet, har IBM en begrebsliste her: [40]

# Test

Software er en foranderlig størrelse. Når et system udvikles er det med et formål i tankerne; at løse et eller flere problemer og opfylde de krav man har til systemet. I løbet af hele udviklingsprocessen kan der opstå fejl; fra udspecificering af krav, over design af løsningen, til udførelsen af designet. For nok så mange tests af den skrevne kode kan ikke forhindre en forkert kravspecifikation i at resultere i det forkerte produkt.




*Figur 9: Viser hvor fejl kan opstå, og hvordan fejlene udmynter sig i produktet, hvis de ikke fanges.*
*Kilde: [FoST], figur 1.1, side 5.*

I dette kapitel vil vi beskæftige os med, hvordan man kan forhindre, at fejl opstår, mens man bygger løsningen på problemet.

Systemer som er i brug, interagerer ofte med omverden og har det med at forandre sig over tid.

Kompleksiteten i et projekt stiger i takt med ”Lines of Code” (LOC[7]). Øget kompleksitet kan med tiden blive bekostelig uden sikkerhed for kvalitet. Koden bliver sværere at vedligeholde, og ændringer giver større risiko for fejl. Derfor er test en vigtig aktivitet for ethvert projekt. Det er gennem test, at man kvalitetssikrer det produkt man laver. Der er mange kategorier af test. Helt overordnet set er der dynamiske og statiske test.

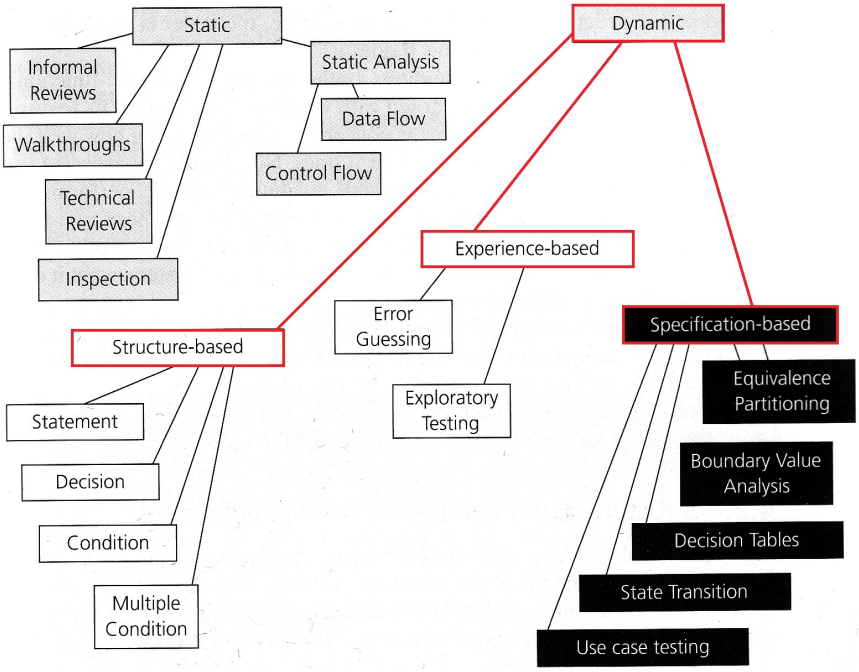
I dynamiske test sammenligner man input og output mod hinanden for at få et forventet resultat. Disse tests kan for de flestes vedkommende automatiseres, så man ikke skal udføre dem manuelt. Statiske test er baseret på en manuel gennemgang af arbejdsprodukter. Det kan eksempelvis være en gennemgang af kundens krav op imod systemets funktionalitet for at se, om systemet opfylder kravene. Derimod er dynamiske test mere beskæftiget med at finde fejl under produktionen af kildekoden, mens statiske test har større fokus på at finde fejl før eksekveringen af koden. Deres forskelligartede natur gør, at de komplementerer hinanden ved at finde forskellige typer defekter.

Vi vil i dette kapitel ikke beskæftige os med statiske testteknikker, men derimod rette fokus på de dynamiske tests som kan automatiseres.

Under dynamiske test er der tre underkategorier; strukturbaseret, erfaringsbaseret og specifikations baseret, som også er illustreret på figur 10.

Specifikations baseret testning går ud på at tage de krav, som projektet skal opfylde, og lave tests ud fra disse. Kravene kan være helt konkrete, såsom at ingen under 15 år må oprette sig som betalende bruger på den applikation, der skal laves. Det kan også være en gennemgang af en hel use case, hvor man specificerer både main succes scenario[8] og alternative path senarier.

Erfaringsbaseret test, er en metode, der bygger på den individuelle testers erfaring i, hvor der hyppigt forekommer fejl i programmer. Disse test er individuelt udarbejdet og der findes derfor ingen faste guidelines.




*Figur 10: Definition af test teknikker opdelt i henholdsvis dynamiske og statiske teknikker.*
*Kilde: [FoST], Figur 4.1, s. 85*

7 Et begreb der tit bliver brugt til at udtrykke hvor mange fejl et givent system har pr. 1000 LOC. Kan også bruges i Projektledelse hvor man bruger LOC til at estimere et projekts størrelse på.

8 ”It’s called the main success scenario because it describes what happens when everything goes right” - Matt Terski [5]

Strukturbaseret testning går ud på at måle testenes dækning af koden gennem en række metrikker. Metrikkerne kan for eksempelvis være om en kodelinje er kørt ved en bestemt test, eller om alle kombinationer af en metodes if-sætninger er kørt igennem.

Ved at benytte en metode til at teste med er du ikke sikret et fejlfrit resultat i den sidste ende. At lave mange tests er ikke det samme som at finde alle fejl. Men ved at bruge systematiske metoder har man redskaber til at opnå den grad af kodekvalitet man tilstræber, samt bedre metoder til at analysere hvor ens kode har brug for test.

Der findes en bred vifte af test værktøjer. Vi vil i de næste par afsnit beskrive de værktøjer som vi har valgt at inkludere i vores opsætning. Vores fokus er på værktøjer som er nemme at integrere i et udviklingsmiljø, giver hurtig feedback til udviklerne og understøtter dynamiske test.

Vi starter med at beskrive hvilken rolle test har spillet i vores udviklingsprojekt. Dernæst skildrer vi hvad unit test er, og herunder hvordan vi har brugt JUnit til at skrive vores test.

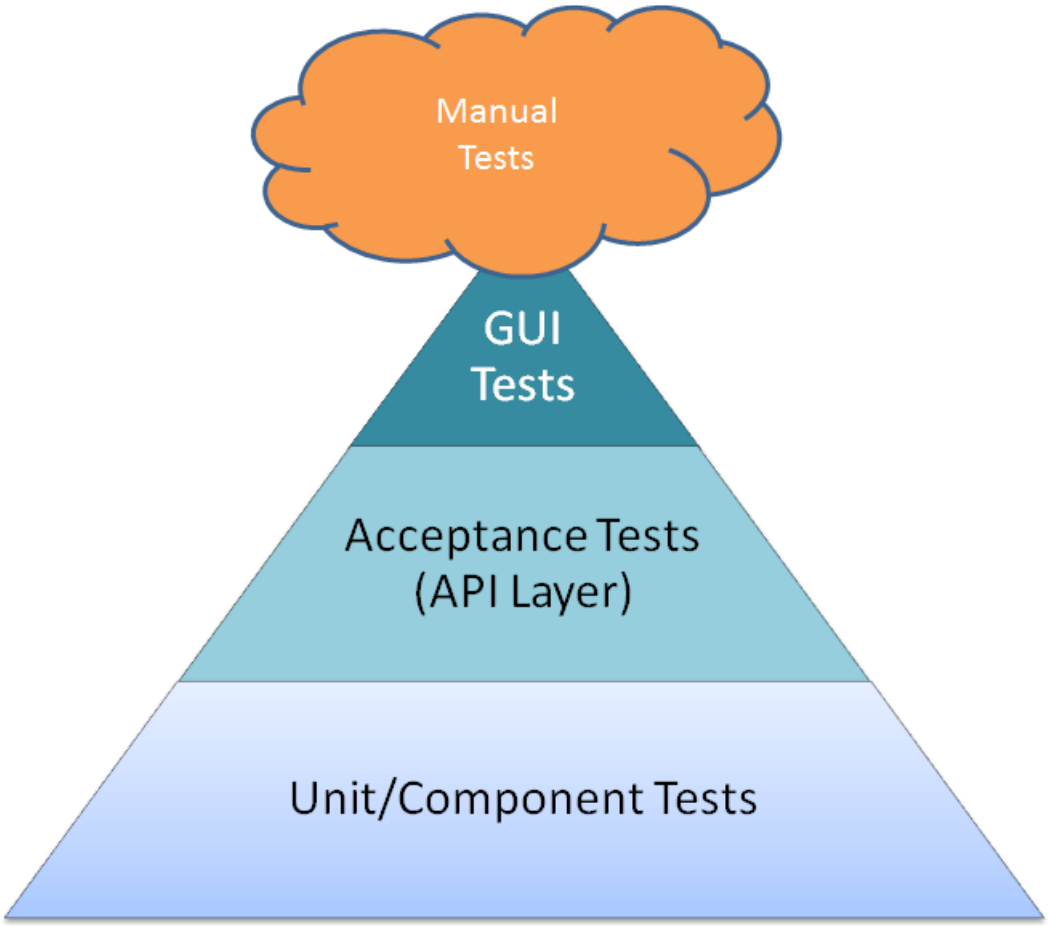
Til sidst kigger vi på koncepterne code coverage og mocking, som vi finder vigtige at tage stilling til i forhold til test. Her kommer vi også med to konkrete frameworks der kan understøtte disse koncepter; Cobertura og Mockito.

## *Testlag*

Automatiserede tests har en lang række fordele over manuelle; De kan gentages når som helst, giver eksakt de samme inputs til programmet ved hver kørsel, og de kræver ikke menneskelig interaktion. Ved at skrive et script, en testklasse, eller lignende, kan man lave en test suite, som automatisk afvikler sine test. Her kan man frit definere inputparametre, rækkefølgen testene skal afvikles m.m. Det gør det nemmere for udvikleren at fokusere på udviklingen, når alt han har brug for at gøre, for at se om koden er ”rigtig”, er at køre en test suite og på en gang få vist hvad der fejler og hvad der består.

Man kan lave tests på flere niveauer. Vi ser disse som opdelt i lagene vist i følgende test pyramide[9]:

9 Figuren er taget fra [EoTA], s. 21. Selve figuren er udarbejdet af Mike Cohn og han gennemgår figuren i en selvstændig artikel her:[6]




*Figur 11: Test Automation Pyramid, som defineret af Mike Cohn.*

Figuren viser hvordan man opdeler sine test i fire lag; tre automatiserede hovedlag og et manuelt lag.

Unit test er, der hvor man tester enheder uafhængigt af hinanden. Det er her ofte på metodeniveau, men der er intet til hindring for at definere units af en anden størrelse. Det vigtige er, at de kan køres selvstændigt, dvs. at der ikke må være krav om, at en eller flere test skal være kørt før nogen andre.

Acceptance test, også kaldet integration- eller service test, er hvor man tester den underliggende forrentnings logik. Disse test spænder ofte over flere komponenter. Selv om alle unit tests, der tester komponenterne internt, består, kan man stadig fejle de dertilhørende acceptance test. På den måde kan man finde ud af, om der er problemer med forståelsen af de interne kontrakter, komponenterne imellem.

GUI test bruges til at sikre kvaliteten af brugerens interaktion med systemet.

I en automatiseret test suite kan man bruge et scripting sprog til at definere en række handlinger, som skal foretages. Alternativt kan man også bruge et værktøj, som optager en rigtig brugers handlinger og afspiller dem med forskellige inputparametre.

Selvom automatiske test har mange fordele, så har manuelle test også deres berettigelse. Der kan til tider stadig være brug for user test i forbindelse med UIen, og udviklerne vil stadig bruge print lines og lignende til at foretage deres egne små hurtige tests. User acceptance tests, hvor en repræsentant for kunden godkender dele eller hele produktet, er også en del af de manuelle tests.

Pyramidens opbygning viser også ,hvordan vægtningen skal være. Man skal have mange unit tests, være moderat med acceptance tests og spare på GUI tests. Det er fordi, unit test kan optimeres til at blive hurtigt afviklet, acceptance test strækker sig over komplekse sammensætninger af forretningslogikken, og GUI tests er historisk set skrøbelige, langsomme og svære at vedligeholde. For eksempel kan en ændring af variabelnavne få testen til at fejle, selvom funktionaliteten stadig er den samme. Det skal understreges, at pyramiden ikke er et udtryk for, hvor besværligt det enkelte lag er, eller hvor meget tid der skal bruges på dem i forhold til hinanden. Ofte bruges der lang mere tid på GUI test end unit test af førnævnte grunde.

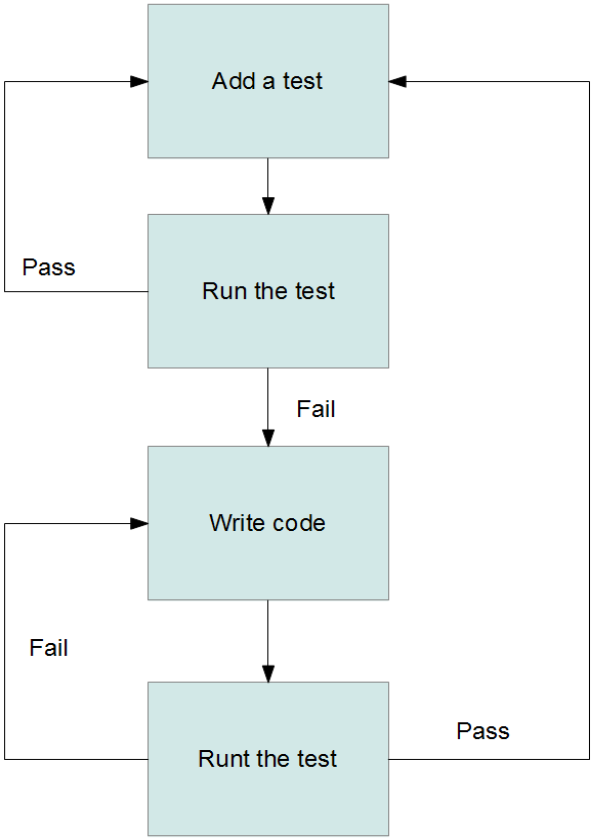
## Test-driven development

Ud over at vælge metoder der kan guide i hvordan man laver ens test, kan man også have en metode der foreskriver, *hvornår* ens tests skal skrives.

Vi valgte i vores projekt at stifte nærmere bekendtskab med test-driven development, herefter TDD, i så vid udstrækning som det var muligt. Erfaringer fra tidligere projekter, hvor test havde fået en mindre rolle i udviklingsprocessen, havde overbevidst os om at prøve denne fremgangsmåde.

Overordnet set kræver TDD et specielt flow gennem udviklingen. Som navnet antyder starter man med at skrive sin test. Allerførst skriver man en test som dækker en lille del af det problem, man ønsker at løse. Så skriver man den simpleste kode man kan for at bestå testen. Dernæst skriver man en ny test, som udvider problemstillingen lidt. Herefter tilføjer man til det første stykke kode så man kan bestå test nummer to. Man fortsætter så på den måde, indtil man ikke kan teste mere. Til sidst kan man refaktorere koden, så det passer ind med de kodestandarder, som projektgruppen har defineret.

En af de udviklingsmetoder, der bruger test first, og som har hjulpet med til TDDs popularitet, er extreme programming. Her er unit test sammen med test first en af




*Figur 12: Test-driven development livscyklusen*

hjørnestenene. Faktisk må man ikke integrere kode, som ikke har tilhørende unit test[10].
Selvom det teoretisk virker som en simpel måde at inkrementere sin kildekode på, så er vores oplevelse, at den ikke er uden vanskeligheder. Den største udfordring var at skrive test uden at vide helt præcist, hvordan det skulle implementeres i koden. Det gør sig gældende for eksempel hvis man ikke kender det bibliotek, der skal bruges, til fingerspidserne, og man samtidig bruger mocking til at emulere biblioteket. I sådanne tilfælde bliver det svært ikke at skrive et første udkast af koden, før man skriver testen, for at opnå et vis kendskab til API kaldene. Dette fik os til at lave vores egen lidt anderledes version af TDD for at afspejle de forhindringer, den originale TDD definition indeholder.

## Test first fremgangsmåde

Vores TDD scenarie ser sådan her ud:

1. Analyser Use casen ud fra happy path og alternative path
2. Opret de klasser der skal med, samt skriv skeletkode til metoderne. Hvis der skal være interfaces, så skriv disse også.

1. Skriv test med mockobjecter til alle metoder og klasser der lige er blevet skabt

- De næste punkter sker iterativt
  1. Skriv test for de interne metoder og kald der skal ske
  2. Skriv kode
  3. Bestå interne tests
- Afslutningsvis
  1. Bestå UC tests
  2. Overdrag UC til godkendelse hos en anden
  3. Godkend UC ved at holde Cobertura raporten op imod de krævede % af testcoverage, samt analyser om der er skrevet test til de rigtige steder.

1. UC er færdig

Vi opfatter det ikke som at skrive kode når der bliver lavet skeletter til klasser.

*Figur 13: Uddrag af vores team kontrakt, hvor vi beskriver hvordan vi skal lave Test driven development. Begrebet skelet skal forstås som gennemløb af koden, som beskrevet ovenfor.*

## *Unit testing*[11]

Unit testing er den testform, der tester de mindste subsets af et projekt, en ”unit”. Fordi unit tests' fokus er så snævert, gør det dem meget egnede til at blive automatiserede. Der findes et overordnet navn for alle de frameworks, der forsøger at hjælpe udvikleren med at skrive unit tests; xUnit.

Navnet xUnit blev udledt af navnet JUnit, som var det første af denne type test frameworks, der

10 ”Code without tests may not be released. If a unit test is discovered to be missing it must be created at that time.” [7]
11 Baggrundmateriale til afsnittet: [8,9,10]

blev almindelig kendt. JUnit blev undfanget af Erich Gamma og Kent Beck under en flyvetur mellem Zurich og Atlanta i 1997. Efterhånden som det slap ud til udviklere, blev det voldsomt populært og har i høj grad været med til at forme Extreme Programming og Test Driven Development.

Filosofien bag JUnit er at gøre processen med at teste mere strømlinet og intuitiv. På den måde vil det være nemmere at benytte test som et integreret element af udviklingen. I visse udviklings metodikker er test således et af de første elementer der benyttes. Her tænkes specielt på agile processer som extreme programming. Langt størstedelen af tilgængelige unit test frameworks i dag er baseret på netop denne type[12]. Centralt for disse typer af værktøjer er automatisering af test uden behov for at skrive de samme test flere gange og uden at huske resultatet af hver enkel test. Ved ikke at skulle huske resultater er det nemt at verificere, at kode, som blev skrevet på et andet tidspunkt i udviklingsprocessen, stadig fungerer. En uundværlig feature i et produktionsmiljø hvor koden ofte undergår forandringer. Selvom der er individuelle forskelle mellem de forskellige xUnit værktøjer, så indeholder de dog fælles elementer som en del af deres basale arkitektur. Herunder test cases og assertions. Vi giver en nærmere forklaring af centrale begreber i eksemplet længere nede i kapitlet.

I et tidligt oplæg fra Kent Beck gennemgår han nogle af de aspekter som for ham, gør unit testing til et oplagt redskab i ethvert udviklingsmiljø[13].

Han beskriver, hvordan han på tidligere projekter har stiftet bekendtskab med test som i høj grad var baseret på interface test. Dermed kunne en mindre, omend kosmetisk, ændring i interfacet få en masse test til at fejle. Det betyder også, at man bruger urimelig meget tid på at lave ændringer i test i stedet for i produktionskoden.

Den store ændring ved at bruge et xUnit framework er, at man flytter sine test ned i koden ved at teste metoder og klasser. Så i stedet for en større test af de integrerede elementer koncentrerer man sig om de enheder (units), som systemet er bygget op af.

En interessant anbefaling fra Kent Beck er, at udviklerne bliver ansvarlige for deres egen kode og derfor dedikerer 25-50% af deres tid på at udvikle test. En anbefaling som bestemt er ført videre i TDD, som Kent Beck er krediteret for at have 'genopdaget'[14].

### *JUnit*

JUnit er som sagt en del af de test frameworks som klassificeres under betegnelsen xUnit.

For at beskrive de centrale begreber i JUnit tager vi udgangspunkt i følgende eksempel.

12 For en komplet liste af unit frameworks, se [11]

13 ”When I was on a project where we used user interface testing, it was common to arrive in the morning to a test report with twenty or thirty failed tests.”,[10]

14 ”He has pioneered software design patterns, the rediscovery of test-driven development, as well as the commercial application of Smalltalk.”, [12]

```
import static org.junit.Assert.*;

import org.junit.After;
import org.junit.Before;
import org.junit.Test;
import java.util.*;

public class GroceryAppTest {

private List groceryList;

@Before
public void setup(){
	groceryList = new ArrayList();
}

@After
public void tearDown(){
	groceryList = null;
}

@Test
public void testAddItems(){
	groceryList.add("milk");
	assertTrue(groceryList.size()>0);
}
}
```

*Figur 14: Et eksempel på en JUnit test case.*

For at kunne arbejde med JUnit bliver man først nødt til at importere de relevante biblioteker **(1)**. Herefter definerer man en test klasse **(2)**. Klassen skal være public, men der er ingen krav til navngivning. Det er dog god praksis at navngive sine klasser *XXXTest*. Da vi har en liste, som vi ønsker at bruge i flere test, erklærer vi en global variabel **(3)**. For at kunne bruge listen skal den instantieres. I stedet for at gøre det i hver test metode kan vi bruge en test fixture. Det fungerer simpelthen ved at lave en setup metode med annotationen *@Before*. Denne metode bliver så kaldt før hver test metode **(4)**. Alle ressourcer som allokeres i setup metoden skal frigives efter hver test. Det bliver gjort i tearDown metoden, som bruger @After annotationen **(5)**. Når man skal angive at den følgende metode er en test, bruger man annotationen *@Test*. Der er igen ingen krav til metodenavnet, men her plejer det at være god praksis at bruge en *testXXX* navngivning.

Så kalder vi den metode, der skal testes **(6)**. Fordi setup metoden er blevet kaldt før test metoden, så ved vi, at vi har en tom liste. Derfor kan vi nu lave vores verificering af metoden. Det gøres med *assertTrue* metoden som er sand, hvis størrelsen af vores liste er mere end 0. Sagt med andre ord, hvis vores metode *groceryList.add()* har tilføjet et element på listen så er testen positiv **(7)**.

Nu er eksemplet en meget simpel Unit test, men ikke desto mindre er de centrale begreber beskrevet her. En test case er simpelthen test metoden, og en assertion er det, vi bruger i bunden til at checke vores test resultat.

Unit test er det vigtigste test værktøj i vores toolbox. Både code coverage og mock-objekter er med til at gøre det nemmere for udvikleren at teste sin kildekode, men de kan ikke fungere uden unit test. Derudover er det værd at bemærke, at hvis man vil lave automatiseret unit test i Java, er det svært at komme uden om JUnit. Der findes alternativer som for eksemsempel TestNG[15]. Men langt de fleste er enten direkte udvidelser af JUnit eller kommercielle alternativer.
Hvordan de andre værktøjer understøtter JUnit, kigger vi nærmere på i de følgende afsnit.

## *Code Coverage*

Brugere af software er interesseret i gode og stabile produkter. De vil med andre ord ikke have et produkt med fejl, ligeså lidt som bilejere vil have biler med defekter. Måden man undgår fejl på er ved at teste produktet, inden det kommer i slutbrugerens hænder. Gennem en grundig og systematisk gennemgang af softwaren kan man mindske antallet af fejl og dermed øge kvaliteten. Måden man  måler den grundighed på er gennem code coverage.

Når man måler, i hvor høj grad koden er blevet testet så får man dækningen målt i procent. Der er dog mange måder at vurdere dækning  på. Man kan eksempelvis måle på flere niveauer. En måling af dækning på acceptance test niveau kan være funktionelle krav, mens dækning på integrationsniveau kan sige noget om interfaces. Selv på unit/component test niveau er der mange forskellige parametre at måle på. Line coverage går ind og beskriver om du har været inde og kalde hver enkel linje i et program, hvor, mens statement coverage beskriver hvor mange udsagn, der er blevet kaldt. Kort sagt kan dækning måles på mange måder, og de giver allesammen en beskrivelse af koden.
Dækning måles som:

$$\text{Coverage} = \frac{\text{Number of coverage items exercised}}{\text{Total number of coverage items}} \times 100\%$$

*Figur 15:  Udregning af testdækning i procent.*
*Kilde: [FoST] side 106.*

Selvom der er delte meninger om graden af dækning, kan man dog ikke sætte lighedstegn mellem en 100% dækning og 'fejlfri' kode. To forskellige tests kan sagtens opnå samme dækning, men teste på forskellige parametre. Samtidig skal man have for øje at kodedækning kun fortæller noget om hvilken linje, der er blevet testet, ikke hvilke værdier den er blevet testet med. Ved at skrive test kan man forbedre sin kode, men det gør sig naturligvis kun gældende for den del af koden som rent faktisk bliver testet. Desuden tester man kun koden isoleret set, og ikke de krav som koden er til for

15 For mere information om testNG, se [13]

at honorere, som figur s. 21 figur 10 også illustrerer.

At udregne dækningen er en simple opgave, som sagtens kan udføres manuelt. Men efterhånden som kodebasen vokser, så bliver det en uoverskuelig opgave. Ved at automatisere processen med at afgøre hvilken kode der er dækket af test bliver code coverage en naturlig del af udviklerens rutiner. Samtidig gør processen det nemt for udvikleren at identificere problem områder i koden.

Selvom code coverage kan assistere i kvalitetssikringen af koden, så giver det ingen garanti for at finde alle fejl eller mangler. Man kan sagtens afdække en metode komplet og samtidig have undladt at teste områder, som kan medføre fejl i programmet. Disse problemer skyldes, at man ikke har taget højde for det i test koden. Så selv om værktøjer kan bruges til at forbedre såvel test som kode, så fritager det ikke udvikleren for at gennemtænke sit design.

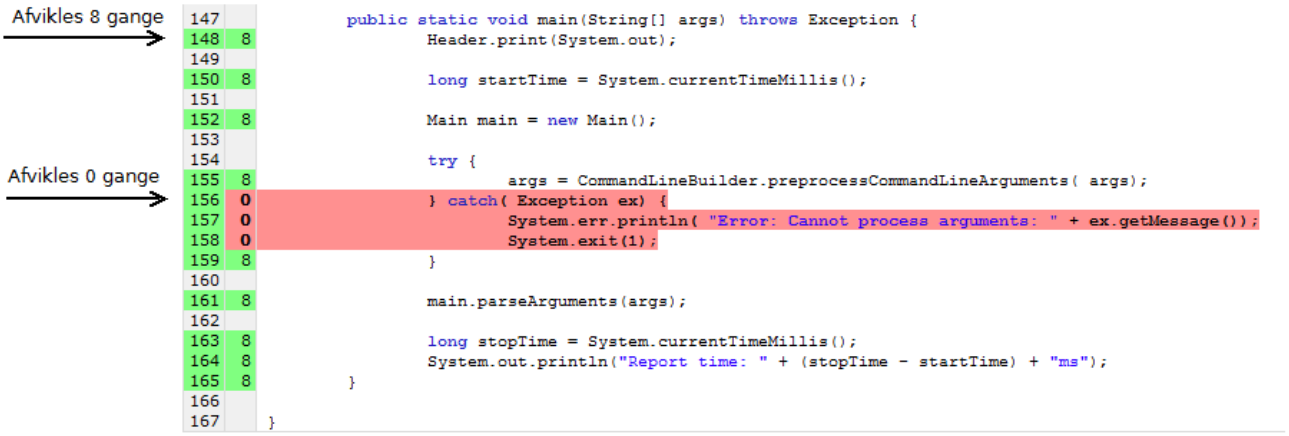
Som før nævnt brugte vi en TDD tilgang til vores udvikling. Inden vi gik i gang med at kode, lavede vi alle tænkelige test scenarier, som vi kunne finde på. Derefter implementerede vi koden, der skulle bestå de test. Hvad vi opdagede var, at når vi bare kodede derudaf, fik vi undertiden kodet funktioner, som vi ikke havde en test til. Grunden var ikke nødvendigvis en mangel på disciplin, men at eftersom man fik en større og mere kompleks kodebase, blev det svært at overskue, hvad der var dækket af test.

## Cobertura[16]

Derfor giver det mening at bruge et værktøj som Cobertura, når man ønsker at automatisere processen med at sikre en høj kvalitet af ens software. Det er et open source værktøj, som er fokuseret på test coverage. Navnet Cobertura kommer fra spansk og betyder dækning.

Cobertura fungerer ved at gennemgå ens kørende test suite for at identificere uafprøvet kode. Selve instrumenteringen foregår på Java byte-kode niveau. Således lægger den instruktioner i de kompilerede Java klasser. Når Java Virtual Machine eksekverer koden inkrementeres en tilhørende tæller, så man kan se, hvor mange gange hver enkel linje er blevet eksekveret. På figur 16 ses, hvordan dette grafisk ser ud.

16 Baggrundmateriale til afsnittet [14,15]




*Figur 16: En grafisk repræsentation af testdækningen på en main metode.*

I marginen kan man se at linje 148 eksempelvis, er blevet eksekveret 8 gange. Koden fra linie 156 til 158 er derimod ikke eksekveret. Udvikleren kan nemt se at, for at øge dækningen, skal der skrives en test, som kommer ind i catch blokken.
Ifølge udviklerne bag Cobertura er denne fremgangsmåde hurtigere end at arbejde på kildekode niveau og kompilere koden to gange eller bruge en modificeret udgave af Java Virtual Machine.

Opsamling af code coverage er gjort nemt i Cobertura, idet der kan autogenereres HTML rapporter, som giver en grafisk repræsentation af dækningen. Herfra er det enkelt at se dækning på klasser, pakker og hele projektet.

**Coverage Report - All Packages**

| Package | # Classes | Line Coverage | Branch Coverage | Complexity |
|---|---|---|---|---|
| **All Packages** | 55 | 75% 1625/2178 | 64% 472/738 | 2.319 |
| net.sourceforge.cobertura.ant | 11 | 52% 170/330 | 43% 40/94 | 1.848 |
| net.sourceforge.cobertura.check | 3 | 0% 0/150 | 0% 0/76 | 2.429 |
| net.sourceforge.cobertura.coveragedata | 13 | N/A N/A | N/A N/A | 2.277 |
| net.sourceforge.cobertura.instrument | 10 | 90% 460/510 | 75% 123/164 | 1.854 |
| net.sourceforge.cobertura.merge | 1 | 86% 30/35 | 88% 14/16 | 5.5 |
| net.sourceforge.cobertura.reporting | 3 | 87% 116/134 | 80% 43/54 | 2.882 |
| net.sourceforge.cobertura.reporting.html | 4 | 91% 475/523 | 77% 156/202 | 4.444 |
| net.sourceforge.cobertura.reporting.html.files | 1 | 87% 39/45 | 62% 5/8 | 4.5 |
| net.sourceforge.cobertura.reporting.xml | 1 | 100% 155/155 | 95% 21/22 | 1.524 |
| net.sourceforge.cobertura.util | 9 | 60% 175/291 | 69% 70/102 | 2.892 |
| someotherpackage | 1 | 83% 5/6 | N/A N/A | 1.2 |

*Figur 17: Et samlet overblik over en række pakker som man kan se på venstre side. Udover line og branch coverage er der udregnet cyklomatisk kompleksitet som er et mål for kodens kompleksistet.*

I vores projekt havde vi i opstartsfasen besluttet en dækning, som vores test skulle overholde. Vi havde besluttet følgende mål.

- 75% code coverage
- 65% branch coverage

Alle de kilder, vi undersøgte, havde forskellige forslag til test dækning, og det gjorde det vanskeligt at træffe en beslutning om, hvilket mål vi skulle sættes os. Specielt fordi ingen af os havde arbejdet

med test dækning i et projekt før. Derfor baserede vi vores endelig mål på denne diskussion fra programmerings forummet stackoverflow[17].

Når man arbejder med code coverage og laver beslutninger om test dækning, skal man være opmærksom på ikke at blive blændet af de procentsatser man har valgt. Der kan hurtig opstå en tendens til at fokusere på at opnå sine mål for dækningen og så glemme at skrive effektive test. Som tidligere nævnt er der ikke en direkte sammenhæng mellem test dækning og fejlfri kode. Det er nemt at forestille sig test, hvor en enkelt randværdi måske opfylder dækning af koden men ikke er effektiv nok til at opdage andre fejl i grænseområdet.

I en undersøgelse af coverage værktøjer foretaget i 2007[18] er en af ulemperne ved denne type værktøj for meget fokus på test dækning og for lidt fokus på at skrive god test kode. Derfor er deres anbefaling, at man lærer sit coverage værktøj at kende og er bevidst om de faldgruber, som der er med dette type værktøj.

## *Mock-objekter*[19]

Når man arbejder med objektorienteret programmering, har man at gøre med objekter, som har en indbyrdes sammenhæng. Selvom man ønsker at indkapsle hvert objekt, så man skaber en veldefineret afgrænsning, så er det nødvendigt for objekterne at interagere for at skabe en funktionel kode. Det forhold har også betydning, når man skaber test.

Forholdet mellem unit test og ens kode er tæt forbundet. Det betyder også at afhængigheder i koden skaber afhængigheder i ens test. Den sammenhæng påvirker vores måde at skrive test på.
En unit test skal være stabil og kunne køres ofte for at kunne bruges af udviklerne. Derfor bliver man nødt til at kunne kontrollere de elementer, som indgår i ens test. Men hvad gør man, når der er afhængigheder til eksempelvis eksterne ressourcer som en database eller en netværksforbindelse?

Det første modul, vi arbejdede på, var Dataretrieval. Langt størstedelen af det modul gør brug af funktioner, som ikke umiddelbart var tilgængelige for os. Heriblandt adgang til Notes databasen, som indeholdt de informationer, vi skulle bruge. At bruge det eksterne bibliotek i vores test for at få fat i data var ikke en løsning. Dertil kunne der gå for meget galt; fejl på serversiden, fejl i netværket, uforudsete ændringer i testdata'erne og så videre.
Vi blev derfor nødt til at finde en anden måde at teste vores metoder på. Det vi skulle bruge var mock-objekter.

Teoretisk set går mock-objekter ud på, som navnet siger, at skabe en imitation af noget man ikke har kontrol over. Generelt virker mocking på følgende måde:

Man erklærer en variabel til den klasse, der skal mockes, og instantierer denne variabel med en mock-version af klassen:

```
Database mockDB = mock(Database.class);
```

Man giver mock-instansen forventninger til at der kommer bestemte kald til den og derefter

17 Diskussionen som vores dækningsmål er baseret på: [16]
18 PDF'en hvori undersøgelsen ligger, kan hentes på [17]
19 Baggrundmateriale til afsnittet [18]

beskriver man de svar den skal returnere.

```
when(mockDB.isOpen()).thenReturn(true);
```

Til sidst kører man sine tests som man havde villet det hvis det havde været et ægte objekt af den pågældende type.

Den kode der bliver testet kommer derved aldrig til at kende forskel mellem rigtige objekter, og så mock-objekterne.

```
if (!db.isOpen()) {
      db.open();
}
```

Strategien med mock-objekter gør det muligt at teste kode som ikke er skrevet endnu, så længe der eksisterer et interface. Undertiden kan det være vanskeligt at teste dele af ens kildekode. Dels fordi det man ønsker at teste er for tæt forbundet med andre elementer, man ikke ønsker at teste, en anden klasse eksempelvis, og dels fordi det ville være upraktisk eller umuligt at teste, en database eller et interface som ikke er implementeret endnu. For at løse denne problemstilling benytter man mock-objekter, som kan simulere 'rigtige' objekter. Dette er en udbredt metode som ofte benyttes i TDD. Man undgår kort sagt interferens med andre metoder eller klasser. I vores tilfælde betød det, at vi kunne teste metoder i vores Dataretrieval modul uden at have adgang til Notes databasen.

Aspektet med at isolere den kode man tester, er en væsentlig grund til at benytte mocking. Ved at undgå interaktion med andre metoder eller klasser kan man skabe små fokuserede tests, som bedre kan isolere koden og give samme resultater ved hver eksekvering. Konstruktionen af flere men mindre tests skal give mere præcis feedback, hvis der opstår fejl.

Mange af de fordele som man opnår ved at bruge et mocking framework kunne også opnås ved at bruge en stub i stedet. En stub er noget kode som bruges i stedet for den ønskede kode. Formålet er at simulere funktionaliteten af den færdige kode. På den måde kan man, ligesom med mocking, teste metoder og klasser, som endnu ikke er blevet implementeret. Der er dog en væsentlig forskel mellem disse to teknikker. Med stubbing kan man kontrollere tilstanden af en metode, således at den eksempelvis returnerer det, man forventer. Men man kan ikke kontrollere opførsel ligeså nemt. Det vil sige at en stub returnere den værdi som man har programmeret den til, men den kan ikke skifte værdi under en test. Det samme metodekald til stubben giver det samme svar hver gang. Sådan forholder det sig ikke med med mock-objekter. Her kan angive hvilken værdier der skal returneres på hvilket tidspunkt. Man kan med andre ord kontrollere dets opførsel.
Essentielt set bruges begge teknikker for at isolere ens test til det element man ønsker at teste, men fremgangsmåderne er forskellige. I Martin Fowlers essay "Mocks Aren't Stubs"[20] argumenterer han for, at valget mellem stubbing eller mocking i høj grad afhænger af ens test stil. Han definerer to typer testpersoner. Dem der hælder til klassisk TDD og dem der bruger mock TDD. I den klassiske TDD vil man så vidt muligt bruge rigtige objekter og en stub eller mock, når det er for vanskeligt at teste det rigtige objekt. Mens man med mock TDD vil mocke alle objekter med en betydningsfuld adfærd.

20 ”This difference is actually two separate differences. On the one hand there is a difference in how test results are verified: a distinction between state verification and behavior verification. On the other hand is a whole different philosophy to the way testing and design play together, which I term here as the classical and mockist styles of Test Driven Development.”[18]

Valget af mocking framework er, som med mange andre værktøjer, påvirket af personlige præferencer. Hvilke funktioner har man brug for, og hvor nemt er værktøjet at bruge. I vores valg af værktøj måtte vi igennem to mocking frameworks, før vi fandt det, som passede til vores behov. Så hvad var forskellen mellem disse tre valgmuligheder, og på hvilken baggrund valgte vi det framework, som blev en del af vores anbefalede teknologier?

## jMock

Da vi indså at det var nødvendigt at bruge et mocking framework, gik vi i gang med at undersøge forskellige valgmuligheder ud fra den referenceramme, som vi havde. Derfor søgte vi inspiration i vores litteratur og de frameworks, der blev præsenteret var jMock og easyMock. Her virkede jMock mere etableret end easyMock, og derfor blev det vores valg.

Der gik dog ikke længe før vi måtte erkende at det var nødvendigt at bruge et andet mocking framework. Det primære problem med jMock var det besværlige setup. Opsætningen af mock-objekter i test koden krævede meget test kode. Det gjorde dels, at det tog lang tid at skrive den nødvendige testkode, men også at det var sværere at læse efterfølgende. Ikke en egenskab vi anså som positiv. I vores dagbog fra perioden lavede vi følgende optegnelse over skrevne kodelinjer til en testmetode lavet med jMock:

- Selve koden: **34** linjer, med formateringslinjer.
- JUnit: **9** linjer (dette tal ville selvfølgelig være et par linjer større uden Mock, men er stadig repræssentativt.)
- Mock: **52** linjer, nogle af disse linjer har to metodekald på en linje, da de hænger sammen.

Derved skulle en metode der havde 34 linjer i sig, have følgeskab af 61 linjers testkode. Et særdeles uligevægtigt forhold mellem reel kode og testkode.

Test skal jo ikke kun bruges under udviklingen men også i den efterfølgende vedligeholdelse, og alene gennemgangen af mange linjers test kode ville blive unødvendig besværlig. Efter gennemgangen af mocking frameworks giver vi et eksempel på forskelle og ligheder mellem jMock, og det framework vi endte med.
Udover det vanskelige setup, så var vi ikke i stand til at mocke klasser[21], uden at skulle tilføje yderligere flere biblioteker, hvilket ikke var tiltænkt.

Alt i alt virkede jMock for omstændigt og vi var overbevist om, at der var værktøjer, som lå tættere på vores behov. Derfor gik vi i gang med at se os om efter et andet værktøj, som kunne leve op til vores forventninger.

21 ”Because it uses Java's standard reflection capability, the default configuration of the jMock framework can only mock interfaces, not classes.”[19]

## JMockit

I stedet for at tage udgangspunkt i vores litteratur igen, begyndte vi at lede efter et framework, som havde den funktionalitet og brugervenlighed, som vi manglede. Efter nogen søgning fandt vi frem til JMockit.
JMockit er en samling af test værktøjer og har udover mocking funktionaliteten også code coverage. Det er skabt som et alternativ til blandt andet jMock[22] og forsøger at overkomme de begrænsninger, som man finder i blandt andet det værktøj. Kombinationen af et alternativ til jMock og muligheden for at få inkluderet code coverage var nok til at vække vores interesse. Nu var code coverage ikke det springende punkt. Vi brugte i forvejen Cobertura, men vi var interesseret i at se hvad JMockit kunne tilbyde.

Det levede op til vores forventninger om mere funktionalitet, men det havde, i vores optik, stadig problemer med brugervenligheden. Til langt de fleste test var det for tidskrævende og vanskeligt til at give os den værdi, vi søgte. Vi stod simpelthen med et værktøj, som havde det samme fundamentale problem som jMock. Det var for besværligt at bruge og tog alt for lang tid i forhold til fordelene ved brugen.
Det vi ledte efter skulle være mere funktionelt end jMock og mere simpelt at bruge end JMockit og jMock.

## Mockito

Kombinationen af et simpelt og funktionelt værktøj fandt vi i Mockito. Mockito er et mocking framework som tilsigter en simpel og intuitiv fremgangsmåde i test-forløbet. Det blev udviklet ovenpå easyMock frameworket af Szczepan Faber, men er siden blevet næsten komplet omskrevet[23]. Den første version blev tilgængelig i januar 2008 og har siden udviklet sig fra version 0.9 til 1.9.0.

Sammenlignet med andre mocking frameworks[24] er Mockito på mindst samme niveau som easyMock, men mangler features i forhold til JMockit. Man kan eksempelvis ikke mocke statiske metoder eller final klasser. Features som vi ikke har haft brug for i vores forløb.

For at illustrere nogle af vores argumenter i diskussionen om forskellige mocking frameworks giver vi her eksempler på dele af vores test kode.
Det første eksempel viser den lighed, der var mellem alle tre værktøjer. Figur 18 viser opsætningen af mock-objekter, som det ser ud i jMock. Alle mock-objekter skal erklæres som final. Herefter laver man en reference til de interfaces, som skal mockes.

22 ”The JMockit approach is an alternative to the conventional use of "mock objects" as provided by tools such as EasyMock and jMock.” [20]
23 ”Initial hacks on Mockito were done on top of the EasyMock code but since then I rewrote most of the codebase.” [21]
24 Sammenligningen kan ses her; [22]

```
final Session mockSess = context.mock(Session.class);
final Database mockDB= context.mock(Database.class);
final View mockView= context.mock(View.class);
final ViewNavigator mockNav = context.mock(ViewNavigator.class);
final ViewEntry mockViewEnty = context.mock(ViewEntry.class);
```

*Figur 18: Opsætning af mock-objekter med jMock.*

Det er næsten identisk til den opsætning, som er på figur 19. Her er det erklæringen af mock-objekter i Mockito. Det er ikke nødvendigt at bruge nøgleordet final her, men bortset fra det er der ikke nogen forskelle at spore. Det ekstra mock-objekt, mockViewColumn, er med, fordi udformingen af resten af testen er anderledes opbygget **(1)**.

```
Session mockSession = mock(Session.class);
Database mockDB = mock(Database.class);
View mockView = mock(View.class);
ViewNavigator mockNav = mock(ViewNavigator.class);
ViewEntry mockViewEnty = mock(ViewEntry.class);
ViewColumn mockViewColumn = mock(ViewColumn.class); 1
```

*Figur 19: Opsætning af mock-objekter Mockito.*

Den næste del er til gengæld væsentlig anderledes. På figur 20 specificeres forventningerne (expectations) til testen. Den første linje i kodeblokken angiver, at mock-objektet *mockSess* skal retunere værdien *serverName*, når metoden *getServerName* bliver kaldt på objektet **(1)**. *OneOf* erklærer at handlingen kun skal ske én gang. Det er basalt set syntaksen der bruges, når man definerer forventningerne.

```
context.checking(new Expectations(){{

oneOf(mockSess).getServerName(); will(returnValue(serverName));  1
oneOf(mockSess).getDatabase(serverName,dbName);
will(returnValue(mockDB));
oneOf(mockDB).isOpen(); will(returnValue(true));
oneOf(mockView).remove(); will(throwException(new NotesException()));
oneOf(mockDB).createView("SemaphorSA_ListFormsWithNumbersOfDocuments",
""); will(returnValue(mockView));
oneOf(mockView).removeColumn(1);
oneOf(mockView).createColumn(1, "Form", "Form");
atLeast(1).of(mockDB).getView(with(any(String.class)));
will(returnValue(mockView));
oneOf(mockView).recycle();
oneOf(mockView).createViewNavMaxLevel(1); will(returnValue(mockNav));
atLeast(1).of(mockView).refresh();
atLeast(1).of(mockViewEnty).recycle();

//1.ViewEntry
oneOf(mockNav).getFirst(); will(returnValue(mockViewEnty));
atMost(4).of(mockViewEnty).isCategory(); will(returnValue(true));
oneOf(mockViewEnty).getColumnValues(); will(returnValue(collum1));
oneOf(mockViewEnty).getDescendantCount();
will(returnValue(colums.get(listOfKeys.get(0))));  2

//2.ViewEntry
atMost(3).of(mockNav).getNext(); will(returnValue(mockViewEnty));
oneOf(mockViewEnty).getColumnValues(); will(returnValue(collum2));
oneOf(mockViewEnty).getDescendantCount();
will(returnValue(colums.get(listOfKeys.get(1))));  3
```

*Figur 20: Testkode med jMock. Bemærk her at der på nogle af linjerne er flere metodekald samlet. Dette gør vi, fordi de to metodekald hører sammen, og det øger derved læsbarheden af koden.*

Som man kan se på jMock eksemplet, kan det hurtigt kræve meget kode at lave opsætningen. Så på hvilke punkter adskiller Mockito sig fra jMock? Først og fremmest kan man definere flere forventninger som parametrer. De sidste to linjer i *1.ViewEntry* angiver en returværdi når metoden *oneOf(MockViewEnty).getDescedantCount()* kaldes **(2)**. Den samme linje bliver gentaget i *2.ViewEntry* men med en anden returværdi **(3)**.
I Mockito kan begge returværdier defineres på én gang ved at tage dem med som parametrer. På figur 21 kan man se, hvordan det ser ud i tredjesidste og næstsidste linje af koden **(1)**.

```
when(mockSession.getServerName()).thenReturn(serverName);
when(mockSession.getDatabase(serverName, dbName)).thenReturn(mockDB);
when(mockDB.isOpen()).thenReturn(true);
when(mockDB.createView(viewName, "")).thenReturn(mockView);
when(mockDB.getView(viewName)).thenReturn(mockView);
doNothing().doThrow(npe).doNothing().when(mockView).remove();
when(mockView.createColumn(1, "Form", "Form")).thenReturn(mockViewColumn);
when(mockView.createViewNav()).thenReturn(mockNav);
when(mockNav.getFirst()).thenReturn(mockViewEnty);
when(mockViewEnty.isCategory()).thenReturn(false);
when(mockViewEnty.getColumnValues()).thenReturn(collum1, collum2);
when(mockViewEnty.getDescendantCount()).thenReturn(
colums.get(listOfKeys.get(0)), colums.get(listOfKeys.get(1)));
when(mockNav.getNext()).thenReturn(mockViewEnty).thenReturn(null);
```

2

1

*Figur 21: Mockito. Opsætning af testen. Bemærk forskellen i Lines of code mellem jMock og Mockito.*

Et andet sted hvor denne fremgangsmåde bliver brugt, er i den linje som starter med *doNothing* **(2)**. Her er der ikke mindre end tre forskellige returværdier, når metoden *mockView.remove()* bliver kaldt. Første gang metode kaldes skal der ikke ske noget (doNothing). Anden gang skal der kastes en exception (*npe* refererer til en exception der er erklæret længere oppe i testen). Tredje gang skal der igen ikke gøres noget. En meget elegant måde at lave forskellig opførsel på.

Udover de funktionelle forskelle, så betød det også noget for os, at der var god dokumentation af værktøjet. Eftersom ingen af os havde arbejdet så koncentreret med et mocking framework før, havde det stor betydning, at vi kunne få de nødvendige informationer nemt og hurtigt. Specielt med open source software er det ofte problematisk at finde god dokumentation. I en artikel fra januar 2005[25] argumenterer Michelle Levesque og Jason Montojo for at open software kan blive mere tilgængelig, hvis der bruges mere energi i fællesskabet på at skabe brugbar dokumentation. Men eftersom det er frivilligt arbejde at udvikle open source software, så kræver det også en aktiv opbakning til projektet. Derfor var vi også interesseret i at finde ud af hvad vores mocking projekters status var. Der er flere måder at få information om de enkelte projekters status. Den mest direkte måde er at kigge på projektets hjemmeside, men det kan være svært at finde den relevante information, man leder efter. En anden måde er at bruge en tjeneste som Ohloh.

Ohloh[26] er en webside som er dedikeret til at overvåge udviklingen hos open source projekter. Ved at følge aktiviteten bag et projekt kan man få en ide om, hvor stor opbakning der er bag projektet. Derudover har brugerne af websitet mulighed for at markere et projekt, som de bruger. Ved at kombinere de statistiker som bliver produceret, får man et så objektivt billede af projektets fremtid og nutidige støtte som muligt.

25 Link til artiklen: [23]
26 Ohloh kan findes på følgende side; [24]

| | Mockito | jMock 2 | JMockit |
|---|---|---|---|
| **General** | | | |
| Ohloh Data Quality | ⚠ Updated 4 months ago | ⚠ Updated about 1 year ago | ✓ Updated about 21 hours ago |
| Homepage | code.google.com | www.jmock.org | code.google.com |
| Project License | MIT | jMock | MIT |
| Estimated Cost | $671,349.00 | $89,555.00 | $897,118.00 |
| **All Time Activity** | | | |
| Committers (All Time) View as graph | 22 developers | 2 developers | 3 developers |
| Commits (All Time) View as graph | 1,799 commits | 121 commits | 1,901 commits |
| Initial Commit | ✓ over 4 years ago | ✓ over 5 years ago | ✓ over 2 years ago |
| Most Recent Commit | ✓ 5 months ago | ✗ about 5 years ago | ✓ 5 days ago |
| **12 Month Activity** | | | |
| Committers (Past 12 Months) | ✓ 13 developers | ✗ 0 developers | ✗ 1 developer |
| Year-Over-Year Commits | ⚠ Decreasing | ✗ No Activity | ✓ Stable |
| **30 Day Activity** | | | |
| Committers (Past 30 Days) | 1 committer | No Data | 1 committer |
| Commits (Past 30 Days) | 2 commits | No Data | 34 commits |
| Files Modified | 3 files | No Data | 94 files |
| Lines Added | 12 lines | No Data | 2,609 lines |
| Lines Removed | 6 lines | No Data | 1,768 lines |
| **Code Analysis** | | | |
| Mostly Written In | Java | Java | Java |
| Comments | Average | Very Low | Very Low |
| Lines of Code View as graph | 50,525 lines | 5,844 lines | 67,486 lines |

*Figur 22: En sammenligning af de mocking værktøjer vi har brugt.*

*Kilde:[25]*

En sammenligning af de tre frameworks viser en større gruppe af udviklere på Mockito. Kode analysen viser også en langt højere grad af kommentarer i koden. Blandt Ohloh brugere er det muligt at give en bedømmelse af projektet udover at indikere, om man er bruger af softwaren. Igen har Mockito fået den største samlede score såvel som flest antal brugere. De største fremtidige udfordringer for Mockito er en faldende aktivitet af ændringer og udvidelser til projektet.

## *Sammenfatning*

De test værktøjer som vi har gennemgået i dette kapitel, er på mange måder anderledes end de andre værktøjer i vores toolbox. For det første er de ikke integreret i RTC, som andre af de værktøjer vi

har brugt. Der er ingen dedikeret test frameworks indbygget i RTC. Her bruger IBM en af deres andre produkter Rational Quality Manager[27], som er deres miljø til at skabe test og sikre kvalitet. Derfor har vores valg af test værktøjer ikke været et fravalg af RTC komponenter, men i stedet et tilvalg af frameworks, som vi mener, er vigtige for udviklere på et projekt.
Fordi der har været tale om tilvalg, er det også værd at bemærke, at RTC ikke er nødvendig for at bruge vores test setup. Man kan således bruge den samme opsætning med andre konfigurations frameworks. Man kan endda undlade at bruge de selvsamme værktøjer, som vi har benyttet. Men som vi har skitseret i dette kapitel, er vores setup præget af tests, som på en hurtig og nem måde giver værdi til projektet. Vi vil endda mene, at det er vanskeligt at kvalitetssikre ens kildekode på mindre projekter uden at bruge et lignende sæt værktøjer som minimum.

Det er vigtigt at nævne at der findes mange alternativer til de værktøjer, vi har beskrevet. I løbet af vores projekt har vi selv stiftet bekendtskab med et par stykker af dem og måtte igennem procesen med at udvælge dem, som gjorde vores udvikling hurtigere og bedre. Man bør naturligvis bruge de værktøjer som understøtter ens udvikling bedst muligt og derfor også være åben for alternativer. Når det kommer til open source produkter er det særlig vigtigt. Netop i open source fællesskaber er den forsatte udvikling afhængig af personer med en passion for produktet. Det betyder, at nogen værktøjer modtager konstant opbakning, mens andre blomstrer op og forsvinder, og atter andre forbliver ukendte. Derfor kan de frameworks, man bruger i dag, være inkompatible med fremtidige projekter.
Når det er sagt så har vi, når det har været muligt, valgt test-værktøjer, som udover deres funtionalitet og brugervenlighed, stadig bliver opdateret. Hermed håber vi, at denne del af vores toolbox, som har gavnet vores projekt, også vil kunne give værdi til fremtidige projekter.

Uden brug af automatiseret unit test havde vi ikke kunnet implementere test-driven development i vores udvikling. En fremgangsmåde som kom til at betyde, at vi hele tiden følte os sikre på stabiliteten og kvaliteten af vores kode. Ved at have vores test som et sikkerhedsnet blev det væsentligt nemmere at refaktorere, og det forbedrede i høj grad kodens niveau. Det kunne vi ikke have gjort uden at bruge unit test. Vi ville også nødigt have været foruden Code coverage. Udover at sørge for at vores test dækning levede op til kravene i vores team contrakt, så forbedrede det også vores test. Ved at se på det grafiske feedback som Cobertura gav, kunne vi nemt se, om vores test kode var tilstrækkelig grundig. På den måde var det enkelt at overskue vores tests kvalitet.
Udvikling uden brug af mock-objekter ville have forsinket vores udvikling meget, og øget usikkerheden idet meget af vores skrevne kode benytter sig af eksterne biblioteker, som vi ikke vil teste. Derudover kunne vi konstatere, at det ville blive umådelig vanskeligt hvis en projektgruppe skulle arbejde og teste på forskellige dele af det samme system uden at bruge et mocking framework.

Et framework til testning skal gøre en i stand til at skrive testen hurtigere end den reelle kode. Vi vil klart anbefale Mockito som mocking-framework, over alternativerne. Mockito er det eneste af dem vi har testet, der er hurtigt nok at arbejde med. Det er fleksibelt, og udviklerne har været meget opmærksomme på at man ikke skal skrive unødvendig meget kode. Mocking er et vigtigt værktøj til separation af ens kode mod eksterne biblioteker, og det ville have været praktisk taget umuligt at teste lige så store dele af vores kode uden et sådan værktøj.

27 Mere info om Rational Quality Manager findes her; [26]

# Projektstyring

Ethvert projekt er en investering. Når en organisation påbegynder et projekt, dedikerer de tid, penge og ressourcer for at gennemføre projektet. Forventningen er naturligvis, at investeringen vil give en øget værdi tilbage til organisationen. Nu er projekter af natur komplicerede størrelser. Der er mange variabler, som skal justeres løbende for at sikre projektets succes. Denne håndtering er projektstyring. Begrebet projektstyring dækker over en bred vifte af områder, som influerer det endelige resultat af projektet. Det er alt fra sammensætning af projekthold til udviklingen af et budget.

I projekter har de involverede forskellige roller og forskellige arbejdsopgaver, og de derfor ofte brug for forskellige typer information. En projektleder skal eksempelvis monitorere projektet og sørge for, at det overholder de rammer, man har aftalt. Sagt med andre ord så skal tid, ressourcer og mål balanceres. For at gøre det nemmere for en projektleder at udføre sit job, har man efterhånden udviklet en masse værktøjer. Værktøjer som passer til projektledelse og bidrager til, at projektlederen kan leve op til sin rolle. Disse værktøjer er ikke de samme, som en udvikler skal bruge. Her er det en anden rolle med andre arbejdsopgaver og derfor også andre værktøjer. Men selvom rollerne og dermed også arbejdet er forskelligt, så er der stadig behov for at bevæge sig mod det samme mål. Så hvordan kan man samarbejde på tværs af roller og hvordan understøttes det i RTC?

Vores fokus er på udviklingsmetode, projektroller og vidensdeling. Disse emner er særlige relevante for værktøjerne i vores opsætning og på områder, som har bidraget til vores udviklingsprojekt. Derfor har vi først en beskrivelse af vores centrale værktøj, Rational Team Concert (RTC), og de elementer af det, som er relevante i forbindelse med projektstyring.
Efter det kommer en diskussion af udviklingsmetoder, og hvordan de fungerer sammen med RTC. Her bruger vi vores proces til at vise koblingen mellem værktøj og proces.
Dernæst har vi en gennemgang af roller på et projekthold samt deres sammenspil med RTC og tilsidst et afsnit om vidensdeling og samarbejde som indeholder erfaringer fra vores projekt. Herunder relevante værktøjer som understøtter samarbejdsprocessen.

## *Rational Team Concert*

Inden vi begynder at se på, hvordan RTC kan understøtte projektstyring, vil vi give en teknisk gennemgang af de funktionaliteter, som relaterer sig hertil. Vi kigger på RTCs opbygning, to interfaces man kan bruge til at arbejde med det, og work items.

Rational Team Concert blev udgivet første gang i juni 2008. Den er nu på version 3, med version 4 kommende i løbet af 2012.

RTC er en del af IBMs Jazz platform, som omfatter hele application lifecycle management, fra krav i Rational Requirements Composer (RRC), over udvikling i Rational Team Concert (RTC), til kvalitetssikring i Rational Quality Manager (RQM).

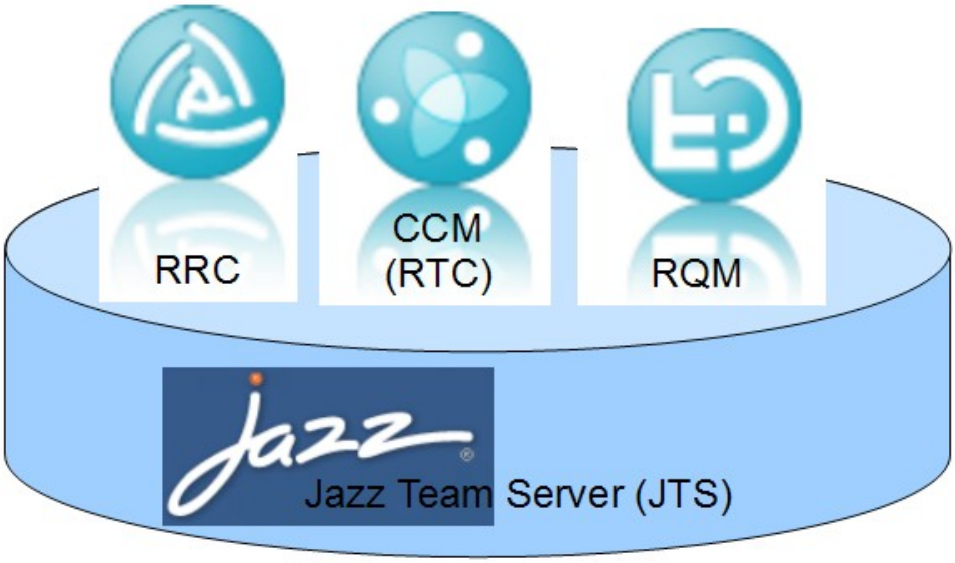
Derudover har IBM en lang række andre programmer i og omkring Jazz, såsom Rational Software Architect, der er en pakke med en lang række udvidelser og tilføjelser, som er bygget oven på Eclipse.

RTCs fokus er som skrevet selve udviklingsforløbet af et produkt.

## RTCs opbygning

RTC består af to applikationer; Jazz Team Server (JTS), og Change and Configuration Management (CCM).

JTS er mediator applikationen mellem alle Jazz produkterne, Rational Requirements Composer, Rational Team Concert og Rational Quality Manager. Dens primære ansvar er at holde styr på brugerhåndtering, projekthåndtering på et overordnet plan, og være service gateway mellem de forskellige Jazz produkter.



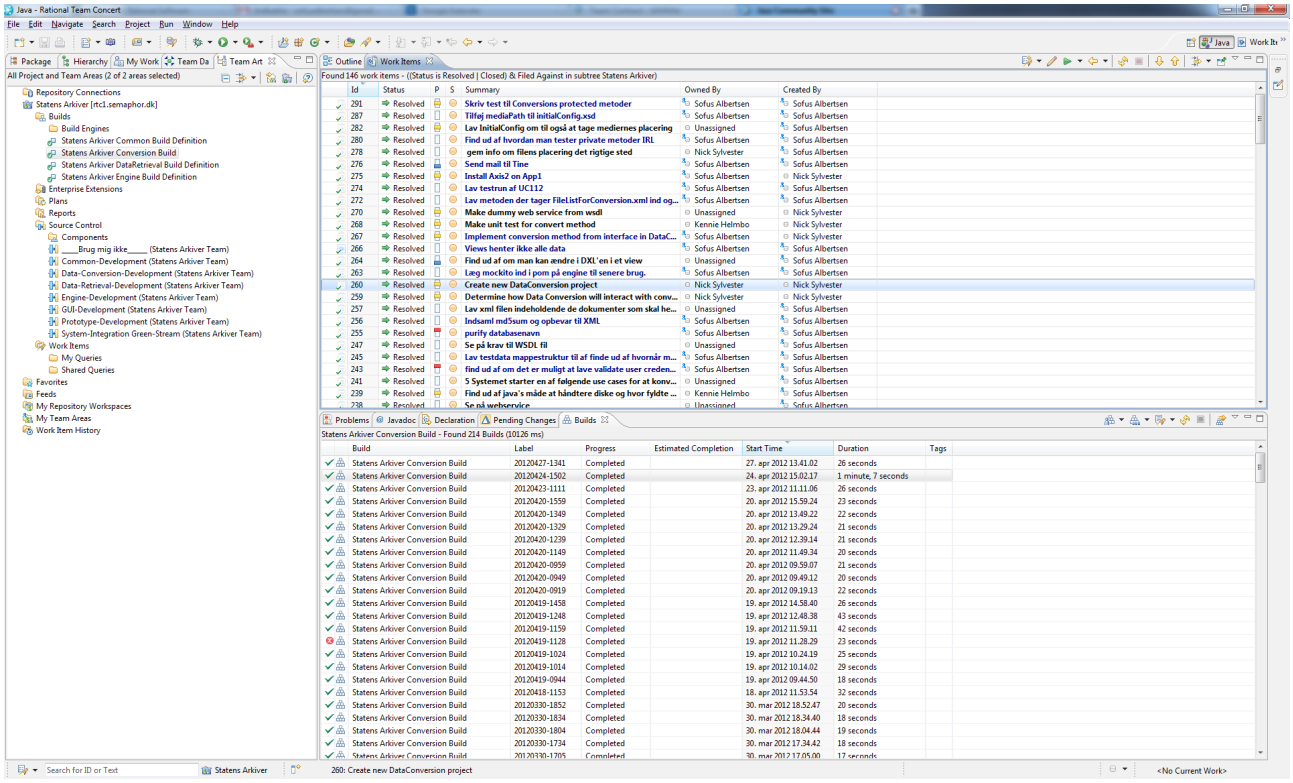

*Figur 23: Opbygningen af Jazz foundation. Øverst Rational Requirements Composer, Change and Configuration Management (RTC's hovedkomponent), og Rational Quality Manager. Disse tre bygger alle sammen på, og benytter sig af, Jazz Team Server for at kunne kommunikere med hinanden.*

CCM har i grove træk alle de funktioner som gør RTC til RTC. Den håndterer work items, kodestyring og planlægning m.m. Det er derfor den del af RTC, som man arbejder med i forbindelse med udvikling og planlægning.

Begge applikationer har hver deres backend med hver deres databaser og er på data niveau helt frakoblet hinanden, så de kan deployes til hver sin serverinstans.

## Interface; Web vs. IDE plugin

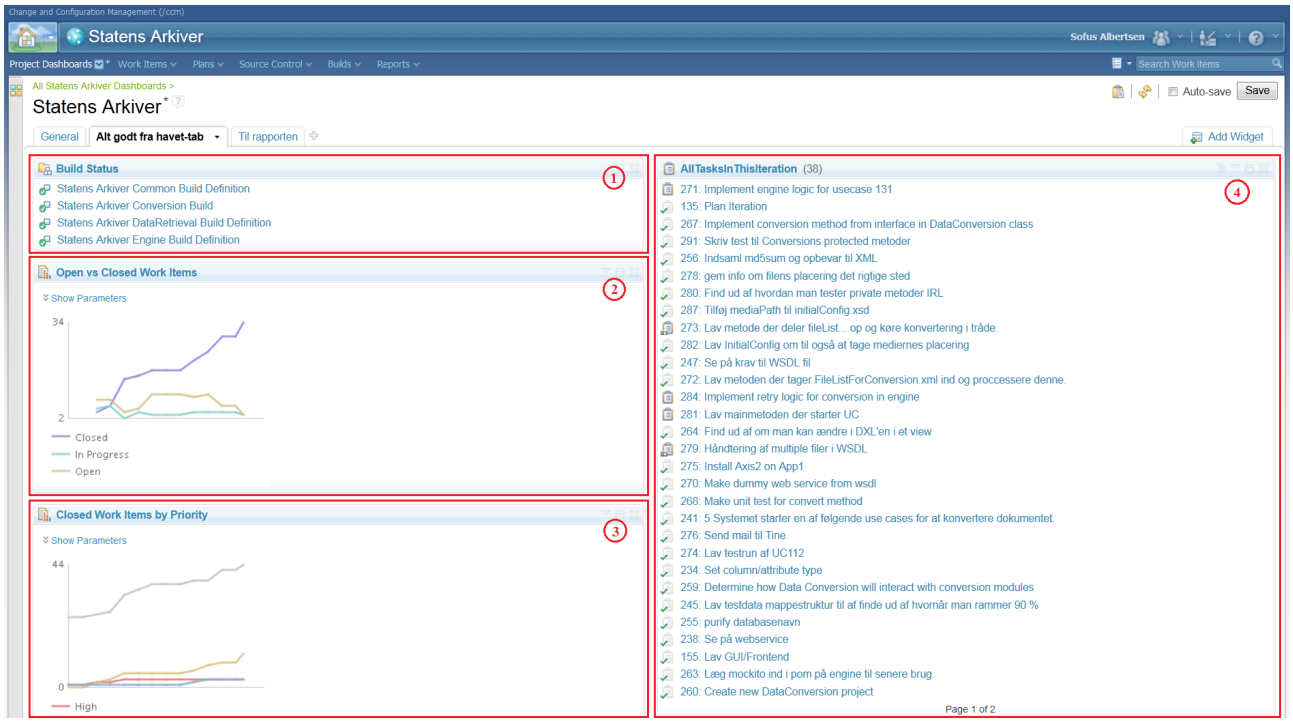
RTC kan tilgås på to måder; igennem et IDE, og gennem et webinterface[28].



*Figur 24: Eksempel på Eclipse interfacet af RTC.*

På figur 24 ses hvordan RTC ser ud i Eclipse. Venstre fane viser vores projekt, med underfaner til blandt andet builds, source control og work items. Øverste faneblad viser resultatet af en forespørgsel på de work items, der har status af "Resolved". Nederst er en oversigt over de seneste builds, der er foretager med Conversion modulets build definition.

28 Der findes også et kommandolinje interface, men det bliver mest brugt til mainframe udviklerne.




*Figur 25: Figur 3:Eksempel på et project dashboard i RTC webinterface.*

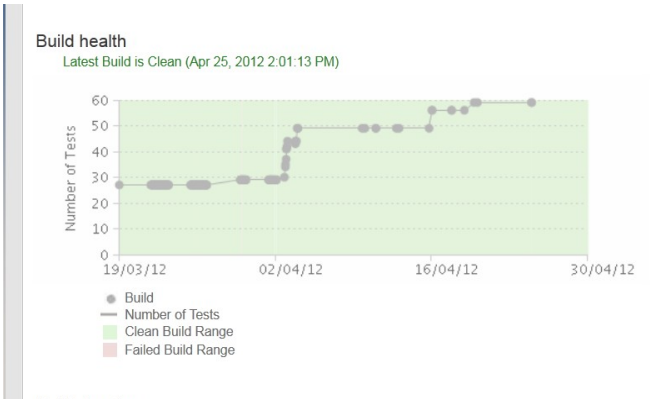
På figur 25 ses et eksempel af, hvordan man kan tilrette RTCs web interface til at skabe overblik over sit projekt. Her ser vi:

1. Status på hver enkelt build definitions seneste build.
2. En graf der viser hvor mange workitems vi har, fordelt på status.
3. Antallet af work items, der er ”Resolved”, fordelt på deres angivne prioritet.
4. En liste af alle de tasks der er blevet angivet til at tilhøre den nuværende iteration, samt et ikon der indikerer status på hver enkelt.

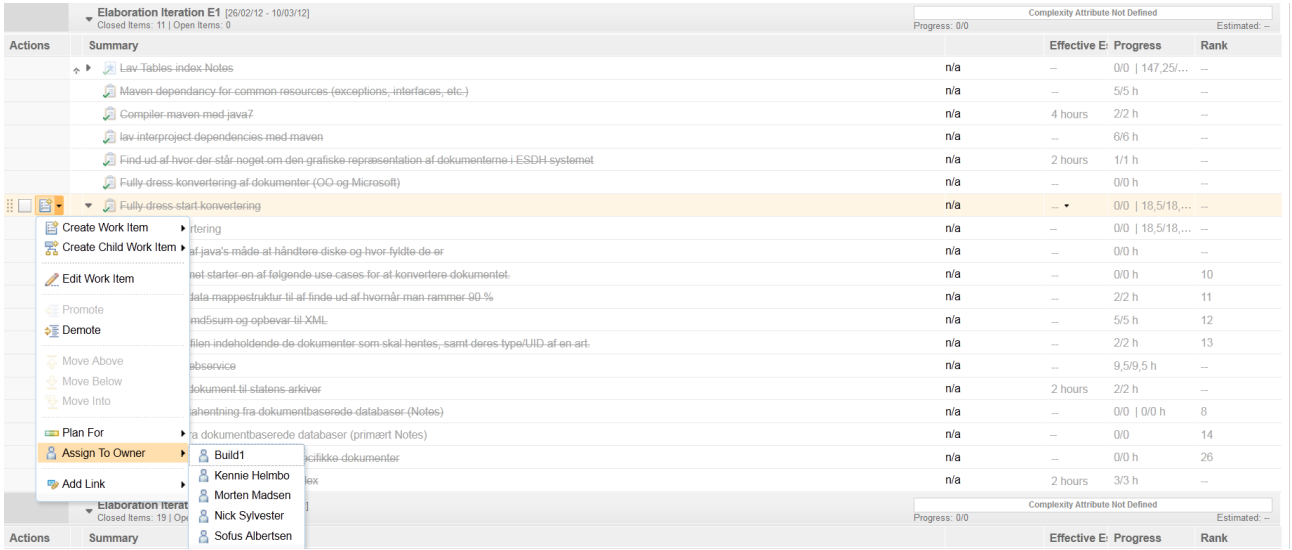
Kernefunktionaliteterne i de to måder at interagere med RTC på, er meget ens. Det interface som er mest optimalt at bruge, afhænger af ens rolle. IDEet er primært tiltænkt folk, der arbejder med udviklingen såsom udviklere, testere og lignende. Funktionaliteterne her går på selve kodedeling/versionering, build definitioner og arbejdet med work items. Desuden er der værktøjer, der gør det nemt at tage et screenshot og gemme det som en del af et eksisterende work item eller et nyt. Det er specielt smart i situationer, hvor det kan være svært at beskrive problemet, eller man ønsker at vise en fejl, som kun opstår periodisk. Med værktøjet kan man hurtigt lave en fejlrapportering og så gå videre med sin initiale opgave uden at miste for meget fokus.

Web interfacet er anderledes bygget op, og giver i højere grad et overblik over hele projektet. Med projektholdets dashboard kan man have forskellige rapporter, der måler relevante metrikker. Det kan for eksempelvis være en burndown chart, hvis man udviklede efter XP/Scrum metodikken, eller en graf der viser hvor mange opgaver hver enkelt medarbejder har oprettet eller løst indenfor en given periode.

Planlægningssiden giver også overblik over hvilke work items, der skal løses i hvilke iterationer, samt mulighed for at angive arbejdsbelastning og timer for hver enkelt medarbejder. Det er desuden også gennem web interfacet, at man kan få en overordnet status på servernes helbred.




*Figur 26: Eksempel på en detaljeret graf fra RTC, der viser antallet af tests gennem tiden.*

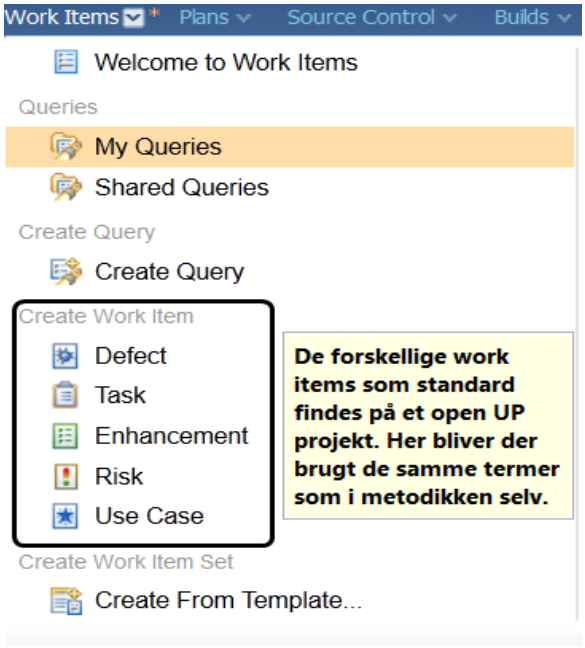



*Figur 27: Eksempel på en iterationsplan fra RTC. Øverst har vi en use case der er brudt op i en række work items. Dernæst en liste med tasks der skal laves i denne iteration. Der er markeret et work item, hvor man i kontekstmenuen kan tildele denne task til en given person, der er med i projektet. Man kan også lave sub-tasks, hvis man vil lave en opsplitning af den enkelte task.*

Det er ikke tænkt sådan at en udvikler dermed aldrig kommer til at bruge webinterfacet, men vedkommende vil i det daglige bruge sit IDE, som alligevel står klar. Det samme gælder for projektlederen, der kan have behov for at tilgå nogle funktionaliteter som IDEet stiller til rådighed.

## Work items

Måden, man koordinerer udviklingen på, er gennem work items. Et work item er en type arbejdsopgave. Hvis man for eksempel arbejder på et SCRUM projekt, så kan et work item være en user story. Arbejder man derimod på et UP projekt, så har man i stedet for et work item, som er en use case. En af de spændende aspekter ved work items er muligheden for at skabe relationer mellem dem. Det kan for eksempel være en use case, der bliver brudt ned i en række enkeltstående tasks. Dermed kan man skabe en Work Breakdown Structure, herefter WBS.
Et projekt består af flere faser. Antallet af faser afhænger af den udviklingsmetode man bruger. I OpenUP har man eksempelvis fire faser, som så igen består af iterationer. Alle faserne har leverancer. En leverancer kan for eksempel være et antal use cases, som skal være implementeret før man går til den næste fase. Når en del af forløbet afsluttes kaldes det en milestone. Det repræsenter en signifikant begivenhed i projektet.

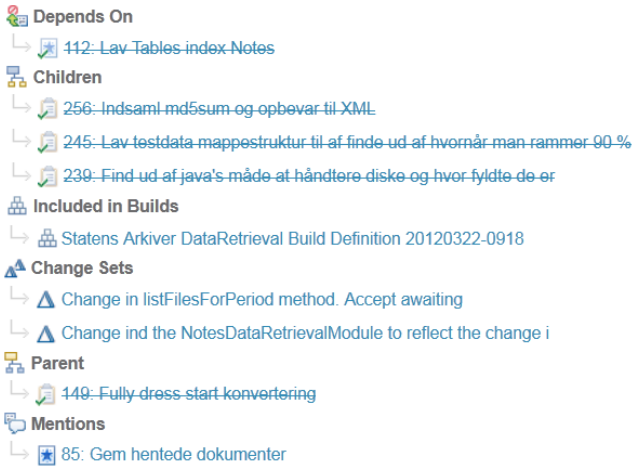



*Figur 28: Oprettelsen af et work item fra webklienten på et OpenUP projekt.*

```
 Projekt
|
`- Fase
  `- Leverance
  |  |
  |  |- Aktivitet/Opgave
  |  |
  |  |- Milestone/Leverance afslutning
  |
  `- Milestone – Fase afslutning
```

WBS er altså en måde til at opdele projektet i mindre komponenter og dermed strukturerer projektarbejdet. Den samme hierarkiske struktur kan man replikere med work items.

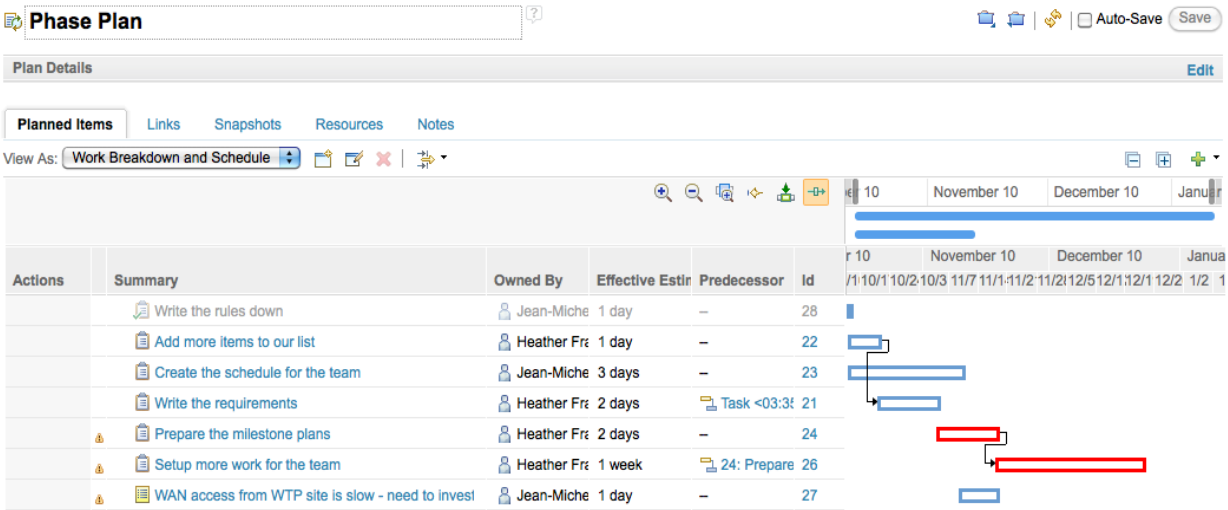
Hvert work item bliver automatisk tildelt et unikt id når det oprettes. En af fordelene er muligheden for at knytte relevant information sammen med det work item. På figur 29 kan man se hvilke ændringer, der er tilknyttet det work item, eller om der er andre opgaver, som refererer til det. Denne reference bliver til




*Figur 29: Eksempel på relationer mellem et givent work item og andre. Her er alle linjer med blå skrifttype links til andre work items, builds, eller change sets. Derved er der fuld transperans i hele systemet.*

direkte links, som gør det nemt at skabe forbindelser på tværs af arbejdsopgaver.

Nedbrydningen af arbejdsopgaver, og relationen mellem dem, indgår naturligt i moderne projektledelse. Selvom nedbrydningen ikke umiddelbart er svær at forstå, så kan overblikket hurtigt forsvinde, når mange opgaver skal planlægges. En af de mere udbredte måder at modarbejde dette på er ved at bruge en Gantt chart til at planlægge arbejdet.

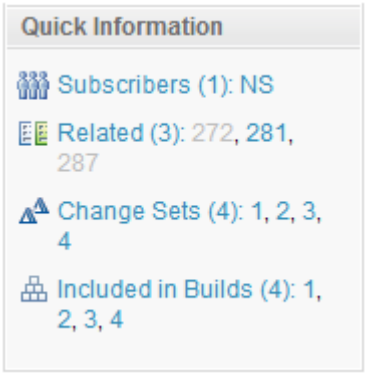



*Figur 30: Overblik over work items set fra RTCs web interface.*

Eksemplet ovenfor viser, hvordan et Gantt chart kan se ud i webklienten af RTC. I venstre side er der en liste over work items. De har alle sammen en person tilknyttet, som er ejer af opgaven. Derudover har de en estimeret tid for gennemførelsen, samt et id. Enkelte af opgaverne har en afhængighed til andre opgaver. Eksempelvis kan work item 26 først udføres, når work item 24 er klaret. I højre side er en tidsplan. Ud af x-aksen er en tidslinje og ned af y-aksen de forskellige aktiviteter og deres varighed.

Figur 31 viser information om et work item i vores projekt. Man kan se at der er 3 work items, som er relaterede til dette. Alle referencerne er links, som giver direkte adgang til de pågældende work items. Herudover har der været 4 change sets[29] med tilknytning til dette work item. Igen er der hurtig adgang til disse ændringer, således at det er nemt af få information om alle relaterede handlinger med dette work item.

Ved at sørge for at alle work items bliver knyttet til relaterede work items, og at alle kodeændringer bliver knyttet til disse, skaber man sporbarhed i projektet.




*Figur 31: Eksempel på de informationer der tilknyttes hvert enkelt work item.*

29 Et change set er en samling af kodeændringer, som man afleverer til sit code repository. Det kan sammenlignes med

## *Udviklingsmetode*

For at forstå hvorfor det er relevant at koble en udviklings livscyklus sammen med RTC, må vi starte med betydningen af en udviklingsmetodik. Så hvor vigtig er valget af udviklingsproces for succesen af ens projekt? Alle udviklingsprojekter har en udviklingsproces. Fra de meget fast definerede metoder til de meget fleksible metoder. Selv fravalget af en process er en måde at håndtere projektet på. I større projektgrupper er der fra planlægningen til leveringen af projektet behov for at styre projektet, så længe der er en arbejdsindsats, der skal koordineres.
Ifølge en rapport foretaget af Standish Group[30] så er valget af en formaliseret process blandt de 10 vigtigste kriterier, hvis et projekt skal lykkes[31]. I et interview med Jim Johnson og Gordon Divitt fra Standish Group bliver der sat fokus på, at en vigtig faktor i stigningen af succesfulde IT-projekter er skiftet fra de ældre modeller, som vandfalds modellen[32], til de inkrementelle udviklingsmetoder, som SCRUM.

Samtidig er udviklingsprocessen med til at skabe rammerne for projektforløbet, og dermed også planlægningen, selvom dette område kun udgør en lille del af projektstyring. Det gør det vigtigt at vælge en form, som passer til opgaven. I sidste ende er det projektets succes, der er målet, og processen er midlet til at opfylde det mål.

I vores projekt var vi ikke opmærksomme på den kobling i starten af forløbet. I stedet blev vi meget fokuserede på de redskaber vi ville bruge. Det betød, at vi blev nødt til at revurdere vores valg af udviklingsmetode og lave de fornødne ændringer. Set i bagklogskabens klare lys skulle vi naturligvis have været mere opmærksomme på koblingen mellem mål og middel.

## Valg af metode

I udviklingen af et software system, som kræver samarbejde mellem flere interessenter, er det nødvendigt at planlægge projektet. Derfor var en af vores første udfordringer at vælge en udviklingsmetode. Som nævnt i kapitlet ”Test”, benyttede vi test driven development. Derfor mente vi initialt, at det bedste valg ville være en agil udviklingsproces, som kunne understøtte den fleksibilitet, som vi gerne ville have. Da der var aspekter af både extreme programming og SCRUM som vi var interesserede i, besluttede vi at vælge de dele som understøttede vores udvikling. Eksempelvis ville vi bruge test first men fravalgte som udgangspunkt pair programming. Dog var der tidspunkter senere hen, hvor vi på skift dannede par for at diskutere dele af koden. Vi valgte altså de dele fra XP og SCRUM, som vi mente ville gavne vores udvikling. Mindre regelsæt og mere rationel tænkning var vores rationale.

Selvom vi havde en del overvejelser omkring valget af elementer i forhold til den udviklingsmetodik, som vi ville bruge, så havde vi ikke vores projekts karakteristika i tankerne. Som beskrevet tidligere så havde projektet med Statens Arkiver meget veldefinerede krav. Vi stod således i en situation, hvor vi havde en kunde, som vidste præcist, hvad vedkommende ønskede, og som ikke ville lave radikale ændringer undervejs.

commits i SVN. Mere herom i kapitlet ”Konfigurationsstyring”

30 [27] The Standish Group er en uafhængig gruppe som primært laver forskning og analyse af it-projekters resultater.

31 [28] Link til det samlede interview.

32 For en overordnet introduktion til vandfaldsmodellen, se [29]

Efter vi havde opdelt kravene i user stories og lavet estimeringer, skulle det besluttes hvilken en vi ville starte med. I en kundeorienteret process ville man som en naturlighed spørge kunden om deres ønske og dermed starte med den del af systemet, som havde mest betydning for pågældende. Men det var fuldstændig irrelevant i vores situation. Den konktrakt der ligger mellem en kommune og Statens Arkiver foreskriver, at enten så har man hele kravspecifikationen udmøntet i en fuldstændig valid arkivversion, eller også har man ikke. Dette gjorde vores aflevering til et ”alt eller intet projekt”. Vi overvejede om en meget kundeorienteret process virkelig var det bedste for vores projekt. Det, vi egentlig havde brug for, var en udviklingsmetode som fokuserede på risikodrevet udvikling. Vi havde at gøre med et ukendt og komplekst domæne, med ukendte teknologier, og vi var i starten af processen ikke engang klar over, om vi kunne levere hele løsningen rent teknisk. Vi havde derfor brug for at have inkrementelle faser hvor vi fokuserede på at nedbringe usikkerheden, og derved også risikoen for fiasko. Med den fremgangsmåde i tankerne var det oplagte valg at bruge en version af Unified Process. Så hvilken egenskaber gjorde udslaget? Selvom udviklingen ikke kunne være kundeorienteret så var den stadig iterativ, inkrementel, risikodrevet og arkitektur-centreret. Nu er der mange versioner af Unified Process, idet processen fungerer som en platform. Da vi stadig var interesseret i en metode, som var fleksibel og pragmatisk, valgte vi Open Unified Process, herefter OpenUP[33].

## OpenUP

OpenUP er et udviklings framework, som er udviklet af Eclipse Foundation. Målet er at beholde centrale elementer fra Rational Unified Process RUP og Unified Process UP, men samtidig gøre det så agilt som muligt. Dette vil man gøre ved at benytte en tilvalgsmodel i stedet for en fravalgsmodel. RUP er en komplet metodik, hvor du skal kende det hele for at vælge de dele, der passer dig (fravalg). OpenUP foreskriver derimod, at man ikke fjerner dele fra det, men sagtens kan tilføje andre elementer (tilvalg).

Blandt de elementer, som er bevaret fra RUP, er de fire faser i livscyklusen: Inception, Elaboration, Construction, Transition. Alle faser har så tilknyttet et antal artefakter.
Der lægges vægt på at OpenUP er til mindre udviklingshold, som arbejder sammen på den samme lokation, idet der kræves daglig face-to-face kommunikation. Holdet består af alle væsentlige aktører lige fra interessenter til systemarkitekter. Alle i teamet bestemmer selv, hvad de vil arbejde på, og hvordan de bedst kan løse interessenternes behov.
Selvom OpenUP gør meget for at give guidelines, checklister og milestones og i det hele taget hjælpe udviklere med at følge udviklingsprocessen. Samtidig opfordres der stadig til, at man tilpasser processen så, det kan leve op til det enkelte holds behov.

De tre karakteristika for OpenUP er minimal, komplet og udvidelig. Det er altså kun det grundlæggende indhold, der er med. Det er en proces at opbygge et system, og indholdet kan udvides eller skræddersyes efter behov. Kort sagt er OpenUP rettet mod udviklingshold, som ønsker at:

- Anvende den minimalt nødvendige process, der bringer værdi.

33 Link til Open UP frameworkets hjemmeside findes her; [30]

- Undgå at blive overbebyrdet med uproduktive[34] formelle arbejdsprodukter.
- Bruge en proces der kan tilpasses og udvides efter behov.

Netop disse udsagn var i god tråd med de behov og forventninger som vi havde, men på grund af ændringen af vores fokus fra udvikling til opsætning, blev mange af de forslåede elementer aldrig indarbejdet i vores arbejdsrutiner. Det var dog væsentligt at bibeholde de centrale begreber for at holde vores referenceramme konsistent med de begreber, som vi brugte  i RTC.

## Kobling mellem metode og RTC

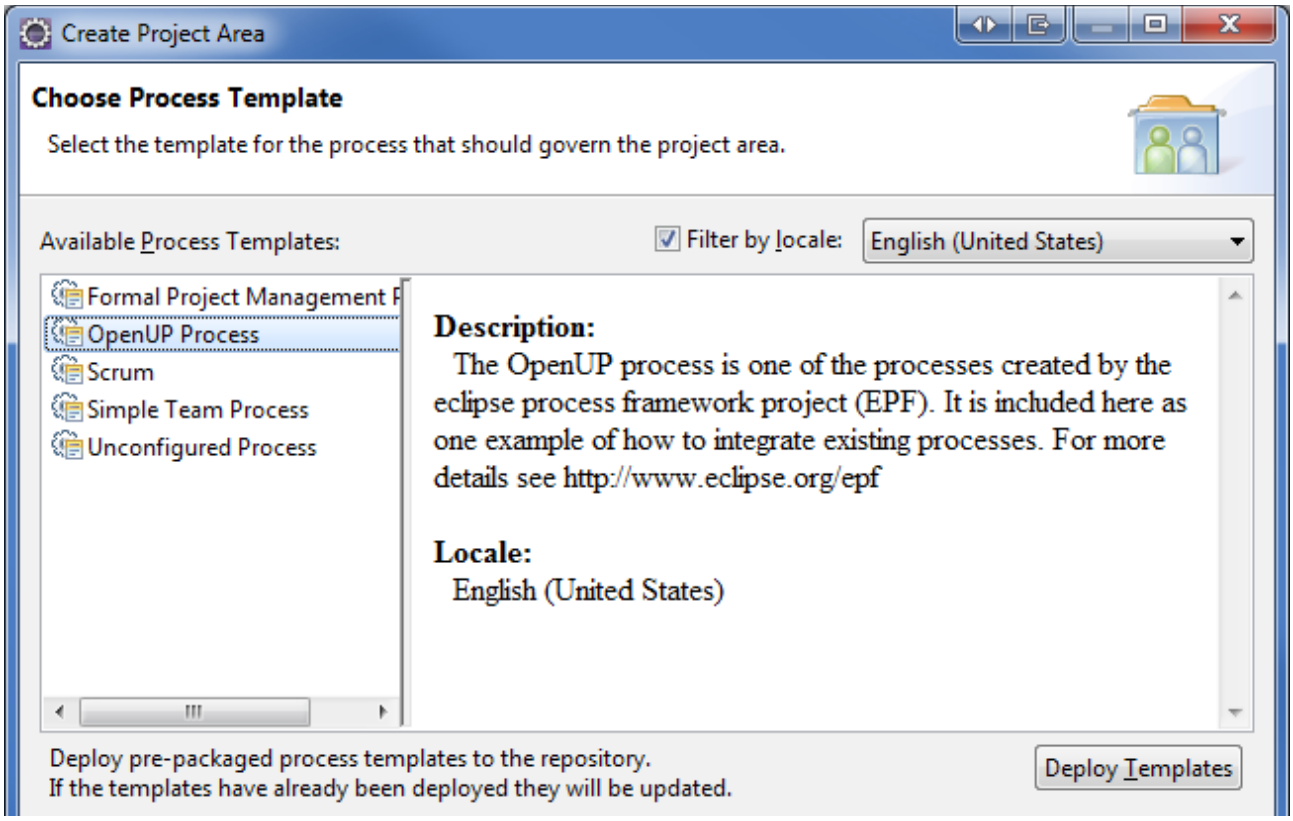
I RTC har planlægning af projekter været inkorporeret siden den første version. I starten var det primært rettet mod agile projekter og dermed målrettet mod at håndtere krav. Men der er over tid blevet tilføjet nye elementer, som nu også understøtter mere traditionel projektplanlægning.
Der er derfor i den nyeste version af planlægningskomponenten i RTC både support til agile og traditionelle projekter. For agile projekter kan man skabe sprints, backlogs og så videre for projektholdet, og individuelle planer for den enkelte udvikler. Work item relationer, Gantt Charts og deslige er understøttet i de traditionelle projekter i RTC. Udover opsætning af de mest almindelige udviklingsmetoder er der også mulighed for at redigere og skabe sin egen proces eller lave en hybrid mellem agile og traditionelle.

## Process Templates

Vi har nu nævnt user stories og use cases, som er udtryk, der stammer fra udviklingsmetoder. Vi vil nu se på, hvordan det er muligt at lave RTC projekter i overensstemmelse med lige netop den udviklingsmetode, man bruger, ved at bruge process templates.

En process template er en skabelon for en udviklingsprocess. Når man opretter et nyt projekt i RTC, har man muligheden for at tilpasse projektet. Valget af proces bestemmer, hvilke work items man starter med og i det hele taget strukturere opsætningen af projektet. På den måde kan man hurtigere komme i gang med oprettelse af planer og skabelsen af artefakter. Hvis man for eksempel bruger OpenUP som udviklingsmetode på sit projekt, kan man i RTC oprette et projekt ud fra en OpenUP proces template.

34 Uproduktive skal her forstås som at nytten ikke svarer til udgiften ved at udføre arbejdet.




*Figur 32: Trinet oprettelsen af et "Project Area" i RTC via Eclipse, hvor man vælger, hvilken proces man vil bruge.*

Nu er en effektiv proces ikke nødvendigvis lig en standard skabelon eller fast definerede artefakter. Mange organisationer, som søger at forbedre kvaliteten af deres software, vælger derfor også at tilpasse deres proces over tid. Det kunne eksempelvis være efter hvert sprint i et SCRUM projekt, hvor man skal vurdere sin metodikker. Et andet eksempel kunne være Alistair Cockburn's Just In Time metodologi[35], hvor man skræddersyr metodikken specielt til hvert enkelt projekt. Denne konstante transformation af udviklingsmetodik understøttes af RTC, idet alt kan konfiguereres på et hvilket som helst tidspunkt i projektet. Ethvert projekt, som bliver påbegyndt, giver erfaringer, hvad enten projektet er en succes eller fiasko. De erfaringer kan til sidst ende som best practices, og det er der taget højde for i RTC. Man kan tilrette en af de eksisterende skabeloner, oprette sin egen fra bunden eller importere skabeloner andre folk har lavet. På den måde kan man få lige præcis de work items, planer, m.m. som passer lige netop til ens projekt. Eftersom det ikke er noget, vi selv har arbejdet særligt meget med, kan vi ikke give en detaljeret beskrivelse af det. Dog synes vi det er nævneværdigt, så man ikke opgiver RTC, fordi den ikke har en skabelon, som passer til ens projekt.

35 For mere information om Just-in-time metodologien, se [31]

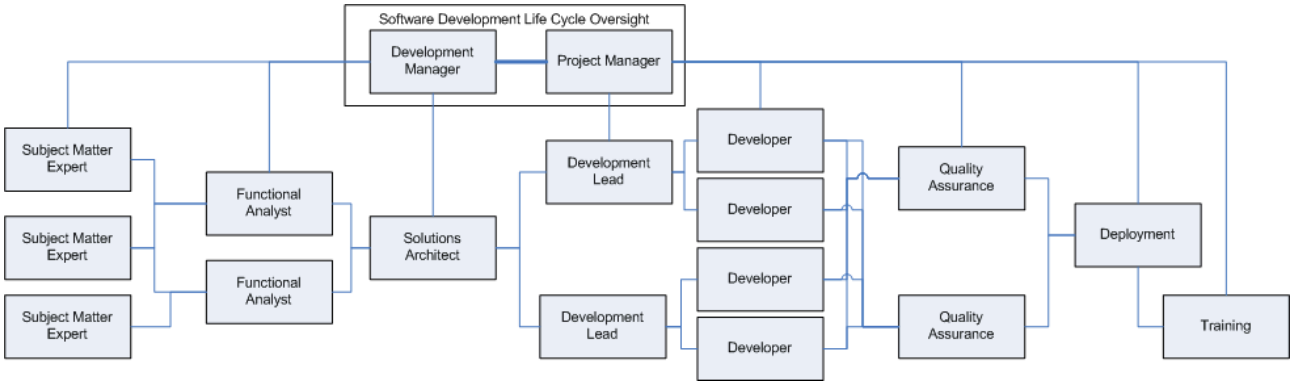
## *Roller i projektet*[36]

En væsentlig del af en udviklingsmetode er, hvordan man fordeler arbejdet. Der er en række ansvarsområder som skal håndteres igennem et projekts levetid. Vi vil her beskrive hvordan rollefordeling understøttes af RTC.

Ansvarsområder og rettigheder på et projekt defineres gennem roller i et team. Afhængig af projektets karakter kan én person godt udfylde flere roller. Disse roller og fordelingen af dem vil variere fra firma til firma og måske endda fra projekt til projekt. Det kan være at man på ét projekt har brug for specialister indenfor arkitektur og test, men på et andet projekt kan slå de to roller sammen til en enkelt udvikler-rolle. Uanset hvordan rollefordelingen tager sig ud i det enkelte projekt , så er der nogle aktiviteter, som altid er vigtige at tage højde for. For størstedelen af alle projekter skal man blandt andet:

- Forstå domænet (forretningsprocesser)
- Lave en arkitektur ud fra domænet
- Udvikle en løsning indeholdene arkitekturen
- Teste løsningen
- Implementere løsningen (deployment)

Dette er kort sagt nogle generelle aktiviteter, som de fleste projekter indeholder. Rækkefølgen, de står i, er ikke nødvendigvis den rækkefølge, de skal laves i. XP, som eksempel, benytter test driven development, hvilket betyder at ”Test løsningen” vil ske før ”Udvikle en løsning indeholdene arkitekturen”. Derudover kan man også slå flere af disse aktiviteter sammen til en enkelt rolle. Kigger man på XP igen, vil udviklingen af løsningen og de tilhørende test altid udføres af udviklerne. Først skriver udvikleren test til den del af løsningen, han skal gå i gang med, og derefter skriver han koden. To forskellige aktiviteter i den samme rolle. Følgende figur er et eksempel på, hvordan rollefordelingen kunne være på et projekt:



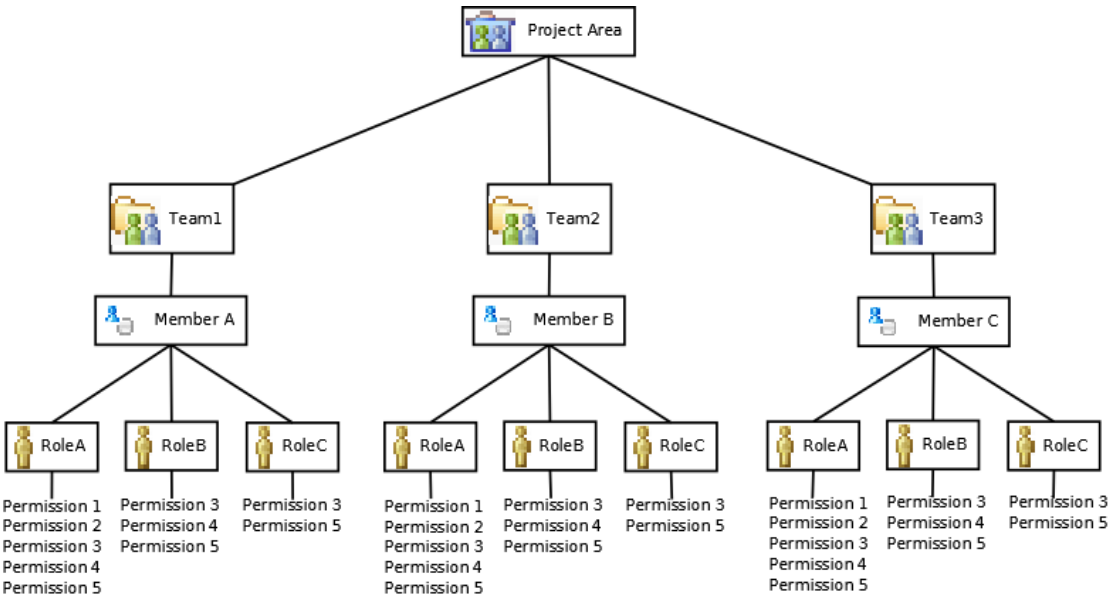

*Figur 33: Organisations diagram som viser, hvordan de forskellige roller relaterer sig til hinanden.*

Som tidligere nævnt er der ikke en endegyldig opdeling. Figuren kunne være en retvisende

36 Artiklen ”Cracking the Code: Breaking Down the Software Development Roles” er blevet brugt som baggrundsmateriale for dette afsnit. [32]

repræsentation af rollerfordelingen på et UP projekt. Derimod ville det før omtalte XP projekt have ”Functional Analyst”, ”Solutions Architect”, ”Developer” og ”Quality Assurance” (herunder test) slået sammen.

Når man har investeret tid og energi i at definere roller og ansvarsområder for et projekthold, er det naturligvis vigtigt, at det værktøj man bruger til at monitorere projektet også understøtter denne opdeling. Her kan RTC skræddersyes til at passe til sit projekt. Den overordnede opbygning af et projekt i RTC ses her:

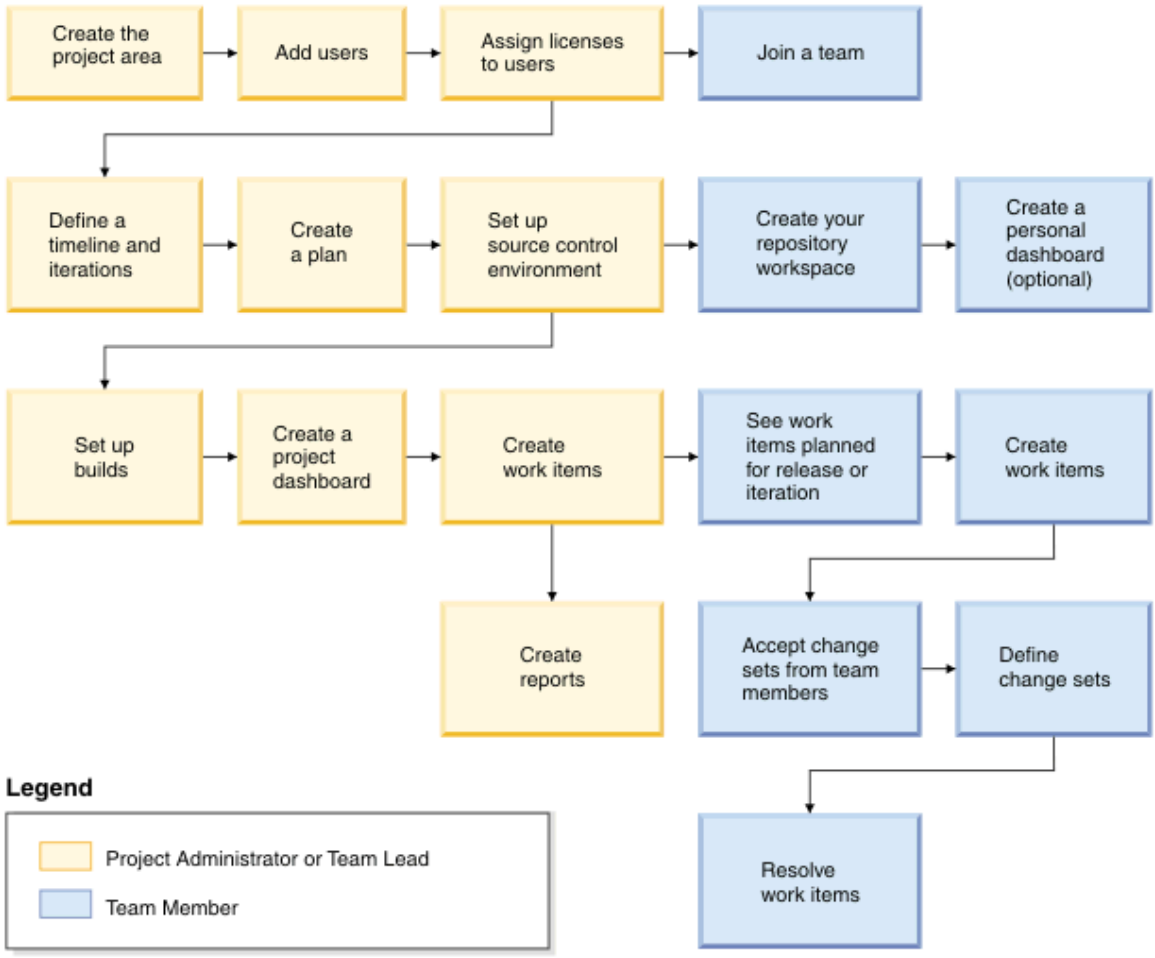



*Figur 34: Figur 3: Opbygningen af et RTC projekt.*

Først opretter man et projekt (Project Area) i RTC. Roller, artefakter, m.m. kommer til at høre herunder. Hertil kan man oprette projekthold (Team) til projektet. Det kunne for eksempel være 3 forskellige SCRUM teams. Et projekthold i RTC er altid koblet til ét betstemt projekt. Der findes altså ikke globale projekthold. Hvis man for eksempel har en test afdeling, som altid består af de samme mennesker, og som arbejder på flere projekter ad gangen, må man gå ind i hvert projekt og definere et projekthold til dem. Hvert projekthold har nogle medlemmer (Team Member). Figuren viser kun et medlem på hvert hold for overblikkets skyld, men der kan naturligvis være flere. Medlemmerne bliver oprettet globalt[37] og er derfor den samme entitet fra projekt til projekt. Hvert medlem, som skal arbejde på projektet, kan blive tilføjet til et projekthold i RTC. De får så tildelt de roller (Role) som er relevante for dem. Et medlem kan have flere roller. For eksempel kan nogle udviklere have roller som tester, udvikler og systemarkitekt, hvor andre på holdet kun har én af disse. Disse roller tildeles så rettigheder i RTC, hvilket bestemmer de aktiviteter en person har lov til at udføre. Rettighederne, man har, er summen af de rettigheder, ens roller tildeler en. Det vil sige at hvis man har flere roller, skal bare én af dem give rettigheder til en given aktivitet, for at man kan få lov til at udføre den. De rettigheder en role har, er helt konfigurerbare. Det vil sige, at to

37 Med globalt mener vi globalt i RTC sammenhæng, altså på tværs af projekter.

roller med samme navn kan give helt forskellige rettigheder på hver deres projekt. For eksempel kunne der være to project areas, som hver har en ”Member” role. Det kunne være sådan, at man på det ene projekt kunne få lov til at oprette work items med den role, men ikke på det andet. Følgende er et billede af, hvordan oprettelsen af et projekt kunne se ud:

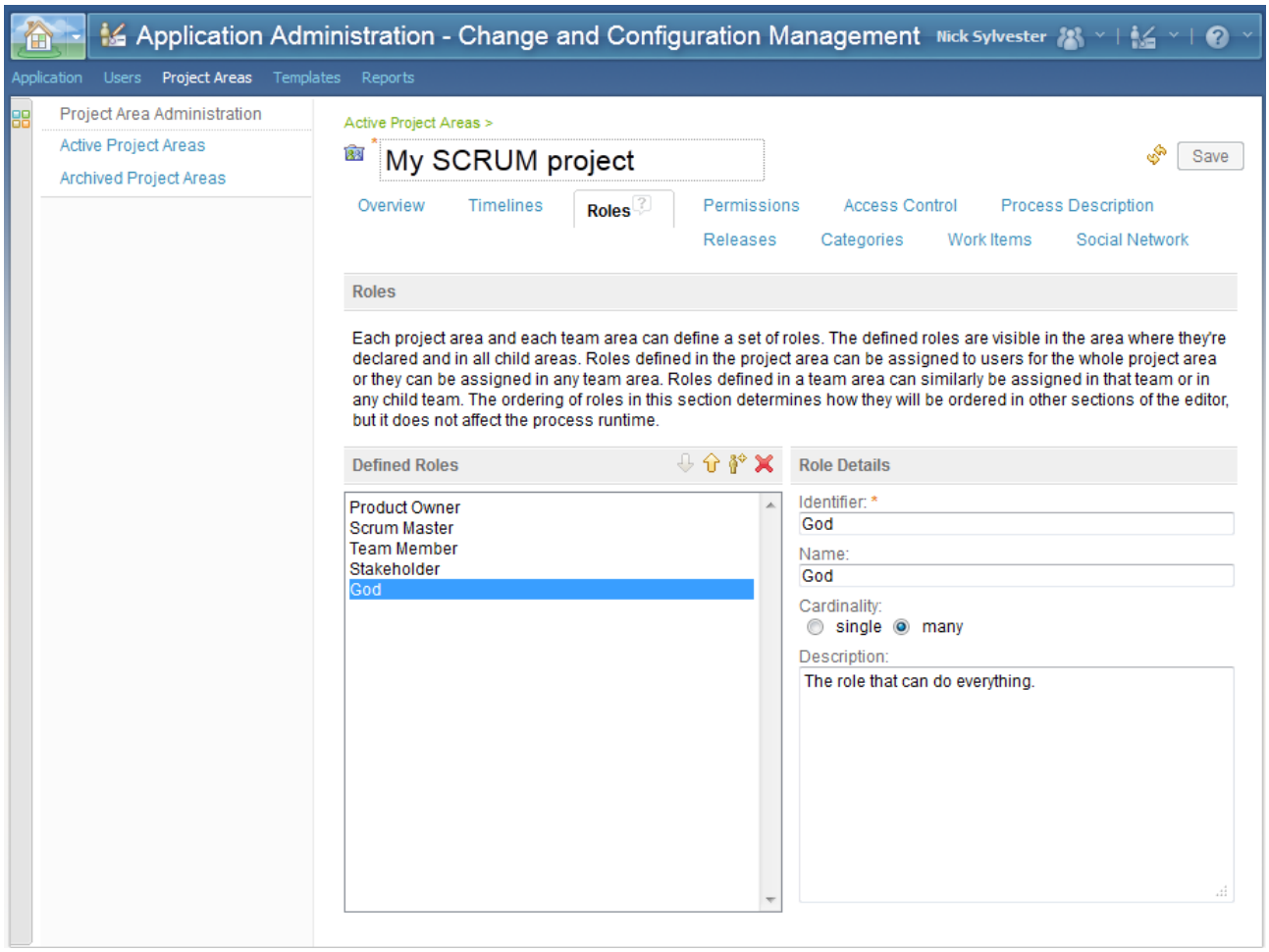



*Figur 35: RTC Workflow diagram.*
*Kilde: [33]*

De gule felter repræsenterer opgaver som en projektleder eller lignende foretager, mens de blå viser hvad en udvikler eller lignende laver. Projektlederopgaverne har meget med strukturen af selve projektet, for eksempel opgaver som at lave en plan over projektet, samt at indkalde folk til projektet. Udvikleren skal derimod koble sig mere mod produktion indenfor den ramme som projektlederen har lavet. For eksempel løser vedkommende opgaven ”Reslove work item” efter at vedkommende har hentet projektressourcerne ned på sin maskine. IBM skriver på figurens hjemmeside at skønt de fleste arbejdsopgaver kan løses i begge interfaces, så er webinterfacet

bedste løsning til arbejdet som projektleder, hvor Eclipse mere retter sig mod udvikleren[38].

Den konfigurationsfrihed man har i forbindelse med roller gør, at man i RTC kan ”duplikere” sit projekts organisering. Følgende billede viser, hvordan man kan redigere, oprette og slette roller:



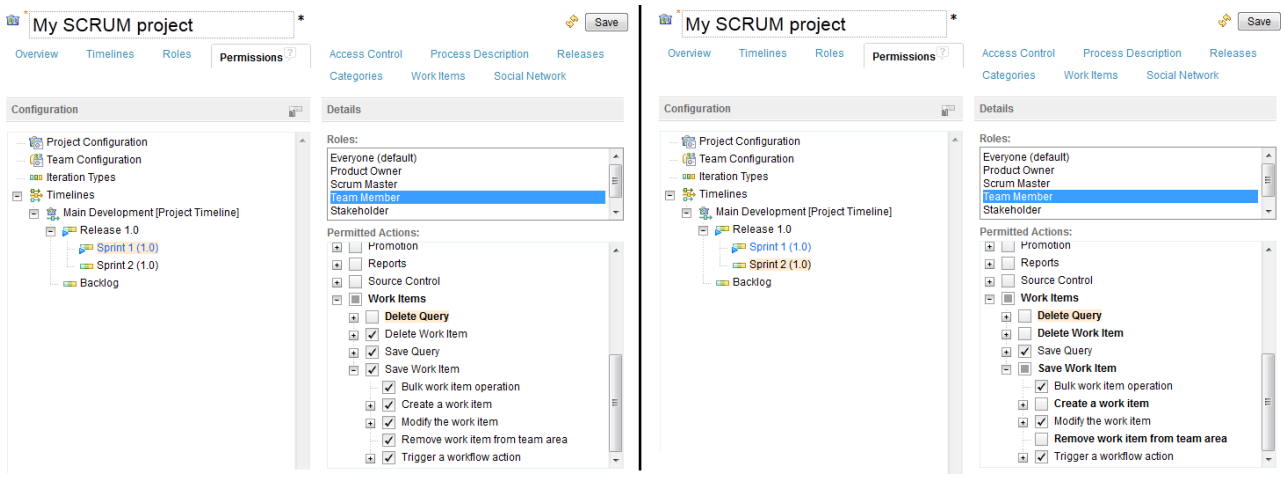

*Figur 36: Vindue til tilføjelse, fjernelse og redigering af roller. De 4 første er oprettet som default for et SCRUM projekt og den sidste er tilføjet efter.*

Det er især rart, når man bruger en metode som afviger fra standarden, at man kan tilpasse sit værktøj til at afspejle dette.

Til disse roller skal der selvfølgelig også høre nogle forskellige rettigheder. Alle skal ikke kunne det samme, hverken i organisationen eller værktøjet. Ved at give de roller, som man har defineret i RTC, rettigheder som passer til deres funktioner, får man mulighed for at kontrollere adgangen til information og konfigurationsmuligheder. Udover at opstille nogle overordnede rettigheder for folk

38 ”Although many of the steps can be done using either the web interface or the Eclipse client, the steps for project administrators and team leads link to topics that refer to the web interface and the steps for team members link to topics that refer to the Eclipse client.” [33]

har RTC også den smarte funktionalitet, at man kan styre rettigheder over tid. Hvis en rolle skal ændre karakter i løbet af projektet, har man mulighed for at specificere hvordan en rolle skal se ud på et givent tidspunkt. For eksempel kan man forestille sig, at man i starten har alle udviklere til at have de fleste rettigheder på projektet. Men efter et par iterationer skal nogle af disse udviklere ikke længere lave tasks, de skal kun fikse bugs. Denne beslutning kan man så få RTC til at overholde ved at lave en ændring i udviklernes rolle.



*Figur 37: Defineringen af rollen "Team Member" for to forskellige sprints.*

På billedet ser vi hvordan man kan ændre de rettigheder, som ”Team Member” rollen har for to forskellige sprints. For eksemplets skyld forestiller vi os en iteration på 2 sprints.
På venstre side bliver rettighederne sat for ”Sprint 1”.  I det sprint skal alt arbejdet, som skal foregå, identificeres og udspecificers. Her vil vi gerne have, at alle kan bidrage til den aktivitet, så den rolle får alle rettigheder vedrørende håndtering af work items.
På højre side bliver rettighederne sat for ”Sprint 2”. Her er det meningen, at alle opgaver har tilsvarende work items. Nu vil vi gerne begrænse adgangen til work items og gør det derfor kun muligt at redigere dem, som eksisterer i forvejen. Nu er det udelukkende folk med disse roller som kan ændre status, tilføje kommentarer, m.m., og ikke længere oprette eller slette work items.

Projektstyring er et af RTC's kerneområder, og folkene bag har gjort meget ud af at lave de medgivne templates, så alt fungerer ud af boksen, som man forventer. Samtidig har man mulighed for at konfigurere alting ned til mindste detalje, så det kommer til at passe til lige netop ens egen udviklingsmetodik.

## *Vidensdeling og kollaboration*

Et af de punkter hvor vi mener RTC falder igennem er, når man skal kommunikere i realtid og dele større samlinger af viden. Vi vil her beskrive de krav vi havde, som RTC ikke kunne imødekomme, og hvordan vi håndterede dem.

Med RTC går IBM langt for at samle nøglepersoner på et udviklingshold. Således kan både udviklere, systemarkitekter og projektledere arbejde gnidningsløst sammen på et projekt. Opgaver

kan uddelegeres og projektet kan løbende overvåges og styres. Det har dog en central begrænsning. Der er ingen steder at samle information, der vedrører projekter, men som ikke er relateret til en bestemt arbejdsopgave. Eksempelvis kan man ikke samle artefakter som klasse- eller sekvensdiagrammer. Al direkte brugerskabt viden er koblet op på work items. Disse work items er i deres natur en opgave (task, bug, enhancement, user story m.m.), som man opretter, for derefter at afkrydse det når opgaven er løst.

Mens vi arbejde på vores opgave var der imidlertid nogle spørgsmål, som skulle besvares hen ad vejen. Deres svar skulle derefter samles et centralt sted, så vi ikke skulle lede flere steder. Vi prøvede at lave to nye work items til at starte med: et Question  , og et Information  . Hver question skulle så fungere som en task, altså være en arbejdsopgave, man tog sig af, og hvis man fandt det endelige svar, så oprettede man et information work item. De to skulle så linkes i beskrivelsen af question, og gennem RTC's related work items. Dette var desværre langt fra optimalt, da vi på den måde ikke havde noget centralt sted at finde alle informationer på, samtidig med at det var meget svært at lave layout og indsætte grafik i work items. Det kunne sikkert blive bedre hvis man vred systemet mere end vi gjorde, men det var der ingen grund til, når der findes løsninger som er skabt til den slags vidensdeling.

## WIKI

Derfor var vi interesseret i at finde en central måde at samle vores informationer på. En af de samarbejdsværktøjer til deling af information som i stigende grad har vundet indpas er en wiki. En wiki er basalt set en hjemmeside, som giver brugerne mulighed for at skrive og redigere websider. Ved at linke mellem siderne kan man organisere indholdet på en enkel måde. Filosofien bagved er at det skal være let at redigere websiderne og dermed opfordre brugerne til at bidrage til den samlede vidensdeling.

Brugen af en wiki ved projekter har vist sig at være en effektiv måde at dele viden på. I en artikel af Ramon Ray fra december 2008[39] undersøger han effektiviteten af en wiki som værktøj. Et godt eksempel er en afdeling af investeringsbanken Dresdner Bank AG. Her er det lykkedes at reducere email trafikken med 75% og halvere tiden for møder. Ved at samle og vedligeholde informationer gør man det nemmere for alle involverede personer i et projekt at være opdaterede.

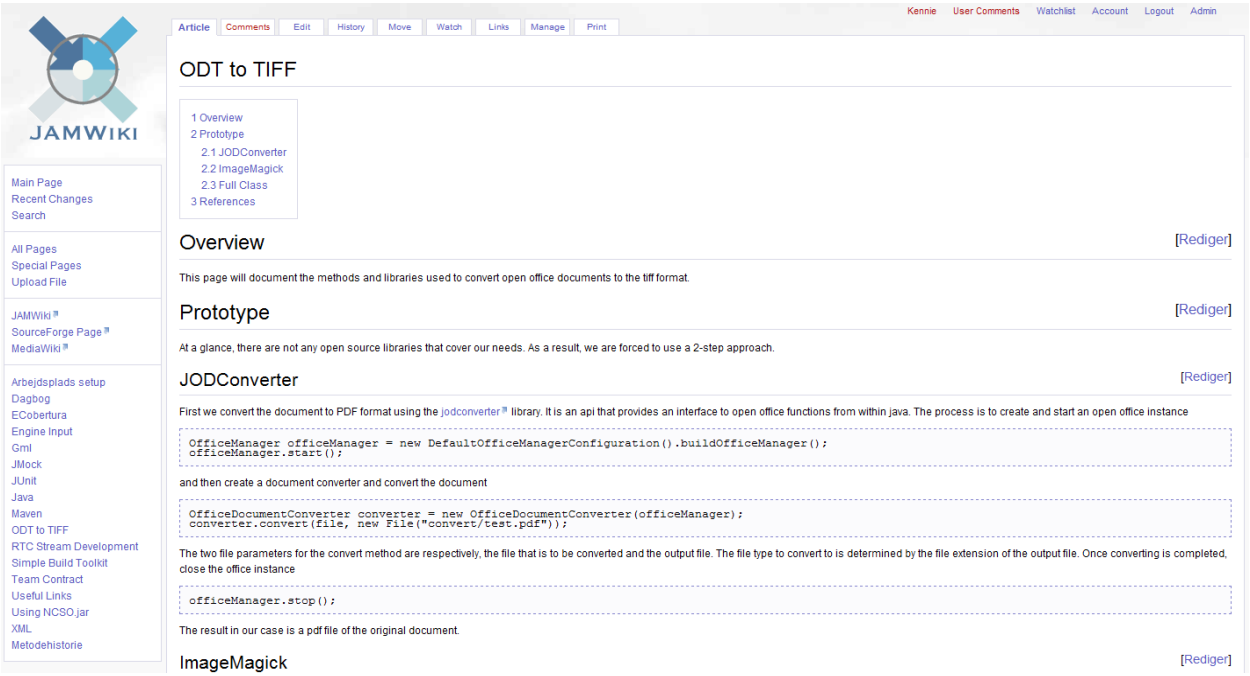
Nogle af mulighederne ved at bruge en wiki i software udviklingsprojekter belyser Panagiotis Louridas i artiklen ”Using Wikis in Software Development”[40]. På det mest enkle niveau kan en wiki fungere som et samlingssted for projektets dokumentation. Man kan simpelthen lægge alt dokumentationen op og linke til den relevante kildekode. En mere avanceret brug kunne være en platform for diskussioner. Således kan diskussioner sammenkobles med det relevante emne og derved gøre det lettere for alle at holde sig ajour med emner som har betydning for den enkelte bruger. Alle kan derved være med i rationalet bag ændringer i informationen hvis diskussionen ligger offentlig. En tredje mulighed er at bruge wikien som et redskab til møder. I stedet for at sende en agenda rundt per email og samle input sammen, kan der oprettes en side, som alle kan bidrage til. Man kan ydermere samle referat og notater efter mødet og uploade dem til selvsamme side, så

39 Artiklen kan findes her [34]
40 Artiklen kan læses her: [35]

folk, der har noget at tilføje til referatet, selv kan gå ind og tilføje det.
Der er kort sagt ikke nogen korrekt måde at bruge en wiki på, og det er den grad af fleksibilitet, der gør den velegnet som værktøj til udviklingsprojekter. Udviklingsholdet afgør selv, i hvor høj grad de vil anvende wikien.

I vores projekt ville vi have et sted til at samle viden, som havde relevans for projektet men ikke var et direkte produkt af opgaven. Så research om emner, dagbog over forløbet og guide til opsætning var alt sammen emner, der faldt i denne kategori.
Når det kommer til valget af wiki software, er valgmulighederne mange og brede. Der findes et væld af software til alle typer projekter og behov[41]. Vi valgte Jamwiki[42]. Det er skrevet i Java og bruger samme syntaks som Mediawiki[43], som er den wiki software der bliver brugt af Wikipedia.

Alternativer som e-mail, fildelingssystemer[44], mundtlig kommunikation, osv., bliver svære at vedligeholde. At bruge en wiki til informationssamlinger er en belejlig og overskuelig måde at håndtere vidensdeling på. Til vores projekt har vi haft stor glæde af vores wiki.




*Figur 38: Eksempel på en af vores wikisider, hvor vi beskriver den viden, vi har tilegnet os i, hvordan vi konverterer Open Document Text (Open og Libre Office's tekstbehandlingsformat) om til TIFF. Informationen skal senere bruges til at lave den webservice, der skal stå for konverteringen.*

41 For en liste af forskellige wiki-software, se [36]
42 JAMWiki findes på denne adresse [37]
43 MediaWiki findes på denne adresse [38]
44 Et eksemepl på dette kunne være dropbox [39]

## *Sammenfatning*

Vi har i dette kapitel vist hvordan RTC, uanset metodikken, kan hjælpe til med at styre et projekt fra start til mål. Ved at kunne integrere projektstyringsværktøjet i udviklingsmiljøet, øger man chancerne for, at udviklerne bruger det i forbindelse med deres daglige gøremål.

Der er fuld sporbarhed gennem hele udviklingsforløbet, mellem work items, builds, change sets osv. Det gør at de alle sammen får merværdi, fordi det er så nemt at se hvilke linjer, der er ændrede med en bugfix, eller se hvilke nye features der er i det nyeste build.

Samtidig gør RTC sig til grundstenen i de nyeste og mest benyttede metodikker som vandfald, XP UP og snart Kanban (link til RTC 4.0). Derved kan man tage den metodik, der passer en, og få alle de diagrammer og regler der er indenfor denne direkte i udviklingsmiljøet.

Dynamisk tildeling og ændring i både roller og metode giver store fordele, hvis man ofte evaluerer ens metode, eller hvis projektet skifter fase.

Med wiki får vi et samlet sted for den information, der er vedblevne og ikke er temporært knyttet op til en task eller bug, samtidig med at vi kan linke mellem wiki sider og work items.

# Konfigurationsstyring

Når software systemer skal udvikles, starter man en udviklingsproces. I løbet af udviklingen ændres systemet. Ny kode bliver tilføjet, fejl bliver rettet, og eksisterende kode blive refaktoreret. Så hvordan sikrer man sig, at en fejl som rettes i én version af systemet også bliver rettet i den nyeste version? Hvordan sikrer man sig, at alle dele af ens produkt fungerer sammen? Hvordan skaber man sporbarhed fra kundens krav til det færdige system? Dette gør man ved at håndtere ændringer systematisk således, at systemet bevarer en konsistent form. Denne praksis kaldes konfigurationsstyring.
Firmaer der leverer software i flere versioner ved hvor vigtig konfigurationsstyring er. Store produkter som for eksempel Adobe Reader findes i talrige varianter, alt efter platformen man bruger, og hver af disse varianter har et utal af versioner. For et firma som Adobe, der leverer Adobe Reader, er konfigurationsstyring alfa omega for at være sikker på at kunne levere nye funktionaliteter og patches i et hurtigt tempo.

Men konfigurationsstyring er også vigtig for mindre udviklingshuse. For så snart et stykke software er blevet solgt, og nye funktionaliteter skal udvikles, hvordan holder man styr på, hvilken del af koden hver enkel kunde har installeret? Og hvordan sikrer man sig at have rettet fejl i alle versioner af koden? Er den version der ligger oppe på kodeversionerings systemet nu også gennemtestet og klar til at blive sendt af sted?

Det overordnede problem kan bedst beskrives som ”Hvordan håndterer man flere forskellige versioner af ét produkt[45], så man let kan finde en bestemt version frem og arbejde med den?”. Den konceptuelle løsning kan udtrykkes således; vi identificerer alle produktdistributioner entydigt med et versionsnummer, og arkiverer dem når de udgives.

I dette kapitel vil vi prøve at besvare de ovenstående spørgsmål ved at give en gennemgang af, hvordan man kan konfigurere en række konkrete værktøjer, der tilsammen kan håndtere både versionering samt sikring af kodekvalitet. Til sidst vil vi komme med en generel diskussion af de fordele konfigurationsstyring har for en virksomhed, og perspektivere til hvilke projekter der særligt kan have gavn af dette.

Konfigurationsstyring spænder som begreb meget vidt og arbejder med emner fra kodestyring til projektledelse[46]. Fokus i dette kapitel vil være på kildekoden, og hvad der ligger i umiddelbar nærhed dertil som for eksempel versionering, kodekvalitet med videre. Selv med vores fokus er konfigurationsstyring et meget bredt begreb, som er kompleks når man tager fat i konkrete løsninger. Vi har bestræbt os på at gøre dette kapitel så læsevenligt som muligt. Dog må vi advare om, at der er mange tekniske begreber og komplicerede funktioner i spil, så det kræver tålmodighed at komme igennem kapitlet. Fordi vi beskriver en konkret løsning med konkrete værktøjer, bliver

45 Produktet er her forstået som alt, der har med udviklingen af et produkt at gøre; et stykke kode, en manual, en fejlrapport osv.

46 Her forstået på den måde at projektledelse kommer til at influere den måde, man driver konfigurationsstyring på; Hvis udviklingsmetoden er XP sikrer man sig hurtigt at have et integrationsmiljø, hvor hvis du bruger vandfaldsmetode, hører integration og versionering først til i slutfaserne, hvor alt er bygget ”helt rigtigt”.

der også brugt termer, der er specifikke for disse værktøjer. Vi vil bestræbe os på at forklare disse begreber, når de opstår[47]. To gennemgående begreber i dette kapitel er Component og komponent. Disse to termer bruger vi med forskellige definitioner;
Komponent defineres som en del af et samlet system, altså en samling af funktionalitet og information, der har en indbyrdes relation til hinanden.
Component defineres som et begreb indenfor RTC, og beskriver en tilstandsløs samling af kode i versioneringsværktøjet[48].

Disse to begreber er ikke diametrale modsætninger, men beskriver to meget forskellige ting, og skal derfor ikke sammenkædes uden for kontekst.

## *Værktøjer*

De to følgende afsnit om Apache Maven og RTC er en beskrivelse af de værktøjer, der er centrale for vores konfigurationsstyring. Vi ønsker at præsentere værktøjerne og deres termer, inden vi går ned i en mere detaljeret diskussion om konfigurationsstyring. Vores håb er, at læseren, ved at blive præsenteret for værktøjerne først, har et bedre udgangspunkt for at forstå de efterfølgende afsnit.

### Apache Maven

Maven er et værktøj til at bygge[49] software projekter, ved at håndtere projektets build, rapportering og dokumentation på en centraliseret måde. Formålet er at skabe en standardiseret måde at bygge projektet på. At bygge betyder at tage de ressourcer, der tilsammen udgør programmet (xml filer, class filer, eksterne biblioteker mv.) og pakke dem ind i det rette format, som fx rå filstruktur, War fil eller Jar fil. Samme procedure sker der når man vælger at køre sit program indenfor sit IDE, dog laver den ikke til slut en færdig pakke med ens program.

Maven kan desuden skabe en centraliseret samling af projektinformationer som unit test rapporter, log filer og project dependencies. Udover samlingen af informationer har Maven en målsætning om at skabe retningslinjer for best practices[50] i forbindelse med udvikling. Det gøres for eksempel ved at holde test- og kildekoden adskilt. Således deler standardindstillingen i Maven projektet op i en mappestruktur, der adskiller kildekode og test.

47 Hvis der er specifikke begreber indenfor RTC som ikke er beskrevet, har IBM en begrebsliste her: [40]
48 En mere fyldestgørende forklaring fås længere nede i kapitlet.
49 Byg og build bliver i dette kapitel skiftevis brugt til at definere det samme.
50 Best practices er et begreb som betegner en række guideline for hvordan man bedst bruger en teknologi, eller et værktøj. Det er måder at gøre nogle bestemte ting på som man anser for at være de bedste måder at løse problemer på.

```
my-app
|-- pom.xml
`-- src
    |-- main
    |   `-- java
    |       `-- com
    |           `-- mycompany
    |               `-- app
    |                   `-- App.java
    `-- test
        `-- java
            `-- com
                `-- mycompany
                    `-- app
                        `-- AppTest.java
```

*Figur 39: Mappestrukturen for maven projektet "my-app".*

### *Project Object Model*

Et af grundelementerne i Maven som gør det nemt at konfigurere er dennes Project Object Model (POM).
POM er en XML fil som indeholder detaljer om konfigurationen af projektet. Maven har en default POM som hedder en Super POM. Alle andre POMs arver fra denne, inklusiv den man laver til sit eget projekt.

En POM skal som minimum indeholde følgende elementer:

- project root
- modelVersion
- groupId
- artifactId
- version

Dermed kunne en mulig POM se således ud:

```
<?xml version="1.0" encoding="UTF-8"?>

<project>
      <modelVersion>4.0.0</modelVersion>
      <groupId>dk.semaphor.sa</groupId>
      <artifactId>sa</artifactId>
      <version>1</version>
</project>
```

Arkitekturen i Maven er plugin baseret, og det gør det muligt at skrive plugin grænseflader til alle typer værktøjer i hvilket som helst sprog. I praksis er det dog primært værktøjer til Java, der bliver

brugt. For at kunne benytte et plugin skal der laves en reference som opnås ved at tilføje en dependency i POM filen.

Hvis man for eksempel vil benytte JUnit til at teste med, laver man følgende tilføjelse til sin POM fil.

```
<?xml version="1.0" encoding="UTF-8"?>
<project>
	<modelVersion>4.0.0</modelVersion>
	<groupId>dk.semaphor.sa</groupId>
	<artifactId>sa</artifactId>
	<version>1</version>
	<dependencies>
		<dependency>
			<groupId>junit</groupId>
			<artifactId>junit</artifactId>
			<version>3.8.1</version>
		</dependency>
	</dependencies>
</project>
```

Maven håndterer så alt arbejdet med at hente den specifikke version af JUnit ned, lave projekt-referencen og pakke den med, når koden skal bygges eller køres. Herefter er det muligt at kompilere projektet og afvikle unit test. Samtidig er man nu sikret, at de test som tilføjes til projektet bliver eksekveret hver gang projektet bygges.

## Maven Life Cycle

Processen med at bygge og distribuere et projekt har en fast defineret form som kaldes build lifecycle. Maven har som standard tre livscykluser: default, clean og site. Default cyklusen håndterer projektets implementering, clean cyklusen håndterer oprydning i projektet, og site håndterer projektets website dokumentation. Alle disse tre livscykluser består af byggefaser (build phases)[51]. Eksempelvis har en clean livscyklus 3 faser.

- pre-clean - udfører processer, der er nødvendige forud for den aktuelle projekt rengøring
- clean - fjerner alle filer, der blev genereret af det tidligere build
- post-clean - udfører processer, som er nødvendige for at færdiggøre projektets rengøring

51 En komplet liste af build phases kan ses her; [41]

Faserne udføres sekventielt, og selvom det i ovenstående eksempel kun giver mening at udføre alle faser, så er det ikke tilfældet i en default livscyklus. Nogle faser har  nemlig også tilknyttet mål (goals). I en clean livscyklus er fasen clean et mål. Kigger man derimod på en default livscyklus, er der flere mål som kan afvikles. Her er målene for en default livscyklus.

- process-resources
- compile
- process-test-resources
- test-compile
- test
- package
- install
- deploy

Man kunne forestille sig, at man ikke ønsker at bruge den sidste fase, deploy, som kopierer den endelige pakke til et centralt repository eller lægger pakken op på en server, så den kan gå i produktion. Det er måske rigeligt at bruge den næstsidste fase, install, som kopierer pakken til et lokalt repository, så den kan bruges i andre projekter. Ved at bruge install afvikler man dog samtidig alle foregående faser ned til install.

Der er meget naturligt mange værktøjer, som har de samme grundlæggende funktioner som Maven. Af værktøjer, der fungerer i sammenspil med Java, kan nævnes Gradle[52], Buildr[53] og Ant[54]. Lige netop Ant er interessant, når man sammenligner med Maven.
Begge værktøjer benytter XML til at konfigurere indstillinger. Men hvor Maven benytter en grundlæggende ide om konvention over konfiguration[55], så er Ant mere fokuseret på mulighederne for tilpasninger. Selvom det ultimative mål er det samme, så foregår der stadig en 'krig' mellem dem, der foretrækker Ant og dem som foretrækker Maven[56].

Med tanke på mantraet i Mavens filosofi om mindre behov for konfiguration, er anbefalingen for udviklere uden større erfaring med disse værktøjer at begynde med Maven, og hvis man ønsker flere muligheder for tilpasninger, så gå over til Ant.

## Rational Team Concert

I kapitlet ”Projektstyring” introducerede vi Rational Team Concert, dens grundlæggende opbygning, og de funktioner som RTC havde i forhold til dette. Vi vil her fokusere på de funktionaliteter, som vi gør brug af i konfigurationsstyring af vores kode.

52 Et groovy-script baseret byggeværktøj kilde; [42]
53 Apaches groovy-script baserede byggeværktøj, der har inkoorporeret mange ting fra Maven for at opnå høj kompatibilitet med alle Mavens biblioteker. Kilde; [43]
54 System magen til Maven, men med langt flere indstillingsmuligheder. Kilde; [44]
55 Dybere forklaring på konvention over konfiguration henvises til [45]
56 Der refereres her til en artikkel omhandlende emnet; [46]

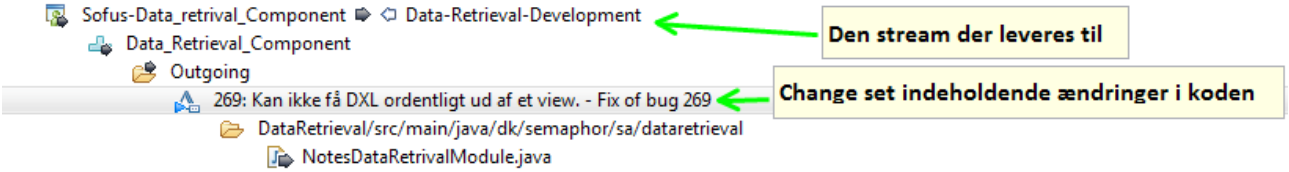
### *Source control*

Dette afsnit er skrevet til læsere, der ved hvad SVN[57], CVS[58], samt GIT[59] er. Der vil flere gange forekomme steder hvor vi, for bedst at kunne beskrive funktioner i RTC, forklarer disse ting ud fra deres lighed, eller mangel på samme, i forhold til ovennævnte.

RTC's sourcecontrol er et 3. generations kodestyrrings værktøj, der går under navnet Software Configuration Management[60] SCM. Dens opbygning minder på nogle punkter om GIT. For eksempel arbejder begge med deltaer[61] til at ændringen af en fil, i modsætning til SVN og CVS, der tager hele filen som revision.

Selve opbygningen af SCM'en er som følger startende fra udviklerens maskine:
Ændringer til koden bliver registreret lokalt på maskinen, når udvikleren har foretaget disse.

De ændringer til koden, som skal deles, checkes ind til et change set . Change settet lægger sig automatisk ind i brugerens repository workspace, som er en kopi af brugerens lokale workspace.

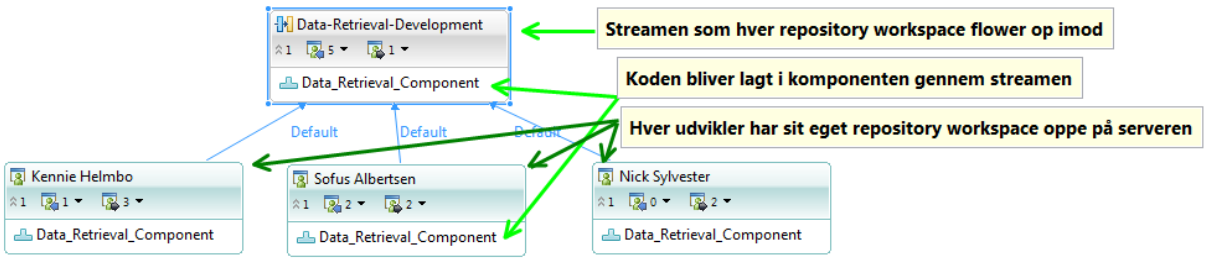



*Figur 40: "Pending Changes" vindue, med et udegående change set, set fra Eclipse.*

Figur 40 viser hvordan et change set og en component interagerer. Gående nedefra og op, ser vi først filen, der er ændret, dernæst mappen som filen hører til. Change settet kommer derefter, og oven over er der en indikation af, at der er et change set kommende fra udvikleren op til streamen. Næstøverste linje angiver hvilken komponent i workspacet der er ændret, for til sidst at angive hvorfra og hvortil change settet skal.

Når brugeren vil dele change sets med andre, skal vedkommende deliver change settet til en stream . Change settet bliver så gemt i den component som streamen peger på. Alle streams gemmer ned i en component, men flere streams kan også dele en component.

57 Subversions hjemmeside kan findes her; [47]
58 Det første versioneringssystem som vandt stor udbreddelse i udviklerkredse [48]
59 Git's hjemmeside [49]
60 For mere information om begrebet, se; [50]
61 I GIT kaldet diff. Beskriver programmatisk ændringen mellem to filer. Hvis der er ændret 10 linjer i en fil på 100 mb, er det kun de 10 linjer der bliver gemt.




*Figur 41: Beskriver yderligere hvordan udviklernes repository workspaces kan pushe change sets til en component gennem en stream.*

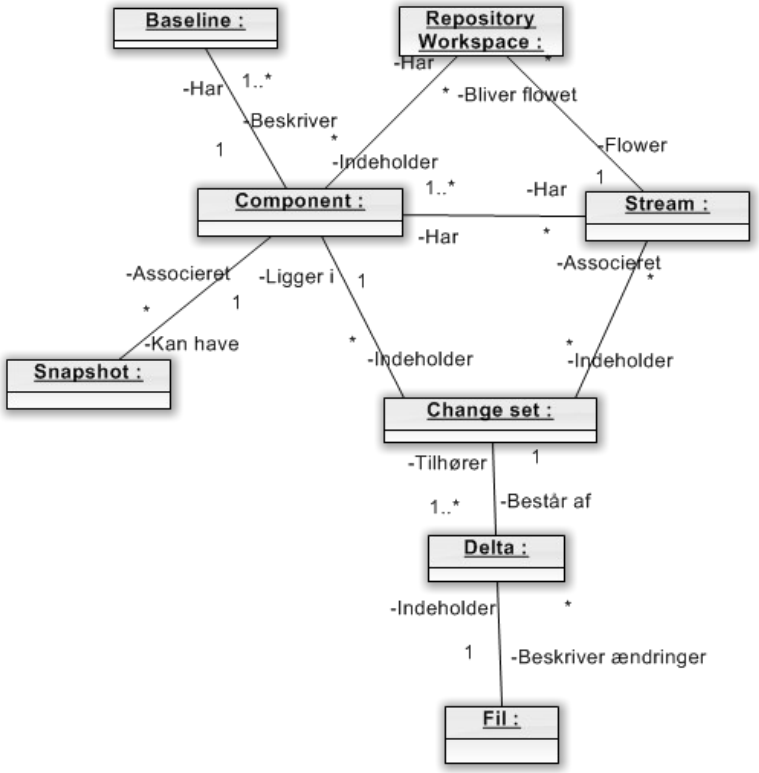
En component indeholder alle de change sets som brugerne har lagt op gennem forskellige streams, men har ikke noget bestemt stadie. Dette bliver afgjort af, hvilken stream som brugeren har valgt at koble sig op på. Et flow er en forbindelse mellem to workspaces eller mellem et workspace og en stream. Derfor kalder man det i RTC at ”flowe” mod en stream, når en bruger sender og modtager sine changeset til en stream.

Sagt på en anden måde er en component det sted, som opbevarer alle change sets. En component har mindst én baseline , som er en permanent kopi af et stadie. Den kan enten være tom, hvis man laver en ny component, eller være en baseline lavet af en anden baseline og tilhørende change sets.

En stream har en eller flere components i sig og beskriver derfor en gruppering af change sets. Denne samling af change set udgør tilstanden, som streamen befinder sig i. En stream kan bedst sammenlignes med en branch/trunk i SVN[62].

Et snapshot er ligesom en stream, et sted hvor man beskriver et stadie ved at have en component, der har en baseline, samt tilhørende change sets. Forskellen er at streamen er dynamisk; den ændrer stadie hver gang, der bliver lagt et nyt change set, og snapshottet er statisk; der kan ikke blive ændret på samlingen af change sets efter tilblivelsen.

62 IBM har lavet en liste over sammenlignelige begreber i henholdsvis SVN og RTC:[51]

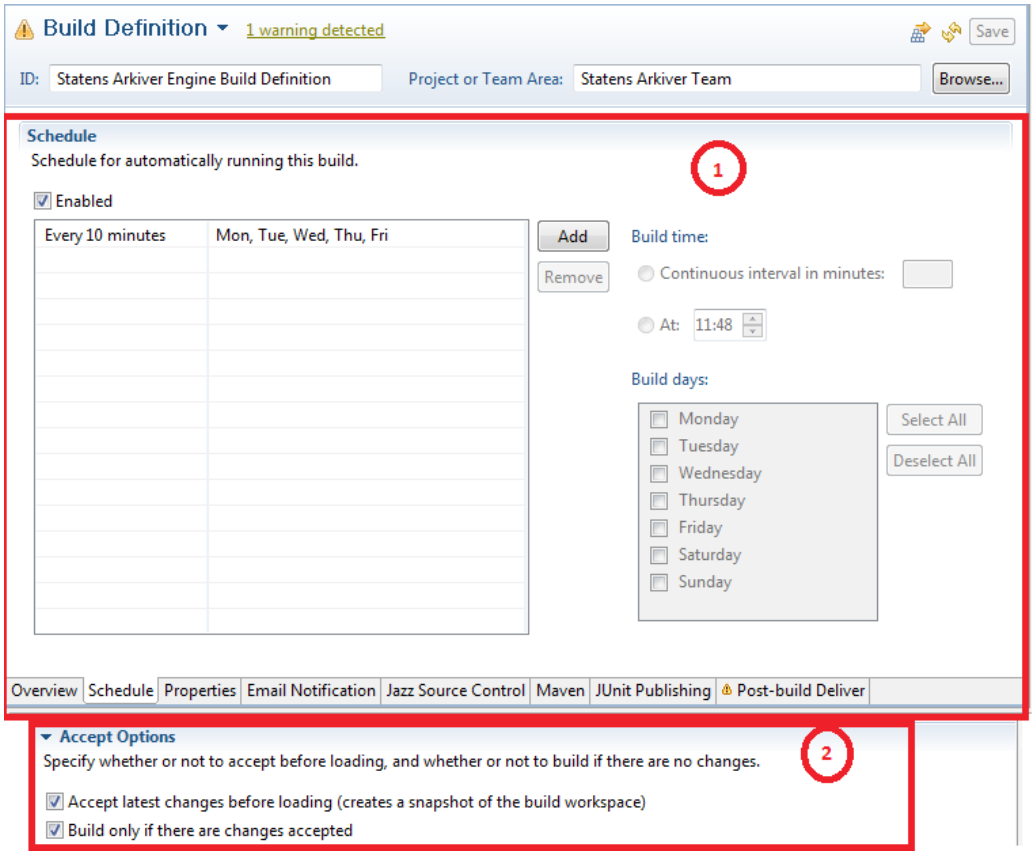



*Figur 42: UML diagram over de forskellige begreber indtroduceret i source control afsnittet.*

### *Build engine*

RTC har direkte integration med mange forskellige byggesoftware, bland andre Ant, Maven og deres eget Rational Build Forge[63]. Det gør at man kan sætte RTC op til at lytte på ændringer i koden og bygge hver gang der kommer en ændring. Dette kan ske efter et givent interval, på et bestemt tidspunkt eller en kombination af disse.

63 For mere information om buildforge, se; [52]




*Figur 43: Eclipse udgaven af en build definition*

Figur 43 viser vores Build Definition for Engine komponenten. Ramme 1 viser, hvordan man kan indstille hvornår en given build definition skal bygge. I ramme 2 kan man bestemme om bygget skal lave tjek-in af ændringer inden bygget, og om der skal bygges, lige meget om der er sket ændringer i koden. Eksemplet er sat til at bygge hver 10. minut i løbet af ugedagene. Nederst kan man se, at vi har markeret en checkbox, som angiver, at der ikke skal buildes, hvis der ikke er kommet nogle ændringer til koden.

Hvert build bliver angivet med et snapshot, som er et øjebliksbillede af hvordan kodebasen så ud på byggetidspunktet. Det giver mulighed for at hente den version af koden ned på ens lokale maskine og undersøge, hvad der eventuelt fik et byg til at fejle.

## *Kvalitet igennem konfigurationsstyring*

Centralt for dette afsnit er to begreber; konfigurationsstyring og continous integration.

Konfigurationsstyring er kunsten at kunne holde styr på flere versioner af samme kodebase. Bogen [OOSE] beskriver konfigurationsstyring således:

”Configuration Management is the discipline of managing and controlling change in the evolution

of software systems"[64]

Den opstiller 4 aktiviteter, der identificerer konfigurationsstyring:

- **Identification of configuration items.**

  Evnen til unikt at kunne identificere og navngive hver enkel del og version af denne del i forhold til projektet som hele. Samtidig også en overordnet beskrivelse af hvilke artefakter der skal indkluderes i konfigurationsstyringen.

- **Change control**

  Når der sker ændringer, skal kvaliteten af den nye kode sikres. Dette kan gøres på mange forskellige måder, blandt andet unit tests, walkthroughs med flere, og kan ske på alle planer fra programmøren til lederen.

- **Status accounting**

  Markeringen af status på alle de dele der bliver arbejdet med, fra kode til opgaver, fejlindberetninger, og funktionalitetsforbedringer. Derved bliver det lettere for ledelsen at holde styr på udviklingen af deres projekt, samt mulighed for et bedre samarbejde udviklerne imellem.

- **Auditing**

  Revidering af bestemte versioner af komponenterne for at sikre kvaliteten og rigtigheden af disse.

Ud over disse aktiviteter beskriver den også build management. I vores optik ligger build management som en naturlig del af konfigurationsstyring, og vi vil uddybe begrebet i dette kapitel.

Continous integration ligger som en del af konfigurationsstyring, da det, som før omtalt, berører alle de fire hovedaktiviteter. Continous integration er en del af eXtreme Programming metodologien, hvor hyppige og hurtige afleveringer af fungerende kode er blandt grundprincipperne[65]. Continous integration handler om at undgå "big bang[66]" integration ved at have kodebasen samlet ét sted, og gøre det til praksis at alle udviklere synkroniserer deres lokale kodebase mindst en gang pr. dag. Samtidig bliver kodebasen testet ofte, så udviklerne får hurtigt feedback på hvor i koden, der er sket fejl. Derved fanger man fejl og misforståelser, før de kan nå at slå rod i koden, og udvikle sig til store problemer senere hen.

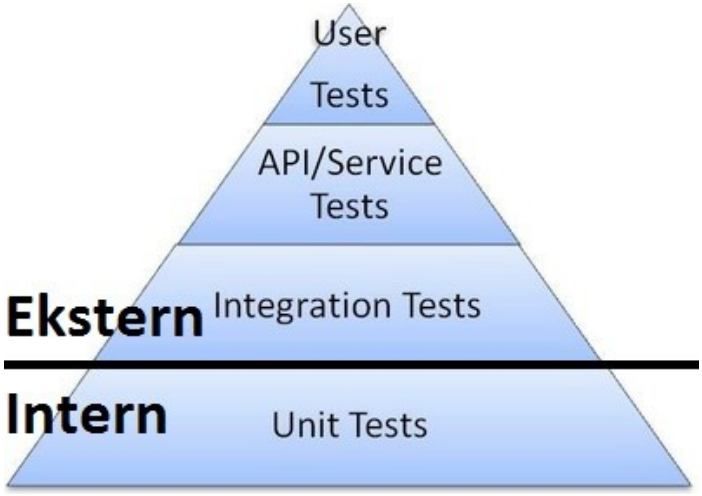
Continous integration handler derfor i høj grad om kodekvalitet. Vi deler kodekvaliteten op i to dele; intern og ekstern.
Intern kodekvalitet fokuserer på hver komponent isoleret set. Det spænder fra de simple ting, som en sikring af at koden kan kompilere, over til de mere avancerede, og mere konfigurerbare ting, som at koden er dækket godt nok af testmetoder.

64 [OOSE] Afsnit 13.2 ,"An overview of Configuration Management"

65 "Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale. " kilde; [53]

66 Martin Fowler beskriver sin første oplevelse med big bang integration således: "... I was told that this project had been in development for a couple of years and was currently integrating, and had been integrating for several months. My guide told me that nobody really knew how long it would take to finish integrating." kilde:[54]

I ekstern kodekvalitet er fokus på systemintegrationen i et projekt. Det handler her om at teste på tværs af komponenter for at sikre sig, at ændringer, der er sket i én komponent, ikke har en negativ impakt på integrationen.




*Figur 44: Vores viderefortolkning af figur 51.*

Hvis vi kigger på intern og ekstern kodekvalitet, kan vi se, at intern kodekvalitet dækker den nederste del af pyramiden. Ekstern kvalitet, forstået som kvalitet komponenterne imellem, bliver sikret af integrationstests, API/service tests, og user tests.

I de følgende afsnit vil vi beskrive, hvordan man kan håndtere intern og ekstern kvalitet.

Vi har ikke i vores projektforløb arbejdet med forskellige produktversioner. Vi har desuden kun sikret den interne kvalitet i forhold til vores ovenstående definition.

Vores opbygning tager dog højde for en fremtidig udbygning og vil kunne håndtere ekstern kvalitet, som bliver behandlet senere i kapitlet. Dermed fungerer den som grundlaget i vores senere diskussion af, hvordan et integrationsmiljø kunne se ud.

## Håndtering af intern kvalitet

Vores opbygning af konfigurationsstyring består af en kombination af tre værktøjer:

- RTC Stream

  Det sted hvor hver udvikler kan koble sig op på og dele koden centralt.

- RTC's build engine, kaldet Java Build Engine (JBE)

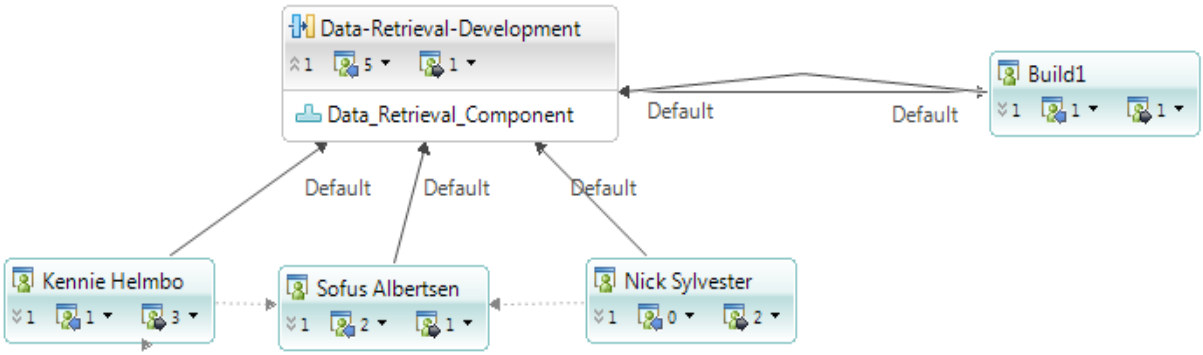
  Applikationen som henter koden ned, kører den, analyserer resultatet og handler ud fra dette resultat. Det er den som har ansvaret for at bestemme, hvornår et build skal køre[67].

- Maven

67 Figur nr. 43 viser hvilke indstillingsmuligheder der er gældende for JBE.

Vores build automation værktøj som bliver eksekveret af JBE.

Vores projekt er som før beskrevet delt ind i en række delkomponenter; Engine, Data retrieval, Common med flere. Hver delkomponent har sin egen stream uden relation til de andre.

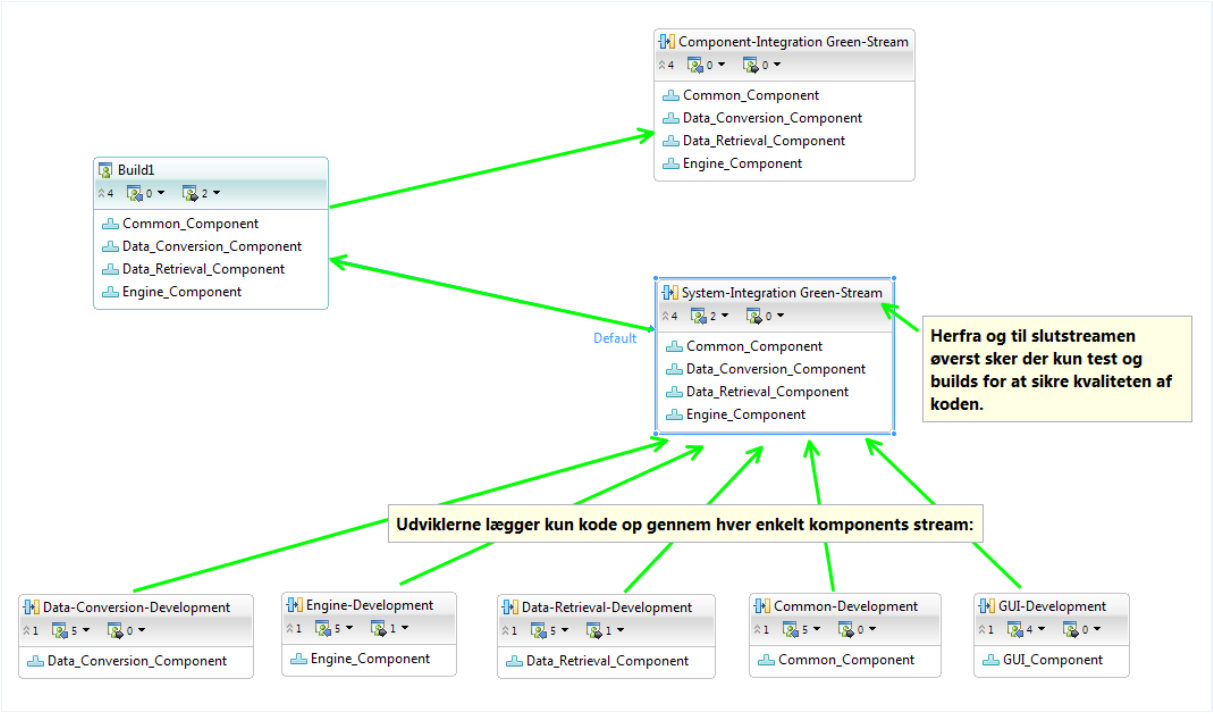



*Figur 45: Beskriver forholdet mellem udviklernes repository workspaces, og en komponents stream.*

Disse streams, og tilhørende components, er isoleret set blot code repositories; der er ikke nogen sikring af, at den kode, der bliver lagt i dem, kan køre eller kan integreres med de andre komponenter[68].

Det er fra disse streams at alt kode kommer ind i systemet. Alle efterfølgende streams har samme kode, blot sikret forskelligt gennem test. Den øverste med sikring for både intern og ekstern kvalitet i hver enkelt komponent. Billedet nedenfor illustrerer systemet.

68 RTC giver mulighed for, gennem nogle processregler, at sætte en række kriterier for, hvornår man må lægge kode op. Bla. Må man ikke lægge kode op, der importerer biblioteker, der ikke bruges eller er syntaktisk fejlskrevet. Disse regler kan man slå til og fra efter behov.




*Figur 46: Kun gennem de nederste 5 streams kommer der ny kode ind i systemet. Resten sikrer kvaliteter på forskellige niveauer.*

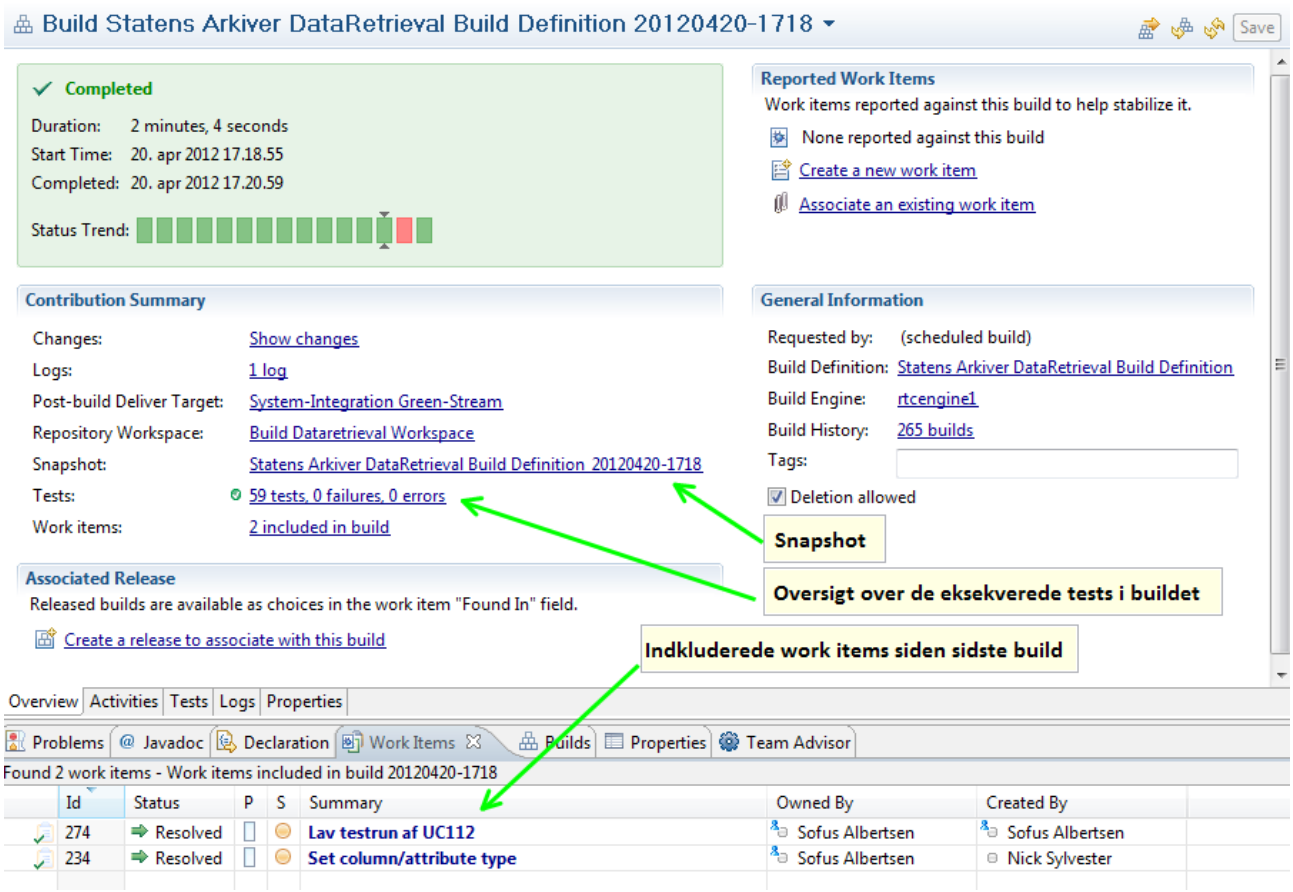
Næste trin i konfigurationsstyringen står build engine for. Den tager ansvar for kvalitetssikringen af koden internt i komponenten. Via Maven kan man give den en række parametre for, hvornår en Maven fase er kørt succesfuldt, blandt andet ved at angive hvilke testklasser der skal køre. Man kan også via pluginnet til Cobertura[69] angive hvor høj en branch og line coverage, givne pakker eller filer skal have, før de skal have lov til at køre igennem Maven uden fejl.

```
...
      <groupId>dk.semaphor.sa</groupId>
      <artifactId>dataretrival</artifactId>
      <version>0.0.3-SNAPSHOT</version>
      <packaging>jar</packaging>
...
      <dependencies>
            <dependency>
                  <groupId>dk.semaphor.sa</groupId>
                  <artifactId>common</artifactId>
                  <version>0.0.4-SNAPSHOT</version>
            </dependency>
...
      </dependencies>
      <build>
            <plugins>
                  <plugin>
```

69 For uddybning af værktøjet, se kapitlet ”Test”

```
                        <groupId>org.codehaus.mojo</groupId>
                        <artifactId>cobertura-maven-plugin</artifactId>
                        <version>2.5</version>
                        <configuration>
                              <check>
Herfra angiver vi, med hvilken      <branchRate>85</branchRate>
coverage vi vil have udfra          <lineRate>85</lineRate>
forskellige parametre, som for      <haltOnFailure>true</haltOnFailure>
eksempel hvor mange linjer der      <totalBranchRate>85</totalBranchRate>
skal være dækket af test, før       <totalLineRate>85</totalLineRate>
at bygget skal gå godt.             <packageLineRate>85</packageLineRate>
                                    <packageBranchRate>85</packageBranchRate>
                              </check>
...
                        </configuration>
                        <executions>
                              <execution>
Her angiver vi at testen-->         <phase>test</phase>
skal køre under testfasen.          <goals>
                                          <goal>cobertura</goal>
                                    </goals>
                              </execution>
                        </executions>
                  </plugin>
...
            </plugins>
      </build>
</project>
```

Build enginen kører de faser af Maven, der er defineret og laver en rapport over buildet, så projektholdet kan se, hvordan det er forløbet.




*Figur 47: Build rapport taget fra DataRetrieval build defiitionen. Viser forskellige aspekter af bygget, såsom overordnet status, test-status, angivelse af snapshot med mere.*

Øverst i venstre hjørne kan man se, hvor lang tid det har taget at bygge, start og stop tidspunktet, samt en grafisk repræsentation af de sidste 15 byg som engine har lavet. Der markeres med grønt for ingen fejl, og rødt for fejl af en eller anden art.

Ved hvert byg bliver der taget et snapshot af koden, som den så ud ved byggetidspunktet. Derved kan man gå tilbage og hente en kopi af koden, som den så ud på nettop det tidspunkt. Man kan også lade en ny stream tage udgangspunkt i snapshottet og derved skabe en afgrening.

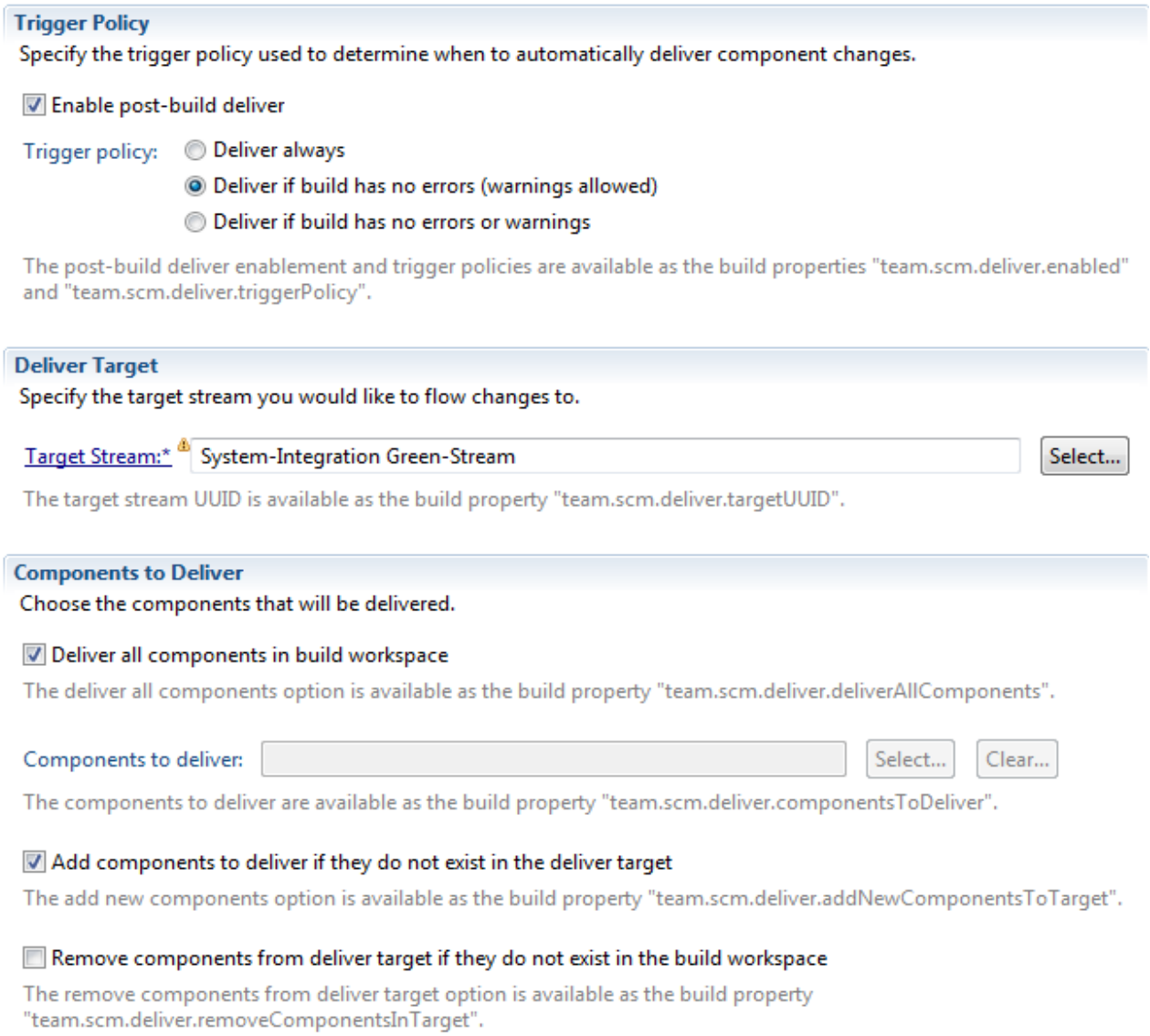
Derudover vises der statistikker over hvor mange og hvilke work items, der er submittet change sets op imod siden sidste byg, samt om bygget løser nogle bugs, eller skaber nogle (øvre højre hjørne). Fordi hvert build bliver koblet så tæt sammen med kodeversioneringen, har man også mulighed for at se, hvor der er ændret kode, helt ned på linjeniveau.

Det er en god fremgangsmåde at optimere unit tests til at blive afviklet hurtigt[70], så man hele tiden kan bygge. Det giver hurtigt feedback til udviklerne, hvilket gør dem mere motiverede til at rette

70 S. 26, afsnit 1.4.3 ,[EoTA]. Her beskrives der at deres JUnit test tager 8 minuter, og deres integrationstests tager 60-90 minutter. Når de to samles med GUI test i et såkaldt ”full build”, skal det køre natten over.

fejlen med det samme. Samtidigt bliver det mere klart for alle, hvad der eventuelt har ødelagt et build, hvis der ikke går timer fra man har skrevet koden, til man får svar.

Hvis buildet går godt, har man mulighed for at specificere, hvor koden skal lægges op gennem build engines post-delivery fane:

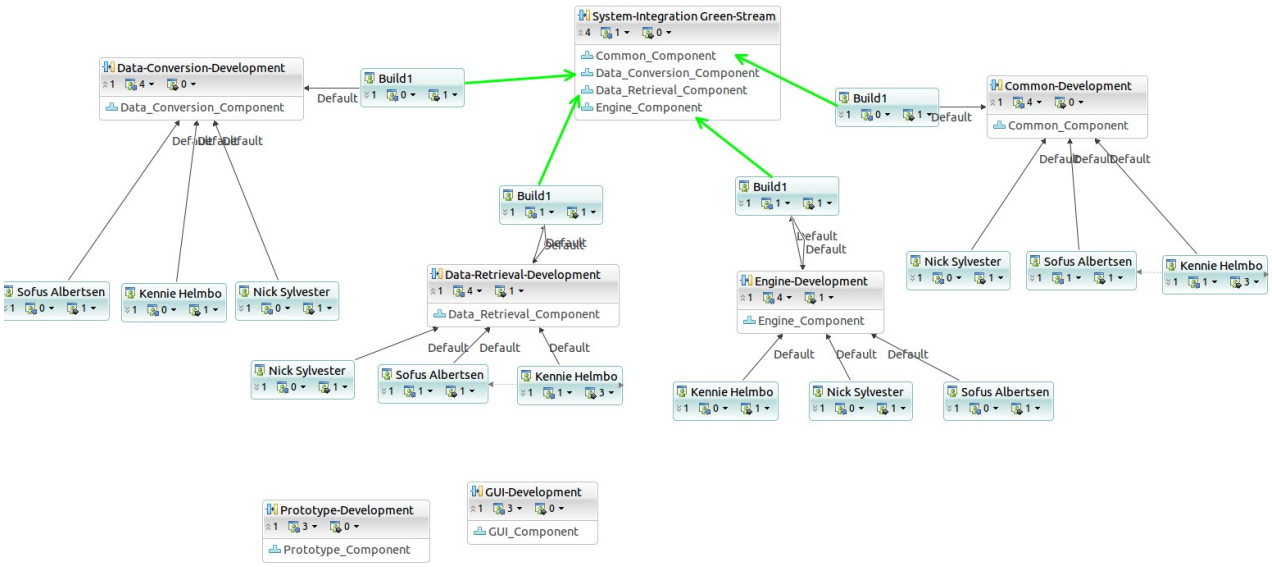



*Figur 48:Et faneblad under "Build definition". her kan et build defineres til at accepte (hente) kode fra én stream og deliver (gemme) koden på en anden stream, under forudsætningen af, at den givne trigger policy, angivet i øverste afsnit af definitionen, er overholdt.*

Derved kan man lave en en-vejs strøm som går opad, hvor koden gradvist bliver verificeret, jo længere den kommer op.

Hvert enkelt subkomponent bliver fra build lagt op til en fælles stream, hos os navngivet ”System-

integration-green-stream”:



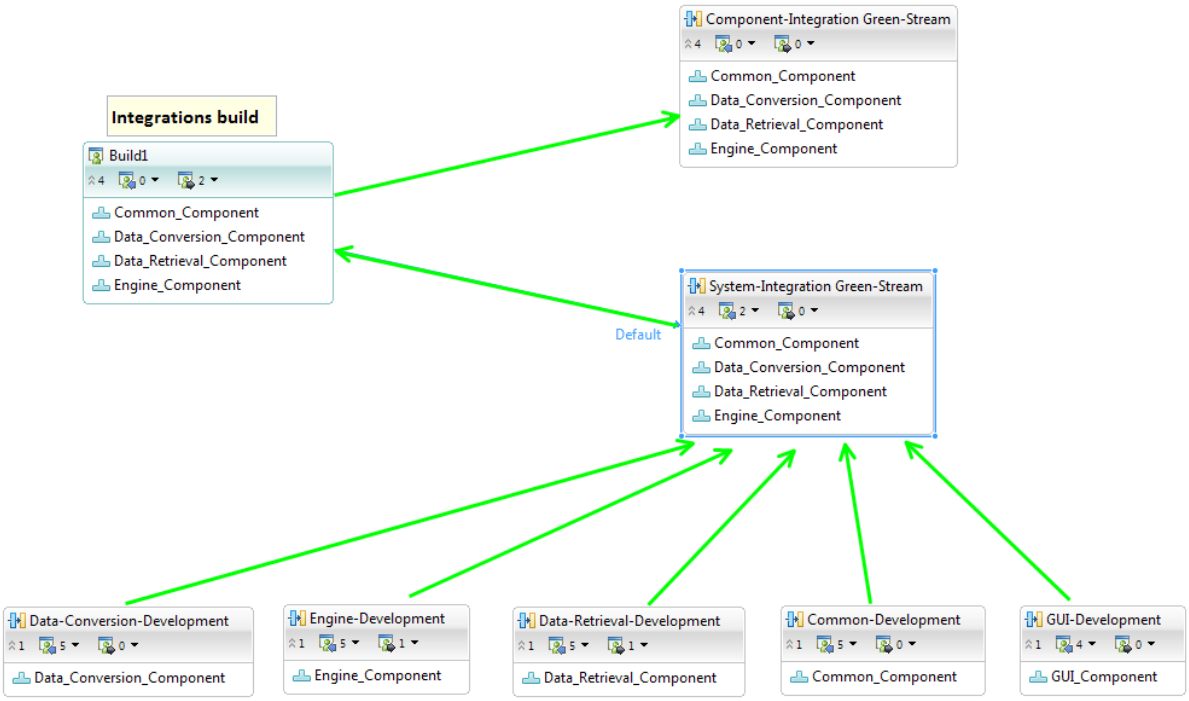

*Figur 49: Fuldt overblik over vores opbygning af streams og components. Hver udvikler flower med de forskellige komponenters streams. Hver stream har en build defition som bygger, og lægger fejlfri kode op på integrations streamen.*

I denne integrations stream har vi gennem unit test sikret komponenternes interne kvalitet, men ikke deres eksterne. Der kan derfor godt opstå situationer, hvor ændringer til den ene komponent gør, at den får en anden komponent til at fejle, enten ved ændring i interfacet mellem komponenterne, eller ved en ændring af de datastrukturer der bliver udvekslet, som for eksempel XML filer.

Vi har gjort os nogle overvejelser om, hvordan vores opbygning i fremtiden kan udbygges til at tage højde for denne problemstilling. Følgende er en beskrivelse af dette.

## Håndtering af ekstern kvalitet

For at kunne håndtere ekstern kvalitet i vores opsætning er vi nødt til at skabe endnu en build engine og endnu en stream:



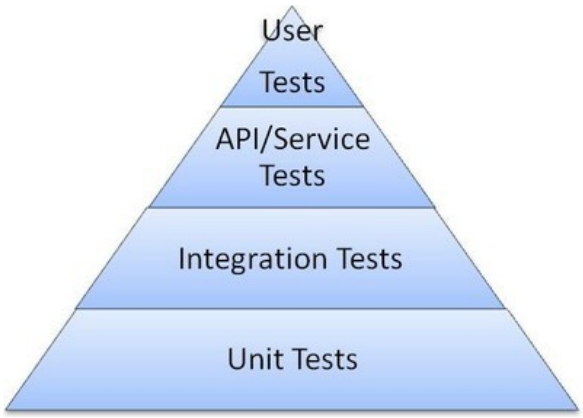

*Figur 50:Beskriver hvordan man kan opbygge et hieraki af build engines for at sikre forskellige aspekter af kodekvalitet. Mellem de nederste 5 streams og "System-Integration Green-Stream " ligger der en build definition for hver stream og pusher godkendt kode op til streamen. Herefter ligger endnu en build engine og bygger samlingen af alle komponenterne med integrationstest mv. Hvis den går godt, så lægges den nye kode op på "Component-Integration Green-Stream"*

Ud over de komponenter som er vist på illustrationen ovenfor, skal der bruges et projekt som indeholder integrationstestene. Dette projekts POM-fil kommer til at importere samtlige komponenter, for derefter at køre testklasserne. Nedenfor ses et eksempel på hvordan disse projekter ville blive defineret i pom filen:

```
<groupId>dk.semaphor.sa</groupId>
	<artifactId>integrationprojekt</artifactId>
	<version>0.0.3-SNAPSHOT</version>
	<packaging>jar</packaging>

	<dependencies>
		<dependency>
			<groupId>dk.semaphor.sa</groupId>
			<artifactId>common</artifactId>
			<version>0.0.4-SNAPSHOT</version>
		</dependency>
		<dependency>
			<groupId>dk.semaphor.sa</groupId>
			<artifactId>engine</artifactId>
```

```
            <version>0.0.4-SNAPSHOT</version>
        </dependency>
        <dependency>
            <groupId>dk.semaphor.sa</groupId>
            <artifactId>dataretrieval</artifactId>
            <version>0.0.4-SNAPSHOT</version>
        </dependency>
        <dependency>
            <groupId>dk.semaphor.sa</groupId>
            <artifactId>conversion</artifactId>
            <version>0.0.4-SNAPSHOT</version>
        </dependency>
...
    </dependencies>
</project>
```

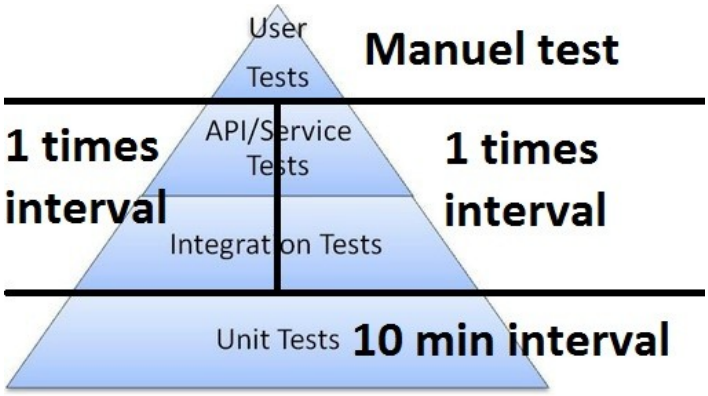



*Figur 51:Fordeling af tests efter andel i pyramiden. Mange unit tests, få user tests*

De tilhørende testklasser skal teste på tværs af delkomponenterne. Det er ikke kun unit tests, men også integration test og API/Service test, for builddelens vedkommende.

Den øverste del, User Tests, er et manuelt trin og kommer som regel efter at disse byg er lavet[71].

Integrationstests går ikke som unit tests kun på enkelte funktionskald, men i langt højere grad på test i forhold til forretningsgangen, og testene her tager derfor lidt længere tid[72], da de simulerer brugerinput. Det samme gør sig gældende med API/Service tests, der kan inkludere drivere som for eksempel Selenium[73]. Derfor er det vigtigt at lave dette split mellem de hurtige unit tests og de langsommere




*Figur 52: Forslag til indeling af tests i forskellige builds.*

71 Denne pyramide er en variation over den originale pyramide (se kapitlet ”Test” figur 11). I den originale er hele pyramiden automatiseret, men da vi her definerer User Tests som brugere der fysisk interagere med systemet, kan disse ikke blive automatiseret.

72 Da vi ikke selv har lavet hverken integrationstests eller API/service, refererer vi her til [EoTA], specielt s. 27 afsnit 1.6.

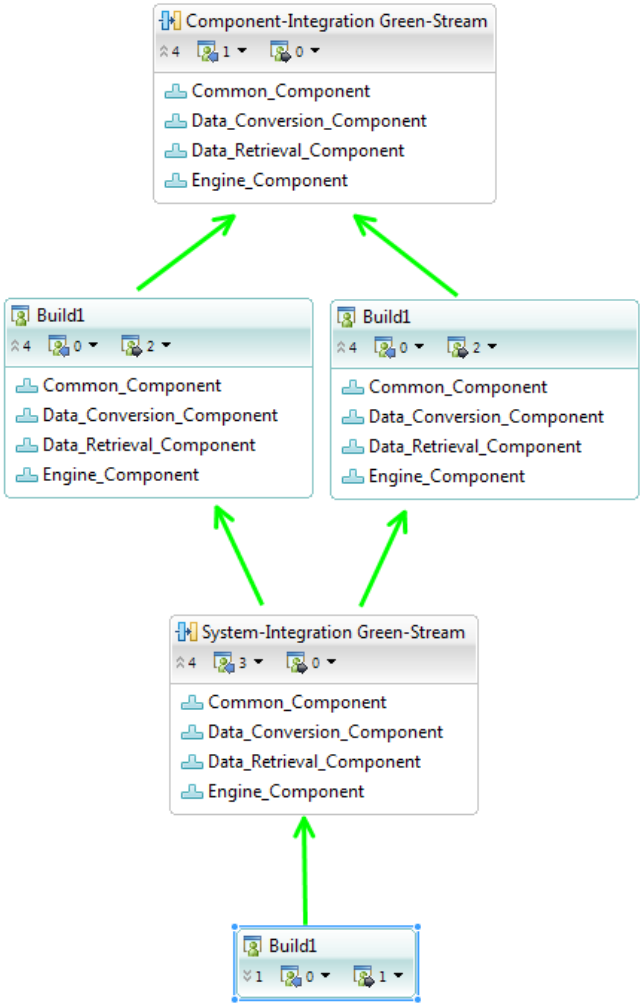
73 For mere information om selenium, se; [55]

integrationstests.

For at få det fulde udbytte af continous integration skal alle builds kunne køres igennem på maks et par timer. Hvis ens builds begynder at overstige det, kan man gøre to ting; opgradere byggeserveren med mere ram, CPU og lignende eller split testen op i to dele, og kør dem parallelt.

På den måde sikrer man sig en hurtigere feedback, som igen er med til at styrke integrationen mellem det, som udviklerne lige har implementeret, og det som rapporterne giver dem svar på.

Vi har nu vist, hvordan man ved hjælp af streams og build definitioner kan opbygge et miljø, som automatisk og konstant tjekker koden for kvalitet, både internt og eksternt. Herfra vil vi beskrive, hvordan man nemmest kan rette fejl igennem flere versioner af samme produkt, samt hvordan man kan støtte og styre revideringer i kodebasen.

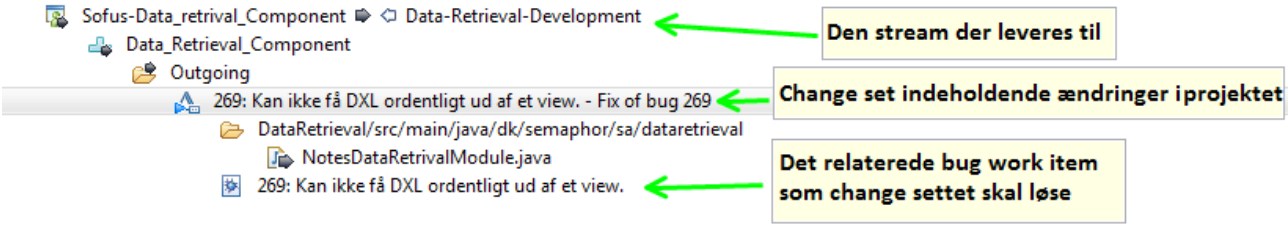



*Figur 53:Eksempel på hvordan man kan opbygge vores streams, så de kan have to paralelle byggemaskiner.*

## Håndtering af fejl og ændringer.

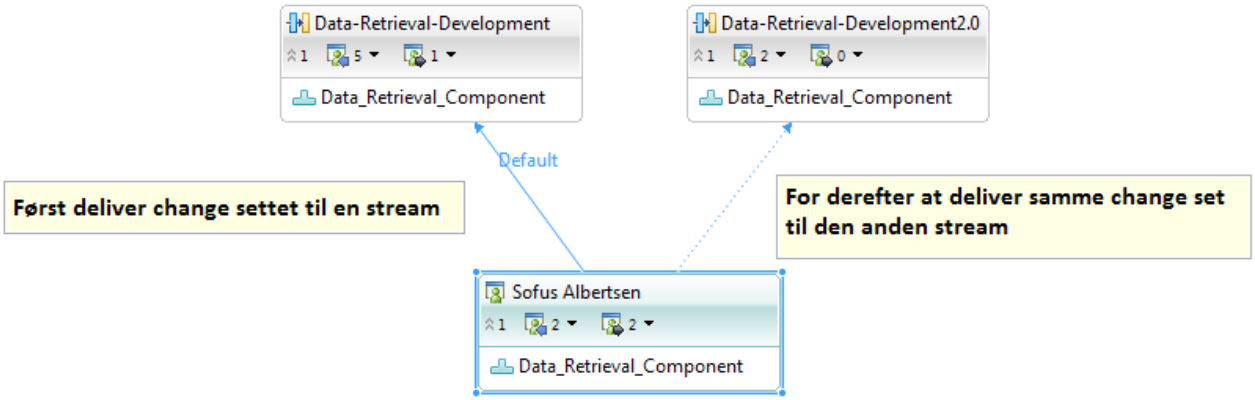
Når enheder af kode kombineres for at danne et softwareprodukt så opstår der ofte fejl. Forudsætningen for at kunne rette disse, gennem alle versioner af koden, er, at versionerne følger en kronologisk rækkefølge. Således kan man sikre, at en fejl som rettes forbliver rettet versionerne imellem.

Alle ændringer i kodebasen bliver, som kort omtalt i afsnittet ”Værktøjer”, repræsenteret som deltaer af filer, pakket ind i change sets. Disse change sets kan så relatere til et work item som for eksempel en ”Bug”.




*Figur 54: Eksempel på et changeset der har en Bug relation til sig. RTC holder styr på hvilke change sets, der har hvilke work items tilknyttet.*

Efter udvikleren har rettet fejlen, der er angivet i en bug, kan vedkommende så deliver sit change set til den første version, som er berørt af fejlen. Dernæst vælger han at flowe med den næste stream, som repræsenterer en anden version. Change settet vil efterfølgende dukke op under pending changes, og udvikleren kan deliver, og slutte af med at flowe tilbage til den oprindelige stream og fortsætte sin udvikling der.



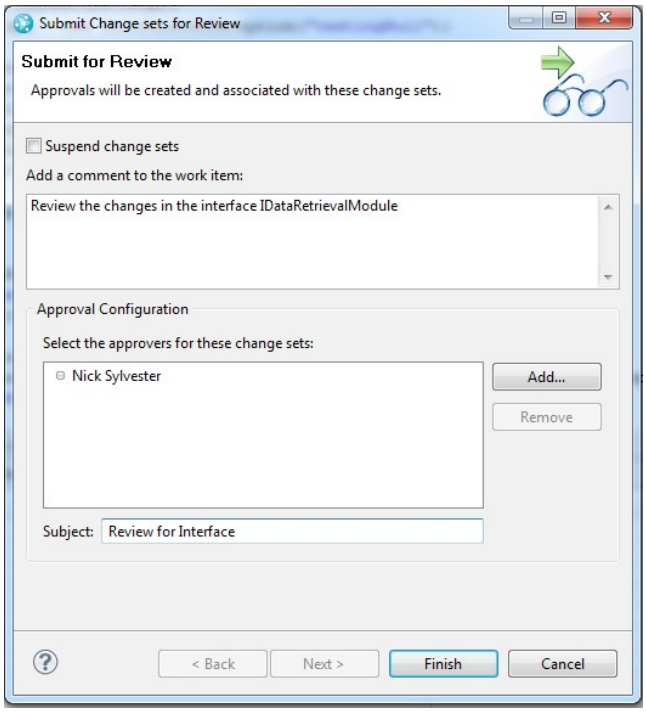

*Figur 55: Eksempel på hvordan man kan flowe med forskellige streams så ens change sets bliver lagt på begge streams, og i dette tilfælde, begge versioner af en komponent*

Revideringer sker efter stort set samme princip.

RTC har en funktion, der hedder review, som man kan føje til et change set.

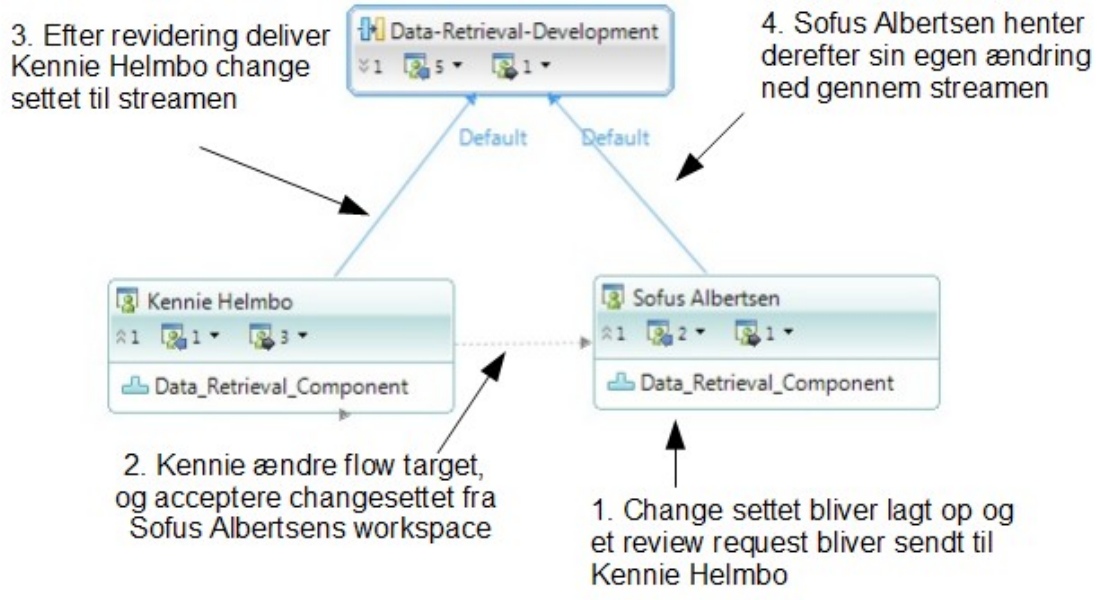
Her kan man specificere hvem, der skal lave revidering, samt forklare hvad der skal revideres på change settet. Man kan også vælge at suspende change settet. Suspend betyder, at change settet bliver lagt op på udviklerens repository workspace, men markeret og fjernet fra arbejdskopien af koden. Udvikleren arbejder herfra på koden, som den så ud før patchen blev lavet. Det er smart, hvis reviewet tager lang tid, og udvikleren skal arbejde på andre opgaver imens.

Vedkommende der skal revidere ændringen, ændrer så sit flow til udviklerens repository workspace, henter det suspendede change set ned på sin egen maskine, og begynder at revidere det.




*Figur 56: Eksempel på en review ansøgning af et change set i Eclipse*

Hvis ændringerne godtages, bliver change settet lagt op på streamen, hvor alle derefter henter det ned ved næste accept af ændringer. Følgende billed viser flowet, som det er beskrevet.




*Figur 57:Work flowet ved et review hvor den ene udvikler flower med den anden udviklers workspace for at få change settet. Efter positivt review ændrer vedkommende igen flowtarget til streamen og deliver changesettet.*

Opbygningen af delta'er, change sets, komponenter og streams gør, at vores opbygning kan tilbyde en lang række af funktionaliteter som gør kodeudvikling på tværs af versioner enkelt og ligetil. Samtidig kan funktionalitet som review understøtte den måde, som en organisation driver udvikling på, og gøre det muligt for udviklerne hurtigt at gå fra én opgave til den næste.

## Sammenfatning

Vi har i dette kapitel, vist hvordan man ved hjælp af streams, build engines og testklasser kan sikre en ensartet måde at sikre kodekvaliteten i et udviklingsmiljø. Vi har gennem builds og tilhørende rapporter vist, hvordan man nemt kan sammenkæde et funktionalitetsønske eller fejlindberetning med et bestemt build, og vist hvordan man gennem disse builds kan lave nye forgreninger af koden. Gennem en udviklers skift af stream har vi vist, hvordan man hurtigt kan rette samme fejl på tværs af forskellige versioner af samme produkt. Til sidst har vi vist, hvordan man gennem reviews kan lave revideringer som skal godkendes af andre, før de bliver lagt ind i kodebasen.

Samlet set har vi vist, hvordan man kan lave et komplet bud på continous integration, samt bud på værktøjer der kan hjælpe organisationen med at lave effektiv konfigurationsstyring på deres kode.

Der hersker formentligt ikke nogen tvivl om, at konfigurationsstyring er af afgørende betydning for store firmaer. Særligt interessant bliver det, hvis man er en virksomhed, der leverer webapplikationer, som Gmail, Facebook, Twitter osv. Her er versioneringen mere flydende, og nye

funktionaliteter kan hurtigt blive sendt ud til brugerne. Men med samme tempo kan fejl også komme ud i produktionsmiljøet. Det er derfor vigtigt med konfigurationsstyring så man hurtigt kan rulle en fejlbehæftet version tilbage, men også være i stand til at gardere sig bedst muligt mod disse fejl, inden de indtræder i produktionsmiljøet.

Men inden man går ud og investerer tid og penge[74] i et sådan setup, er det naturligt at stille sig spørgsmålet: ”Passer dette setup til mine projekter og/eller mit firma?”.

Svaret er sandsynligvis ja, så det egentlige spørgsmål er mere, hvor meget af det man kan drage nytte af. Selv små udviklingsvirksomheder har stor gavn af at få styr på deres kodebase, og implementeret nogle automatiske kvalitetssikringer i den måde de udvikler på. Hvis man udvikler kode, som er et samlet produkt, der skal ud til flere kunder, kan man drage nytte af både kvalitetssikringen, versioneringen, og muligheden for hurtig og automatisk feedback på dine ændringer.

Konfigurationsstyring er en meget kompleks disciplin at arbejde med, fordi den både spænder vidt i teknologisk avancement, og bredt i dækningen af mange forskellige områder. Men det er meget nødvendig, hvis man vil opnå og fastholde god kodekvalitet og overblik.

74 RTC er kun gratis for op til 10 udviklere. Hvis man er 11 udviklere, skal der betales fuld pris for alle 11. Samtidig findes der også open source alternativer der tilsammen har lignende funktionalitet. Til disse gælder de generelle koncepter omkring opdeling af streams og byg stadig, hvor de forklarede termer og billeder af gode grunde ikke ville være retvisende.

# Opsætning af toolboxen

Dette kapitel tjener som vores leverance tilbage til communitiet. Denne leverance skulle oprindeligt være en teoretisk vinkel på en given problemstilling. Vores bidrag kommer dog til at være en praktisk gennemgang af hvilke værktøjer vi har valgt at inkludere i vores toolbox. En toolbox der skal tjene som starthjælp til de, som gerne vil have et udviklingsmiljø der støtter dem bedst muligt. Toolboxen har en række værktøjer som giver projektet støtte indenfor kodekvalitet, overblik og versionshåndtering.

## *Læsevejledning*

I dette afsnit vil vi beskrive det udviklingsmiljø, og dets opsætning, som vi endte med efter vores projektforløb. Udover den endelige opsætning vil vi komme ind på alternativer, som vi forsøgte os med, og atter andre som kun blev overvejet at inkludere. Vi har bestræbt os på at skrive kapitlet så det kan læses selvstændigt uden at man læser resten af rapporten. Derfor vil der forekomme gentagelser og pasager fra andre kapitler, for at få meningsbærende pointer med. Derved kan denne del af rapporten forhåbentligt bruges som en vejledning til andre, der ønsker at opsætte et lignende miljø. Hvis det tages i brug på den måde, skal det understreges, at det er vigtigt at læse hele kapitlet igennem, før man prøver at efterligne opsætningen. Kapitlet er delt op i emner, og forsøgt skrevet kronologisk indenfor hvert emne. Hermed kan det forekomme, at nogle afsnit giver indtryk af, at vi har brugt nogle værktøjer eller versioner af værktøjer, som vi slet ikke har, hvis det læses ude af kontekst. Da kapitlet omhandler vores erfaringer med opsætningen af toolboxen, har kapitlet en fortællende stil.

Vi må advare om, at der er mange tekniske begreber og komplicerede funktioner i spil, så det kræver tålmodighed at komme igennem dette kapitel. Der er komplicerede fagområder i spil som kan være vanskelige at gøre letlæselige og mange termer som kan være svære at forstå. Fordi vi beskriver konkrete løsninger med konkrete værktøjer, bliver der også brugt termer, der er specifikke for disse værktøjer. Vi vil bestræbe os på at forklare disse begreber, når de opstår[75].

Vi behandler kun opsætningen af miljøet her, ikke brugen. For mere information om brugen af de enkelte værktøjer henvises til de andre kapitler i rapporten.

## *Formål*

Resultatet af vores arbejde er blevet en samling af værktøjer, som vi kalder en toolbox. Det har krævet mange måneder at gøre os bekendt med og opsætte de nye værktøjer. En udgift som, hvis det var 3 fuldtidsansatte der skulle udføre det samme arbejde, ville blive en mærkbar udgift for et mindre firma. Men som praktikanter, har vi ikke været en mærkbar byrde for Semaphors[76]

75 Hvis der er specifikke begreber indenfor RTC som ikke er beskrevet, har IBM en begrebsliste her: [40]

76 Se afsnit "Kort beskrivelse af virksomheden" i kapitlet "Beskrivelse Af Den Underliggende Opgave" for en kort beskrivelse af det firma, projektet blev udarbejdet i.

ressourcer og har på den måde kunnet give firmaet en indsigt, de ikke selv ville have haft råd til at få. Sådan en investering kan for mindre firmaer være svær at retfærdiggøre på kort sigt. Hvis firmaet derimod kan få det til at løbe rundt, mens de investerer i et forbedret udviklingsmiljø, er vi sikre på at det vil betale sig på lang sigt. Vores håb med dette kapitel er at kunne formindske udgifterne for andre, ved at beskrive de faldgrupper, afvejninger med mere, som vi har stødt på i forbindelse med denne opsætning.

Vi vurderer, at vores opsætning egner sig til små og mellemstore projekter. Afsnittet vil ikke gå i detaljer med selve installationen af de forskellige værktøjer, men fokusere på guidelines og tips til, hvad man skal være opmærksom på, når man laver den opsætning, som vi beskriver her. Vi vil også komme med en analyse af, i hvor høj grad denne opsætning ville være moden nok til et enterprise miljø[77], og komme med forslag til forbedringer i forbindelse med dens opskalering.

## *Kriterierne for udvælgelse af værktøjer*

Når vi har skullet vælge mellem forskellige værktøjer med sammenlignelige funktionaliteter, har vi haft nogle kriterier, der har bestemt hvilke, der blev valgt. Disse kriterier er baseret på følgende ræsonnement:

Da det oftest er opstartsvirksomheder der endnu ikke har lavet investeringen i deres egen toolbox, er det dem vi fokuserer på. Grundet opstartsvirksomheders natur, med få økonomiske midler, skal omkostningerne til opsætningen af en toolbox være så lave som muligt. Det betyder, i de fleste tilfælde, open source værktøjer. Til disse værktøjer vægter vi et stort community og brugervenlighed lige højt.

Emnerne kommer i denne rækkefølge; vores arbejdspladsopsætning (det vil sige klient siden), og dernæst servermiljøets opsætning. Til slut vil vi komme med en generel betragtning af, hvad der ville være af yderligere behov, hvis man skulle opskalere vores opsætning ud fra begge emner.

77 Enterprise miljø skal her forståes om et firma med tocifrede antal ansatte i udviklingen af en eller flere produkter.

## *Valg af værktøjer*

Følgende er vores fremstilling af et miljø, som kan understøtte udviklingen af forskelligartede Java projekter. Vi vil beskrive de værktøjer, som man benytter i et projekt, for at opnå bedre resultater indenfor **test**, **projektstyring** og **konfigurationsstyring**.

Hvert enkelt værktøj har sine måder at blive konfigureret på, samt krav til interaktionen med andre værktøjer. Ved hver tilføjelse af et nyt værktøj er den tid som skal bruges for at inkorporere disse til det eksisterende setup derfor stigende, fordi kompleksiteten af miljøet øges.

Der har været tre forudsætninger som firmaet har udstukket på forhånd; operativsystemet skal være Ubuntu, programmeringssproget skal være Java, og Rational Team Concert skal benyttes som versionerings og samarbejdsværktøj. Derfor vil vi ikke vægte alternativer til disse beslutninger, da sådanne ikke har været på tale. Valget af Linux gør også, at nogle af de tips vi giver kan have en anden betydning i for eksempel et Windows miljø. Vi bestræber os dog på at gøre afsnittet så operativsystemsneutralt som muligt, for at flest mulige kan drage nytte af dette kapitel.

Vi vil her komme med en kort opsummering af de værktøjer, vi vil beskrive i dette kapitel, samt de funktionaliteter de hver især opfylder.

**Eclipse** er vores IDE, der hjælper med at skrive hurtigere, mere fejlfri kode, samtidig med at det hjælper os med at holde overblik over projektet.

**RTC-Udvidelsen** giver os direkte adgang til alle RTC's funktioner, direkte i udviklingsværktøjet, og støtter derved tankegangen om en meget tæt sammenkobling mellem udviklingen af kode, og aktiviteterne i RTC.

**Maven** hjælper os til at holde styr på projektafhængigheder på et højt niveau, og gør os i stand til at lave automatiske builds gennem RTC.

**Cobertura** viser konkret, hvor vores unit tests dækker i koden, og hjælper os derved til at have en bedre kvalitetssikring i de områder af koden, hvor den gør mest gavn.

**RTC** som server applikation er hele fundamentet, som alle de andre ting bygger udenpå. Den integrerer versionering, bug tracking, projektstyring, og rapportering af selvvalgte metrikker.

**JBE** sikrer en kontinuerlig testing af vores ændringer, og hurtig tilbagemelding på de builds vi sætter op. I og med at den er helautomatisk medføre det, at test bliver gjort konsekvent og ens hver gang, der sker ændringer.

## Arbejdsstation

Arbejdsstationen er der, hvor hvert enkelt udvikler sidder og arbejder. Vi vil ikke beskrive de fysiske rammer en udvikler skal have, men i stedet klargøre hvilke udviklingsværktøjer en udvikler behøver, for at få så god en støtte som muligt.

### *Integrated development enviroment (IDE)*

Et IDE hjælper udviklerne med at skrive hurtigere, og bedre kode ved at have funktionaliteter såsom at samle klasser og ressourcer i projekter, syntaksfejl highlighting, auto completion, osv.

Der findes mange forskellige IDE'er til at lave Java udvikling med, alle sammen med de samme grundfunktionaliteter, og alle sammen med et stærkt community bag sig. Blandt de mest benyttede IDE'er til Java kan man nævne Eclipse, IntelliJ[78], Netbeans[79] og JDeveloper[80]. Forskellene mellem disse er deres opbygning, hvilke ekstra features de har, eller kan have, samt hvor god deres kobling er til eksterne applikationer. For eksempel har JDeveloper, som bliver udgivet at Oracle, ikke overraskende gode værktøjer til at tilgå deres egen database samt applikationsserver. IntelliJ, hvis udgivere ikke har egen applikationsserver eller lignende, fokuserer meget på de nyeste sprog, såsom Scala og Groovy, der begge kører på Javas virtuelle maskine.

Vores valg var stort set givet på forhånd, da Rational Team Concert kun understøtter Eclipse som IDE til javaudvikling.

Kort fortalt er Eclipse et modulært IDE, som man kan tilslutte plugins til. Eclipse kommer i mange varianter, og grundudgaven, kaldet Eclipse Classic, har ligesom alle andre udviklingsmiljøer de ovennævnte funktioner som syntaksfejl highlighting, auto completion, osv. Hvis man ønsker yderligere funktionalitet, kan man finde og installere et plugin, som indeholder den funktionalitet, man har behov for. Det kunne for eksempel være man var i gang med at udvikle en Java servlet. Dertil kan man hente et Java EE[81] plugin, som gør det muligt at oprette et webprojekt, skabe en servlet klasse indeholdende metodehovederne og afvikle programmet på en lokal server, som kan integreres i Eclipse. Derved kan hele udviklingsprocessen klares et sted. Disse plugins kan enten hentes fra nettet i zip filer, som man udpakker direkte til roden af Eclipse, eller man kan bruge det integrerede marketplace, hvor man kan søge efter, downloade og installere plugins ét samlet sted fra. På den måde kan man skabe et udviklingsmiljø, som passer lige til ens behov.

IBMs Rational Team Concert produkt er, som beskrevet i kapitlet ”Projektstyring”, bygget ovenpå Eclipse platformen. De har i den forbindelse udgivet en version af Eclipse, som udover standard funktionerne også indeholder funktioner, som understøtter brugen af RTC. Det betyder, at mange af de aktiviteter, man normalt ville lave igennem RTCs web interface, kan laves direkte i Eclipse. Det kan for eksempel være at oprette en task, få en liste af use cases, se hvad andre udviklere arbejder på, osv. RTC versionen er bygget på Eclipse Classic og mangler derfor mange af de funktioner, der er behov for til Java EE udvikling. Derudover er den Eclipse, den bygger på, 0.2 versioner bagud[82]. Da den version blev udgivet, var Java stadig på version 6, og derfor ville nogle af de nye udviklingsmuligheder, som Java 7 har, ligne syntaksfejl for det gamle IDE. Dette var et problem for os, eftersom vi netop gerne ville benytte Java 7. Grunden til dette beskrives nærmere i næste afsnit.

På baggrund af disse mangler valgte vi at bygge vores egen variant af Eclipse op ved at stykke

78 IntelliJ's hjemmeside; [56]
79 Netbeans hjemmeside;[57]
80 Jdevelopers hjemmeside; [58]
81 Java Enterprise Edition. SDK/JRE'en der understøtter kompilering og kørsel af enterprise teknologier som servlets, JSP, JSF, Enterprise Beans mm.
82 RTCs Eclipse bundle bygger på Eclips 3.5 (codename Galileo), hvor den nyeste version af Eclipse på skrivetidspunktet er 3.7 (codename Indigo).

følgende pakker sammen:

- Eclipse for Java EE[83] - Indeholder en lang række værktøjer, som letter udviklingen af Java EE projekter.
- RTC p2 bundle - Alle værktøjer fra RTC versionen kan også hentes som et plugin, som dernæst kan installeres på alle versioner af eclipse[84].
- M2E & M2E-WTP - Integrationspakker til Ecplise, som muliggør udvikling af maven projekter.

Denne opsætning gør os i stand til direkte fra IDE'et at benytte de mange teknologier, som vi har beskrevet i de forrige kapitler.

### *Java 7*

Efter omkring 3 års venten udsendte Oracle en ny version af Java den 7. juli 2011, kaldet Java 7[85]. Den nye version byder på en række forbedringer af sproget. Blandt disse forbedringer er et nyt bibliotek til at håndtere filer (java.nio), nogle avancerede trådning features, understøttelse af switch på Strings, og såkaldt syntaktisk sukker[86] som diamond operators, hvor du ikke behøver at typeinferere på instantieringen af et objekt.

Da vores udviklingsprojekt ville indeholde meget med trådning og manipulering af filer, var vi opsat på at benytte Java 7.

Det kunne argumenteres, at Java 6 ville være tilstrækkeligt til vores behov, da det har biblioteker til både håndtering af filer og tråde, som har været uændrede de sidste mange versioner, og som derfor er velkendte og veldokumenterede. Derudover betragtes Java 7, på trods af næsten et år på markedet, stadig som ny og relativ uprøvet. Det var vores vurdering, at den tid, som vi ville bruge på at sætte os ind i de nye biblioteker, ville blive vundet tilbage, og mere til, i sidste ende. Det, kombineret med et ønske om at bruge de nyeste teknologier der er til rådighed, var grundlaget for vores valg om at benytte Java 7.

Da Java er kerneteknologien som alle vores andre værktøjer bygger oven på, skal disse værktøjer også understøtte den nyeste udgave af udviklingssproget. Dette falder så tilbage til de communities, der vedligeholder de værktøjer som vi bruger. Der er som sagt gået tre år siden sidste store Java version, og mange værktøjer har virket i de år, selvom communitiet bag er faldet fra. En god huskeregel er her at se på antal opdateringer i kildekoden på projektet over det sidste halve år. Desuden vil det ikke give meget mening at lave en opsætning, som ikke følger med tiden, og derved er forældet ved tilblivelsen.

RTC udgaven af Eclipse virkede, som beskrevet før, ikke med Java 7. Den nød knækkede vi ved at inkludere de plugins der var RTC specifikke oven på den nyeste version af Eclipse for Java EE. Nu

83 Link til Eclipse for Java EE; [59]
84 Det kræver at nogle få ekstra plugins er installeret, men det er Jazz.net communitiet meget hjælpsomme til at give en.
85 For mere information om Java's historie, se; [60]
86 Syntaktisk sukker er slang for elementer i et programmeringssprogs syntaks, som kan bruges til at lave elegante løsninger, men som kan være sværere at forstå og ikke tilføjer værdi til sproget.

havde vi vores source control integreret i vores IDE og var klar til at udvikle i den ønskede version af vores valgte sprog. Det var så tid til at tilføje et build værktøj til at hjælpe os med automatisering og håndtering af projekt afhængigheder.

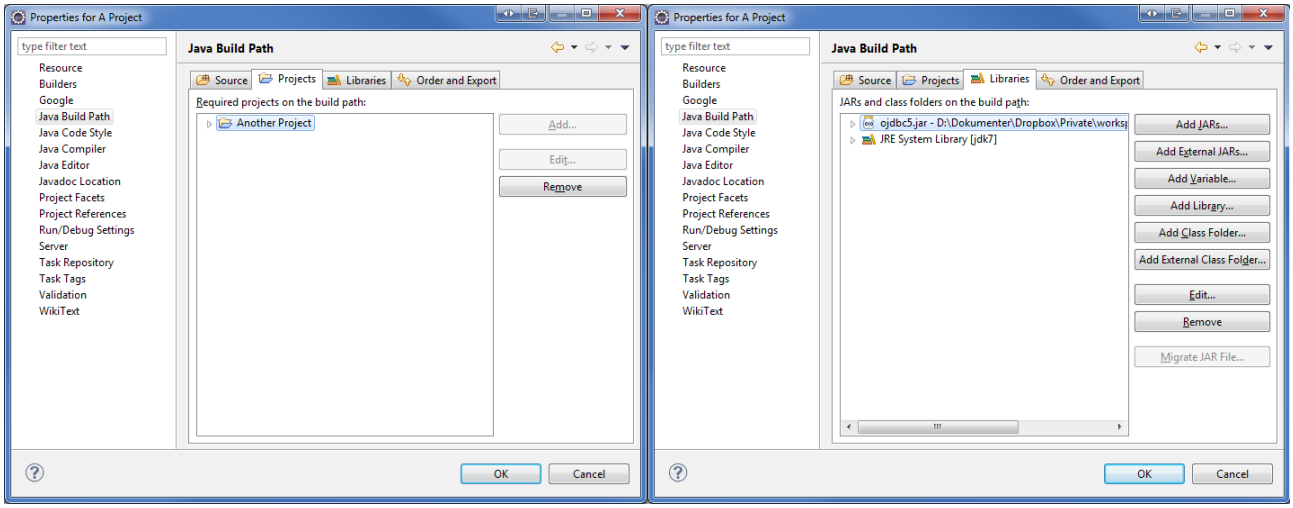
### *Projektafhængigheder*

I større projekter med mange udviklere kan det være en fordel at dele programmet op efter funktionalitet eller udviklerteams. Det letter udviklingen, når alle udviklerne ikke skal kode på tværs af hinanden, men i stedet kan fokusere på en isoleret komponent. Man mindsker også den mængde information, man skal have fremme på en gang. Hvis man arbejder på et uopdelt program, bliver man nødt til at have samtlige af programmets klasser, ressourcer, m.m. loadet ind i sit udviklingsmiljø. Ved at lave en opdeling kan man nøjes med kun at loade de ressourcer, man skal bruge til den pågældende komponent, og de klasser den indeholder. Man får også en form for kontraktbaseret udvikling, idet de forskellige teams ikke behøver at vide, hvordan de andre komponenter er implementeret, men kun skal kende interfacene[87] ved programmets snitflader. Man bliver nødt til at have en metode til at lave ændringer til interfacene og formidle disse ændringer til udviklerne. Der kan være en firma regel, som siger, at alle ændringer skal drøftes ved et projektledermøde, hvor det besluttes, om de skal med eller ej. Udover at være enig om interfaces, har man blot brug for en måde at benytte sig af de funktioner som er implementeret i de relevante komponenter.

Vi vil her kigge på to forskellige løsningsforslag til denne opgave; Eclipses indbyggede funktionalitet, og værktøjet Maven.

**Eclipse** har en indbygget funktion til at inkludere flere Eclipse projekter i ét projekt. Det er en simpel måde til at få adgang til ny funktionalitet, som man ikke selv har skrevet. Udover projekter, kan man også inkludere eksterne Java biblioteker, også kaldet jar filer, som er vist på figuren nedenfor. Det kan for eksempel være, at der skal bruges nogle funktioner til at tilgå en database. I stedet for selv at skrive det hele fra bunden, kan man inkludere en jar fil, som indeholder alle de funktioner, man skal bruge til at tilgå og modificere databasen. Denne løsning er fin nok for en enkel udvikler ved én maskine, og denne funktion har været flittigt brugt under vores uddannelsesopgaver.

87 Interface er her ikke nødvendigvis tænkt som et Java interface (dog kan det være det i nogle tilfælde), men som en måde at beskrive hvordan to komponenter skal kommunikere sammen.




*Figur 58: To forskellige udtag fra build path konfigurerings menuen for et Eclipse projekt. Til venstre ses at et andet projekt er blevet inkluderet. Til højre ses at et eksternt bibliotek er taget i brug.*

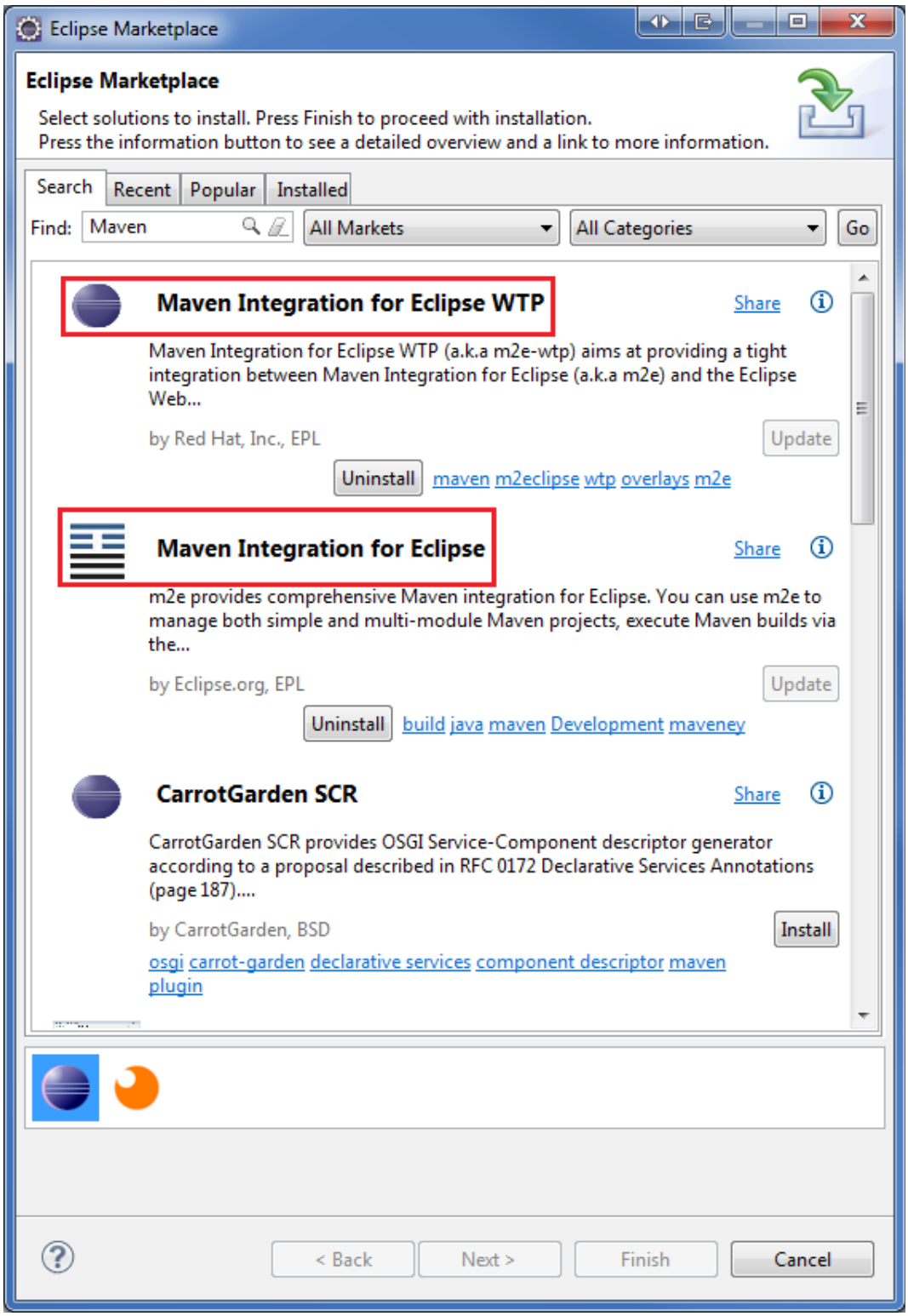
**Problemerne** med denne løsning præsenterer sig, ligeså snart man tilføjer en ekstra udvikler. Man begynder hurtigt at støde ind i problemer med absolutte stier, for eksempel \home**[user name]**\workspace\... hvilket kun er en gyldig sti for den enkelte udviklers maskine. En løsning på det kan være at oprette såkaldte miljøvariabler, som på hver enkel maskine bliver mappet over til en konkret sti på den givne computer. Ulemperne ved det er dels, at man ikke har nogen konvention til at beskrive, hvilken version af de enkelte afhængigheder man bruger, samt at man heller ikke viser, hvor en given afhængighed kan hentes henne.

**Maven** er en anden løsning. Funktionaliteten er indlejret i en POM[88] fil, og beskrives ved hjælp af XML-tagget <dependency>. Maven har selv et stort repository, hvor man kan submitte biblioteker til. Dette gør, at man ikke har behov for at kende URL'en til et givent bibliotek. Maven sørger for at hente biblioteket fra sit eget repository, og lægge den i projektets build path. Ydermere er versioneringen en grundlæggende del af Maven, og du har derfor præcis kontrol over hvilken version af de eksterne biblioteker, du bruger. Disse informationer ligger som en del af projektet, så Maven ved selv, hvad der skal hentes og hvor, når du beder den om at builde. Dette er specielt smart, hvis du har flere versioner af samme projekt kørende, som hver bruger forskellige versioner af andre projekter. Alt det kan Maven hjælpe dig med at håndtere.

Som tidligere fortalt, benyttede vi M2E pluginnet til at integrere Maven i Eclipse. Installationen var ganske ligetil. Vi benyttede Eclipses ”Marketplace” funktion og søgte efter Maven. Som vist i Figur 59, installerede vi både m2e og m2e-wtp. M2e er integrationen mellem Maven og Eclipse Classic, mens m2e-wtp er de yderligere integrationspunkter der ligger mellem Maven og Eclipse, når man har Web Tools Platform med som en af sine plugins[89].

88 For mere information om Maven og POM filer henvises til afsnittet "værktøjer" i kapitlet "Konfigurationsstyring", eller til Mavens hjemmeside.

89 WTP er et projekt som Eclipse for Java EE inkluderer i deres variant af Eclipse.




*Figur 59: Installation af m2e og m2e-wtp fra Eclipse Marketplace.*

Pluginnet har sin egen indlejret version af maven, så du er klar til at benytte Maven i Eclipse, så snart pluginnet er installeret. Vi valgte at installere maven som stand alone direkte på operativsystemet for at kunne få fuld adgang til dens kommandolinjeværktøj. Det gjorde vi, fordi vi derved selv havde styr på versionen af Maven, samt bedre var i stand til at kunne identificere om det var Maven eller pluginnet der fejlede, hvis der skete uforudsete hændelser. Installationen af Maven er lige til, og Apache har en udførlig guide til hvordan du skal installere det på både Windows og Unix[90].

Pluginnet går ind og ændrer mange af de forud definerede project dependencies i Eclipse. Maven har en fast måde at lægge både klasser og ressourcer i et projekt på. Default strukturen med at lægge source filerne direkte i /src mappen og lægge klasse- og andre outputfiler i /bin mappen bliver ikke længere brugt. Det bliver overskrevet med Mavens mappestruktur, der adskiller kildekode og testkode.

```
my-app
|-- pom.xml
`-- src
    |-- main
    |   `-- java
    |       `-- com
    |           `-- mycompany
    |               `-- app
    |                   `-- App.java
    `-- test
        `-- java
            `-- com
                `-- mycompany
                    `-- app
                        `-- AppTest.java
```

*Figur 60: Mappestrukturen for maven projektet "my-app".*

Derudover går Maven nu på bestemte steder ind og ændrer den måde, Eclipse normalt kører på. Maven får ansvaret med at kompilere, samt holde styr på build path, som ovenfor beskrevet. Det kræver, at man tænker anderledes end normalt. Man skal sørge for, at test klasserne ligger det rigtig sted, så de alle sammen bliver kørt, når man kører sin kode. I den forbindelse, skal man også huske at bruge en af Maven kørslerne[91] i stedet for den typiske Eclipse run knap. Vi brugte ”Maven Test”, når vi blot skulle sørge for at koden overholdt vores unit tests og ”Maven Install”, når vi skulle lave en ny pakke med opdaterede ændringer, som de andre projekter kunne gøre brug af.

Som beskrevet i ”Beskrivelse af den Underliggende Opgave” opdelte vi vores udviklingsprojekt i

90 [61], nedenfor de forskellige download links. Det er rigtig vigtigt at huske at sætte Path, så operativsystemet ved, hvor Maven ligger henne.

91 En grundigere gennemgang af de forskellige Maven kørsler kan findes i afsnit "Maven Life Cycle" i kapitlet "Konfigurationsstyring".

flere moduler. Disse moduler blev udviklet i hver deres Eclipse projekt. Det betød, at vi fik en masse duplikering af ressourcer, fordi der var nogle, som skulle bruges i flere projekter. Vi valgte derfor at samle disse i et projekt (kaldet Common), som de andre projekter så inkluderede. Her brugte vi også Maven. Ligesom at tilføje en dependency til et ethvert andet Java bibliotek, tilføjede vi en afhængighed til vores Common projekt. Den del af POMen for Engine ser således ud:

```
<dependency>
	<groupId>dk.semaphor.sa</groupId>
	<artifactId>common</artifactId>
	<version>0.0.4-SNAPSHOT</version>
</dependency>
```

På den måde åbner vi også op for at migrere vores udvikling til et andet miljø, idet det ikke længere er bundet op på Eclipses build path konfigurering.

Det, at Maven er et uafhængigt kommandolinjeværktøj, gør også, at vi kan bruge det til vores build engine. Dette emne beskriver vi nærmere i afsnittet ”Server-side”.

Valget mellem disse to alternativer har vi begrundet således: Både Maven og Eclipse er open source og har et stort og sundt community bag sig. Begge har også mange års udvikling bag sig, der gør dem til solide valg i firma sammenhæng.

Vores valg blev Maven, fordi det giver os en række konkrete fordele overfor den integrerede løsning som man finder i Eclipse[92]. For det første er man ikke længere bundet til Eclipses måde at håndtere projekterne på[93], men kan bruge forskellige IDE'er, hvis man lyster. For det andet er dens måde at håndtere projektafhængigheder på meget mere avanceret, med fokus på versionering, nedhentning af biblioteker, samt maskineuafhængighed opsætning. Endelig er Mavens funktion i forhold til build engine nærmest uundværlig.

### *Code coverage*

I forbindelse med vores fokus på test, havde vi brug for en måde, hvorpå vi kunne måle vores test dækning. Vi kunne selvfølgelig udregne dækningen i hånden, men dette vil dels være ualmindeligt tidskrævende, hvilket igen ville føre til at dette simpelthen ikke ville blive gjort. Samtidig findes der værktøjer, der automatisk kan lave disse udregninger for dig.

Både HP[94] og IBM[95] har store testværktøjer hvis målgruppe er store udviklingshuse, med mange hundrede udviklere på enkelte store softwareprojekter. Disse platforme er både alt for avancerede og dyre til det behov, som små firmaer har. Hvis vi kigger på værktøjer der integreres med Eclipse,

92 Der findes mange værktøjer der kan det samme som Maven. For en hurtig gennemgang af disse henvises til kapitlet "Konfigurationsstyring".

93 Da Eclipse er en af de største spillere på markedet, har blandt andre Jdeveloper og Netbeans importfunktioner, der kan transformere sådanne projekter om til deres respektive opbygninger. Med Maven ville dette ikke være nødvendigt, og man ville kunne arbejde transparent i alle IDE'er der har understøttelse af Maven.

94 For mere information om HP's produkt; [62]

95 [26]

og hvis kernefokus er på code coverage til Java, finder vi Clover[96], Emma[97], JMockit[98] og Cobertura[99]. Vi kommer hurtigt med en gennemgang af disse forskellige værktøjer og vores vurdering af dem.

**Clover** er den eneste i feltet, der ikke er open source. Faktisk koster det 300 dollars (~1.700 kr.) pr. maskine, eller 2200 dollars (~12.900 kr.) for 10 maskiner, eller servere. Skønt Clover har et aktivt firma bag sig til at udvikle produktet, stemmer både licensen og prisen dårligt overens med vores kriterier for udvælgelse. Clover blev derfor ikke valgt.

**Cobertura** blev startet i 2005, og er i skrivende stund på version 1.9.4. Den seneste udgivelse skete marts 2010, og communitiet bag er ikke aktivt mere, hvilket gør at det ikke har understøttelse for Java7.

**Emma** er det ældste open source værktøj som vi har taget med[100]. Udviklingen af det holdt op i 2005-6, og har derfor de samme begrænsninger som Cobertura. Samtidig med er deres oversigt ikke nær så letlæselig som Cobertura, og vi har derfor kun behandlet det overfladisk ved at søge information på internettet.

**JMockit** er som beskrevet i kapitlet ”Test” oprindeligt set blot et Mocking framework, men har derudover også biblioteker til at måle code coverage. Vi prøvede det hurtigt, for at se om det kunne håndtere Java7, men fik i mindst én klasse nogle tal, som var stærkt upålidelige, både hvad angik hvor mange linjer der reelt var i koden, og hvor stor dækning der var. Det var ret graverende, at værktøjet havde optællingsfejl, og vi valgte ikke at bruge mere tid på at finde ud af, om man kunne ændre i opsætningen for at hjælpe JMockit med at forstå koden.

Da vi første gang valgte Cobertura, vidste vi ikke, at det manglede support for Java7. Det følgende afsnit er skrevet kronologisk, og beskriver hvordan vi brugte værktøjet i vores opsætning.

Som beskrevet i afsnit ”Cobertura” i kapitlet ”Test”, producerer Cobertura en HTML-fil, som grafisk illustrerer hvilke linjer af koden, der er afviklet under kørslen af Unit testene. Udover HTML-filerne bliver der også produceret en .ser-fil. Denne fil kan via pluginnet E-Cobertura linkes med ens kode, for at man direkte i IDE'et kan se hvilke linjer kode, der er berørt af test, og hvilke der ikke er. Vi valgte dog at gå væk fra pluginnet. Problemet er, at pluginnet ikke kan sættes op til automatisk at tage informationer fra en given .ser-fil. Ej heller starter pluginnet med at lede i projektets rod, men i brugerens rodmappe, på linux /home/[user name], som gør det meget tidskrævende at skulle bruge værktøjet. Vi endte med at bruge det via tabs i vores browser, hvor man havde en tab åben med stien til de HTML-filer, der blev genereret. Derved skulle man bare opdatere siden og gå ind i den bestemte klasse, man havde arbejdet i for at se resultatet. Det strider imod vores ønske om at have så meget som muligt integreret i vores udviklingsmiljø, især når det rent faktisk er muligt at gøre det, men det var i vores øjne simpelthen ikke besværet værd.

Udover dette ”kosmetiske” problem, stødte vi ind i et meget mere seriøst problem i forbindelse tilføjelsen af Cobertura til vores værktøjspalette. Cobertura har ikke haft aktivitet på deres kode i

96 For mere information om Clover, henvises til [63]
97 For mere information om Emma, henvises til [64]
98 For mere information om JMockit, henvises til [65]
99 For mere information om Cobertura, henvises til [66]
100 Hvornår det præcist startede, har vi ikke kunne finde ud af, men det blev først registreret på sourceforge i maj 2004.

over et år[101]. Det gør at der ikke er lavet nogle ændringer til koden, der kan imødekomme de nye tilføjelser i Java 7. Fordi Cobertura arbejder med at instrumentere bytekoden[102], kan den ikke finde ud af at blive afviklet under den nye version af Java. Det betød helt konkret, at vi måtte rykke til Java 6 for alle subprojekter. GUI projektet kørte dog videre på Java 7, da der allerede der var sket en stor del udvikling, som var bygget op på Java 7[103], og eftersom der ikke blev brugt unit testing, var det unødvendigt at have Corbertura i det projekt.

Under tilblivelsen af denne rapport er vi stødt på et andet projekt, som efter sigende er fuldt integreret med Java7[104]. Projektet hedder EclEmma, er open source og er under aktiv udvikling. Projektet startede som et Eclipse plugin til værktøjet Emma, men skiftede omkring 2009 over til deres eget værktøj, kaldet JaCoCo (stående for Java Code Coverage).

JaCoCo

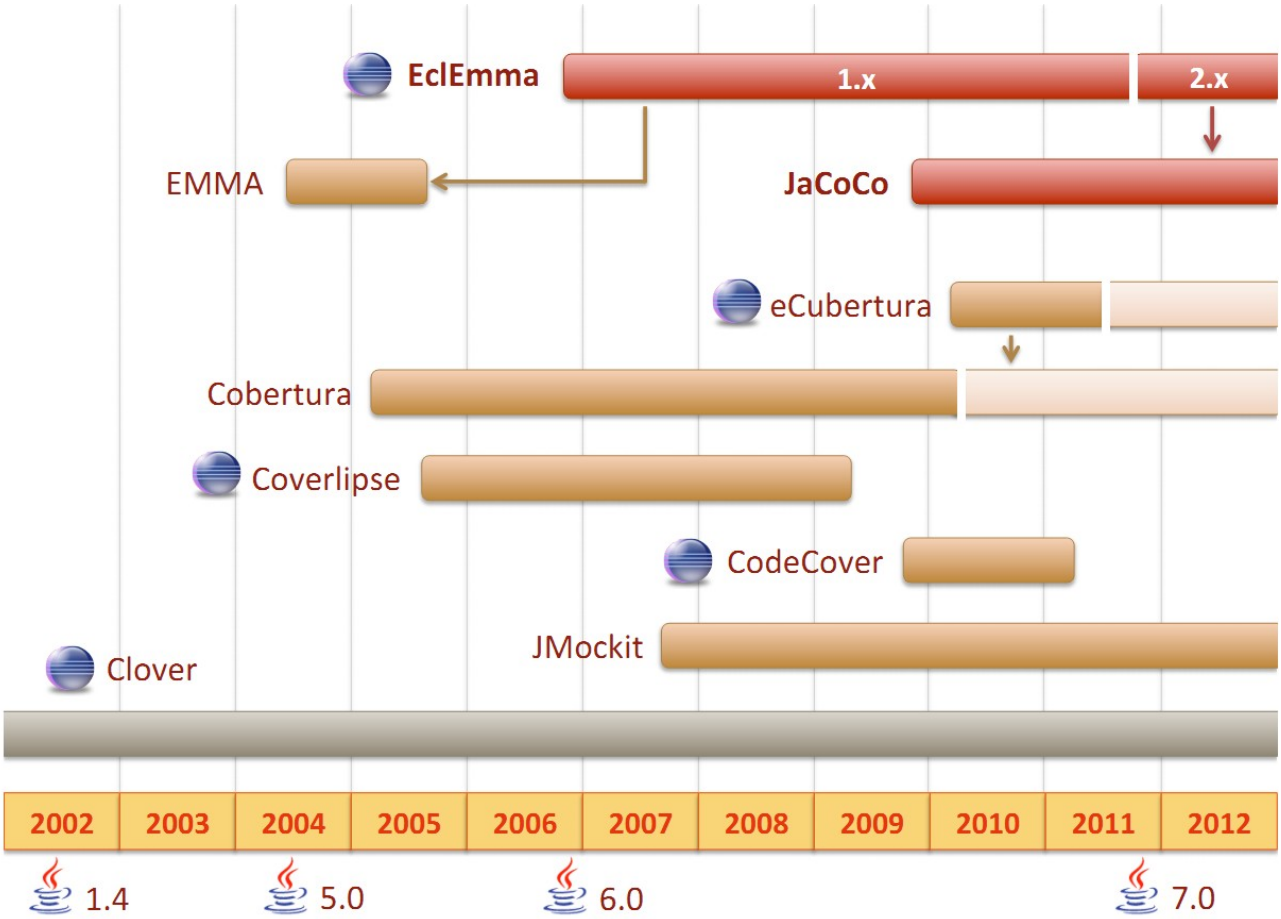
**JaCoCo**

| Element | Missed Instructions | Cov. | Missed Branches | Cov. | Missed | Cxty | Missed | Lines | Missed | Methods | Missed | Classes |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| org.jacoco.agent.rt | | 79% | | 78% | 21 | 80 | 42 | 191 | 12 | 53 | 2 | 12 |
| org.jacoco.core | | 98% | | 100% | 26 | 702 | 30 | 1,649 | 24 | 465 | 0 | 76 |
| jacoco-maven-plugin | | 90% | | 78% | 18 | 75 | 12 | 163 | 2 | 37 | 0 | 4 |
| org.jacoco.report | | 99% | | 98% | 7 | 445 | 9 | 1,162 | 2 | 305 | 0 | 56 |
| org.jacoco.ant | | 99% | | 99% | 3 | 126 | 7 | 403 | 2 | 88 | 0 | 16 |
| org.jacoco.agent | | 85% | | 75% | 3 | 11 | 5 | 33 | 1 | 7 | 0 | 1 |
| Total | 405 of 14,353 | 97% | 38 of 941 | 96% | 78 | 1,439 | 105 | 3,601 | 43 | 955 | 2 | 165 |

*Figur 61: Eksempel på visning af code coverage i JaCoCo*

En af JaCoCo's udviklere ved navn Marc R. Hoffmann holdt på EclipseCon 2012 et foredrag om EclEmma og JaCoCo, hvor han havde lavet en tidslinje der viser projekternes kodeaktivitet kronologisk med javaversionerne[105].

101 For en uddybende graf, se [67]
102 For en uddybende forklaring, se afsnit "Cobertura" i kapitlet "Test"
103 Se d. 12/3 i "Bilag 2 - Dagbog" på CD'en.
104 De har endnu ikke lavet understøttelse for de nye dynamiske variabeldelklæringer som JVM'en tilbyder til dynamiske sprog. Men alt hvad du skriver i Java7 er understøttet.
105 Billedet er taget fra hans slides som finde her; [68]




*Figur 62: Grafik over hvilke code coverage værktøjer der er på markedet, og hvornår de aktivt er blevet udviklet på hver især.*
*Kilde: [68]*

Hvis vi kigger på udviklingsgraferne som hvert værktøj har, kan vi se at der kun er tre der er under stadig aktiv udvikling; JaCoCo, JMockit og Clover[106]. Eftersom Clover koster penge, blev det fravalgt. JMockit er også under aktiv udvikling, men vores test af det viste, at man ikke i vores tilfælde kunne stole på de metrikker, der blev vist. En så graverende fejl gjorde, at det også blev fravalgt. Derved er der kun ét af værktøjerne som reelt kan have en chance; JaCoCo.

Anbefalingen gives med den note at vi ikke selv har brugt det, men vores råd til andre ville være at starte med at kigge på dette værktøj før andre. Hvis det ikke opfylder de givne kriterier, står vores anbefaling mellem Clover, hvis du har råd, og Cobertura, hvis man kan leve med kun at have Java 6 support.

Vi har nu beskrevet hvilke konkrete værktøjer, vi har anvendt i vores opsætning. Ud over disse har

106 Både Cobertura og E-Cobertura ser ud som om de har en form for udvikling på grafen, men denne udvikling er ikke at se i deres respektive repositories, og derfor må begge værktøjer anses for ikke-aktive.

vi selvfølgelig brugt teknologier som mail, chat og VoIP[107], men da disse i opsætningen ikke er andet end at klikke sig igennem en installations setup eller gå ind på en hjemmeside og tilmelde sig, vil vi ikke behandle dem her.

## Server-side

Et centralt placeret kodestyringssystem er alfa omega for en organisation med udviklere[108]. Det samme gælder for build værktøjer, der ud over at lave færdigpakkede eksekverbare filer, også kan bruges til at rapportere fejl[109]. Begge er med til at styrke programmøren indenfor kodekvalitet, overblik og versionshåndtering.

I dette afsnit vil vi behandle to ting; Rational Team Concert som server applikation, og Jazz Build Engine (JBE) som er en del af IBM's tilføjelser til RTC.

Grundet at RTC var givet til os på forhånd, vil vi ikke gå ind og diskutere alternativer her, kun beskrive vores opsætning. Det samme gør sig gældende for JBE. Skønt der findes mange build værktøjer som er open source, eksempelvis Hudson[110] og Jenkins[111], og som har en langt større brugerbase bag sig[112], vil vi benytte JBE fordi integrationen til RTC klart synes at være det nemmeste.

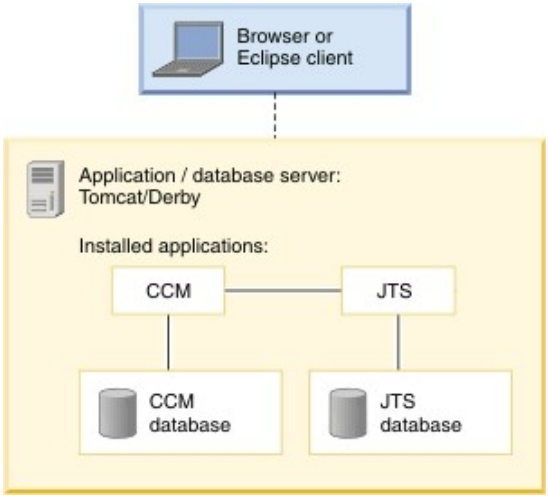
### *RTC*

IBM har lavet en opsætningsguide, som viser den mest hensigtsmæssige måde at sætte sit miljø op, der afvejer performance mod vedligeholdelse, alt efter hvor mange der skal bruge RTC. De har delt det ind i tre hoved topologier; Evaluation, Departmental og Enterprise[113]. Semaphor kommer både nu og i nærmeste fremtid ikke til at have mere end ti javaudviklere. Derfor har vi valgt at køre med en Evaluation topologi, der dels er den nemmeste at administrere, og som også går ind under IBMs 10 gratis udviklerlicenser pr. organisation. Vi vil her beskrive Evaluation topologien; Departmental og Enterprise vil blive behandlet i kapitlets sidste afsnit.

107 Voice over IP.
108 Undtagelsen er GIT, der er et decentralt kodestyrringsværktøj som virksomheder også kan bruge. Dog er der, set ud fra et forretningsmessigt synspunkt, en god pointe i at have koden et sted hvor man har mulighed for at tage reglmæssige backups, og kan danne sig overblik, samt andre former for governance.
109 Denne del vil ikke blive behandlet i dette kapitel. Vi henviser kapitlerne "Konfigurationsstyring" og "Test" for mere information. Hvis disse kapitler ikke er tilgængelige bliver emnet generelt behandlet i [OOSE], kapitel 13: Configuration Management.
110 For mere information om Hudson, henvises til [69]
111 For mere information om Jenkins, henvises til [70]
112 Bloggen her snakker om Hudson vs. Jenkins, men kommer også ind på hvor mange der skriver plugins mm. til de to applikationer: [71]
113 For dybere forklaring på de forskellige topologier, se [72]

**Evaluation:**

Denne topologi kører normalt med Tomcat som servlet container, og Derby som database[114].
Både Tomcat og den indbyggede Derby er bundlet med i installationen af RTC, og kan derfor nemt blive installeret på serveren. Da Derby databasen ikke er en "enterprise ready" database, har IBM valgt at lægge et loft på 10 udviklere for, at brugerne ikke skal opleve for dårlig performance, hvilket går meget godt i spænd med IBM's licensering om op til 10 udviklere er gratis på platformen.




*Figur 63: RTC's evaluation topologi hvor hele RTC er samlet på en server, for bedste afvejning af pris og performance for små udviklingsfirmaer*

Vi prøvede, efter ønske fra Semaphor, at installere en DB2 database for at mindske eventuelle performance problemer med Derby, men måtte efter mange forgæves forsøg gå tilbage til den embeddede database. Om denne udfordring skyldtes, at vi kørte på et ikke supporteret operativsystem, eller en fejl i konfigurationen, fandt vi aldrig ud af. Det skal dog også siges, at vi ikke har kunne mærke derby databasen som flaskehals på noget tidspunkt, og vi kan derfor godt anbefale bare at arbejde med den som database, hvis man er under 10 udviklere.

RTC er en Java EE applikation, og den kan derfor køre på alle de platforme, som har en servlet container. IBM har support for alle nyere Windows server versioner, samt Red Hat- og Suse Enterprise Linux Server. Semaphors installation kører på en Debian server uden problemer, og det samme må formodes med langt de fleste andre Linux-distributioner[115].

Da RTC er så meget andet end kun versionering af kode, fylder det langt mere end SVN, og kræver langt mere CPU samt ram. Den fremtidige version af RTC, 4.0, kommer til at kræve to server-cpu kerner, 4 GB ram, samt et 64 bit operativsystem, som minimum[116]. Til sammenligning bruger Semaphors gamle SVN-server kun 256 mb, og en 1GHz CPU, hvor den i dvale kun brugte få MHz. Det er vigtigt at huske på, hvis man vil sætte sin egen server op, at ens operativsystems arkitektur skal passe til ens applikation, da 64 bit software ikke kan installeres på et 32 bit operativsystem.

Det er vigtigt, at man gør sig klart hvilke funktionaliteter, man har brug for. Hvis man kun søger funktionalitet indenfor kodeversionering er RTC skudt helt over målet. Der vil GIT, eller SVN være langt billigere i hardware, og læringskurven sandsynligvis mindre stejl. For langt de fleste vil det være tiden det tager, at lære et nyt system at kende, der koster langt de fleste penge for firmaet.

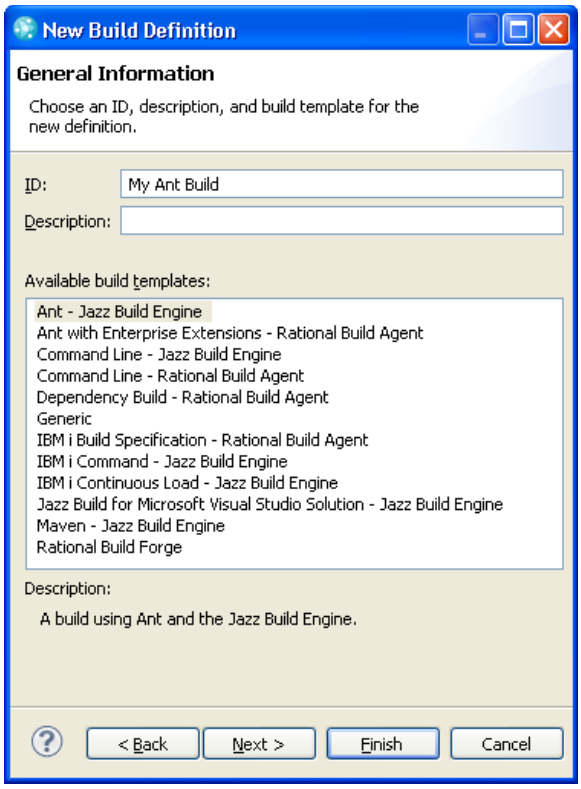
RTC er meget mere end dette, og for at få fuld udbytte af det, skal man bruge kombinationen af de forskellige funktioner; versionering, bugtracking, planlægning, build enginge osv.

114 Både Tomcat og Derby er produkter fra Apache Foundation. En fuld liste af deres projekter kan ses her: [73]
115 Debian, Suse og RHEL er tilsammen de distributioner som langt de fleste andre distributioner lægger sig op ad. Fx er Ubuntu en fork af Debian, og deler derved systemopbygning, pakkehåndtering mm.
116 For en komplet gennemgang af systemkrav, se [74]

### *Build*

IBM's build engine til RTC hedder som før beskrevet Jazz Build Eninge (JBE). JBE har integration til ANT, Maven, IBM's eget Rational Build Forge, samt alle andre build automation værktøjer, der kan interageres med gennem kommandolinjen.

Vi har brugt builds til dels at sikre, at den  kode vi lagde op altid blev testet, selv hvis den enkelte udvikler en gang imellem glemte at gøre det, og for at sikre at der ikke var nogle skjulte konfigurationer, der gjorde, at noget der virkede hos en ikke virkede hos de andre[117].
Vores byggeinstans blev lagt på samme server som RTC for nemhedens skyld, dels hvad angår opsætning, og vedligeholdelse af operativsystemet, men også ud fra den tro, at det var nemmere at vide, at der kun var én server, man skulle fejlsøge på. Det viste sig imidlertid at vi gentagende gange har måtte genstarte RTC delen, fordi byggedelen af serveren krævede det. Det er ikke andet end et lille irritationsmoment, når man sidder i samme lokale og kan koordinere med alle ved at snakke sammen, samt gøre det i den fælles frokostpause. Omvendt er det fuldstændig uacceptabelt, hvis man kørte med store udviklingsafdelinger, som vi vil skitsere i enterprise afsnittet.

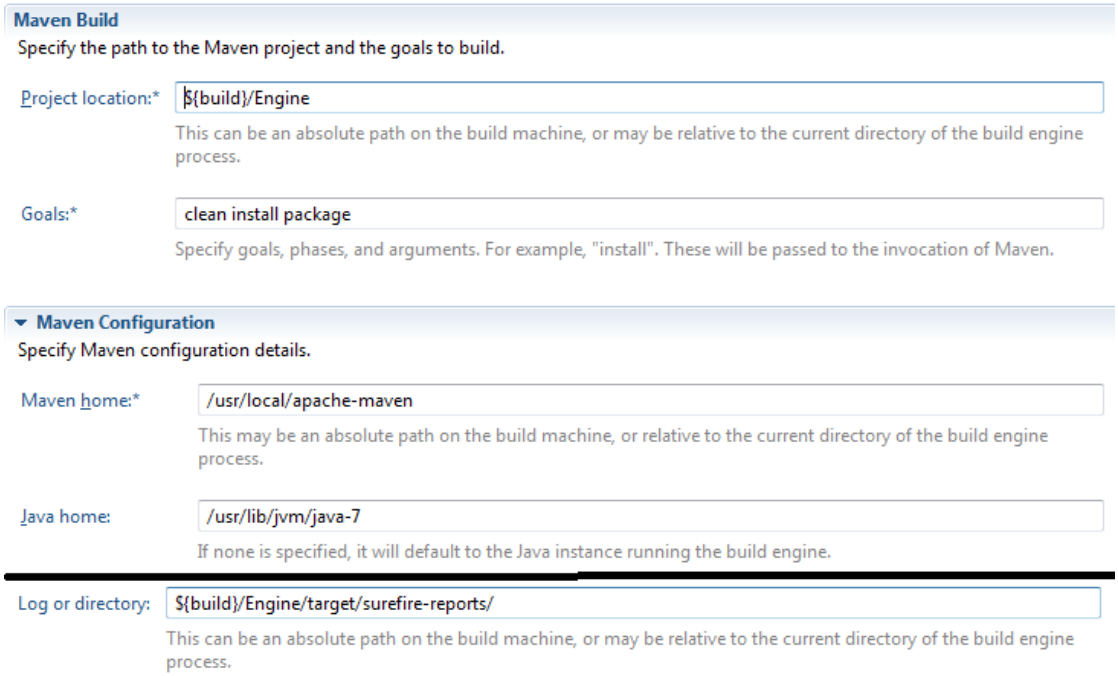



*Figur 64: Eksempel på eclipse wizzard til at lave build defitioner med. Her kan man vælge hvilken build template man vil have udfra hvilket build automation tool man har valgt at bruge i sit projekt.*

Da build enginen skal ”efterligne”, hvad en arbejdsstation ville gøre for at bygge koden, kræver det, at den har de samme værktøjer til rådighed. Derfor installerede vi Maven, samt Java 7 på serveren. Måden du installerer disse værktøjer på afhænger i høj grad af dit operativsystem, og vi vil derfor ikke gå nærmere i dybden med dette her. Selve JBE kan man installere via samme installationspakke som RTC serveren selv, eller du kan hente en pakke med programmet på Jazz.net. Efter JBE er startet på din server, skal du ind og kæde den sammen med RTC, som en build engine.

Når det er sket kan du gå i gang med at definere de forskellige builds du skal have; et per stream der skal buildes, plus et pr. integration mellem de forskellige streams.

117 For mere information, se kapitlet "Konfigurationsstyring" i rapporten.




*Figur 65: Maven indstillinger i en given build defition. man skal udspecificere hvor projektet ligger, hvilke Maven faser der skal køres, Hvor Maven og Java ligger installeret på serveren, samt hvor JUnit resultaterne ligger*

Opsætningen her er ligetil, så længe du er opmærksom på, hvor JBE lægger dit projekt. IBM fokuserer primært på at rette JBE mod Ant og Buildforge, og man kan mærke, at der ikke er gjort meget for at gøre det lettere for de folk, der bruger Maven. Et eksempel er afrapportering af JUnit tests fra et build. Da Maven bygger på konvention over konfiguration er det helt fastlagt, at de XML-filer som skal bruges til at lave afrapporteringen ligger i [Project location]/target/surefire-reports/. Men det skal man selv vide, samt plotte ind. Det tager ikke lang tid at taste det ind, men det tager lidt tid at gætte sig frem til, hvilke filer JBE leder efter.

Enginen skal også kunne se de samme ressourcer som udviklerne bruger i testøjemed. Så hvis du tester forbindelsen til en database, skal denne database også være synlig for build enginen, og derved dens server.

Vores anbefaling er at have build engine og server skilt ad, da kravene til de to servere er meget forskellige. Hvor RTC skal være klippe stabil og have en høj oppetid, skal build engine være i stand til at kunne konfigureres hurtigt efter behov og kunne klare at gå ned, uden at det påvirker andet. Vi har været ude for, at engine gjorde RTC ustabil, da den kørte en test, der ved en fejl loopede og derved tog langt mere ram end normalt.

## Opsummering på client- og serverside

Vi vil her komme med en kort opsummering af de værktøjer, vi har beskrevet, samt de

funktionaliteter de hver især opfylder.

**Eclipse** er vores IDE, der hjælper med at skrive hurtigere, mere fejlfri kode, samtidig med at det hjælper os med at holde overblik over projektet.

**RTC-Udvidelsen** giver os direkte adgang til alle RTC's funktioner, direkte i udviklingsværktøjet, og støtter derved tankegangen om en meget tæt sammenkobling mellem udviklingen af kode, og aktiviteterne i RTC.

**Maven** hjælper os til at holde styr på projektafhængigheder på et højt niveau, og gør os i stand til at lave automatiske builds gennem RTC.

**Cobertura** viser konkret, hvor vores unit tests dækker i koden, og hjælper os derved til at have en bedre kvalitetssikring i de områder af koden, hvor den gør mest gavn.

**RTC** som server applikation er hele fundamentet, som alle de andre ting bygger udenpå. Den integrerer versionering, bug tracking, projektstyring, og rapportering af selvvalgte metrikker.

**JBE** sikrer en kontinuerlig testing af vores ændringer, og hurtig tilbagemelding på de builds vi sætter op. I og med at den er helautomatisk medføre det, at test bliver gjort konsekvent og ens hver gang, der sker ændringer.

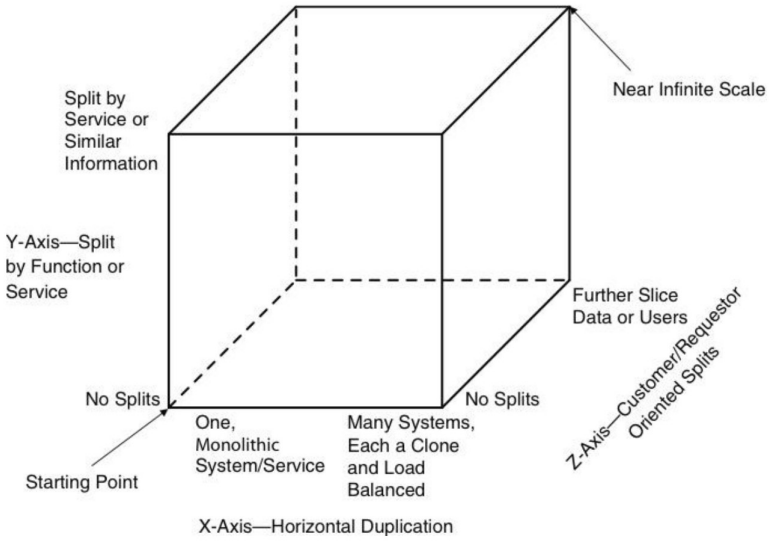
Ud over disse har vi brugt Skype, XMPP chat, WIKI og mail til at udfylde de funktionaliteter indenfor vidensdeling og kommunikation, som RTC ikke har indbygget. Vi vil ikke beskrive disse ting nærmere, da installationen er triviel og ikke har nogle opsætningsmæssige udfordringer.

## *Opskalering af vores Toolbox*

Vi ved nu af erfaring, at vores toolbox kan bruges af et hold på 3 udviklere. Men hvad med større projekter? Hvor vil evt. flaskehalse være, og hvordan skal skaleringen foregå, hvis det skulle vise sig nødvendigt? Det er disse spørgsmål vi prøver at afdække med dette afsnit.

## Metode

For at identificere hvilken grad af opskalering de forskellige værktøjer i toolboxen besidder, tager vi udgangspunkt i Abbot og Fishers kube[118].
Kuben skal hjælpe en systemudvikler med at finde ud af, hvordan de forskellige dele af systemet skal kunne skalere. Kuben består af 3 akser; X, Y og Z. Hver akse beskriver en egenskab i opskalering, og det er muligt, at et system kan have egenskaber på flere akser samtidigt.
X aksen beskriver en kloning af den samme instans for at kunne håndtere et øget brug. Det kunne fx være flere kasser i et supermarked. Hver kasse er en kloning af de andre og har derved samme funktioner og mål, som de andre.




*Figur 66: Viser de forskellige akser i Abbot og Fishers kube, samt forklaringer på hvad aksernes egenskaber er.*

Y aksen beskriver en deling i forhold til specialisering af opgaver. Det svarer til en separat bagerafdeling, eller kiosk, der kun sælger bestemte varer som resten af supermarkedet ikke sælger.
Z aksen beskriver en opdeling i forhold til brugeren af systemet. Det svarer til hurtigkasser i visse supermarkeder, hvor folk med mindre end et vist antal ting kan stille sig i kø og blive hurtigere betjent.

## Klientside

På klientsiden er det forholdsvist enkelt at beskrive opskaleringen, da der her skaleres i forhold til X aksen på kuben. Det vil blot være at lave mange dubletter af de samme udviklermaskiner. Administrationen af disse kunne lettes ved enten at have Landscape[119] for Ubuntu-Linux maskiner, eller have Windows server med policy management i et windowsmiljø. Det bliver dog vigtigt at have nogle klare definerede arbejdsprocesser. Vi har kunnet tillade os at være mindre disciplinerede med for eksempel oprettelse og vedligeholdelse af work items i RTC, samt rettelser i kontrakterne mellem komponenterne, fordi vi kun har været 3 mennesker, som har siddet i samme lokale. Det betød, at nogle informationer var nemmere at videregive verbalt. Den går ikke, når man sidder mange udviklere fordelt på flere afdelinger og lokationer, og det ikke er muligt at kommunikere sammen fysisk/face-to-face. Der bliver det vigtigt at alle følger de retningslinjer, som organisationen har opstillet, for at informationer kan flyde frit.

118 [TAoS], kap. 22.
119 Landscape er Cannonical's (Firmaet bag Ubuntu) enterprise desktop-management værktøj. Link: [75]

Vores beskrevne opsætning ville i et rent Java udviklingshus være tilstrækkeligt. Men i næsten alle tilfælde hvor man har med store udviklingsafdelinger at gøre, har man også C#/.NET udvikling. Til det har IBM også et plugin til RTC, så man direkte i Visual Studio har de samme interaktionsmuligheder med RTC, som man har i Eclipse. En liste over alternativer til de redskaber, som udgør vores toolbox, kan findes på ”Bilag 1 - Værktøj alternativer”. Til decideret webudvikling, samt andre sprog, findes der varianter af Eclipse, der er beregnet til disse. Efterfølgende kan man bare lægge de plugins, som man skal bruge fra RTC ovenpå, og på den måde opnå den samme funktionalitet som vi har med Java.

## Server-side

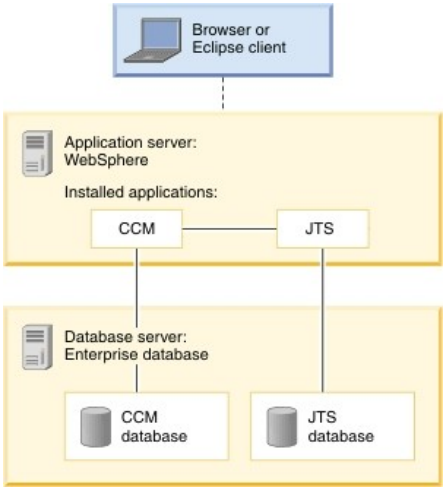
En skalering af servermiljøet er anderledes i forhold til klientsiden, fordi det her gælder om at have få centraliserede ressourcer som klienterne kan arbejde op imod. Vi vil, ligesom i afsnittet om vores egen opsætning, her tage fat på RTC og build.

### *RTC*

IBM har som tidligere omtalt lavet en opsætningsguide, som viser den bedste måde at sætte sit miljø op på, målt på performance kontra vedligeholdelse og pris, alt efter hvor mange der skal bruge RTC. Den har de som sagt delt ind i tre hovedtopologier[120]; Evaluation, Departmental og Enterprise:

**Departmental:**

I denne topologi har man lavet en dedikeret applikations og database server. Derved opskalerer man gennem Y aksen. Fordelen er her, at man kan have flere brugere på samme servermiljø, uden at performance går for meget ned. Desuden kan man skræddersy de to servere til hver deres brug; databaseserveren med masser af disk og ram, applikationsserver med masser af ram og CPU kerner. Til slut kan der være nogle gode grunde til kun at udstille applikations serveren i DMZ[121], mens DB ligger internt, uden adgang til omverdenen andet end gennem applikationsserveren.



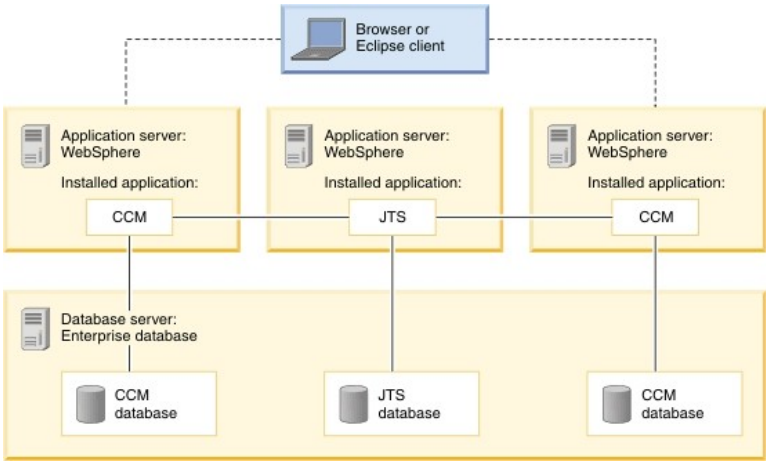

*Figur 67: RTC's Departmental topologi, hvor database of applikationsserven er adskilt for øget hastighed.*

120 Der er med vilje ikke sat nogen brugerantal på departmental og enterprise topologierne, fordi det i høj grad kommer an på hvilken hardware som serverne køre på. IBM har lavet en sizing test, og har i den forbindelse beskrevet nogle anbefalinger:[76]

121 De Militarized Zone. Betegnelse for den del af et firmas IT infrastruktur der ikke står bag en firewall, og derved kan rammes direkte udefra. DMZ bruges til alle de servere der skal være offentlige tilgængelige som for eksempel

**Enterprise:**

Her skalerer man både på X, og på Y, ved dels at have multiple instanser af CCM, en dedikeret JTS, og en  dedikeret DB server.
Fordelen her er, udover at have mere regnekraft til rådighed også at have mulighed for at have forskudte servicevinduer på de to CCM servere.
Dette kan være en fordel hvis man har distribueret udvikling, uden dog at have follow the sun[122] arbejde. Derved kan man udnytte de timer der er stilstand på et projekt til at lave service på maskinerne[123].




*Figur 68: RTC's Enterprise topologi, hvor applikationerne er splittet ud på flere servere. Dette gøres for at kunne håndtere flere brugere.*

RTC kan naturligt vokse alt efter behov, da man her kan have en kombination af X og Y akse skalering i forhold til kuben. Selve opskaleringen er mulig, fordi IBM netop har lavet RTC skalerbart fra starten. Hvilken skalering der bruges afhænger i høj grad af topologien, der passer til ens behov.

### *Build*

Store systemer tager ofte timer at bygge. Derfor har man dedikerede byggeservere, hvis eneste formål er at køre alle de tests, der er skrevet op imod koden, og returnere resultatet.
Hver byggeserver kan have ansvaret for en eller flere byggedefinitioner, som beskriver, hvad der skal hentes, og hvad der skal testes, samt hvad der skal ske efterfølgende med koden. Hvert enkel delprojekt i et stort projekt bliver bygget og kørt, for derefter at gå videre gennem SCM'en til integrationsbyg, først på subsystemniveau, og senere hen på systemniveau.

En opskalering her vil være en kombination af X og Y skalering, alt efter hvordan det gøres. Hvis alle byggeserverne er ens, og i princippet kan det samme, er det en klar X skalering. Hvis man derimod har klynger af byggeservere, der for hver gruppe har forskellige egenskaber, og bruges til at bygge på forskellige tidspunkter i forløbet, vil det være en kombination af X og Y.

Mange firmaer, der ikke producerer shrinkwrapped[124] software, bruger det sidste skridt i deres byggeprocess til at flytte koden over til en eller anden form for slutbrugertest, for derefter at flytte

mail- og webservere.

122 Follow the sun er en betegnelse for en arbejdsorganisering hvor der bliver arbejdet hele døgnet på et givent projekt, ved at distribuere arbejdet ud på tre eller flere geografisk separerede lokalisationer.

123 JAZZ teamet beskriver det selv således: "With the read permission and scalability support in 3.0, you can now host many projects on the same server. But where does this end, will all 30,000 developers in IBM end up running on one giant server in the cloud? We don't think so." link:[76] afsnittet "multiple repositories".

124 Generel betegnelse for software som man køber fysisk i en butik, der tidligere havde meget fastlåste versioner såsom windows 95, og 98, hvor opdateringer kom i form af cd'er man kunne købe efterfølgende.

det til produktion. Derfor er byggemiljøet af høj betydning, fordi det i så stor grad indgår i test og modning af koden.

## *Sammenfatning og perspektivering*

Vores toolbox kan håndtere alle de vigtigste aspekter i et udviklingsforløb; kvalitetssikring, projektstyring og konfigurationsstyring. Dette gør den uafhængig af metodik og opgave, og kan derfor passe på alle udviklingsprojekter, uanset deres process. Vi har også vist hvordan toolboxen kan blive tilpasset, og derved være kernen i et projekt der vokser. Samtidig har vi givet råd og vejledning til andre, der vil prøve at implementere disse værktøjer. Vi har også givet vores erfaringer med afvejning af forskellige værktøjer op mod hinanden, og beskrevet deres styrker og svagheder set ud fra de kriterier vi har opstillet. Toolboxen kan forhåbentligt være med til at mindske indlæringskurven, og tjene som en guide til at bestemme, om et givent værktøj kan det, som vedkommende forventer af det.

Vi ser ingen hindringer i at en lille virksomhed bruger dette udgangspunkt, hvis de samtidig gerne vil have en mulighed for senere at opskalere. Det har vi vist gennem RTCs måde at skalere på, vores udvalg af standardværktøjer som Eclipse, og Maven, der også bliver brugt i langt større firmaer, samt erkendelsen af, at denne toolbox sagtens kan fyldes op med flere værktøjer, så snart behovet for nye funktionaliteter opstår.

# Konklusion

Vi har i denne rapport beskæftiget os med at sammensætte en toolbox, der kan hjælpe ethvert Java udviklingshold med at udvikle kvalitets applikationer, indenfor en fast defineret tidsramme, som er nemme at vedligeholde.

Sammensætningen af relevante udviklings værktøjer og etableringen af et udviklingsmiljø er noget som alle organisationer skal igennem. Men hvorfor skal nye organisationer starte fra bunden hver gang? Hvorfor ikke samle en række informationer som gør det nemmere for sådanne organisationer at vælge konkrete værktøjer baseret på deres behov? Primært fordi det tager tid at tilegne sig viden indenfor dette emne, dels fordi informationerne er spredt, og hver organisation har sin egen process med sine egne sæt af værktøjer.

Udfordringen ligger i at der ikke findes én måde at løse en problemstilling på. For hver problemstilling som kodestyring, unit test, code coverage med mere findes der en stor mænge værktøjer. Alle med deres fordele og ulemper. De forsøger alle at være relevante for udvikleren, men virker forskelligt og har forskellige opsætningsprocesser.

Vi giver her, med vores toolbox, et eksempel på hvordan det kan gøres. En toolbox som både kan anvendes i sin helhed, eventuelt med tilføjelser, eller kun delvist.

Vi har kigget på det generelle problem, fundet en generel løsning, samt givet vores bud på en konkret opbygning af specifikke værktøjer, baseret ud fra vores vurdering af dem i daglig brug.

Toolboxen fokuserer på de steder, hvor støtte til udviklingen er vigtigst, nemlig i håndteringen af projektstyringsprocesser, kvalitetssikringsprocesser og versionsstyringsprocesser.

Vi har vist, hvordan man effektivt med automatiserede værktøjer kan udføre unit tests i hverdagen, for at opnå større sikkerhed i sin kode. Med hjælp fra det rigtige mocking framework kan man isolere ens test fra resten af ens kode og derved i langt højere grad identificere, hvor fejlen ligger, hvis en sådan opstår. Med code coverage værktøjer kan man hurtigt gå ind og se hvilke dele af koden, der er utestet. Derved hjælper værktøjet med, at vise hvor der kan være brug for yderligere test. På den måde får man bedre testdækning.

I kapitlet ”Projektstyring” viste vi hvordan man ved hjælp af RTC, uafhængigt af udviklingsmetode, kan skræddersy projektet til ens behov og effektivt bruge det til at monitorere udviklingsforløbet. Samtidig giver RTC mulighed for høj agilitet ved at kunne skræddersy både rollerne og selve udviklingsprocessen, mens den er i gang. Vi giver også et bud på, hvordan virksomheden kan opretholde en kollektiv samling af viden ved brug af en wiki, der understøtter samarbejde og vidensdeling.

Konfigurationsstyring er en vigtig, men tit overset disciplin. Vi har vist, hvordan man ved hjælp af en fast defineret opbygning kontinuerligt kan sikre kvaliteten af kode og hele tiden være ajour med projektets helbred. Samtidig demonstrerede vi, hvordan sporbarhed gennem hele udviklingen kan hjælpe til med at skabe overblik. Ved at koble work items med change sets kan hver enkel linje kode henføres til en fejlrapport eller et change request indeholdende et ønske om ny funktionalitet

fra kunden. Dette kan også bruges i kunderelationen til at give større gennemsigtighed i udviklingsprocessen og give kunden et redskab til selv at involvere sig direkte i udviklingen.

Vores toolbox er, med sin åbenhed for til- og fravalg, ikke hængt op på en bestemt metodik. Den kan bruges af både iterative og inkrementelle metodikker, som vi har prøvet med XP og Open UP, men også sekventielle metodikker som vandfald m.m. Hvis man har en proces, der for eksempel ikke bruger test, og sikrer kodekvalitet på anden vis, med for eksempel kontraktbaseret udvikling, kan man blot fjerne dette element fra toolboxen og bruge resten.

Vores toolbox er ikke blot et sæt værktøjer som er tilfældigt valgt. Toolboxen er et resultat af en udskillelsesproces, som er foregået gennem måneders arbejde med en konkret case som grundlag. Vi arbejdede med hvert enkelt framework, afvejede dette i forhold til dets brugbarhed og funktionalitet, og skiftede det ud med andet, hvis ikke det kunne opfylde vores krav.

Som vi konkluderede i kapitlet "Opsætning" kan toolboxen også opskalere og derved vokse dynamisk sammen med firmaets projekter, der benytter den.

Vi mener, at toolboxen kan være kernen i et udviklingsmiljø, sammen med andre moduler/værktøjer hægtet på, alt efter den enkelte organisations eller projekts behov. Samtidig er de enkelte dele udvalgt sådan, at toolboxen kan tilpasse sig dynamisk efter projektets karakter og metodik. Vi håber, at denne rapport kan give andre det indblik i værktøjer og frameworks, der kan hjælpe til med at højne kvalitet, overblik og styring af leverancer i et udviklingsforløb.

# Litteraturliste

## Bøger

Formattet vi har benyttet til at liste bog referencerne er som følger:

Forfatter. (år). Titel: Undertitel (udgave). Udgivelsessted: Forlag. ISBN:[nummer]

Udgave skrives kun på listen, hvis det ikke drejer sig om 1. udgave.

**[FoST]** Dorothy Graham, Erik Van Veenendaal, Isabel Evas, Rex Black. (2009). Foundations of Software Testing (Revised Edition). England: Cengage Learning EMEA. ISBN: 978-1-84480-989-9

**[JiA]** Petar Tahchiev, Felipe Leme, Vincent Massol, Gary Gregory. (2011). JUnit in action (second edition). Stamford: Manning Publications. ISBN: 978-193-518-202-3

**[OOSE]** Bernd Bruegge, Allen H. Dutoit. (2010). Object-Oriented Software Engineering: Using UML, patterns, and Java (3. udgave). Upper Saddle River: Pearson Education. ISBN: 978-0-13-815221-5

**[EoTA]** Dorothy Graham , Mark Fewster. (2012). Experiences of Test Automation: Case Studies of Software Test Automation. Boston: Addison-Wesley. ISBN: 978-032-175-406-6

**[TAoS]** Martin L. Abbott, Michael T. Fisher. (2009). The Art of Scalability: Scalable Web Architecture, Processes, and Organizations for the Modern Enterprise. Boston: Addison-Wesley ISBN: 978-013-703-042-2

## Internet ressourcer

Der kan findes kopier af de refererede hjemmesider på den vedhæftede CD i .pdf format. Stien er: ”~/Web Pages”, hvor "~" er roden på mediet. Filnavnene svarer til nummereringerne i dette dokument. F.eks. Indeholder "1. Apache Maven.pdf" en kopi af Wikipedia siden om Maven. For at åbne filerne, kræves der Adobe Acrobat Reader. Det kan hentes gratis her: http://get.adobe.com/uk/reader/.

Formattet vi har benyttet til at liste referencerne er som følger:

[Ophav] ([seneste opdatering[125]]). [*Titel*]. Sidst besøgt d. [besøgsdato]
URL: [url]

125 Det skal bemærkes, at mange moderne hjemmesider benytter dynamisk indhold. Det vil sige, at siden bliver opdateret hver gang den åbnes. I de tilfælde hvor opdateringsdatoen fremgår af siden, er denne naturligvis blevet brugt, ellers har vi brugt datoen for indførelsen i denne liste.

1. Statens Arkiver (2012, 4. jun). *Arkivering af data efter bekendtgørelse nr. 1007 om arkiveringsversioner*. Sidst besøgt d. 4 juni 2012
   URL: http://www.sa.dk/content/dk/for_statslige_myndigheder/aflevering/it-systemer/aflevering_efter_bekendtgorelse_nr_1007
2. Statens Arkiver (2012, 22. maj). *Vejledning til bekendtgørelse om arkiveringsversioner*. Sidst besøgt d. 4 juni 2012
   URL: http://www.sa.dk/media%284185,1030%29/Vejledning_til_bekendtg%C3%B8relse_1007_maj2012.pdf
3. IBM (2005, 24. maj). *Which style of WSDL should I use?* Sidst besøgt d. 12 maj 2012
   URL: http://www.ibm.com/developerworks/webservices/library/ws-whichwsdl/
4. Oracle (). *Java Architecture for XML Binding (JAXB)*. Sidst besøgt d. 4 juni 2012
   URL: http://www.oracle.com/technetwork/articles/javase/index-140168.html
5. Matt Terski (2009, 23. mar). *Writing Use Case Extensions*. Sidst besøgt d. 4 juni 2012
   URL: http://blog.casecomplete.com/post/Writing-Use-Case-Extensions.aspx
6. Mountain Goat Software (2009, 17. dec) *The Forgotten Layer of the Test Automation Pyramid.* Sidst besøgt d. 22 maj 2012
   URL: http://www.mountaingoatsoftware.com/blog/the-forgotten-layer-of-the-test-automation-pyramid
7. Don Wells (2012, 4. jun). *Unit Tests*. Sidst besøgt d. 4 juni 2012
   URL: http://www.extremeprogramming.org/rules/unittests.html
8. Mike Clark (2006, 20. feb). *JUnit FAQ*. Sidst besøgt d. 4 juni 2012
   URL: http://junit.sourceforge.net/doc/faq/faq.htm
9. Martin Fowler (2012, 24. maj). *Xunit*. Sidst besøgt d. 4 juni 2012
   URL: http://www.martinfowler.com/bliki/Xunit.html
10. Kent Beck (2012, 4. jun). *Simple Smalltalk Testing: With Patterns*. Sidst besøgt d. 4 juni 2012
    URL: http://www.xprogramming.com/testfram.htm
11. Wikipedia (2012, 1. jun). *List of unit testing frameworks*. Sidst besøgt d. 4 juni 2012
    URL: http://en.wikipedia.org/wiki/List_of_unit_testing_frameworks
12. Wikipedia (2012, 31. mar). *Kent Beck*. Sidst besøgt d. 5 juni 2012
    URL: http://en.wikipedia.org/wiki/Kent_Beck
13. TestNG (2012, 7. apr). *TestNG*. Sidst besøgt d. 5 juni 2012
    URL: http://testng.org/doc/index.html
14. Mark Doliner (2012, 5. jun). *Cobertura*. Sidst besøgt d. 5 juni 2012
    URL: http://cobertura.sourceforge.net/
15. Elliotte Rusty Harold (2005, 3. maj). *Measure test coverage with Cobertura*. Sidst besøgt d.

5 juni 2012
URL: http://www.ibm.com/developerworks/java/library/j-cobertura/

16. Stack Overflow (2009, 26. maj). *Branch Coverage*. Sidst besøgt d. 5 juni 2012
URL: http://stackoverflow.com/questions/908123/branch-coverage
17. The British Computer Society (2007, 25. maj). *A Survey of Coverage-Based Testing Tools*. Sidst besøgt d. 5 juni 2012
URL: citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.101.3739&rep=rep1&type=pdf
18. Martin Fowler (2007, 2. jan). *Mocks Aren't Stubs*. Sidst besøgt d. 5 juni 2012
URL: http://martinfowler.com/articles/mocksArentStubs.html
19. jMock.org (2010, 2. jul). *Mocking Classes with jMock and the ClassImposteriser*. Sidst besøgt d. 5 juni 2012
URL: http://www.jmock.org/mocking-classes.html
20. JMockit developer team (2012, 2. apr). *The JMockit Testing Toolkit*. Sidst besøgt d. 5 juni 2012
URL: http://jmockit.googlecode.com/svn/trunk/www/about.html#coverage
21. Szczepan Faber (2008, 14. jan). *Mockito*. Sidst besøgt d. 5 juni 2012
URL: http://monkeyisland.pl/2008/01/14/mockito/
22. JMockit developer team (2011, 15. maj). *MockingToolkitComparisonMatrix*. Sidst besøgt d. 5 juni 2012
URL: http://code.google.com/p/jmockit/wiki/MockingToolkitComparisonMatrix
23. Michelle Levesque og Jason Montojo (2010, 5. nov). Sidst besøgt d. 5 juni 2012
URL: http://www.crosstalkonline.org/storage/issue-archives/2005/200501/200501-Levesque.pdf
24. Black Duck Software, Inc (2012, 5. jun). *Your Guide to Open Source*. Sidst besøgt d. 5 juni 2012
URL: http://www.ohloh.net/
25. Black Duck Software, Inc (2012, 5. jun). *Compare Projects*. Sidst besøgt d. 5 juni 2012
URL: http://www.ohloh.net/p/compare?project_0=Mockito&project_1=jMock+2&project_2=JMockit
26. IBM (2012, 30. maj). *Rational Quality Manager*. Sidst besøgt d. 6 juni 2012
URL: https://jazz.net/products/rational-quality-manager/
27. The Standish Group (2012). *The Standish Group Blog*. Sidst besøgt d. 6 juni 2012
URL: http://blog.standishgroup.com/
28. Deborah Hartmann Preuss (2006, 25. aug). *Interview: Jim Johnson of the Standish Group*. Sidst besøgt d. 6 juni 2012
URL: http://www.infoq.com/articles/Interview-Johnson-Standish-CHAOS
29. Wikipedia (2012, 5. jun). *Waterfall model*. Sidst besøgt d. 6 juni 2012
URL: http://en.wikipedia.org/wiki/Waterfall_model

30. The Eclipse Foundation (2012, 30. maj). *Introduction to OpenUP.* Sidst besøgt d. 6 juni 2012
URL: http://epf.eclipse.org/wikis/openup/

31. Alistair Cockburn (2000, 1.jan). *Just-in-time methodology construction*. Sidst besøgt d. 6 juni 2012
URL: http://alistair.cockburn.us/Just-in-time+methodology+construction

32. Robert Bogue (2005, 22 mar). *Cracking the Code: Breaking Down the Software Development Roles.* Sidst besøgt d. 26 maj 2012
URL: http://www.developer.com/mgmt/article.php/3490871/Cracking-the-Code-Breaking-Down-the-Software-Development-Roles.htm

33. IBM (2012, 6. jun). *Getting started with your projec*t. Sidst besøgt d. 6 juni 2012
URL: http://pic.dhe.ibm.com/infocenter/clmhelp/v3r0/index.jsp?topic=%2Fcom.ibm.team.concert.doc%2Ftopics%2Ft_getting-started.html

34. Ramon Ray (2008, 4. dec). *Beyond Wikipedia: Using Wikis for Collaboration & Project Management*. Sidst besøgt d. 6 juni 2012
URL: http://smallbiztechnology.com/archive/2008/12/wikis.html/

35. Panagiotis Louridas (2006, marts/april). *Using Wikis in Software Development*. Sidst besøgt d. 6 juni 2012
URL: http://kiwiwiki.co.nz/pmwiki/uploads/Technology/Software/Using%20Wikis%20in%20software%20development.pdf

36. Wikipedia (2012, 26. maj). *Comparison of wiki software*. Sidst besøgt d. 6 juni 2012
URL: http://en.wikipedia.org/wiki/Comparison_of_wiki_software

37. JAMwiki (2012, 27. maj). *JAMWiki*. Sidst besøgt d. 6 juni 2012
URL: http://jamwiki.org/wiki/en/JAMWiki

38. MediaWiki.org (2012, 3. jun). *MediaWiki*. Sidst besøgt d. 6 juni 2012
URL: http://www.mediawiki.org/wiki/MediaWiki

39. Dropbox (2012, 6. jun) *Dropbox*. Sidst besøgt d. 6 juni 2012
URL: https://www.dropbox.com

40. IBM (2012, 6. jun). *Rational Software glossary*. Sidst besøgt d. 6 juni 2012
URL: http://pic.dhe.ibm.com/infocenter/clmhelp/v3r0/index.jsp?topic=/com.ibm.team.concert.doc/topics/glossary.html

41. The Apache Software Foundation (2012, 6. jun). *Lifecycle Reference*. Sidst besøgt d. 6 juni 2012
URL: http://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html#Lifecycle_Reference

42. Gradleware (2012, 6. jun). Gradle.org. Sidst besøgt d. 6 juni 2012
URL: http://www.gradle.org/

43. The Apache Software Foundation (2012, 6. jun). *Apache Buildr*. Sidst besøgt d. 6 juni 2012

URL: http://buildr.apache.org/

44. The Apache Software Foundation (2012, 23. maj). *Apache Ant*. Sidst besøgt d. 6 juni 2012 URL: http://ant.apache.org/
45. Wikipedia (2012, 6. maj). *Convention over configuration*. Sidst besøgt d. 6 juni 2012 URL: http://en.wikipedia.org/wiki/Convention_over_Configuration
46. Christoph (2010, 20. feb). *Maven vs. Ant: Stop the Battle*. Sidst besøgt d. 6 juni 2012 URL: http://javamoods.blogspot.com/2010/02/maven-vs-ant-stop-battle.html
47. The Apache Software Foundation (2012, 6. jun). *Apache Subversion*. Sidst besøgt d. 6 juni 2012 URL: http://subversion.apache.org/
48. Wikipedia (2012, 6. jun). *Concurrent Versions System*. Sidst besøgt d. 6 juni 2012 URL: http://en.wikipedia.org/wiki/Concurrent_Versions_System
49. Software Freedom Conservancy (2012, 6. jun). *Git*. Sidst besøgt d. 6 juni 2012 URL: http://git-scm.com/
50. Wikipedia (2012, 12. maj). *Software configuration management*. Sidst besøgt d. 6 juni 2012 URL: http://en.wikipedia.org/wiki/Software_Configuration_Management
51. Andrew R. Freed (2011, 29. apr). *Comparing concepts between Subversion and Rational Team Concert*. Sidst besøgt d. 6 juni 2012 URL: https://jazz.net/library/article/639/
52. IBM (2012, 6. jun). *Rational Build Forge family*. Sidst besøgt d. 6 juni 2012 URL: http://www-01.ibm.com/software/awdtools/buildforge/
53. Agile Alliance (2007, 7. aug). *Principles behind the Agile Manifesto*. Sidst besøgt d. 6 juni 2012 URL: http://agilemanifesto.org/principles.html
54. Martin Fowler (2006, 1. maj). *Continuous Integration*. Sidst besøgt d. 6 juni 2012 URL: http://martinfowler.com/articles/continuousIntegration.html
55. Selenium (2012, 6. jun). *Selenium*. Sidst besøgt d. 6 juni 2012 URL: http://seleniumhq.org/
56. JetBrains.com (2012, 6. jun). *IntelliJ IDEA 11 - The most intelligent Java IDE*. Sidst besøgt d. 6 juni 2012 URL: http://www.jetbrains.com/idea/
57. NetBeans (2012, 6. jun). *NetBeans IDE*. Sidst besøgt d. 6 juni 2012 URL: http://netbeans.org/
58. Oracle (2012, 6. jun). *Oracle JDeveloper*. Sidst besøgt d. 6 juni 2012 URL: http://www.oracle.com/technetwork/developer-tools/jdev/overview/index.html
59. The Eclipse Foundation (2012, 6. jun). *Eclipse IDE for Java EE Developers*. Sidst besøgt d. 6 juni 2012

URL: http://eclipse.org/downloads/packages/eclipse-ide-java-ee-developers/indigor

60. Wikipedia (2012, 6. jun). *Java SE 7*. Sidst besøgt d. 6 juni 2012
URL: http://en.wikipedia.org/wiki/Java_version_history#Java_SE_7_.28July_07.2C_2011.29
61. The Apache Software Foundation (2012, 6. jun). *Download Maven*. Sidst besøgt d. 6 juni 2012
URL: http://maven.apache.org/download.html
62. Hewlett-Packard Development Company (2012, 6. jun). *HP Application Lifecycle Management Consulting*. Sidst besøgt d. 6 juni 2012
URL: http://www8.hp.com/uk/en/software/software-product.html?compURI=tcm:183-937009&pageTitle=application-lifecycle-management-consulting
63. Atlassian (2012, 6. jun). *Clover*. Sidst besøgt d. 6 juni 2012
URL: http://www.atlassian.com/software/clover/overview#
64. Vlad Roubtsov (2006, 15. jan). *EMMA: a free Java code coverage tool*. Sidst besøgt d. 6 juni 2012
URL: http://emma.sourceforge.net/
65. JMockit developer team (2012, 6. jun). *JMockit*. Sidst besøgt d. 6 juni 2012
URL: http://code.google.com/p/jmockit/
66. Mark Doliner (2012, 6. jun). *Cobertura*. Sidst besøgt d. 6 juni 2012
URL: http://cobertura.sourceforge.net/
67. Sourceforge (2012, 6. jun). *Cobertura*. Sidst besøgt d. 6 juni 2012
URL: http://sourceforge.net/projects/cobertura/stats/scm?repo=SVNRepository&dates=2011-05-11+to+2012-05-11
68. Marc R. Hoffmann (2012, 28. mar). *Code coverage revised*. Sidst besøgt d. 6 juni 2012
URL: http://www.eclipsecon.org/2012/sites/eclipsecon.org.2012/files/20120320%20EclEmma%20on%20JaCoCo.pdf
69. Hudson (2012, 6. jun). *Hudson*. Sidst besøgt d. 6 juni 2012
URL: http://hudson-ci.org/
70. Jenkins (2012, 6. jun). *Jenkins*. Sidst besøgt d. 6 juni 2012
URL: http://jenkins-ci.org/
71. Bob Bickel (2011, 12. mar). *Jenkins vs. Hudson - Time to Upgrade!*. Sidst besøgt d. 6 juni 2012
URL: http://bobbickel.blogspot.dk/2011/03/jenkins-vs-hudson-time-to-upgrade.html
72. IBM (2012, 6. jun). *Upgrading to a fully distributed enterprise topology*. Sidst besøgt d. 6 juni 2012
URL: http://pic.dhe.ibm.com/infocenter/clmhelp/v3r0/index.jsp?topic=%2Fcom.ibm.jazz.install.doc%2Ftopics%2Ft_upgrade_rtc2_distributed.html

73. The Apache Software Foundation (2012, 6.jun). *Alphabetical Index*. Sidst besøgt d. 6 juni 2012
URL: http://projects.apache.org/indexes/alpha.html

74. IBM (2012, 7. jun). CLM Workbench System Requirements
URL: https://jazz.net/wiki/bin/view/Main/CLMWorkbenchSystemRequirements

75. Canonical (2012, 6. jun). *Make the most of Ubuntu with Landscape*. Sidst besøgt d. 6 juni 2012
URL: https://landscape.canonical.com/

76. Dave Schlegel, Grant Covell, Jean-Michel Lemieux (2010, 23. nov). *Rational Team Concert 3.0 sizing guide*. Sidst besøgt d. 6 juni 2012
URL: https://jazz.net/library/article/551

## *Forsidebilleder*

- config - 4 people 2 computers - http://www.advisorone.com/2012/05/23/kickstarter-for-wall-street-broker-dealers-get-int
- proj-mgmt - real peeps - http://www.impressionmedia.net/projectcentre
- test - real quality assurance dude - http://www.chapterthree.com/blog/matt_cheney/drupal_development_best_practices_techniques_part_ii