DATALOGISK INSTITUT
DET NATURVIDENSKABELIGE FAKULTET
AARHUS UNIVERSITET

Hovedopgave

Diplom i Informationsteknologi
linien i Softwarekonstruktion

Refaktorisering kontra omskrivning af legacy kode

14. juni 2010

Kim Ginnerup, studerende
Jakob Marager, studerende

Elmer Sandvad vejleder

Datalogisk Institut
Aarhus Universitet
Åbogade 34
8200 Århus N

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

Indhold

1 Indledning

Evaluere refaktorisering af legacy kode i forhold til funktionel omskrivning.

Grundlaget for rapporten er en eksisterende web-service som i dag leverer online lagerstatus. Problemet med systemet er at det ikke længere benyttes som oprindeligt designet. I det oprindelig designoplæg var der lagt op til en funktion som skulle kaldes få gange om dagen med få varer. I dag bliver den kaldt mange gange om dagen og med mange flere varer per kald. Dette giver performance problemer.

2 Motivation

Der bliver talt meget om at refaktorisere legacy kode, men er der noget vundet ved det?
Er der indikatorer som på forhånd kan afgøre om refaktorisering er den bedste vej?
Der kan være mange årsager til at man vil omskrive kode. Vi har valgt at tage udgangspunkt i, at man har besluttet sig til, at refaktorisere en Webservice for at opnå bedre performance.

3 Hypotese

Det er hurtigere og kvalitetsmæssigt bedre at lave en funktionel konvertering af en eksisterende kode end at lave en fuld refaktorisering. Det er derimod hurtigere at benytte refaktorisering, hvis det kun er et del-område af koden som skal ændres.

4 Problemstilling

4.1 Baggrund

Det system vi tager udgangspunkt i, er et kørende system, som har været i drift i flere år.
Systemet er skrevet som en Web service der leverer online lagerstatus for varer. Web servicen bygger ovenpå et ERP system ved navn [Unistar] som er udviklet af Bording Data. [Unistar] ERP har eksisteret siden 1987 og bliver stadig udviklet og vedligeholdt. Systemet var oprindeligt udviklet som et Client/Server system med Ingres DBMS [Ingres] som den underliggende database. Programmerne blev udviklet i ABF (Application By Forms) et karakter baseret 4GL udviklingssystem som ejes af Ingres Corporation [Ingres]. I dag er der stadig ABF programmer, men systemet har spredt sig over flere platforme, alt efter de behov der har vist sig. I dag benyttes både Java, Microsoft DotNET og Ingres OpenROAD. OpenROAD er et objekt orienteret 4GL udviklingssystem som kommer fra Ingres Corporation [Ingres].

Projektet her tager udgangspunkt i lagerstyringen i [Unistar]. Det vil sige en vare ligger på et eller flere lagre eventuelt på flere lokationer. Der er en lagerprofil som giver en dateret oversigt over alle kendte fremtidige til og afgange. Der er en kalender som kan friholde dage så som weekender og helligdage. Hvis en vare ikke er på lager, eller de varer som ligger på lager er reserveret tager man indkøbsoplysningerne med i betragtning såsom leveringstid og klargøringstid. Ligeledes kan man tjekke for om varen stadig kan/må anskaffes.

På basis af disse oplysninger skal vores web-service fortælle hvornår, indenfor en given tidshorisont, en vare kan leveres.

Den nuværende løsning kører for langsomt og har svært ved store forespørgsler. Spørgsmålet er, skal vi skrive løsningen om eller skal vi benyttes os af refaktorisering.

4.2 Interessenter

Der er flere interessenter involveret:

- Kunden som ikke er tilfreds med den nuværende performance.
  Kunden er årsagen til at opgaven overhovedet er igangsat. Kunden er ligeglad med omskrivningen og ser nok helst at der bliver pillet så lidt som overhovedet muligt. Succeskriteriet er udelukkende en forbedret performance og at pris og tid for opgaven overholdes.

- Kundens partnere som benytter den nuværende service. Vi har ikke nogen direkte kontakt med partnerne, men i sidste ende er det deres brug af servicen der har initieret opgaven. Partnerne er ikke økonomisk involveret i dette projekt, men forventer at få løst performance problemet og at deres brug af servicen ikke skal ændres.

- Kode ejeren som har valgt at opgaven skal løses ved at omskrive hele eller dele af koden. Succeskriteriet er en mere *up to date* kode som er lettere og sikrere at modificere og som opfylder kundens krav.

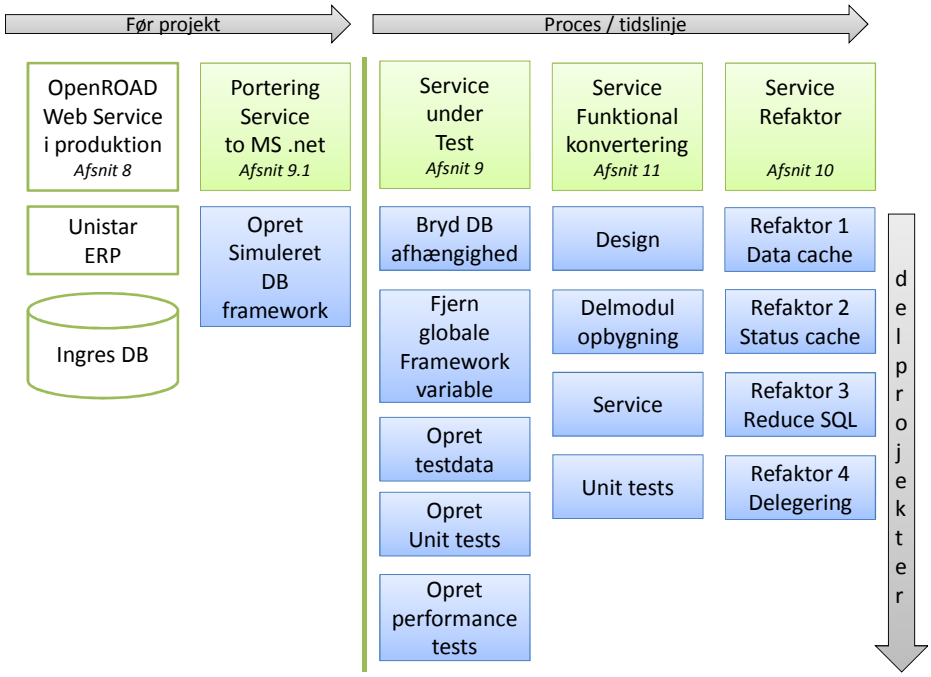
- Den eller de programmører som skal udføre opgaven. Succeskriteriet er at tilfredsstille alle involverede og at sikre der ikke opstår fejl i processen.

Der er derfor vigtigt, at det efterfølgende kan påvises, at den nye version af servicen opfylder kriterierne: Bedre performance, bedre kodekvalitet og at det gældende interface fastholdes. For at kunne bevise dette, skal den eksisterende kode kunne testes efter de samme kriterier som den nye kode. Vi skal altså kun påvise at den nye service i forhold til den eksisterende:

- Er hurtigere
- Har en bedre kodekvalitet
- Har samme interface som den eksisterende

4.3 Projektet

Projektet er delt op i fire hovedopgaver.
Nedenstående Figur 1 viser hvordan projektet er delt op. Venstre del er "før projekt" opgaver, mens kasserne til højre for stregen er projektet. De grønne kasser er hovedopgaverne i projektet. De blå (mørkere) kasser er delopgaver.



Figur 1 Projektopdeling

De i Figur 1 overordnede kasser:

- OpenROAD Webservice i Produktion
  Er det projekt vi tager udgangspunkt i.

- Portering af service til DotNET.
  Er en før projekt opgave, hvor vi etablerer det udgangspunkt, vi vil benytte.
  I den forbindelse er vi nødt at oprette en lille del af det framework, som den oprindelige service er baseret på.
- Service under test.
  Før vi kan komme i gang med refaktorisering skal vi have bragt den eksisterende service under test. Den oprindelige opgave går på at forbedre performance. Vi skal derfor også have etableret et udgangspunkt for den nuværende performance, som vi kan måle op imod.
- Service Funktionel konvertering.
  Dette er omskrivningsdelen af projektet.
- Service Refaktor.
  Dette er refaktoriseringsdelen af projektet.

"Før projekt" delen har ikke så meget med selve opgaven at gøre, men er en nødvendighed for at vi kan gennemføre projektet, som vi gerne vil.

"Service Funktionel konvertering" og "Service Refaktor" er foregået parallelt uafhængigt af hinanden.

4.4 Databasesystemer og performance

Når man arbejder med performanceforbedringer, hvor der er en database involveret, vil det ofte være databasen der fokuseres på og analyseres først. Men det viser sig ofte at problemerne ligger i applikationen og dens måde at tilgå databasen på. Det kan være optimering af de enkelte kald eller en reduktion i antallet af kald. I vores analyse har vi derfor valgt at se bort fra databasen og dens strukturering. Vi laver en simuleret tilgang til databasen, hvor vi kan returnere resultater. Vi finder en model til at emulere tidsforbruget i forhold til produktionskoden.

Det er der to årsager til. For det første har vi ikke adgang til produktionsdatabasen. Det vil sige vi kan godt få fat i data, men kunden ser nødig vi arbejder direkte i deres produktionsdata.
For det andet er det svært at udføre kontrollerede performancetest mod et databasesystem. Hvis vi havde en kopi, ville vi være de eneste der kørte på denne kopi, så det underliggende disksystem vi holde alle vores data i diskbuffere og databasen vil ligeledes holde vores data i cache. Selvom vi fandt en vej rundt om dette vil der være så mange lag, vores test kommer i gennem, at det vil være svært at sammenligne.

En anden del af performance optimeringen kan bestå i caching af data.
Caching i den eksisterende løsning fungerer ikke og kan faktisk afstedkomme fejl.
Vi mener godt, baseret på Unistar ERP-systemet, at vi kan ændre cachingmodellen, så den kan få en positiv effekt og sikre korrekte resultater.

5 Metode

Vi vil, med udgangspunkt i bogen "Working effectively with legacy code" af Michael C. Feathers (2004) [Feathers], få den eksisterende webservice bragt under test. Herefter vil vi, parallelt og uafhængigt af hinanden, arbejde med to løsningsmodeller: en funktionel konvertering og en refaktorisering af den eksisterende service som nu er under test.
På basis af dette arbejde vil vi konkludere om refaktorisering faktisk er en bedre model end en

funktionel konvertering. Vi vil ligeledes se om der er nogle indikatorer der, på forhånd, kan hjælpe med at beslutte, hvilken løsning der er bedst.

Når man har to løsninger, som løser samme problem.
Hvordan kan man så konkludere om kvaliteten af den ene løsning er bedre end den anden.
Vi vil vurdere dette ud fra "S.O.L.I.D. Design Principles" af Robert C. Martin & Micah Martin [Martin&Martin].

5.1 Opgavefordeling

Den praktiske tilgang blev, at Jakob gik i gang med at lave det nye design, mens Kim begyndte at opsætte forudsætningerne for refaktorisering. Det betyder at den nye kode ikke har haft indflydelse på den måde refaktoriseringen er blevet designet på og omvendt. Det er altså designbeslutningerne der er blevet holdt adskilt.

Det skal nævnes at Jakob er en trænet DotNET udvikler og Kim stort set ikke har udviklet andet end små test programmer og rettet lidt i eksisterende kode. Vi har forsøgt at kompensere for dette i vores tidsforbrug for de enkelte opgaver. Jakob har været Kims DotNET ekspert i forbindelse med implementeringen af refaktoriseringen. Jakob har dog ikke fået lov til at påvirke de retninger som Kim har valgt.

I tidsforbrug er der et par ting som man skal holde sig for øje. Vi kender begge den eksisterende service ganske godt. Både på kodeniveau og dens funktion. Det skal også nævnes at Jakob, længe før dette projekt, har overvejet forskellige løsningsmodeller for at løse de nuværende problemer. Derfor har vi sandsynligvis kunnet analysere og komme op med løsningsmodeller både til refaktorerisering og til den funktionelle konvertering væsentlig hurtigere and hvis vi var startet helt scratch. Vi har ikke kompenseret for dette, i vores tidsangivelser, da det, efter vores mening, er umuligt at estimere værdien af denne viden.

5.2 Opsætning af performancetest

I Unistar's framework er der indbygget profiling af metodekald og databasekald. Dette skal vi emulere i vores test. Vi har valgt en simpel model, hvor vi blot indlægger variable delays ind i vores testdataset.

Profiling af databasekald er delt op i:

- en request time som emulerer et round trip til serveren
- En select time som emulerer query kompleksitet. Eksempelvis hvor mange tabeller der indgår.
- En row time som ganges op med det antal rows vi henter.

Ved at have disse tre målepunkter får vi målt konsekvensen af antal kald, SQL kompleksitet og datamængder.

Vi kan nu måle på hastigheden af vores service på samme måde som i produktion. Vi har hermed etableret vores målepunkter i forhold den stillede til opgave. Da vi logger alle databasekald og deres tidsforbrug, kan vi også udlede, hvor det vil være mest fordelagtigt at lægge vores indsats.

Der er et problem her. Vi vælger at kører med aktuelle delays, hvilket betyder at vores test bliver langsomme. Et alternativ vil være at bygge opsamling af "virtuel" tid således at der ikke er delays i koden. [Feathers] mener at test skal køre så hurtigt som overhovedet muligt. Vores delays er konfigurerbare, så der kun er delays i performance tests og ikke i unit tests.

Vi har udført nogle målinger, af de enkelte SQL kald, mod produktionsdatabasen. Baseret på disse resultater har vi opsat nogle delays i vores test, således at de svarer nogenlunde til de faktiske forhold.
Herefter har vi kørt målinger på den kode som er lagt under test. Nedenstående tabel 1 viser et gennemsnitsresultat over 10 kørsler. Baseret på de fabrikerede testdata.

Antal varer der spørges på	Elapsed time for hele servicen målt i ms.	Mål for service i ms.
1	41	Ca. 41
5	113	Maksimalt 113
55	1143	572
109	2218	Mindre end 1109

Tabel 1 Performance analyse

Ovenstående mål kunne nu bygges ind i vores test, således at de fejler indtil målet er nået.
Det har vi ikke gjort fordi målingerne er forholdsvis simple og det er nemt at se, hvornår vi har opfyldt vores mål. Endelig er performancedelen blot katalysatoren for dette projekt og ikke det vi skal undersøge.

Nedenstående tabel 2 angiver de varekombinationsmuligheder som servicen behandler.
Der er visse kombinationer som ikke er mulige, så nedenstående giver 27 valide kombinationsmuligheder. Dertil kommer forespørgsel på varer som ikke eksisterer.

Vi opbygger 4x27 + 1 varer, så vi har 109 varer i vores simulerede datamængde.

De 109 varer danner datagrundlag for alle unit og performance tests i alle projekter.

For at sikre at algoritmen fungerer, skal der laves flere forløb. Første forløb hvor data hentes og caches for første gang. Andet forløb hvor vi trækker på cachen.

Leverings/ klargørings-tid i dage	Varen kan indkøbes	Aktuel beholdning i styk	Fremtidig afgang i styk	Fremtidig tilgang i styk
1, 3, 6	Ja / Nej	0, > 0	0, > 0	0, > 0

Tabel 2 Vare kombinationsmuligheder

Alle testforløb er gennemført 2 gange.
Første gang får vi en måling hvor cache ikke har nogen effekt.
Anden gennemkørsel er der opbygget data i cache.

Ved kun at køre 2 gennemløb og tage gennemsnittet får vi en test hvor cache og ikke cache vægter lige meget. Havde vi kørt igennem 55 gange ville cache få større positiv indflydelse på vores målinger. Det ønsker vi ikke. Med denne metode får vi en forholdsvis konservativ måling af om cache har en effekt.

6 Konklusion

Hvilken løsningsmodel der er bedst afhænger af, hvordan man vægter områderne: tid, kvalitet og projekt sikkerhed.

6.1 Tid

I vores projekt vinder den funktionelle konvertering tidsmæssigt stort over den refaktoriserede løsning. Men generelt vil det afhænge af, hvor dårlig den eksisterende kode er.

6.2 Kvalitet

I vores projekt er kvaliteten af den funktionelt konverterede løsning væsentlig højere, målt med S.O.L.I.D., end den refaktoriserede løsning. Dette tror vi vil være generelt for de to løsningsmodeller uanset projekt, hvis udviklerne er lige dygtige.

6.3 Sikkerhed

I projektsikkerhed vinder den refaktoriserede løsning. Projektet kan stoppes undervejs og det man har lavet kan bruges. Hvis man stopper den funktionelle konvertering inden man er færdig har man spildt det hele.

I funktionssikkerhed vinder den refaktoriserede løsning. Den virkede før man startede og vil stadig skal virke efter hver refaktorisering. I den funktionelt konverterede løsning kræver det at man ned til mindste detalje kender den oprindelige funktion. Hvis den oprindelige funktion ikke er under test, hvordan kan man så være sikker på at de fungerer ens.

6.4 Fordele og ulemper

Den refaktoriserede løsning bruger generelt færre databaseressourcer på grund af caching. Performancemæssigt er det lidt sværere at vurdere, da det afhænger af, hvilken effekt cachen vil få, som igen afhænger af hvordan omsætning og lagerstatus er for de enkelte varer. Hvis cache får en stor betydning vil den refaktoriserede service være meget hurtig. Uden cache er den langsommere end den funktionelt konverterede. Ved en cache effektivitet over 50 % vil den være hurtigst.

Den funktionelt konverterede løsning overholder også performancekravet og har bedre egenskaber i forhold til S.O.L.I.D.
Den funktionelt konverterede service mangler en implementering af databasetilgangen.

Begge løsninger mere end opfylder performance kravet. Den funktionelt konverterede funktion har en væsentlig bedre implementering. Caching i den refaktoriserede løsning giver et lavere ressourcetræk på databasen og vil skalere bedre.

En ting som er anderledes i den funktionelle service i forhold til den refaktoriserede service er at den funktionelle service læser lagerprofilen for 10 varer af gangen. Det giver en kæmpe performancemæssig gevinst. Det er værd at undersøge om de betragtninger der diskuteres i afsnit 10.3.2 faktisk holder vand eller om den løsning som den funktionelle service benytter, er bedre.

I den funktionelle konvertering bevirker strukturen at man kan teste dele afløsningen uafhængigt af hele løsningen. Man kan for eksempel teste kalender separat. I den

refaktoriserede løsning er der ikke helt den samme fleksibilitet. De nye regler vil kunne testes individuelt, men ellers er man er nødt til at teste servicen som et hele.

Nedenstående tabel 3 er en oversigt over fordele og ulemper ved de to løsninger.

Refaktor		Funktionel	
Fordele	**Ulemper**	**Fordele**	**Ulemper**
Sikrere forløb, da man kan stoppe undervejs.	Koden bærer stadig præg af gamle beslutninger	Ny og moderne kodestruktur	Ny og uprøvet kode
Bedst performance. Hvis cache slår igennem.	Overholder ikke S.O.L.I.D. I samme omfang.	Om nødvendigt kan cache implementeres	Læsning af 10 lagerprofiler af gangen kan vise sig at give bagslag
Sikrest i mål	Mere kompliceret løsning	Hurtigst i mål	Kræver en adapter for at overholde oprindeligt interface (krav)
Mindre ressourcetræk på databasen			Ikke alle forstår de nye konstruktioner som er taget i brug Opgaven er i princippet ikke helt færdig Benytter flere tråde, hvilket er svært
Andre kan stadig genkende koden		Behøver ikke at bringe Legacy kode under test	
Benytter let adgang til database grundet framework	Begrænset af framework	Mulighed for flere tråde og anden teknologi	
	Kan ikke bruges i andre sammenhænge	Kan bruges i andre sammenhænge. Se 11.3.5	
Verificerbar		Lettere verificerbar grundet komponent arkitektur	

Tabel 3 Fordele og ulemper ved de to løsninger

6.5 Kan man på forhånd vælge retning?

Er der indikatorer som på forhånd havde kunnet give os en ide om, hvilken af de to løsninger der ville være bedst at vælge? Vi synes ikke, vi kan se nogle meget entydige kendetegn, men der er nogle områder man kan være opmærksom på:

Jo mindre koden er, som skal ændres, desto større sandsynlighed er der for, at en funktionel konvertering vil være den bedste løsning.

En anden indikator kan være kendskab til den eksisterende løsning. Hvis den eksisterende løsning er veldokumenteret, således at der ikke er tvivl om, hvad løsningen skal kunne, kan funktionel konvertering også vise sig er at være den bedste og hurtigste vej til målet.

En tredje indikator er forholdene mellem tid, kvalitet og sikkerhed:

- Refaktorisering har sikkerhed som første prioritet.
  Til dels også tid, da man har mulighed for at stoppe undervejs og stadig kan have opnået noget. Kvaliteten forbedres kun gradvist og hvis man vil have kvaliteten helt i top, ender man med at have skrevet hele koden om. I den situation har man højst sandsynligt brugt væsentlig mere tid end ved en funktionel konvertering. Man har til gengæld gjort det med mindst mulig usikkerhed i processen.
- Funktionel konvertering har kodekvalitet som første prioritet.
  Det kan være svært at vurdere tidsforbruget på opgaven. Der er også en usikkerhed i om den funktionelt konverterede udgave faktisk kan det samme som den oprindelige. Vil man sikre sig kendskab til den eksisterende kode, skal man først bringe den under test. En ting er, at man tror man ved hvad den oprindelige service laver, noget andet er hvad den faktisk gør.

For begge løsninger gælder det dog at al kodeskrivning skal foregå under test.

Umiddelbart ser det ud til at valget mellem de to løsningsmodeller er et valg mellem tid og kvalitet i forhold til sikkerhed. Er man presset på tiden vil man nok vælge refaktorisering fordi man vægter projektsikkerhed højst. Dette på trods af at man måske kunne have skrevet en helt ny og bedre funktion på den samme tid.

6.6 Tidsforbrug

Ser man udelukkende på tidsforbrug, vinder den funktionelt konverterede service stort.
At den er hurtigere at udvikle end den refaktoriserede løsning kommer ikke som nogen overraskelse, men at forskellen er så stor kommer noget bag på os.

I tabel 4 kan man se tidsforbruget for de enkelte opgaver

Projekter	Tidsforbrug i dage
1. Analyse	1
2. Bring service under test	3
3. Refaktorisering	4
4. Funktionel konvertering	2

Tabel 4 Tidsforbrug

Der kan laves flere forskellige forløb:

1. Refaktorisering.
   Her skal man udføre punkterne 1, 2 og 3.
   Totalt forbrug: 8 dage.

2. Funktionel konvertering med stort kendskab til den eksisterende service.
   Her skal man udføre punkterne 1 og 4.
   Totalt forbrug: 3 dage

3. Funktionel konvertering med lille eller ingen kendskab til den eksisterende service.
   Her skal man udføre punkterne 1, 2 og 4.
   Totalt forbrug: 6 dage

4. Lav begge løsninger i parallel.
   Meromkostningen ved at lave begge løsninger er i vores tilfælde kun 25 %.
   Totalt forbrug: 10 dage.

Dette er ret interessant. Vi havde ikke på forhånd forventet at se så stor forskel mellem løsningerne. Fordelen ved at lave begge løsninger er at de begge kan bruge ideer fra hinanden.

Vi ender altså op med at vi betaler ret dyrt for projektsikkerhed og for at have mulighed for at stoppe undervejs. Vores projekt er et lille projekt, så om tallene blot kan skaleres op til større projekter, kan man ikke udlede. Men det vil sandelig være interessant at prøve den samme proces på andre lidt større projekter, for at se om tallene holder. Tallene understøtter vores forventning om at små projekter ikke skal refaktoriseres, men med fordel kan funktionelt konverteres.

6.7 Metoderne

S.O.L.I.D. virker som et sæt fornuftige principper at arbejde med. De kan blot ikke stå alene. S.O.L.I.D. handler udelukkende om struktur. Man mangler noget som kan vurdere koden. Man kan skrive den værste spaghettikode inde i de enkelte metoder og stadig overholde S.O.L.I.D.

Feathers kommer med mange gode underbyggede forslag til, hvordan man skal strukturere en proces som bringer legacy kode under test. En sjov ide er at bruge tests til at finde ud af, hvad et program faktisk laver.

For begge metoder er der en interessant detalje. Det er sammenhængen mellem objekt orienteret design og database tilgang. Hvis man ikke passer meget på kan man, som vi selv har gjort i denne opgave, komme til at gå på kompromis med en effektiv database adgang for at opfylde nogle objekt orienterede design strategier. Vi er normalt meget opmærksomme på dette problem og alligevel fik vi delt VareSet og PartstockSet op i to kald. Først da vi så den negative effekt på performance begynder vi at tænke på databasen.

6.8 Kan løsningerne overføres til den eksisterende service?

Det vil være let at overføre de enkelte refaktoriseringer til den eksisterende service en af gangen. Den funktionelt konverterede løsning kan ikke umiddelbart overføres, da den benytter en masse ny DotNET funktionalitet som ikke umiddelbart kan overføres til OpenROAD, som den oprindelige kode er skrevet i. Skal den funktionelt konverterede kode benyttes, skal den nuværende service overføres til DotNET.

7 Teori

I dette afsnit vil vi gennemgå de teknikker som vi vil benytte i projektet.

Michael C. Feathers [Feathers], ”Working effectively with legacy code”,
vil blive brugt til at bringe den eksisterende service under test.

Robert C. Martin & Micah Martin[Martin&Martin], ”S.O.L.I.D. Design Principles”,
vil blive benyttet til at vurdere kvaliteten af de resulterende løsninger.

7.1 Refaktorisering

Refaktorisering er blevet en teknik som de fleste indenfor softwareudvikling har taget til sig. I hvert fald i teorien.
[Feathers] og [Martin&Martin] refererer begge til Fowler[Refactoring].

Med refaktorisering menes det at ændre på kode uden at ændre programmets eksterne funktionalitet, på en sådan måde at chancen for at introducere fejl minimeres. Refaktorisering benyttes til mange forskellige formål. Det kan være at gøre en kode mere læsbar. At stramme en kode op så den bliver mere sikker. Vi benytter oftest selv refaktorisering til at forbedre koden, så den bliver forberedt på nogle nye krav. Det vil sige først refaktoriseres koden så den er klar til, på fornuftig måde, at få inkorporeret de nye krav.

Refaktorisering går godt i spænd med, at man har koden under test, så man kan påvise, efter en kodeændring, at operationen ikke har ændret på den eksterne funktionalitet eller introduceret fejl. Vi benytter til hverdag også refaktorisering på kode som ikke er under test. Det stiller større krav til den efterfølgende manuelle test, men omkostningen ved at bringe koden under test først er for stor. Om dette er en korrekt betragtning er der sikkert nogle som vil sætte spørgsmålstegn ved, men en sådan beslutning, vil kræve ledelsens opbakning.

7.2 Michael C. Feathers: Working Effectively with Legacy Code

Bogen præsenterer en metodik til at lave ændringer i eksisterende programkode. Bogen indeholder kataloger over metoder og patterns til løsning af de forskellige problemstillinger man står overfor, når man vil ændre på kode, som ikke er under test.

7.2.1 Legacy Code

[Feathers] definerer *legacy code* som værende kode der ikke er under test[1]. Denne definition skal ses fra en software udviklers synspunkt, hvor en ændring skal laves i noget kørende software. Altså ses der bort fra hardware, kodekvalitet, softwareteknologi med mere. Dårlig skrevet kode er ikke nødvendigvis legacy, hvis bare den er verificerbar med test.

Præmisset for at kunne lave ændringer i legacy kode er at få koden under test, hvorefter det så, jævnfør Feathers, ikke længere er legacy kode.

7.2.2 Ændre Software

[Feathers] definerer fire primære grunde til at software ændres[2]:

1. Tilføjelse af ny funktionalitet.
2. Rette en fejl.

3. Forbedrer design.
4. Optimering af ressource forbrug.

Ændringer skal kunne gøres på en sikker og forsvarlig måde, og på en måde så vi ved om ændringerne virker, og er uden sideeffekter. Til dette opsætter [Feathers] en algoritme til at ændre kode[3]:

1. Identificer ændringsområdet.
2. Find test punkter.
3. Bryd afhængigheder.
4. Skriv tests.
5. Lav ændring og refaktoriser.

Samtidig indsnævrer [Feathers] begrebet for hvilke test, der er acceptable, for at kunne bryde ud af legacy kode definitionen: Tests skal være meget hurtige og kunne give øjeblikkelig feedback. Så begrebet test skal fortolkes som unit tests[4], Dermed ikke sagt at andre former for tests ikke kan benyttes.

Så en lidt ændret definition af legacy kode bliver til: Kode som ikke er under *unit* test.

Når ændringen er fortaget og koden er kommet under test, betragtes den ikke længere som legacy kode.

Algoritmen introducere dog et dilemma; Hvis ikke vi må lave ændringer uden test, så kan vi ofte ikke introducere test. Det skyldes at for at kunne introducere test skal vi lave ændringer i koden. Dette dilemma adresserer [Feathers], via såkaldt sikker refaktorisering. Dette er tilladt så længe, det er for at få test på plads[5]. Denne fase er klart den usikre del af et projekt. Fejl her vil skabe fejl i det efterfølgende system.

7.2.2.1 Identificer ændringsområder

Før en ændring kan fortages, skal området kendes. Det kræver altså (rimeligt nok) at man finder de områder som skal ændres. [Feathers] bruger mange analyseværktøjer til at identificere disse områder.

Vi vil ikke gå i dybden med dette område, da vores ændringsområde er let at identificere, nemlig hele servicen/proceduren.

7.2.2.2 Find testpunkter

At finde testpunkter er at lokalisere hvilke dele af koden som skal under test. [Feathers] giver forskellige forslag til hvordan dette kan lokaliseres.

Ved testpunkter forstås, hvilke dele af koden som skal under test, før man kan foretage ændringer. Man skal sørge for at de dele af systemet, som kan blive påvirket af ændringen er under test.

I vores tilfælde har vi en stor procedure, som bliver kaldt uden at denne procedure kalder videre i subsystemer, som vi skal ændre i. Derfor skal vi teste kald til denne procedure. Man kan godt vælge at lægge testpunkter længere inde, så man tester hver delproces i procedure. Vi mener dog ikke dette er nødvendigt, da proceduren ikke er specielt stor.

7.2.2.3 Fjern afhængigheder

[Feathers] definerer en afhængighed som værende noget der hindrer unit test[6]. Dette kan være en database eller anden ekstern form for kommunikation. Unit tests skal ikke tale med sådanne enheder, da unit test kun skal teste en *unit* og ikke kommunikationen til eksterne systemer.

For at komme til at teste systemer som har afhængigheder til eksterne systemer, er det nødvendigt at bryde disse afhængigheder. Det er her, vi ser dilemmaet med at ændre kode før testen er på plads, fordi afhængigheden skal brydes (dermed en ændring uden test) for at kunne lave tests.

Bogen beskriver mange forskellige måder på hvordan en eksisterende kode kan få brudt afhængigheder for at komme under test. Ikke alle metoder dur til alle teknologier, og mange er specielt lavet til ældre sprog, så som C og C++.

7.2.2.4 Skriv tests

[Feathers] gør ikke meget ud af at beskrive dette, men henviser til unit test software og teknikker[7]. Dog gøres der opmærksom på en forskel mellem at skrive tests til ny software kontra det at skrive til legacy kode.

Forskellen går på, at tests til ny kode mest skrives for sikre at koden virker og for at bevare *opførsel*. Altså at eventuelle ændringer ikke introducerer til fejl.

Test til legacy kode er mere et sikkerhedsnet, at støtte op om koden og sikre at andet legacy kode ikke påvirkes af ændringer. Selvfølgelig også at sikre at der ikke introduceres nye fejl.

7.2.2.5 Lav ændringer og refaktoriser

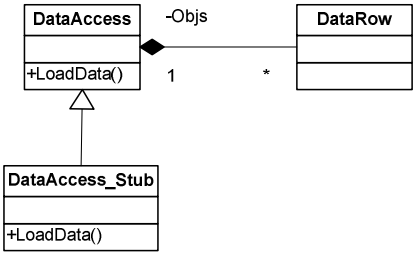
[Feathers] anbefaler *test drevet udvikling* som metode til at fortage ændringer. Man kan dog lave sine ændringer som man vil, så længe det bliver gjort under test. [Feathers] anbefaler også at man laver mange små refaktoriseringer, for at forbedre koden. Med forbedre menes små ændringer i koden, så som at ændre et lokalt variabelnavn fra noget intetsigende ”p” til noget sigende ”vare”. Små forbedringer der primært gør koden lettere at læse og forstå. Hver af disse små ændringer skal verificeres via test.

7.2.3 Metoder til at bryde afhængigheder

Af de mange forskellige metoder som Feathers beskriver, har vi identificeret og brugt fire, som kort vil blive gennemgået her.

7.2.3.1 Subclass and Override Method

For at bryde en afhængighed, er det nogle gange muligt at opnå det ønskede resultat ved at nedarve og overskrive metoder. Som et eksempel kan vi tage en DataAccess klasse som tilgår en database for at hente data. For at komme uden om databasen når man er under test, kan man nedarve fra DataAccess klassen (se figur 2).
Ved at overskrive metoden: LoadData(), kan DataAccess_Stub selv producere de data, som testen skal køre på.



Figur 2 Nedarv for at fjerne database afhængighed

Legacy koden fortsætter med at benytte en variabel erklæret som DataAccess, men når koden er under tests, så kan man benytte en instans af DataAccess_Stub, og dermed undgå kontakt til databasen. Hvordan man så får legacy koden til at benytte den nye instans, er en anden sag.

7.2.3.2 Extract and Override Factory Method

Denne metode afhjælper problemet nævnt i ovenstående afsnit 7.2.3.1. Metoden styrer hvilke konkrete instanser en kode benytter. Følgende constructor i en klasse kald Prg, illustrerer problematikken:

```
DataAccess _dataAccess;
public Prg(string connectionString)
{
   this._dataAccess = new DataAccess(connectionString);
}
```

Hvordan kan vi få variablen _dataAccess til at blive en kontrolleret DataAccess_Stub når vi kører test? "*Extract and override factory method*" forslår at der introduceres en factory method i klassen:

```
public virtual DataAccess CreateDataAccess(string connectionString)
{
   return new DataAccess(connectionString);
}
```

Herefter ændres den oprindelige constructor til at benytte den nye factory method:

```
public Prg(string connectionString)
{
   this._dataAccess = CreateDataAccess(connectionString);
}
```

Herefter kan vi så benytte *subclass and override method* teknikken til at teste klassen med testdata. Vores DataAccess_Stub kan nu overskrive CreateDataAccess metoden, så vi kan bestemme hvordan _dataAccess skal virke. I vores eksempel kan vi returnerer en TestDataAccess som nedarver fra DataAccess, og er udfyldt med test data:

```
public override DataAccess CreateDataAccess(string connectionString)
{
   return new TestDataAccess();
}
```

7.2.3.3 *Parameterize Constructor*

Denne metode adresserer samme problemstilling som *extract and override method*, men på en lidt anden måde. Vi introducerer en ny constructor, samtidig med at vi beholder den gamle (constructor overload). I Prg eksemplet kan den nye constructor se således ud:

```
public Prg(DataAccess dataAccess)
{
   this._dataAccess = dataAccess;
}
```

Det medfører, at kode som benytter den gamle constructor, fortsat vil virke, mens kode som er under test kan styrer den konkrete DataAccess instans ved brug af den nye constructor.

7.2.3.4 *Break Out Method Object*

Opdel koden så dele af den kan bringes under test. Man tager en meget lang metode og flytter den over i et nyt objekt. Det nye objekt er nu lettere at få under test, og den eksisterende kode benytter blot det nye objekt.

7.2.4 Konklusion

At bringe eksisterende kode under test kan være meget svært. Metoderne beskrevet af [Feathers], giver mange gode eksempler på, hvordan man skal angribe legacy kode.

Det mest spændende ved [Feathers] er hans definitionen på legacy kode. Der findes en del fortolkninger på hvad legacy kode betyder[8], men en så klar en definition, som Feathers giver, har vi ikke set før. [Feathers] næsten ufravigelige krav om at al kode skal være under test før den kan ændres, var ikke hvad, vi før opgavens start, havde regnet med.

7.3 S.O.L.I.D.

S.O.L.I.D. er et akronym for en samling af best practices. S.O.L.I.D. står for:

Single-Responsibility Principle

Open/Close Principle

Liskov Substitution Principle

Interface Segregation Principle

Dependency-Inversion Principle

Som en indikator for kodekvalitet har vi valgt at diskutere løsningerne ud fra S.O.L.I.D. [Martin&Martin]
Principperne giver os mulighed for at kikke på nogle strukturelle aspekter af koden og vurdere kvaliteten.

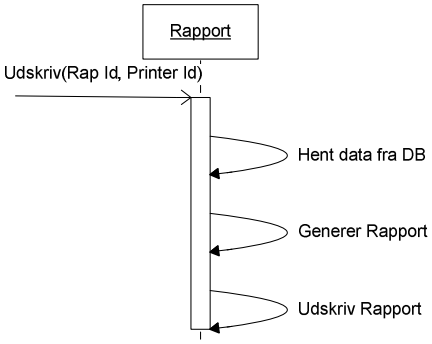
Forfatterne bag bogen [Martin&Martin] er ikke selv ophavsmænd til principperne. Principperne er indsamlet fra forskellige kilder og samlet til den lette og sigende forkortelse S.O.L.I.D. Tilsammen giver det deres bud på nogle overordnede retningslinjer for et godt designet objektorienteret system. Konklusionerne i [Martin&Martin] indikerer at et system som benytter principperne, vil kunne opnå nogle eftertragtede egenskaber: fleksibilitet, genbrugelighed, robusthed, og lettere vedligeholde[9].

7.3.1 The Single-Responsibility Principle (SRP)

SRP[10] siger, at en klasse kun skal have ét ansvar.

Idéen er at hvis en klasse har mere end et ansvar, har den også flere grunde til at skulle ændres. Så hvis flere komponenter benytter en klasse, og denne klasse ændrer sig, kan det få uønskede følger for nogle af disse komponenter. Hvis en klasse kun har et ansvar, vil det typisk også betyde at alle, som benytter klassen, vil ønske ændringen.

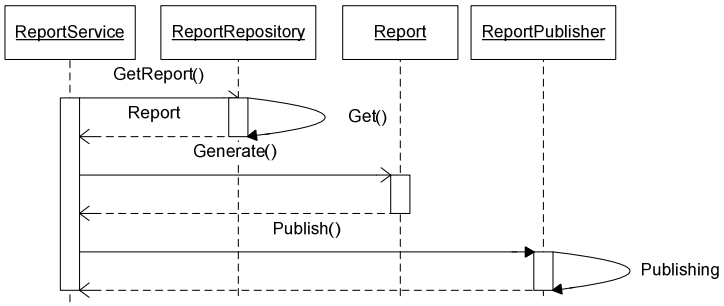
Et eksempel kunne være en rapportklasse. Denne rapport henter data fra en database, danner rapporten og udskriver til en printer (Figur 3).



Figur 3 Rapport klasse som bryder SRP

Rapportklassen bryder SRP, da den har mange grunde til at kunne ændre sig. Vi kan forholdsvist let se tre forskellige ansvar: Database, rapport opbygning og printer kendskab. Hvad nu hvis der kommer et krav om at rapporten skal udstilles som en web service?

Ved at dele ansvaret op i flere klasser (Figur 4), får hver klasse et ansvarsområde, som så benyttes af en overordnet serviceklasse.



Figur 4 ReportService som overholder SRP

Med denne konstruktion, kan man implementere (eller konfigurerer) konkrete services som udgiver rapporter til forskellig brug, uden at påvirke dannelsen af rapporten.

Ovenstående eksempel viser at koden bliver mere fleksibel ved overholdelse af princippet. Det tyder også på, at de enkelte klasser vil kunne genbruges. Systemet er samtidig blevet mere komplekst (en klasse er blevet til fire). Til gengæld er de enkelte klasser, hvor i eventuelle

rettelser skal ske, blevet mindre og tilføjelser til systemet er blevet lettere. De enkelte klasser bør også være blevet mere robuste, da de kun har et veldefineret ansvar hver.

7.3.2 The Open/Closed Principle (OCP)

OCP[11] siger, at en klasse/modul skal være åbent for udvidelse men lukket for modifikation.

Lad os forestille os en klasse som laver beregninger og samtidig skriver mellemregninger ud til en fil.

```
byte[] Calculate(byte[] data, string filename)
```

Klassen er lukket for udvidelse, men også lukket mht. modifikation. Havde den i stedet haft følgende signatur:

```
byte[] Calculate(byte[] data, Stream stream)
```

bliver output bestemt udefra.
Stream en abstrakt klasse som kunne være en FileStream, MemoryStream mm. Ydermere kan vi selv implementere en Stream som skriver til disk og samtidig til en database. Denne version er altså åben for udvidelser, men lukket med hensyn til modifikation (beregningen vil altid blive udført på samme måde).

Med den nye signatur vil metoden kun skulle ændres, hvis kalkulationen skal ændres, men ikke hvis mellemberegninger skal gemmes eller bruges et andet sted.

Med udgangspunkt i ovenstående eksempel er koden blevet lettere at genbruge, mere fleksibel. Vedligeholdelse er også blevet lettere (metoden laver mindre) og er også mere robust (skal ikke håndterer forskellige fil relaterede fejl mm.).

Et andet eksempel er sortering af elementer i en List i DotNET. Her kan man sende en IComparer instans med til Sort metoden, og på den måde bestemme rækkefølgen af sorteringen.

7.3.3 The Liskov Substitution Principle (LSP)

LSP[12] siger: *Subtypes must be substitutable for their base types.*

Basalt set siger princippet at man skal overholde følgende regel: hvis man nedarver fra en klasse/interface, så skal man overholde den kontrakt som supertypen forskriver. Samtidig skal man designe sine supertyper så dette er muligt.

Med udgangspunkt i eksemplet vist i figur 4, kan man forstille sig en helt generel implementation af ReportService klassen. Denne skal kun kende klasserne/interfaces ReportRepository, Report og ReportPublisher. Hvis den behøver konkret kendskab til en konkret nedarvet implementation, bryder vi princippet. Et eksempel på brud er, hvis ReportService er nødt til at gøre noget forskelligt ved bestemte instanser. Hvis ReportPublisher konkret er en PDFReportPublisher. ReportService undersøger så om ReportPublisher implementationen er af typen PDFReportPublisher, så skal der angives et filnavn. Dette er et brud, da ReportService kun bør have kendskab til supertypen ReportPublisher.

Overholdes LSP bliver koden lettere at vedligeholde, og mere robust. Klassehierarkiet er mere genbrugeligt, men der syntes ikke at være vundet meget med hensyn til fleksibilitet.

7.3.4 The Interface Segregation Principle (ISP)

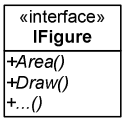
ISP[13] siger, at en bruger af en klasse/interface ikke skal være tvunget til at være afhængig af metoder, som de ikke benytter.

Et interface skal ikke have metoder som klienter ikke benytter. Så interfaces skal implementeres så specifikt som muligt i forhold til deres opgave.

Ræsonnementet er at software skal genleveres, hvis der sker ændringer i de interfaces, som de benytter. Hvis der udstilles et stort interface, vil en vilkårlig ændring i interfacet kræve at alle afhængige software komponenter implementere det ændrede interface. Dette gælder også selv om de ændrede dele ikke benyttes.

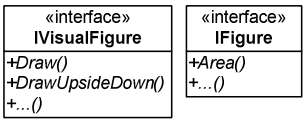
Omvendt hvis komponenten udstiller flere specifikke interfaces og der laves en ændring i en disse, så er det kun dem som benytter det ene interface som bliver påvirket.

Eksempelvis kunne et grafisk geometri bibliotek have et interface, IFigure, som kan tegne sig selv på skærmen og beregne sit areal (figur 5).



Figur 5 IFigure interface

En bruger (klient) af dette bibliotek benytter IFigure, men kun til at lave beregninger med og ikke de grafiske metoder. Brugeren har endog selv lavet figurer som implementerer IFigure. Hvis en ny version af IFigure (og dermed også af biblioteket) tilføjer noget spændene grafisk, eksempelvis ved at tilføje metoden *DrawUpsideDown()*, så er brugere af IFigure nødt til også at implementere denne metode, selvom de ikke bruger metoden.



Figur 6 opsplittede interfaces

Havde interfacet været opdelt som i figur 6, ville ændringer til den grafiske del ikke påvirke brugere af IFigure.

På en måde ligner dette lidt SRP princippet, bare for interfaces.

Ovenstående eksempel viser, hvordan koden bliver mere fleksibel, genbrugelig, robust og lettere at vedligeholde, hvis ISP overholdes.

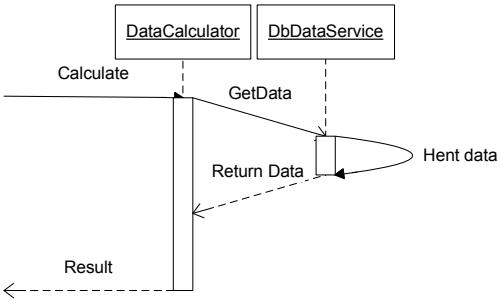
7.3.5 The Dependency-Inversion Principle (DIP)

DIP[14] siger, kort fortalt, at højt rangerende moduler ikke skal være afhængige af lavere rangerende moduler, og at moduler kun skal afhænge af abstraktioner.

Med høj og lav mener [Martin&Martin] abstraktions niveau, hvor lav eksempelvis er en klasse Fil som håndterer fil operationer. En højere rangerende klasse Document, benytter førnævnte Fil håndteringsklasse til at skrive et dokument på disk. Så er Document klassen afhængig af Fil klassen, hvilket er et brud på DIP.

Idéen er at løfte afhængigheder ud af den kontekst som de opstår i. Et eksempel kan være en klasse som henter data fra en database og behandler disse. Her kan problemet være databasen. Hvad nu hvis kravet ændrer sig og nogle data skal hentes via en webservice?

Figur 7 viser denne situation, hvor DataCalculator er hårdt bundet til at benytte en instans af DbDataService. Dette er et brud på DIP, da DataCalculator er afhængig af DbDataService. Vi kan ikke genbruge DataCalculator til at hente data fra en webservice.

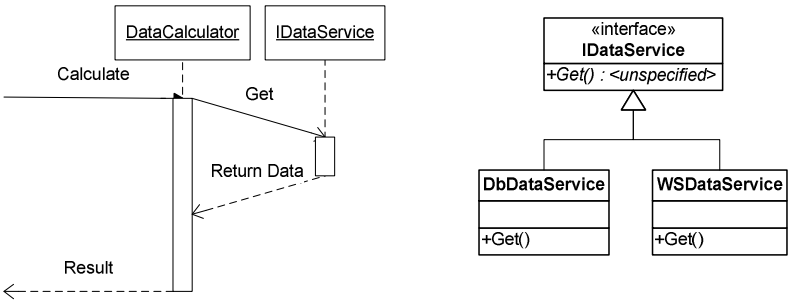


Figur 7 Eksempel på hård konkret afhængighed.

Vi vil kunne lave koden om direkte i DataCalculator. Men det betyder, at dette vil skulle gøres hver eneste gang, der opstår et ønske om at hente data fra en anden kilde.

Hvis vi vurderer at metoderne til at hente data ændrer sig uafhængigt af kalkulationen, kan det være en fordel at vende afhængigheden til DbDataService klassen. En måde at gøre dette på er at tilføje en parameter til klassen, så denne holder et interface *IDataService* som indeholder alle metoder nødvendige for at hente data.

Herefter kan vi frit implementere andre metoder til at hente data. Figur 8 viser en sådan konstruktion. Det er hvad [Martin&Martin] kalder at vende afhængigheden om. Afhængigheden af den lavere rangerende konkrete klasse er flyttet ud til brugeren af klassen.



Figur 8 Eksempel med dependency inversion.

Dette princip er specielt blevet meget udbredt i forbindelse med *test drevet udvikling*[15]. Når der skrives unit tests, har man behov for at kunne simulere eksempelvis database adgang. Til dette kan benyttes: "Inversion Of Control"[16] (IoC), hvilket er der samme som DIP.

Feathers benytter også denne teknik til at bryde afhængigheder med eksempelvis *Extract Interface*[17]. [GOF] benævner dette: *Program to an Interface, not an Implementation.*[18]

Koden bliver klart mere fleksibel og genbrug bliver også lettere. Det største problemer med DIP er den abstraktion som koden får, hvilket kan gøre den sværere at gennemskue.

7.3.6 S.O.L.I.D. Konklusion

Principperne lægger sig meget op af hinanden. Hvis man benytter SRP som vist i figur 4, er der ikke langt til at implementere DIP på en generaliseret service klasse.

Alt i alt, giver brugen af disse principper god mening, i forhold til at gøre kode fleksibel, genbrugelig, robust, og lettere at vedligeholde. Men er det altid givet at principperne fører til disse egenskaber?

Hvis man altid benytter disse principper, eksempelvis DIP, vil der automatisk blive mange interfaces eller abstrakte klasser; mindst en per afhængighed. Altså vokser systemet med en *kodeenhed* per gang dette gøres. Ligeledes bliver systemet sværere at finde rundt i, da disse abstraktioner kan være svære at gennemskue. Det er let at blive grebet af *værktøjer*, men man skal altid holde sig opgaven for øje[19]. Principperne kan let lede til løsninger som er alt for komplicerede. Der er et simpelt eksempel på Dimecasts.net, hvor man gennemgår en simpel printløsning bestående af to klasser. Efter at have været en tur i gennem S.O.L.I.D. maskinen, er den ene af de to klasser blevet til fire interfaces og ti klasser[20].

S.O.L.I.D. Principperne hjælper med at strukturere koden, så man lettere kan foretage ændringer og udvidelser uden uforudsete sideeffekter. Principperne hjælper med at bryde afhængigheder som man ellers risikerer uforvarende at bygge ind i sit system. Som et delresultat gør det ofte koden lettere at unitteste.

Liskov Substitution Principle syntes dog at være en fornuftig regel under alle omstændigheder.

Så svaret er ja. Hvis koden overholder principperne, mener vi at koden generelt vil være mere fleksibel, genbrugelig, robust, og lettere at vedligeholde.

S.O.L.I.D. stiller altså krav om infrastruktur, men ikke kodens kvalitet. Man kan altså skrive noget rigtig dårlig kode inde i metoderne og stadig overholde S.O.L.I.D. Der er altså et område som S.O.L.I.D. ikke dækker.

8 Den eksisterende Service

I forbindelse med udviklingen af web butikker. Vil vores kunder også gerne udnytte deres ERP system til web salg. Den web-service som projektet omhandler, er udviklet for at give vores kunder mulighed for online up to date lagerstatus på deres web-sites.

8.1 Historie

Servicen blev brugt af en af kundens partnere, som dagligt lavede forespørgsler med imellem en og fem varer pr. gang, hvilket ikke gav nogle problemer.

En dag meldte kunden om problemer. En ny partner var begyndt at benytte servicen, og syntes at den var meget sløv, og til tider slet ikke svarede. Det viste sig at den nye bruger af servicen lavede meget store kald (mange varer per forespørgsel), og hvis ikke svaret kom tilbage i løbet af kort tid, forsøgte man med kaldet igen, og hvis ikke svaret kom tilbage i løbet af kort tid, forsøgte man kaldet igen, og …

Populært kaldes dette, set fra vores side, for et ”Denial of Service Attack” (DOS). En belastning af servicen, så den ikke kan nå at svare inden et nyt opkald sættes i kø. Servicen, set

fra kundens side, virkede ikke længere, da den oprindelige partner heller ikke fik sine svar. Den nye partner kunne, mærkeligt nok, ikke genkende vores udlægning om at servicen blev sendt til tælling med mange store kald.

Den hurtige pragmatiske løsning på problemet blev, at der blev oprettet en parallel service, så kundens partnere benyttede hver sin service instans. Dermed kunne de ikke længere blokere for hinanden. Kunden og den oprindelige partner var tilfredse, da systemet virkede igen og den nye partner fik på et tidspunkt styr på sine kald, så de ikke udviste DOS tendenser.

Men performance var ikke tilfredsstillende på store kald. Viljen til at bruge mange timer på analyse og rettelser var ikke til stede. En billig og mulig forbedring (og forringelse) var at cache alle resultater, i servicen til store kald, over en "acceptabel" periode (20 minutter).
Konsekvensen er at vi nu har en hurtigere rutine som så til gengæld ikke altid returnerer korrekt svar. Partnerne vi aldrig opdage at et svar kan være forkert, men kunden kan faktisk miste salg på grund af dette!

En anden problemstilling er at vi nu har to separate services som tilgår databasen og stiller de samme spørgsmål (en med cache aktiveret, og en uden). Vi skal altså svare på de samme spørgsmål mere end en gang. Hvad hvis der kommer en partner mere (meget realistisk), skal vi så oprette endnu en kopi servicen. Dette er en uacceptabel model.

8.2 Funktionens interface

Der skal udstilles en webservice som kan tilgåes af kundens partnere. Servicen skal ud fra en række varenumre svare på, hvornår varerne kan sendes fra lageret. Spørgsmålet er altid: Hvornår kan jeg få leveret 1 styk af en given vare.

Når alt det tekniske xml og web specifikke er skrællet af servicen har den følgende signatur:

```
public List<StockIndicatorResult> GetStockIndication(List<string> EANs)
```

Metoden tager en liste med EAN-numre[21]. Den returnerer en liste af resultater i klassen StockIndicatorResult, som holder to egenskaber: EAN-nummer og en farvekode.

Svaret fremgår af nedenstående tabel 5:

Svar Indikator	Beskrivelse
Sort	Varen kan ikke leveres (mere)
Rød	Der vil gå mere end 5 dage før levering
Gul	Varen kan leveres indenfor 5 dage
Grøn	Varen kan leveres inden for 1 dag

Tabel 5 Returværdier for de enkelte varer

8.3 Algoritme for lagertjek

Først og fremmest skal varens grundoplysninger hentes. Med disse oplysninger kan det afgøres om varen er kendt, salgbar, dens nuværende lagerbeholdning samt summen af fremtidige til og afgange. De enkelte fremtidige til og afgange ligger i en lagerprofil. Ved at slå op i lagerprofilen kan man få en nøjagtig profil af lagerbeholdningen dag for dag.

8.3.1 Regel 1

Hvis varen findes og hvis den aktuelle beholdning er større end summen af alle fremtidige afgange, kan varen leveres med det samme.

Hvis dette ikke er tilfældet, går man videre til regel 2.

8.3.2 Regel 2

Her tjekkes der for om varen stadig kan købes og hvor lang tid det i så fald tager at skaffe varen. Hvis leveringstid + klargøring er indenfor 1 dag, kan varen leveres med det samme.

Hvis vi stadig ikke kan afgøre om varen kan leveres indenfor 1 dag, skal vi have hentet lagerprofilen.
Lagerprofilen indeholder oplysninger om alle til og afgange og hvornår disse skal effektueres.
Her kan man undersøge flere ting.

8.3.3 Regel 3

Kan vi 'stjæle' nogle af de reserverede varer, fordi vi kan nå at bestille nye varer hjem inden reservationen skal effektueres eller fordi allerede bestilte varer når at komme hjem inden.

8.3.4 Regel 4

Hvis vi ikke kan levere nu, hvornår kan vi så levere. Her skal lagerprofilen gennemløbes og der skal laves en oversigt over den fremtidige lagerbeholdning pr. dag. På basis af dette kan vi gå ind og se hvornår vi kan tillade os at levere varen uden det går udover eksisterende fremtidige leverancer. Her skal en kalender over arbejdsdage med i spil, således at man kun regner i arbejdsdage og ikke kalenderdage.

Når man kigger i lagerprofilen (se Tabel 6) er det ikke nok at finde første punkt, hvor der er positivt lager så vi kan tage en vare. Tager vi varen i uge 7, kan det være at beholdningen går negativ i uge 8. Vi er altså nødt til at se lige så langt frem i tid, som det vil tage at fremskaffe og klargøre en ny vare, hvis vi bestiller nu.
Eksempel på en lagerprofil for en vare som det vil tage 5 dage at skaffe hjem:

Dag	Aktuel beholdning	Afgang	Tilgang
1	10	0	0
2	10	5	0
3	5	4	2
4	3	3	0
5	0	0	0
8	0	0	14
9	14	0	0
Total		**12**	**14**

Tabel 6 Eksempel på lagerprofil

Hvis man kigger på: beholdning – afgang + tilgang (10 -12 +14=12), kunne man forledes til at tro man kan levere 2 styk. Men det kan man ikke da det vil betyde at leverancen af de 3 styk på

dag 4 vil få lageret til at gå negativ. Man må altså enten bestille nogle varer hjem, hvorefter man kan levere dag 5, eller man må vente til dag 9 hvor der igen er varer på lager.
Lagerprofil tjekket vil dog ikke kigge længere frem i profilen end den tid det tager at skaffe nye varer hjem. Profil tjekket vil altså ikke se tilgangen på 14 styk på dag 8.

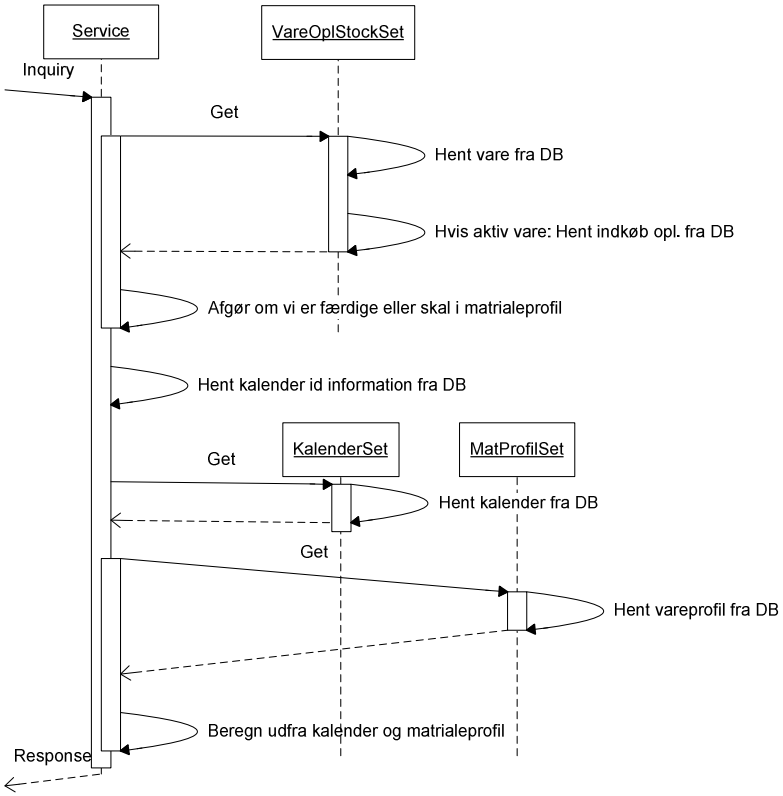
Om man skal bestille nye varer hjem eller vente på dem der allerede er bestilt er ikke en del af servicens opgave. Den skal blot svare på, hvor hurtigt varen kan skaffes.

Regel 3 og 4 er relativt dyre, så dem vil man gerne undgå.

8.4 Den nuværende implementering

Implementeringen af GetStockIndication er lavet som en procedure, som kun laver kald ud til det underliggende framework og klasser baseret på dette framework. Det vil sige at logik vedrørende det at finde resultatet, afgøres i proceduren. Al kommunikation med database/ERP er uddelegeret til framework relaterede klasser.

Frameworket giver generede klasser til database adgang.



Figur 9 Flow i nuværende kode

Proceduren består primært af to loops. Det første loop gennemløber varerne enkeltvis og laver opslag på vare, beholdningsoplysninger og indkøbsmuligheder. Hvis en afgørelse kan træffes

på baggrund af disse informationer er vi færdige med den vare. Hvis det ikke kan afgøres, vil varen blive ført videre til det næste gennemløb.

Andet gennemløb over varer henter nu kalenderoplysninger og lagerprofilen for varen. Beregninger fortages og et resultat er herefter klar.

Opbygningen er konstrueret med to loops for at undgå lagerprofilopslag hvis muligt. En performancevurdering er at lagerprofilopslag er meget dyre. Vi er ikke helt sikre på, hvorfor man oprindeligt vurderede at det var nødvendigt at have dette i to loops. Umiddelbart ser det ikke ud til at være nødvendigt.

8.5 Caching

Der er implementeret caching af varer i den nuværende løsning. Dette har vi slået fra under vores test. På basis af den nuværende brug har den implementerede cache ingen eller meget effekt.
Det er kun den service som benytter et stort antal vare pr. kald som har caching slået til. Problemet er at der godt kan gå mere end 20 minutter inden næste store kald kommer, og så er caching uden betydning da den kun holdes i 20 minutter.

8.6 Database kald

Simple tidsmålinger, i proceduren, har indikeret at database kald, ikke overraskende, tager størstedelen af tiden. Nedenstående tabel giver et overblik over antallet af databasekald, når det antages at varen ikke ligger i cache. Et enkelt databasekald til konfigurationen er ikke taget med i betragtning, da det kun foretages én gang i service instansens levetid.

Som det kan ses af tabel 7, begynder antal af SQL statements at blive et problem. Ved 400 SQL kald vil selv tiden for et round trip til serveren begynde at få betydning. Skal vi have bedre performance skal vi altså reducere antallet af kald til databasen. Dette kan vi gøre på to måder: Caching og læse mere end en vare af gangen.

Regel	Antal SQL kald	Max. antal rækker	Gennemsnitligt antal rækker
1	1	1	1
2	1	1	1
3+4	2	365+40	65+10
Total ved 1 vare	**4**	**407**	**77**
Total ved 5 varer	**20**	**2.035**	**385**
Total ved 50 varer	**200**	**20.350**	**3850**
Total ved 100 varer	**400**	**40.700**	**7.700**

Tabel 7 Databaseomkostning i antal kald og antal returnerede rækker

8.7 Overholdelse af S.O.L.I.D. principperne

8.7.1 The Single-Responsibility Principle

Vores klasse/procedure, har overfladisk ét ansvar; at angive en farvekode ud fra en vares lagerprofil. Går man lidt mere i dybden i koden afsløres det at klassen håndterer ret mange forskellige ansvarsområder; algoritme til servicen (hovedformål), hente data fra databasen med tilhørende fejlhåndtering, dato beregninger m.m.

SRP bliver ikke overholdt. Det eneste ansvar som ikke ligger i klassen er database kommunikation. Det sørger vores framework for.

8.7.2 The Open/Closed Principle

Vi læner os op af [Martin&Martin]'s diskussion vedrørende det at kunne forudse ændringer[22]. Strukturelle ændringer vil skulle fortages når et krav om ændring kommer (vi tager den første kugle)[23].

8.7.3 The Liskov Substitution Principle

Da der ikke er klasser som nedarver fra hinanden i løsningen (bortset fra framework klasser), eksisterer denne problemstilling ikke. LSP er altså irrelevant i den eksisterende løsning.

8.7.4 The Interface Segregation Principle

I servicen har vi ikke denne problemstilling, omend frameworket og dets generere klasser måske har, men det er ikke til diskussion her. Så vi konkludere at ISP er overholdt i servicen og ikke er relevant.

8.7.5 The Dependency-Inversion Principle

Servicen er meget afhængigt af vores framework, og de genererede klasser som frameworket giver os mulighed for. DIP bliver ikke overholdt. Den højt rangerende serviceklasse afhænger af mange konkrete lavere rangerende klasser og har stort kendskab til disse. Eksempelvis benyttes der klasser som vedrører database kriterier, hvilket er et ret tæt kendskab til databasen.

8.7.6 S.O.L.I.D. Konklusion

Med hensyn til robusthed, så har det vist sig at koden ikke er særlig robust. Ændringer har over tid haft effekter andre steder i koden. Servicen har vist sig skrøbelig. Fleksibilitet er der ikke meget af. Koden kan ikke konfigureres eller tilrettes på nogen let måde.

Det har vist sig at være meget svært at vedligeholde servicen. Meget tyder på at fejlrettelser har introduceret fejl.

Servicen vil måske kunne genanvendes, men den er designet meget specifikt i mod en bestemt type af spørgsmål. Forespørgsler vil blive besvaret ved hjælp af koder, ikke præcise datoer.

9 Den eksisterende service sat under test

I dette afsnit vil vi gennemgå de enkelte step, der skal til, for at få den eksisterende service bragt under test. Ligeledes gennemgår vi nogle af de overvejelser som ligger til grund for de ting vi har gjort.

9.1 Flyt fra Ingres OpenROAD til Microsoft DotNET

Det kørende system er kodet i et sprog som hedder OpenROAD[24]. Omend det kunne have været en spændene proces at lave opgaven på denne løsning, har vi vurderet at det ville være en for omfattende opgave.

I stedet har vi valgt at kopiere koden over til C# og tilpasse den så den funktionelt og strukturelt blev magen til den oprindelige kode.

Den ændring er gjort udelukkende til dette projekt og har intet med eventuelle fremtidige ændringer i den eksisterende løsning. Dermed ikke sagt at de ideer og løsninger som kommer frem i dette projekt ikke vil blive benyttet.
Denne opgave er ikke en del af projektet, men kan betragtes som en forudsætning for projektet.

9.2 Database afhængighed

For at kunne lave sikre test, vil vi gerne fjerne afhængigheden til databasen. Der er et problem her da vores "Data Access Layer" er en del af et framework vi ikke kan ændre på. Hvad vi kan gøre er at lave en subclass for hver Data Access klasse, hvor vi faker tilgangen. Vi har valgt en simpel metode, hvor hver klasse kan aflevere forskellige foruddefinerede sæt af data. På den måde er vi ikke afhængige af database adgang og vi har et veldefineret sæt af "database" kald som returnerer de samme data hver gang.

9.3 Databasen en fælles knap-resource

Når vi isolerer vores kode fra databasen, for at skabe et målbart miljø. Fjerner vi også den direkte mulighed for at se konsekvensen af de ændrede SQL-statements. Hvis vi ændrer på tilgangen til databasen, er vi nødt til at teste at de nye SQL-statements performer fornuftigt.
Vi skal altså finde nogle performance tal vi kan putte ind i vores emulerede database tilgang.

Et andet problem er samspillet med resten af systemet. Det kan godt være at vi laver noget dyrere SQL som totalt set, ud fra vores synspunkt, er billigere. Men samspillet med resten af systemet kan blive ramt. Hvis det er vores ejet system kan vi sammenholde det vi gør med de accesspatterns som resten af systemet benytter indenfor samme område.

Men hvad med kundens egen brug af databasen. Det kan være ODBC / JDBC rapporter som vi ikke ved noget om. Hvad med 3.part? hvis de, af kunden, har fået direkte adgang til databasen. Hvordan skal vi garantere noget på de vilkår? Man kan argumentere at det er en uskik at 3. part har direkte adgang. Men dette er kode, fra før man arbejdede med en service orienteret arkitektur. Så det er de vilkår vi har at arbejde ud fra og nej vi kan ikke garantere noget i forhold til 3. parts brug.

9.4 3.parts integration

Når vi isolerer vores kode og bringer den under test. Isolerer vi også vore kode fra 3. part som benytter vores rutiner. Hvordan kan vi garantere at vi ikke skaber problemer i forhold til 3. parts brug. Det kan vi faktisk ikke. Oprindeligt er der udstillet et interface som 3. part kan bruge. Vi har ingen garanti for, at de ikke har fundet på nye kreative måder at benytte vores interface på. En brug som vi ikke er klar over og derfor kan komme til at bryde i forbindelse med vores refaktorisering. Problemet er at, selvom vi kan bevise at det er 3. part som ikke følger reglerne vil dette altid lyde som en dårlig undskyldning fra vores side.

Man bør altså involvere 3. part for at sikre at intet går galt. Hvem skal betale for det. Vi hverken kan eller skal betale. Kunden vil med rette sige, at det er os som har ændret på

spillereglerne og dermed går alle pointene til 3. part. Dette er højest uacceptabelt for os. Hvordan skal dette så løses? En mulighed er at gå til kunden og foreslå dem at den partner som har problemet inviteres til at teste performance. Herved skal vi nok få noget feedback, hvis systemet pludselig ændrer opførsel i forhold til 3. part. Yderligere diskussion om dette, mener vi falder uden for rapportens område.

9.5 Processen med at bringe koden under test

I første omgang skal vi have brudt afhængigheden til databasen. Derefter skal vi sørge for at vi har den profiling, der skal til, for at kunne måle tidsforbruget i de enkelte dele af koden. På basis af dette, bør vi kunne finde de steder, som det er relevante at kigge på. Derudover skal vi have målinger på funktionen som et hele, da det er det, der i sidste ende skal bevise, at vi har nået målet.

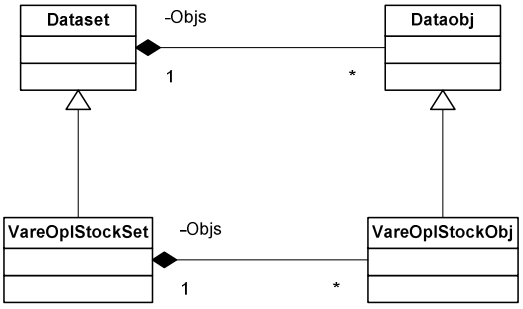
Der er to områder vi skal have brudt afhængigheden på: databasen og globale framework variable.

9.5.1 Bryd database afhængighed

Det første punkt er at få brudt afhængigheden til databasen.
Der er flere grunde til at ville bryde afhængigheden. For det første fordi [Feathers] kræver at vi kører hurtige unit tests, hvilket ikke kan lade sig gøre med databasen. For det andet er det sværere at få ensartede målinger på et live databasesystem. Der bliver simpelthen flere ubekendte såsom: netværksbelastning, databasesystemets interne caching, operativsystemets caching samt andre brugeres belastning af systemet.

For at sikre ensartethed, vil vi være nødt til at genstarte databasen/serveren eller på anden måde sikre at vores data ikke ligger i de underliggende systemers cache. Ved at bryde afhængigheden til databasen og lave en simulation i stedet har vi fuld kontrol over målingerne.

Vi er begrænset af det framework som vi er underlagt. Frameworket styrer adgangen til databasen, og er implementeret meget lig *Record Set*[25]. Det vil sige at adgangen til databasen abstraheres væk og styres ved implementering af nogle klasser (figur 10). Den generelle klasse DataSet, styrer operationer mod databasen og holder resultater fra denne, mens Dataobj repræsenterer de enkelte resultat rækker. Frameworket genererer kode til implementationen af konkrete klasser, såsom VareOplStockSet og VareOplStockObj i figur 10.



Figur 10 Framework database adgang

Da vi ikke kan ændre på de eksisterende dataset og da de ikke er implementeret ved hjælp af interfaces, har vi valgt at nedarve de enkelte dataset som metoden *Subclass and Override Method* (afsnit 7.2.3.1) beskriver. Herved har vi isoleret testkoden i testdataset i forhold til

produktionskoden uden at ændre noget af den eksisterende kode.
Dette skaber dog et nyt problem. I produktionskoden instantieres de enkelte datasets direkte. Det er vi nødt til at abstrahere ud, således at koden kan benytte testklasserne i stedet for. Vi starter med at benytte metoden *Extract and override factory method* (afsnit 7.2.3.2). Det giver os en del nye factory methods, som så skal overskrives i test. Nu skal vi altså både overskrive servicen og test dataset i test. For at simplificere dette benytter vi en variant af *Break out method object* (afsnit 7.2.3.4), sådan at alle factory metoder bliver samlet i et objekt, som vi så danner i servicens constructor. Nu kan vi benytte *Parameterize constructor* (afsnit 7.2.3.3) til at lave en ekstra constructor som tager det nye objekt med alle factory metoderne i, som parameter. Vi ender altså op med, hvad der også kaldes en ServiceLocator [SLDI].

ServiceLocatoren indeholder en factory metode per dataset. Som udgangspunkt leverer ServiceLocatoren blot en instans af de klasser som tidligere blev instantieret direkte i koden. Her bliver vi altså nødt til at ændre i den eksisterende kode. Det er dog en forholdsvis ufarlig ændring. En dataset-instantiering ændres til et kald til serviceLocatoren.
Eksempel:

```
Før: VareSet vs = new VareSet();
Efter:VareSet vs = ServiceLocator.createVareSet();
```

I vores testmiljø initialiseres ServiceLocatoren, til at returnere de tilsvarende testklasser.
Her er vi altså nødt til at ændre på koden for at få den bragt under test.
Vi ændrer altså vores produktionskode til en lidt mere kompleks konstruktion. Noget vi ikke ville have gjort, hvis ikke det havde været for testen. Test påvirker altså vores produktionskode til et mere fleksibelt design, men også et mere komplekst design.
Brug af ServiceLocator er ikke nødvendigvis den bedste løsning, hvis man har mange dataset. Hvert dataset kræver at vi opretter en ny metode i ServiceLocatoren. Her kunne man argumentere for yderligere refaktorerisering som bryder afhængigheden mellem ServiceLocatoren og de enkelte dataset. I vores tilfælde er dette dog unødvendigt. Det vil blot gøre koden endnu mere abstrakt og kompliceret. Her ligger en fare ved refaktorisering. Hvor langt skal man gå? Går man længere end nødvendigt spilder man tiden og gør muligvis koden unødig kompleks. Vi er endnu ikke begyndt på den reelle performance opgave, men kun i gang med forberedelserne til at bringe koden under test. Vi kan altså ikke benytte den oprindelige målsætning til at stoppe refaktorisering på basis performancekrav. Her er vi nødt til at have en meget stram styring for ikke at lav mere end højst nødvendigt.
En målsætning for at bringe kode under test kan være at, man ikke går længere end absolut nødvendigt. Spørgsmål man kan stille:

1. Er afhængigheden brudt?
2. Kan koden testes?

Kan man svare ja til de to spørgsmål, er yderligere refaktorisering ikke nødvendig.

9.5.2 Framework Globale variable

Globale variable kan være et problem. I dette tilfælde har vi nogle globalt instantierede klasser som vi får stillet til rådighed via vores framework. Vi kan altså ikke ændre de globale variable eller indholdet af de klasser som de peger på. Her har vi valgt at benytte en tilsvarende konstruktion som vi benytter ved abstraktion af dataset klasserne. Det vil sige alle de steder hvor en global variabel refereres, ændres dette til en ”getGlobal” klasse. [Feathers] giver nogle metoder til dette, så som at udstille en global *setter* så man kan overskrive konstanten under test. Den er vi ikke meget for, om end der klart vil kunne opstå situationer, hvor det er nødvendigt.

Vi har i vores tilfælde blot benyttet den allerede oprettede ServiceLocator.
Eksempel:

```
Før:  var = Config.get (…);
Efter:var = serviceLocator.getConfig (…);
```

Her gælder alle de samme observation som under dataset abstraktionen.
Vi øger kompleksiteten på grund af test.
Har vi mange globale variable som kræver abstraktion, kan vi ende op med mange metoder i getGlobal klassen. Igen kan dette generaliseres yderligere med øget kompleksitet til følge.
Igen skal man passe på med unødvendig generalisering. Det har vist sig at, vi har kunnet undgå brugen af de globalt framework instantierede klasser i vores projekt.

9.5.3 Afledt konsekvens

Efter at koden er lagt under test viser det sig at der er fejl i en eksisterende kode.

- Der er en simpel situation, hvor programmet returnerer forkert svar.
  Dette problem løser vi.
- Der er en del af koden som aldrig bliver udført.
  Dette lader vi ligge
- Lagerprofil opslaget returnerer et svar lidt afhængig af vindretningen.
  Dette rør vi ikke ved i første omgang. Det er ikke en del af den stillede opgave.
  At svaret er forkert påvirker ikke vores måling omkring performance.

Dette skaber faktisk lidt frustration i processen. Er ovenstående virkelige problemer, eller er der noget vi har misforstået eller er vores tests forkerte. Det har vist sig at ovenstående er reelle problemer.

9.6 Få opgaven gjort målbar

Der ønskes en halvering af den gennemsnitlige svartid. De små kald må ikke blive mærkbart langsommere, mens de store kald skal være hurtigere.

Vi mener at, der er en fare i refaktorerisering. Det er lidt som med prototyping, hvornår er nok nok. Ved at have et veldefineret mål (en halvering af svar tiden) og ved at indlægge profiling-kode kan vi lave nogle målinger på den nuværende funktion. Ligeledes kan de samme målinger udføres efter hver refaktorisering. På den måde kan vi afgøre, hvornår opgaven er løst. Det vil sige at selv om man har alle mulige gode argumenter for yderligere refaktorisering, så kan man ikke godtgøre yderligere arbejde i forhold til den, af kunden, stillede opgave. Det kan godt være at kode ejeren mener at yderligere refaktorisering er nødvendig. I så fald er det arbejde som ikke finansieres af kunden.
Vi har i tabel 8 opstillet 4 sæt af forespørgsler som vi vil benytte i forbindelse med vores test.

Antal varer der spørges på	Beskrivelse
1	Må ikke blive mærkbar langsommere
5	Må ikke blive langsommere
55	Skal halveres i tid
109	Skal mindst halveres i tid

Tabel 8 Forespørgselsstørrelse

9.7 S.O.L.I.D.

Har det at bringe funktionen under test ændret på dens kvalitetsmæssige egenskaber?

Ja det har det. Vi opfylder nu nogle få punkter af S.O.L.I.D. som vi ikke gjorde før.

9.7.1 The Single-Responsibility Principle

ServiceLocator klassen har nu ansvaret for at aflevere relevante instanser af klasserne til servicen. Dette fjerner et ansvar fra servicen. Så på dette område er vi blevet bedre.

9.7.2 The Open/Closed Principle

Her har vi også lidt forbedring. Vi er stadig lukket for modifikation, men blevet mere åben for udvidelse, ved hjælp af den nye ServiceLocator klasse.

9.7.3 The Liskov Substitution Principle

Intet nyt her.

9.7.4 The Interface Segregation Principle

Ingen ændring.

9.7.5 The Dependency-Inversion Principle

Dette princip er hvad vi faktisk er endt op med at implementere, for at få koden under test. Database afhængigheder og globale variable er blevet fjernet fra servicen og kan nu konstrueres med en ServiceLocator instans.

9.8 Tidsforbrug

At bring legacy kode under test har vist sig at være en ikke helt triviel opgave. Faktisk har det været den sværeste og mest tidskrævende del af projektet.

For det første er det svært at gennemskue, hvordan afhængigheder skal brydes. Vi endte eksempelvis op med en række metodevalg, som kunne få servicen under test. Det var ikke indlysende, hvordan metoderne kunne kombineres til en løsning som virkede. For at få det overblik der skal til, kræver det en del analyse af koden.

Den anden problemstilling er at få bygget en test op, som man stoler på. Testen skal jo helst ramme 100 % af servicens udfaldsrum, ellers kan vi ikke verificere at vores efterfølgende rettelser ikke har ændret på den eksterne funktionalitet. Dette har krævet en meget grundig analyse af den eksisterende kode og datagrundlag for at kunne bygge denne test.
Dette bør man tage med i betragtning når man overvejer refaktorisering i forhold til en funktionel konvertering. Ved refaktorisering er det et krav at servicen er under test inden man starter. Ved en funktionel konvertering kan man bygge test op omkring den nye kode og undlade at bringe legacy koden under test. Det kræver dog stadig at man ved hvad man skal lave.

Tidsforbruget er delt op i to. Analyse og programmering. Dette har ikke været en sekventiel proces, men mere en iterativ proces. Så fordelingen af tid mellem de to opgaver er vores bedste bud. Som nævnt under afsnit 5.1 har vi på forhånd haft stor viden om servicen, så vi er sandsynligvis sluppet hurtigere gennem analysen end, hvis det havde været helt ukendt land.

- Analyse
  1 dag
- Refaktorisering
  3 dage

9.9 Konklusion

At få servicen under test har vist sig at være en hård nød at knække. Tvivlen om man nu også havde valgt den rigtige metode, om man kunne have ødelagt noget, er meget frustrerende. Fejl der laves i denne fase, vil jo få afsmittende effekt på hele forløbet og dermed nogle af de beslutninger, der vil blive taget omkring refaktorisering. I processen fandt vi kode som ikke blev udført og kode som tilsyneladende var forkert. Var det fordi vi ikke forstod koden korrekt, eller at de test vi implementerede, ikke var fyldestgørende. Denne fase, hvor man ændrer kode uden at kunne verificere om det er korrekt, kalder [Feathers] *lag time*. I disse situationer kan man blive i tvivl om man kan udlede de korrekte svar af koden.

Status på at få koden under test er at der er fundet fejl i koden og kode som ikke bliver udført. Dette kom lidt som en overraskelse. Vi mener dog at have rettet en graverende fejl, og vi tror på at vi kan bringe systemet i en tilstand, så vi kan verificere af andre fejl, ved hjælp af unit test og refaktorisering.

Dette er efter vores opfattelse den sværeste og mest tidskrævende del af processen. Man føler lidt man er på gyngende grund i en periode. Vores service er endda ret simpel, så dette er klart et punkt man ikke skal undervurdere.

10 Den refaktoriserede service

Denne proces er delt op i 4 overordnede processer. Disse overordnede processer er så yderligere delt op. Efter hver delproces kan servicen testes og skal fungerer. Vi har valgt ikke at skrive større konklusioner for hvert step, men kun der, hvor det er muligt at drage en reel konklusion.

De fire hovedprocesser er:

- Bring data caching til at fungere
- Husk sidste svar for en vare
- Minimer antal database kald
- Uddelegering
  Dette punkt er ikke implementeret i koden.

10.1 Bring data caching til at fungere

I den nuværende model caches alt i 20 minutter. Det er ikke godt da det faktisk giver mulighed for at returnere forkerte svar. De data der caches, er en kombination af stamdata og transaktionsdata. Stamdata og transaktionsdata har som regel ikke sammen stabilitet / levetid. Så ved at ændre dette til kun at cache stamdata fjerner vi behovet for at invalidere cachen hvert tyvende minut og vil samtidig sikre korrekte data. Vi har derfor valgt at dette skal være vores første refaktorisering.

I det bagvedliggende ERP system ligger der, pr. vare, et tidspunkt for sidste beholdningsændring. Dette kan vi drage nytte af i vores cachingstrategi. Ved at cache tidspunkt for sidste beholdningsændring, når vi læser beholdningerne, kan vi næste gang nøjes med at tjekke om tidspunktet har ændret sig. Har det ikke det, ved vi, at det svar vi gav sidste gang, stadig er gyldigt.

I den nye model separerer vi stamdata fra transaktionsdata.
Ved hjælp af tidspunkt for sidste beholdningsændring kan vi afgøre om der er sket en ændring i beholdning siden sidst.

- Cache vareoplysninger (stamdata)
- Cache beholdningsoplysninger med tidspunkt for sidste opdatering og gem det svar vi gav sidste gang. (transaktionsdata)
- Cache indkøbsoplysninger (Stamdata)

Disse ændringer forventes at hjælpe på gentagne kald på de samme varer.
Vi tror ikke dette i sig selv er nok til at klare de store forespørgsler.

Cachen er implementeret som en lazy cache, det vil sige at varen caches første gang, der bliver forespurgt på den. Samtidig bliver yderligere oplysninger hentet efter behov. Det betyder at selvom vi har været helt nede i lagerprofilen ved sidste svar og selv om vi ikke cacher lagerprofilen, Så kan vi blot returnere sidste svar igen, hvis tidspunktet for sidste lageropdatering ikke er ændret.

Først caches vare og beholdning. Hvis det ikke er nok hentes og caches indkøbsoplysningerne.
Når en vare først er cached behøver vi kun at hente beholdningsoplysninger for at se om der er sket ændringer. Her overholder vi ikke helt Single responsibility i S.O.L.I.D., da vi læser beholdninger sammen med læsning af varen. I stedet for at holde beholdningslæsningen helt separat. Separationen er jo allerede lavet, da vi jo nøjes med at opdatere beholdningsoplysningerne, hvis varen allerede er i cache. Men dette accepteres da performance kravet rangerer højere end principper.
Dette er gjort for at opnå den bedst mulige performance på dette punkt.

Vi har ikke implementeret nogen form for udløb af cachen. Servicen genstartes hver nat, hvilket effektivt betyder at varer i vores cache maksimalt lever i 24 timer. Cacheudløb kan let implementeres på et senere tidspunkt hvis dette ønskes.

10.1.1 Cleanup

I dette step fjerner vi loop 2 således at alt foregår i et gennemløb. Det giver et renere forløb og vi slipper for et ekstra array.
Dette har ingen konsekvens for performance. Vores performancetest viser ingen ændring.

10.1.2 Adskille klassen VareOplSet i to

Denne klasse håndterer i dag både vare, summeret beholdning og indkøbsoplysninger.
I første om gang piller vi indkøbsoplysninger fra, Det skubber os i retning af SRI. Dog ikke helt. Vi har valgt at beholde de summerede beholdningsoplysninger. Dette er gjort af performancemæssige årsager. VareOplSet bliver til VareSet og IndkobVareSet. De to klasser som vi ønsker at kunne cache individuelt.
Dette kræver at vi ændrer på servicen. Den skal selv instantiere indkøbsoplysninger. Dette passer dog fint med den underliggende algoritme, da indkøb er et særskilt step.
Koden der skal til, ligger i forvejen i VareOplSet, så det er ikke nogen større ændring.
Dette passer fint med S.O.L.I.D. Vareset ved ikke længere noget om indkøb (SRI). Vareset er

ikke længere afhængig af indkøb som den i realiteten ikke skulle bruge til noget. Indkøb kan nu læses separat og efter behov. Det bliver synligt i servicen at indkøb er et særskilt step.

10.1.3 Bedre separation

I dette step flytter vi hele styringen og fejlhåndteringen for de enkelte dataset ind i det respektive dataset.
ServiceLocatoren afleverer nu de enkelte dataset færdiginitialiserede.

Her bevæger vi os hen imod et par af S.O.L.I.D. principperne.
SRI fordi servicen ikke længere skal håndtere initialisering, fejlhåndtering og cleanup af de enkelte dataset.
DIP fordi servicen ikke længere behøver at vide noget om hvordan dataset håndteres.
Dette har ingen konsekvens for performance. Vores performancetest viser ingen ændring.

10.1.4 Kalender

Kalenderen bliver i dag læst hver gang vi skal i lagerprofilen. Men kalenderen er den samme ved alle forespørgsler. ServiceLocatoren initialiserer nu kalenderen, således at servicen blot skal bede om den. Samtidig lægges kalenderen op som en del af opstarten af servicen således at den kun bliver dannet en gang i servicens levetid.
Her bevæger vi os hen imod et par af S.O.L.I.D. principperne.
SRI fordi servicen ikke skal vide noget om kalenderen, men blot skal benytte den, den får fra ServiceLocatoren.
DIP fordi servicen ikke længere behøver at vide noget om hvordan dataset håndteres.
Dette burde have en minimal forbedring af performance. Vores performancetest viser ingen ændring.

10.1.5 Caching

Alle ovenstående step har været skridt på vejen mod en ny cachestyring.
DataSetCache er en ny klasse som skal kunne holde data fra et vilkårligt dataset.
DataSetCache instantieres i de klasser som skal kunne håndtere caching.
Dette kan laves på et mere generelt niveau, men det kræver at vi ændrer på det underliggende framework. Det vil vi ikke på nuværende tidspunkt. De to klasser: VareSet og IndkobVareSet skal kunne caches. Cachen implementeres i de pågældende DataSet. Det betyder at servicen ikke ved, hvilke klasser der benytter caching. Til gengæld skal man rette i de enkelte dataset hvis man vil benytte caching. Dette kan godt ændres med yderligere separation til følge. De har vi ikke gjort, da det for dette projekt ikke har nogen betydning.
Servicen skal dog rettes, da den i øjeblikket opretter og nedlægger dataset ved hver forespørgsel. Vi flytter derfor alle dataset instantieringerne op, så de bliver udført ved service opstart. Det kan vi tillade os nu, da ansvaret for vedligeholdelse af datasets indhold nu påhviler de enkelte dataset selv.

VareSet og IndkobVareSet cacher nu data efterhånden som de bliver læst. Dette er ikke helt godt for Vareset, da den jo indeholder beholdningsoplysninger som jo må forventes at ændre sig.
Så klassen ændres til at opdatere beholdningsoplysningerne i cache, når servicen henter varen som allerede er cached.
Vi har nu en cache som fungerer og som faktisk har en ganske god effekt på performance.

Nogle nye tests er blevet skrevet for at kunne verificere at cache fungerer.

Performance efter 1. refaktorisering.
Tallene i tabel 9 viser performancemålingerne for første gennemløb, hvor cache er tom og

andet gennemløb hvor cache har fået effekt. Næste kolonne viser servicen gennemsnitlige svartid. Det interessante er at vi nu opfylder to ud af de fire mål. Dette passer fint med det som kunden også har nævnt, at det er de store kald, der er et problem. Dette er hvad refaktor 3 skal adressere.

Antal varer der spørges på	Elapsed time for hele servicen målt i ms. 1. gang	Elapsed time for hele servicen målt i ms. 2 gennemløb cached	Gennemsnit	Mål opfyldt
1	24	14	19	Ja
5	90	35	74	Ja
55	1093	455	767	Nej
109	2138	930	1527	Nej

Tabel 9 Performancemåling efter 1. refaktorisering

10.1.6 S.O.L.I.D.

SRP er ikke overholdt overalt, eksempelvis er VareSet stadig blandet sammen med PartstockSet. For alle dataset klasser gælder det dog at de nu selv styrer fejlhåndtering, og oprydning. ServiceLocatoren håndterer intialisering af data klasser. Så servicen har fået fjernet en masse ansvar.

OCP Ingen ændring i forhold til den eksisterende service bragt under test.

LSP Ingen ændring i forhold til den eksisterende service bragt under test.

ISP Ingen ændring i forhold til den eksisterende service bragt under test.

DIP I servicen er al viden om databasen fjernet. Dataklasserne er derimod stadig hårdt bundet mod en SQL database.

10.2 Husk sidste status pr. vare

Dette er endnu en caching strategi som kan spare nogle af de dyre opslag.
Unistar ERP systemet holder et tidspunkt pr. vare for sidste beholdningsændring. Ved at cache denne oplysning sammen med sidste svar, kan vi spare nogle opslag i databasen.

10.2.1 Fjern beholdningsoplysninger fra vareSet.

Hvorfor nu det? Vi nævnte i 1. refaktorisering at VareSet ikke skulle deles af performancemæssige årsager.
Begrundelsen for at gøre det nu er at, vi skal bruge oplysningen om dato for sidste beholdningsopdatering i servicen. Ved at gøre dette kan vi blot aflevere svaret fra sidste gang, hvis datoen er uændret. Dette kunne ikke lade sig gøre i 1. refaktorisering, da datoen lå skjult inde i VareSet.
Vi har nu mulighed for at verificere om vores påstand om performance faktisk er korrekt.
Tabel 10 viser at vi betaler forholdsvist dyrt for denne adskillelse

Antal varer der spørges på	Elapsed time for hele servicen målt i ms. 1. gang	Elapsed time for hele servicen målt i ms. 2 gennemløb cached	Gennemsnit (ændring i forhold til R1)	Mål opfyldt
1	30	5	18	Ja
5	131	30	75	Ja
55	1440	335	874	Nej
109	2832	654	1721	Nej

Tabel 10 Performance efter opdeling af VareSet

Vi overholder dog trods alt stadig de to første mål, men servicen er nu langsommere på de store kald end den eksisterende service.

Denne ændring er en ren kodekvalitetsforbedring og forberedelse til næste step. Som man kan se af tallene får vi faktisk en dårligere performance. Men de fire dataset er nu tæt på SRI. Hvorfor kun tæt på. Fordi de klasser som benytter caching har denne implementeret direkte. Dette er unødvendigt og kan generaliseres så det bliver en integreret del af det bagvedliggende Framework. Dette ligger dog uden for dette projekt.

VareSet har nu ingen viden om beholdning mere. Den oplysning styres nu af servicen. Det giver ganske god mening da det får de enkelte regler til at stå mere klart i servicekoden.
PartStockSet havde vi i forvejen. Brugen af den er blot flyttet fra VareSet til Service.
Ændringen har også medført en oprydning i VareSet, som ikke længere kender de enkelte felter fra PartStock.

10.2.2 Implementering af status cache

Denne rettelse vil hjælpe på de vare der ikke købes og sælges så hurtigt. Fordi vi kender tidspunktet for sidste beholdningsændring, kan vi tjekke om der er sket noget siden sidste forespørgsel. Er der ikke det, kan vi blot returnere samme svar som sidste gang. Hermed sparer vi de dyre opslag på lagerprofilen, hvis der alligevel ikke er sket noget.

Antal varer der spørges på	Elapsed time for hele servicen målt i ms. 1. gang	Elapsed time for hele servicen målt i ms. 2 gennemløb cached	Gennemsnit (ændring i forhold til R1)	Mål opfyldt
1	30	5	17	Ja
5	136	30	75	Ja
55	1426	330	876	Nej
109	2789	660	1724	Nej

Tabel 11 Performance efter implementering af status cache

Tabel 11 viser at resultatet er ikke helt så positivt som forventet. Dette kan skyldes sammensætningen af vores testdata. Men umiddelbart tyder det på at opsplitningen af vareset og partstockset er voldsomt dyrere end det vi vinder ved at huske sidste svar.

Status cache ændringen er en ren performancemæssig ændring. Den bidrager ikke med at øge kodekvaliteten. Men den øger performance.

Efter implementering af status cache kan man overveje, om denne implementering er den korrekte. En relationel database er meget effektiv til joine tabeller, og her har vi skilt en join ad udelukkende på grund af objektorienteret tankegang og S.O.L.I.D. Havde man kun haft databasebriller på, ville man aldrig have fjernet denne join. Så umiddelbart tror vi at den bedste løsning vil være at beholde denne join, men at placere beholdningsdata i PartStockSet. Hermed kunne vi have bibeholdt den oprindelige performancegevinst vi fik fra 1. refaktorisering og samtidig fået gevinsten omkring status cache. Vi ville i så fald ikke overholde S.O.L.I.D. i samme grad, men her må performance veje tungere. Havde der været mere tid, ville vi nok have lavet en yderligere refaktorisering, hvor dette var blevet implementeret.

10.2.3 S.O.L.I.D.

SRP Vareset håndterer ikke længere viden om PartStockSet og holder heller ikke længere data for partstock.

OCP Er uændret

LSP Er uændret

ISP Er uændret

DIP Er uændret

Så alle DataSet har nu kun et ansvar hver i forhold til det underliggende framework.

10.3 Minimer databasetilgang ved forespørgsel på mere end en vare

I det følgende afsnit vil vi gennemgå refaktoriseringsprocessens tredje del. Det gælder nu om at minimere antal af kald til databasen.

10.3.1 RAT vs. CAT

Dette begreb blev introduceret for os for mange år siden af John Smedley fra Ingres Corporation. Begrebet står for: ”Row At a Time i forhold til Chunk At a Time”. Databaser og SQL er optimeret mod at hente flere data af gangen. Hvis man arbejder med en row af gangen vil databasesystemets overhead få stor negativ indflydelse på performance. Ved at arbejde efter ”CAT” udnytter man disksystemets Read ahead og databasesystemet evne til at behandle mængder af data på en effektiv måde. Hvis man arbejder efter ”RAT” får man ikke den samme gavn af Read ahead og man har heller ikke nogen garanti for at databasesystemet holder de rows man vil hente lige om lidt.

10.3.2 Den nuværende tilgang er RAT

I dag hentes oplysninger for en vare en af gangen. Dette har aldrig været god praksis i en relationel database. Vi vil derfor ændre dette så vi henter op til 10 vare pr. gang. Dette vil reducere antallet af database kald betragteligt.
Hvorfor så ikke blot læse alle varer, der bliver sendt på en gang? Dette er baseret på vores erfaring med SQL statements med lange IN-lister. Både hvad vi selv har forsøgt, og hvad vi har set fra forskellige generelle rapport og datamining værktøjer. Store IN-Lister kan have en meget negativ indflydelse på databaseserverens performance. Ved at holde os på op til 10, får vi forudsigelige svar tider og benytte nogle SQL statements som ikke er alt for asociale.

For at lave denne ændring vil vi første separere database tilgang og forretningslogik. Herved kan vores service blot sende listen med varer til en dataset klasse som har ansvaret for at levere vareoplysningerne tilbage. Om denne dataset klassen så har fundet dem i cache eller har slået dem op en af gangen eller 10 af gangen er servicen uvedkommende.

For varer som allerede ligger i cache, vil det kun være beholdningsoplysninger der bliver opdateret.

Den samme strategi vil blive implementeret for indkøbsoplysningerne.

Lagerprofilen vil aldrig blive cached og vil blive læst pr. vare. Det er der to årsager til. For det første vil et opslag på en vare returnere et ukendt antal rows mellem 0 og ca. 40. For det andet er lagerprofilens datastruktur hashed på varenummer. Et kald med ti varer vil returnere et sted mellem 0 og 400 rows. Da vi ved at data er hashed på varenummer kan vi risikere at den beslutter sig for at udføre en tablescan i stedet for 10 keyed opslag. Det at lagerprofilen kun er hashed på varenummer betyder at vi får clustered den enkelte vares lagerprofil på nogle få sammenhængene pages i databasen. Dette giver nogle performancemæssige fordele og nogle concurrency mæssige fordele. Performance mæssigt fordi data der hører sammen ligger sammen. Concurrency mæssigt fordi data der ikke hører sammen ligger på forskellige pages og derved ikke låser for hinanden.

Alle ovenstående ændringer forventes at gavne alle kald til servicen som indeholder mere en 1 vare. De kald som kun indeholder 1 vare, vil blive negativt påvirket i minimal grad, udelukkende på grund at den ekstra kompleksitet der er lagt ind for at bundle varer i sæt af 10.

Vi vil få reduceret antallet af kald betragteligt. Dette sammenholdt med caching forventer vi løses opgaven.

10.3.3 Processen

Servicen skal ændres fra at køre en vare af gangen til at behandle alle varer på en gang. Så i stedet for et stort loop, er den blevet delt op i et loop pr. regel.
Det gør faktisk koden mere forståelig og de enkelte regler står mere klart. En rettelse som umiddelbart giver en bedre læsbarhed men som ikke kan retfærdiggøres via S.O.L.I.D.

For at kunne håndtere denne ændring skal de enkelte dataset udvides med en ny metode, så de kan håndtere at hente en liste af gangen.
Servicen skal internt deles op i de fire regler, Hvor den efterfølgende regel får en liste af de varer som ikke er afgjort endnu.
Et problem med denne refaktorisering er, at man skal rette ret mange ting på en gang, før blot en enkelt unit test melder ok igen. Ændringen minder lidt om en mini version af den funktionelle konvertering

Dette er en større omgang. Før vi gik i gang med arbejdet, kørte alle tests.
Alle datasets på nær lagerprofil skal ændres til en sæt orienteret tilgang med opbygning af in-lister på op til 10 varenumre og samtidig friholde de varer som allerede ligger i cache.
Servicen skal opdeles i særskilte loops per regel. Alle disse ændringer skal være på plads før vi kan teste igen. Det er altså ikke hvad [Feathers] kalder baby skridt, men lange seje træk. Ligesom da koden skulle under test er vi her ramt af en længere frustrations periode, *lag time.*
Det viste sig efterfølgende også at der skulle rigtig mange testgennemløb og rettelser til før servicen kørte korrekt igen. Samtidig var vi nødt til at indføre nye unit test for at verificere om vare blev cached og fundet igen på korrekt vis.

Tabel 12 viser at performancemålinger er meget positive. Vi har nu en service, der langt overstiger målsætningen i forhold til de stillede performance krav.

Antal varer der spørges på	Elapsed time for hele servicen målt i ms. 1. gang	Elapsed time for hele servicen målt i ms. 2 gennemløb cached	Gennemsnit	Mål opfyldt
1	30	5	18	Ja
5	34	6	19	Ja
55	316	36	191	Ja
109	516	66	330	Ja

Tabel 12 Performancemålinger efter 3. refaktorisering

Nedenstående tabel 13 viser en som forventet en kraftig reduktion i antallet af SQL kald til databasen.

Regel	Antal SQL kald eksisterende	Antal SQL kald Refactoriseret uden cache	Antal SQL kald Refaktoriseret med cache
Total ved 1 vare	4	5	1
Total ved 5 varer	20	5	1
Total ved 55 varer	220	28	6
Total ved 109 varer	436	55	11

Tabel 13 Databaseomkostning i antal kald

Sidste kolonne i Tabel 13 viser den maksimale effekt af cache. I drift vil de enkelte vares ressourcetræk ligge og svinge mellem 1 og 5 SQL kald afhængig cache effektivitet.

10.3.4 S.O.L.I.D.

SRP Er uændret.

OCP Er uændret

LSP Er uændret

ISP Er uændret

DIP Er uændret

Derimod er den interne kodestruktur kraftig forbedret. Servicen er væsentlig mere gennemskuelig og bedre opdelt. Dataset klasserne kan forholdsvis let ændre fra at køre 10 af gangen til et andet antal og det kan individuelt styres for de enkelte klasser.

Her er altså nogle kodemæssige forbedringer som ikke kan påvises med S.O.L.I.D.

10.4 Uddelegering af reglerne

Denne opgave er ikke baseret på kundens krav om performance men i kode ejerens krav i et mere robust system. Vi kan altså ikke godtgøre denne ændring i forhold til kunden, da performancekravene så rigeligt er opfyldt efter foregående afsnit.

10.4.1 De 4 regler

Servicens opbygning med et separat loop pr. regel, gør det oplagt at uddelegere reglerne til et særskilt sæt af klasser (Strategy Pattern [GOF]). Dette gør koden væsentlig mere robust, da vi hermed får adskilt det at hente data ind i servicen og det at udføre reglerne. På denne måde får servicen mere en rolle med at koordinere opgaven i stedet for selv at udføre den. Dette har ingen nævneværdig negativ performancemæssig effekt, men stor effekt i forhold til S.O.L.I.D.
Vi har ikke kodet denne refaktorisering. Men har valgt at tage den med, som om vi havde.
Kompleksiteten i denne refaktorisering er lille, da det mest drejer sig om at flytte kode.
I stedet for selv at sætte statuskoden skal disse blot returnere om varen kan leveres og hvornår.
Det vil medføre at vi kan teste de enkelte regler lettere.

10.4.2 Statuskode

I stedet for at servicen og de fire regler alle skal vide, hvordan status skal opgøres, kan det flyttes ud i en særskilt klasse (Strategy Pattern [GOF]). Klassens interface vil være: kan vare skaffes og hvornår.
Hvis vore service pludselig skal levere en orange farvekode tilbage for en ny status, vil det kun være statuskode klassen som skal ændres.

10.4.3 S.O.L.I.D.

SRP Uddelegeringen af de fire regler bevirker at service ikke længere ved noget om reglerne. Den ved stadig der er fire regler, men reglerne kan ændres uafhængigt.
Statuskode beregningen er ligeledes nu lagt i sin egen klasse og ingen andre klasser ved nu noget om dette.

OCP Er uændret.

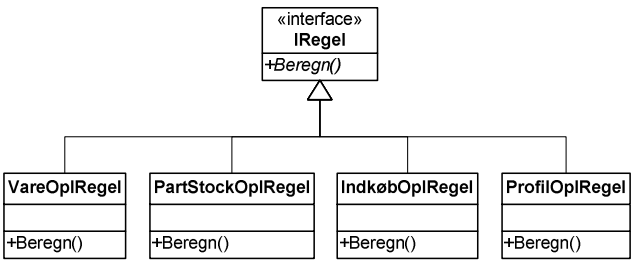
LSP Er uændret.

ISP Vi har nu fået et interface for regler, som de fire regler alle implementerer.
ISP er overholdt, da interfacet er specifikt rettet mod denne ene funktion.

DIP Er uændret.

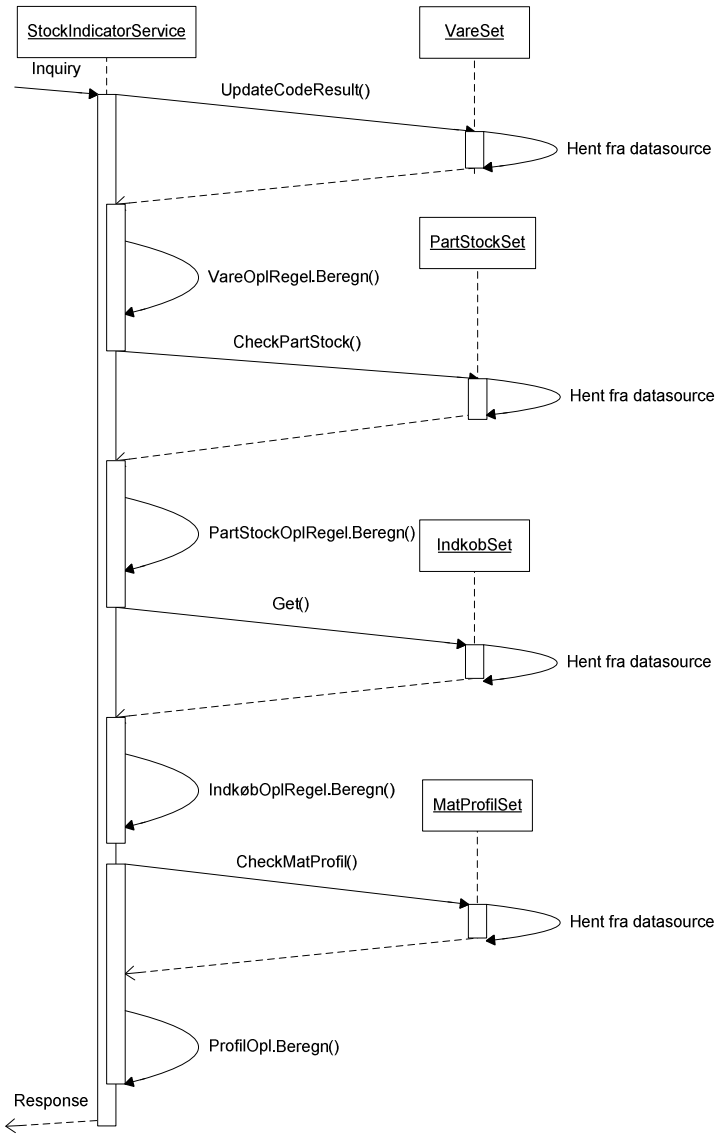
10.5 Den refaktoriserede struktur

Figur 12 viser det nye regelinterface vi implementerede i afsnit 10.4.1. Statuskode implementeringen følger samme mønster, men er ikke vist i figur 11.



Figur 11 Det nye regel interface

Figur 12 er en skitse over algoritmen efter refaktorisering. Som det fremgår, er den overordnede algoritme ikke ændret. Men flowet er ændret. Den enkelte regel udføres nu for alle varer i et loop inden man fortsætter til næste regel.



Figur 12 Refaktoriserede service flow

10.6 Overholdelse af S.O.L.I.D. principperne

10.6.1 The Single-Responsibility Principle

Dette punkt er væsentligt forbedret i forhold til den oprindelige service. Der er stadig plads til forbedringer. Cache styringen kan fjernes fra de enkelte klasser og gøres generel. Antallet af regler kan ændres så regelsættet injiceres ind i servicen.

10.6.2 The Open/Closed Principle

Er uændret.

10.6.3 The Liskov Substitution Principle

Er uændret.

10.6.4 The Interface Segregation Principle

Vi har fået et enkelt interface. Interfacet og klasserne der bruger det overholder ISP.

10.6.5 The Dependency-Inversion Principle

Er uændret.

10.6.6 S.O.L.I.D. Konklusion

Refaktoriseringen har bragt systemet tættere på S.O.L.I.D. Så af den grund kan man udlede at vi har fået et mere robust system. Servicen er blevet lettere at læse og forstå fordi den nu kun arbejder på et niveau. Den ved intet om databaser, regel-implementering eller status beregninger. Servicen har fået en mere koordinerende rolle, mens alt arbejdet uddelegeres.
Man kan sige at S.O.L.I.D. er den indirekte årsag til mange af disse forbedringer.

10.7 Tidsforbrug

Tidsforbruget på den totale refaktorisering er cirka 4 dage. Langt størstedelen er gået til 3. refaktorisering.

10.8 Konklusion

3. Refaktorisering (afsnit 10.3) var langt den sværeste og den som skabte de fleste problemer, men også den der, som forventet, har givet det største udbytte. Opdeling af VareSet og PartStockSet ser ud til af have været en dårlig ide. Den performancemæssige omkostning ved denne ændring er efter vores mening for stor. Ved lav cache udnyttelse vil dette få ret stor negativ effekt. Samtidig mener vi der er forkert at man ikke tilgår databasen på dennes præmisser. Noget vi først blev opmærksomme på, da vi så hvor stor omkostning faktisk var. Vi tror at dette er en fælle man ofte hopper i når man udvikler objekt orienterede programmer. Så den løsning som er skitseret i slutningen af afsnit 10.2.2 vil nok være en rigtigere model på trods af at den reviderede udgave passer pænt ind i S.O.L.I.D.

Baseret på de erfaringer vi har fået fra den funktionelle konvertering kan man sætte spørgsmålstegn ved om al den caching som er implementeret med den dertil øgede kompleksitet virkelig er nødvendig. Jeg tror dog at min beslutning om refaktoriseringsrækkefølgen har været korrekt baseret på den viden, vi havde på daværende tidspunkt. Caching har trods alt en ganske positiv effekt på ressourcetrækket på databasen.

Det er lidt svært på basis af vores test at vurdere, hvad den faktiske performance gevinst vil blive. Vi tror dog den vil blive større end de vi har angivet i rapporten, fordi vi kun ser på gevinsten ved et cache hit. Det må forventes at cachen har større effekt en dette. For mange af de kunder vi arbejder med er der ofte en 80-20 regel som gør sig gældende. Det vil sige at 80 % af omsætningen stammer fra 20 % af varerne. Dette vil ganske sikkert også være tilfældet her. Varer som kommer på kampagne eller står i udsalgsaviser vil stå for en stor del af dagens omsætning.

11 Den funktionelt konverterede service

11.1 Metode

Før vi gik i gang med at få den eksisterende kode under test, lavede Jakob et design og startede implementering for den nye kode. Målet med den nye kode er at forbedre performance og få kode med gode karakteristika.

Designet tager blandt andet udgangspunkt i [Martin&Martin], og benytter af nogle af de metodikker som bogen beskriver.

11.2 Analyse

Målet med den nye implementering er at forbedre performance. Det blev antaget at performance problemet ville kunne løses ved at mindske database tilgangen, ved at hente flere data per database forespørgsel.

For at komme i gang så hurtigt som muligt, er servicen implementeret uden hensyntagen til en specifik database (omend at layoutet af data er mage til det eksisterende). Det medfører at servicen, i den nuværende form, ikke vil virke ”out of the box”, og at en database specifik implementering vil være påkrævet[26].

Analyse af den eksisterende kode og databasen indikerede at der skal bruges tre forskellige objekter for at kunne afgøre, hvornår en given vare kan leveres: StockItem, PurchaceItem og StockProfile. Disse identiteter er magen til dem i den eksisterende løsning, om end med nye navne.

StockItem: En vares basisoplysninger og den nuværende overordnede lagerstatus.

PurchaceItem: En vares indkøbsoplysninger.

StockProfile: En vares detaljerede lagerprofil (lagerstatus).

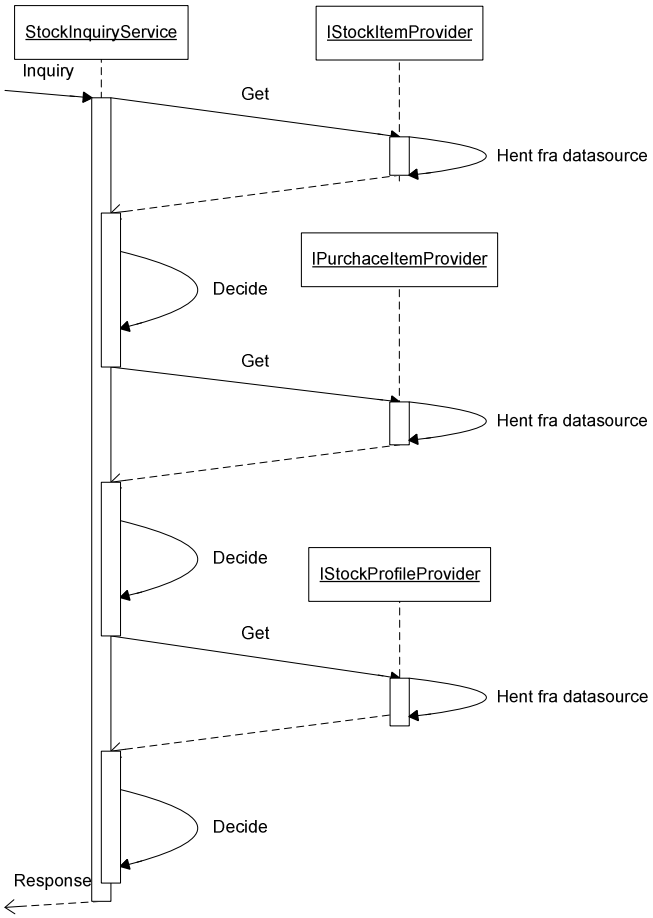
Med de tre identiteter kan en vares lagerstatus afgøres, men status kan i nogle tilfælde afgøres med en delmængde. Rækkefølgen på forespørgsler er StockItem, PurchaceItem og StockProfile. For at kunne afgøre en vares lagerstatus, behøves måske kun StockItem. Hvis ikke det er nok, så skal PurchaceItem benyttes. Hvis dette heller ikke er nok, kan status regnes ud med StockProfile. Algoritmen for afgørelse er altså magen til den oprindelige services algoritme.

To strategier er blevet anvendt for at opnå performance: Samle til bunke og parallelisme.

11.2.1 Samle til bunke

Den gamle service benyttede samme metode for at afgøre en vares lagerstatus, først StockItem (vareOpl), så PurchaceItem (IndkobVare) og til sidst StockProfile (MatProfil).

Den gamle service udfører dette for 1 vare af gangen, hvilket fører til mange databasekald. For at minimerer antallet af kald kan der *samles til bunke*; i stedet for at spørge databasen om en identitet adgangen, hentes der flere identiteter af gangen.



Figur 13 Flow for StockInquiry

Flowet igennem servicen er overordnet skitseret i figur 13. Servicen modtager et kald (inquiry) med EAN numre. Servicen uddelegerer arbejdet med at anskaffe alle nødvendige data fra databasen. Konkret er arbejdet udelegeret til tre interfaces; IStockItemProvider, IPurchaseItemProvider og IStockProfileProvider. Alle interfaces henter en mængde af identiteter.

Efter at have hentet objekter, loopes der over disse og det undersøges for hver vare om yderligere informationer er nødvendige (Decide metoden). Denne proces fortsætter indtil en afgørelse er truffet for alle vare.

Der hentes maksimalt 10 vare per kald til et *provider* interface, for ikke at degradere database performance (se 10.3.1). De enkelte SQL statements er blevet undersøgt for sideeffekter ved at benytte *in clause*, og ingen er fundet.

11.2.2 Parallelisme

En oplagt mulighed at undersøge, for at opnå bedre performance, er parallelisme. Der er blevet identificeret to forskellige områder af servicen, hvor vi kan anvende muligheden for at splitte opgaver ud til flere tråde.

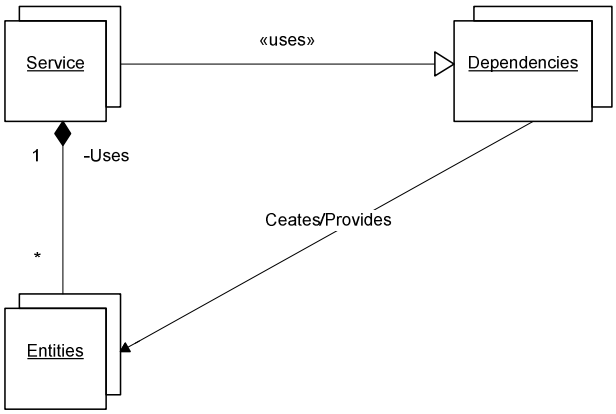
Den første mulighed er ved kald til databasen. Kald til databasen bliver brudt op i grupper af 10. Det betyder at hvis en forespørgsel er på 30 vare, vil det afstedkomme 3 kald til databasen. For hvert kald venter servicen på resultatet kommer tilbage, hvorefter de næste 10 vil blive hentet og så videre.

Alle kald til databasen laves uden låsninger af data. Dermed skulle der ikke være noget problem i at tilgå database parallelt. I figur 13 betyder dette at de enkelte *Get* kald, til de tre *provider* interfaces, kan forgår parallelt.

Det andet område som kan paralleliseres, er beregninger af resultater, (*Decide* loops i figur 13). Denne optimering forventes dog ikke at kaste meget performance af sig, da databasekald er der, hvor vi har identificeret et reelt tidsforbrug, og ikke loop strukturerne i sig selv.

11.3 Implementering

Fra start er det blevet forsøgt at udvikle den nye service efter teknikker beskrevet i [Martin&Martin]. Specielt skal nævnes at der er benyttet unit tests, for at sikre at enkelte delkomponenter virker efter hensigten.



Figur 14 Overordnet arkitektur

Arkitektur er skildret i figur 14. Klasser er inddelt i tre forskellige kategorier; Service, Entities og Dependencies.

Service er de klasser, der er *offentlig* tilgængelige og holder servicens flow. En webservice vil benytte klasser i denne kategori til at finde svar på vares beholdning. Service benytter afhængigheder (dependencies) til at anskaffe objekter/identiteter. Ved at samarbejde med disse identiteter kan servicen udføre sit flow. Kategorien er uddybet i 11.3.1.

Dependencies repræsentere klasser og interfaces, som giver adgang til identiteter. Klasser herunder har intet kendskab til servicen. Kategori har det primære formål at skærme servicen/programmet fra stærke bindinger til andre systemer. Kategorien er uddybet i 11.3.2.

Entities er identiteter som vedrører data, beregninger, altså tilstande. Ud fra objekter i denne kategori afgøres en vares lagerstatus. Kategorien har ingen kendskab til servicen, men nogle kender til *dependencies*[27]. Kategorien er uddybet i 11.3.3.

Generelt for alle elementer som benytter sig af kategorien *dependencies* benyttes DIP. Det vil sige; alle instancer af objekter skal have konkrete *dependency* objekter stillet til rådighed, for at kunne udføre deres operationer.

Implementeringen er lavet på en sådan måde, at hver gang en ny tilføjelse er blevet lagt ind, så er der blevet skrevet en tilsvarende unit test. Det er altså ikke helt *test drevet udvikling* som [Martin&Martin] beskriver. Valget af udviklingsmetode skyldes at Jakob finder denne mest effektiv. [Martin&Martin] beskriver dog også at det vigtigste er at have hurtige unit tests, ikke at lave *test drevet udvikling*.

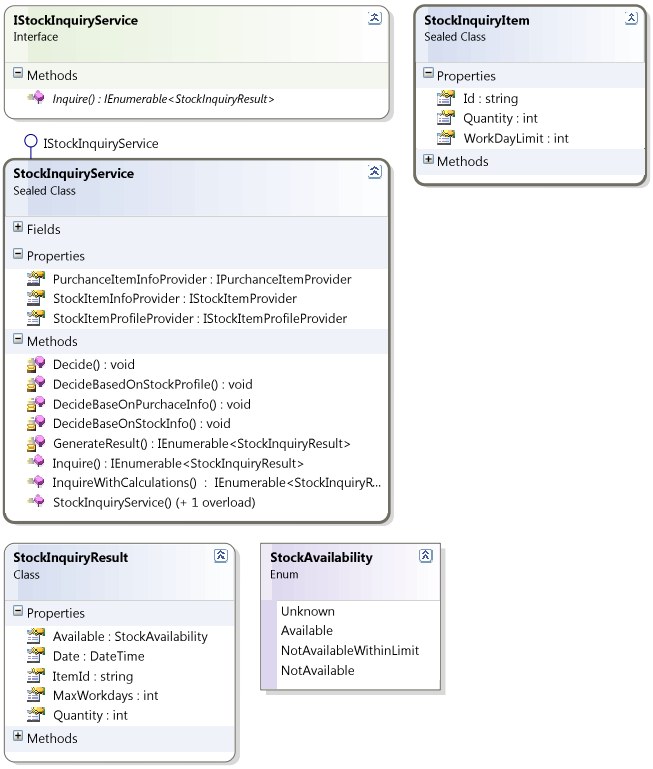
Følgende tre afsnit behandler hver kategori og overholdelse af S.O.L.I.D. principperne.
De efterfølgende to afsnit diskuterer nogle implementeringsspecifikke emner.

11.3.1 Service

Interface først; *IStockinquiryService*, med en metode *Inquire*, som har følgende signatur:

```
IEnumerable<StockItemResult> Inquire(IEnumerable<string>)
```

Inquire metoden tager en mængde af vare EAN numre og returner deres lagerstatus. Klasser i kategorien ses i figur 15.



Figur 15 Service kategori diagram

SRP er overholdt, da hver klasse har et klart afgrænset ansvar.

OCP er overholdt. Den eneste klasse med interessant opførsel er StockInquiryService. Denne er lukket for modifikation, med ikke for udvidelser ved hjælp af *dependency injection* (se 11.3.2).

LSP er trivielt overholdt, da der ikke forekommer nedarvninger.

ISP er overholdt af det ene interface, IStockInquiryService, specielt da den kun har en metode.

DIP er overholdt, da alle afhængigheder er introduceret til servicen udefra (se 11.3.2).

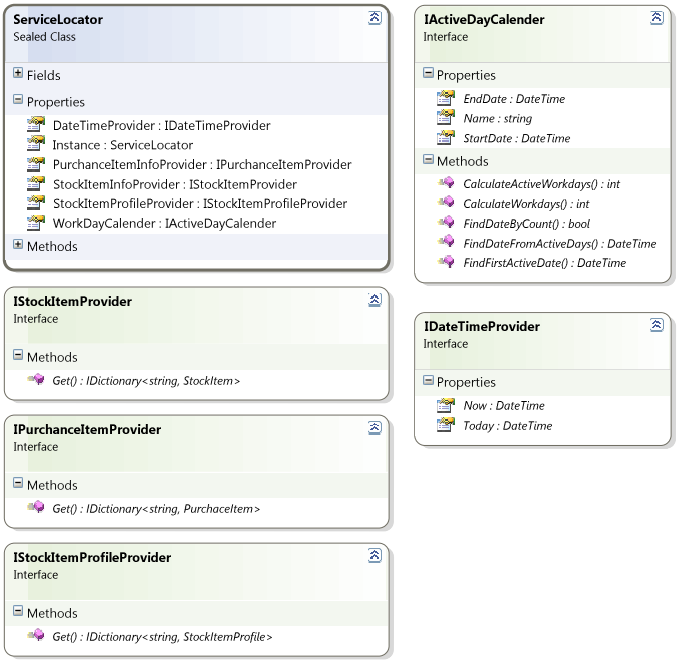
11.3.2 Dependencies

Som vist på figur 13, er strategien for at forbedre performance, at uddelegere database operationer til andre klasse, som kan optimere operationen (samle til bunke). Disse interfaces skal nu introduceres til den konkrete service. I figur 16 ses klassen og interfaces som har med dependencies at gøre.

ServiceLocator[28] pattern er valgt i den nye implementering. En konkret instans af denne kan gives til servicen på konstruktions tidspunkt. ServiceLocator holder så interfaces til alle afhængigheder. Valget af ServiceLocator pattern beror mest på, at det var en måde at komme i gang på. Skulle behovet komme, vil det være relativt let at konvertere til et dependency

injection framework. Så servicen, for givet en instans af ServiceLocator, holder instanserne på de delservices, som der er behov for.

Et klassisk eksempel på *dependency injection*.
Vi introducerer servicen til de nødvendige afhængigheder, ved at serverer en instans af ServiceLocator som en *gateway*[29]. Servicen behøver så ikke at vide noget om, hvor informationerne kommer fra.



Figur 16 Dependencies diagram

ServiceLocator og S.O.L.I.D.

SRP er overholdt, fordi ServiceLocator har et klart veldefineret ansvarsområde, at fungere som *gateway* til konkrete eksterne implementeringer.

OCP er overholdt, da ServiceLocator klassen kan konfigureres med vilkårlige implementeringer af de udstillede interfaces.

LSP er ikke relevant i dette tilfælde, der er ingen nedarvninger.

ISP er overholdt, da hvert interface som er introduceret har et lille og veldefineret ansvar.

DIP er overholdt, og klasserne/interfaces er introduceret til servicen, netop for at overholde dette princip.

Et eksempel på den fleksibilitet S.O.L.I.D. giver mulighed for, kan illustreres med interfacet IDateTimeProvider. Interfacet er en abstraktion over det at kunne finde en dato og et tidspunkt. Interfacet er introduceret af to grunde. For det første fordi kunden har opstillet en lidt speciel regel (som ikke skal implementeres, brugerne er nødt til vide det): Efter klokken 14 på en arbejdsdag, skal kaldet til servicen betragtes som værende fra den efterkommende dag (i dag +

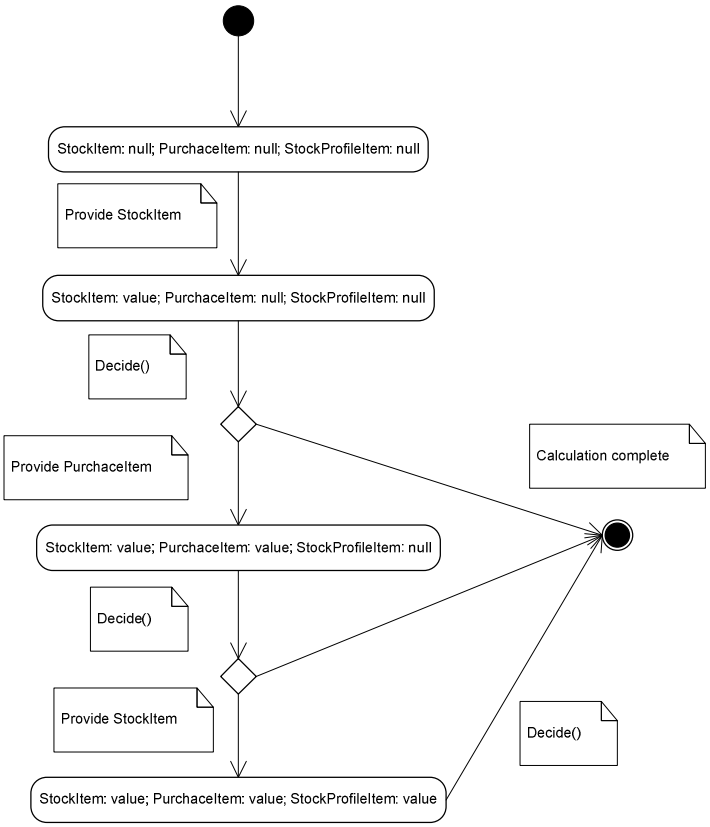
en arbejdsdag). Alle som benytter servicen forventes at kende denne regel. Denne type skrivebordsskuffe regler har en tendens til at give anledning til fejl, og systemet bør kunne håndtere dette, hvorfor IDateTimeprovider interfacet er blevet introduceret. Når servicen benytter dette interface vil vi kunne lave en variant som overholder denne regel.
Den anden grund er test. For at kunne teste systemet, har vi brug for statiske testdata, hvor vi ved hvilke datoer vi har med at gøre. Ved brug af IDateTimeProvider interfacet kan vi introducere et mock object[30] hvor vi ønsker at styre de tidspunkter vi vil teste på.

11.3.3 Entities

Figur 17 viser de væsentlige klasser som servicen benytter. StockItem, PurchaceItem og StockProfileItem er de identiteter som har informationer til at afgøre, hvornår en vare kan leveres. ActiveDayCalender holder styr på, hvilke dage der er arbejdsdage.

Figur 17 Identiteter

StockInquiryResultCalculator klassen har ansvaret for at afgøre, hvornår en vare kan leveres. Afgørelsen træffes efterhånden som den får informationer (StockItem, PurchaceItem og StockItemProfile). Figur 18 illustrerer denne proces. Første *state* indeholder ingen data, hvorfor der ikke kan træffes nogen afgørelse. Næste skridt er at give instanser af basisvare oplysninger, hvorefter *Decide* metoden kaldes. Metoden undersøger så om der er nok information til rådighed til at rræffe en afgørelse. Sådan fortsætter afgørelsesflowet, indtil en afgørelse er truffet.



Figur 18 StockInquiryResultCalculator

SRP er overholdt, da hvert objekt kun har et ansvarsområde.

OCP er ikke nødvendigvis overholdt. Det er ikke muligt at udvide StockInquiryResultCalculator's afgørelses algoritme. Dette kunne introduceres, men det er næppe et område som vil ændre sig. Med hensyn til dato styring, jævnfør diskussion i 11.3.2, så benytter StockInquiryResultCalculator sig af IDateTimeProvider og IActiveDateCalender til at lave dato beregninger, så på det punkt er der mulighed for at påvirke beslutningsprocessen.

LSP brydes ikke. StockInquiryResultCalculator nedarver fra StockInquiryResult, og de bryder ikke princippet.

ISP er overholdt, da der ikke er nogle interfaces i denne kategori.

DIP er overholdt, om end, ligesom for OCP, kunne man ændre StockInquiryResultCalculator til at uddelegere de enkelte beslutnings algoritmer.

11.3.4 Det der mangler

Servicen er på nuværende tidspunkt ikke i stand til at kører mod en faktisk database, da vi ikke har implementeret de interfaces, som skal give dataadgang. Men vi har ikke behov for det. Grundet de *plugin* egenskaber servicen har fået ved brug af *dependency injection,* kan vi på et

senere tidspunkt implementere de nødvendige klasser. P nuværende tidspunkt har vi nok til at kunne vurdere, hvorvidt den nye arkitektur har de ønskede performanceegenskaber.

Den nye arkitektur tillader endog at vi genbruger det eksisterende framework, hvis det var ønskeligt. Ved at benytte et *adapter* pattern kan vi levere konkrete implementeringer af de forskellige interfaces. Disse konkrete klasser kan så benytte frameworket til at skaffe data.

11.3.5 Interface ændring

Hvis vi kikker lidt på dette interface, ser vi en forskel fra den gamle service; signaturen er ændret. Denne service kan mere en den gamle, man kan forespørge med et antal ønskede varer og hvornår disse varer senest skal kunne leveres. Den kan dermed godt svare på det oprindelige spørgsmål (Quantity: 1, WorkDayLimit: 5), og konverter sit resultat til en farvekode.
Hvorfor nu denne udvidelse?

Den beslutning må jeg, Jakob, tage på mine skuldre. Dette er direkte i strid med YAGNI[31] (you ain't gonna need it) princippet. Tilføjelse af funktionalitet som der ikke er brug for. Hvad der er endnu værre, tilføjelse af kompleksitet. Kunden har bestemt heller ikke efterspurgt denne feature. Ved designet af interfacet kunne jeg ikke se nogen større forøgelse af kompleksitet ved tilføjelserne. Faktisk var det indlysende at udvidelsen skulle med for at undgå referencer til konstanter som ligger i den gamle kode. Den gamle kode har kendskab til fem hverdage. Så på det grundlag valgte jeg at lave interfacet mere generelt (og mere fleksibelt), uden væsentlig forøgelse af kompleksitet.

Kompleksitetsforøgelsen består nu i, af at vi nu er nødt til at have et lag som oversætter fra det gamle interface til de nye, både på vej ind i servicen og på vej ud (kan løses ved en Adapter pattern[32]). Det originale serviceinterface er bevaret i kataloget (og namespace) Indicator med en implementering af det oprindelige interface.

Fordelen ligger i en mere generel kode, som vil kunne benyttes i andre sammenhænge.

11.3.6 Cache

Der er ikke implementeret en cache i den nye løsning. Men som en lille øvelse og for at teste fleksibiliteten, vil vi her skitsere, hvordan en cache ville kunne laves.

11.3.6.1 StockInquiryDictionary oversigt

Klassen StockInquiryDictionary er intern datastruktur som benyttes af servicen til at finde hvilke StockInquiryResultCalculator instanser der skal findes data til. Hele dens levetid er vist her i servicens primære metode, Inquire:

```
public IEnumerable<StockInquiryResult>
Inquire(IEnumerable<StockInquiryItem> items)
{
   if (items == null) return new List<StockInquiryResult>(0);
   var dictionary = new StockInquiryDictionary(items, serviceLocator);
   this.Decide(dictionary);
   var result = GenerateResult(dictionary);
   return result;
}
```

Når servicen skal finde hvilke varer, den skal hente data til (*get()* metoderne i figur 13) sker det på følgende måde (hvor stockItemInfos er de varer der er hentet):

```
var stockItemInfos =
StockItemInfoProvider.Get(dictionary.GetUndecidedKeys());
```

Der er en metode på StockInquiryDictionary for alle tre *get()* metode i flowet.

På denne måde kan vi så lave en nedarvning af StockInquiryDictionary som håndterer cache.

11.3.6.2 Implementering

Vi laver en nedarvning af StockInquiryDictionary (SID), ved navn: StockInquiryDictionaryCache (SIDC).
Følgende skal implementeres:

1. Opret en *static IDictionary<StockInquiryResultCalculator>* i SIDC, som skal holde vores resultater.
2. Overskriv SIDC constructor, og opret kun nye StockInquiryResultCalculator instanser, hvis ikke de findes i cache (og indsæt dem), ellers benyt cache instancer.
3. Overskriv SIDC *GetUndecidedPurchaceItemKeys()* til kun at returnere manglende PurchaceItem (null & ikke beregnede). Så henter vi ikke længere PurchaceItem fra databasen, hvis den cached.
4. Tilføj timestamp på StockItem (dermed også ændring i implementeringer af IStockItemProvider).
5. Ændrer servicens første *get()* metoder i flowet. Hvis den hentet StockItem indeholder ændringer i forhold til StockInquiryResultCalculator instansen, så markeres StockInquiryResultCalculator til *undecided* og oplysninger opdateres.
   Denne ændring vil også fungere med den eksisterende.
6. Ændrer servicens constructor til at benytte den nye cache udgave af datastrukturen.

Disse trin er forsimplet, nogle yderligere ændringer er nødvendige, såsom at tilpasse SID til at have virtuelle metoder og lave en cache nøgle.

11.3.6.3 Resultat

Omend løsningen ikke er optimeret, ligesom den refaktoriserede service, vil ovenstående løsning kunne anvendes. Denne variant håndterer ikke optimerede *partstock* database opslag. En fortsat udvikling af cache vil kunne føre til en cache styring med samme performance karakteristika som den refaktoriserede service.

Kun punkt 4 & 5 griber væsentlig ind i den eksisterende kode.

En skitse til en løsning som ovenstående er lavet, men ikke grundigt testet eller fuldt implementeret (eksisterende tests kører dog). På cached varer ser det ud til at være en tidsbesparelse på 47 % i gennemsnit.

11.4 S.O.L.I.D.

Den nye implementation er fortaget med hensyntagen til S.O.L.I.D. principperne. Hvis de er ”implementeret” skulle koden være robust, fleksibel, lettere at vedligeholde og genanvendelig.

11.4.1 The Single-Responsibility Principle

Denne mener vi er overholdt. De enkelte klasser har et afgrænset og klart ansvar.

11.4.2 The Open/Closed Principle

Princippet har været brugt fra starten af designet af den nye kode. Klasserne og modulet er lukket med hensyn til modifikation, men designet til at være åben for udvidelser ved hjælp af *dependency injection*.

11.4.3 The Liskov Substitution Principle

Denne problemstilling opstår ikke i designet. Den eneste klasse som nedarver fra en anden er StockItemResult -> StockItemResultCalculator (figur 17). Og denne relation bryder ikke princippet.

11.4.4 The Interface Segregation Principle

Dette princip menes at være fulgt, da alle interfaces menes at være afgrænset til helt specifikke områder.

11.4.5 The Dependency-Inversion Principle

Princippet er benyttet som en grundsten i modellen af servicen, specielt på grund af unit tests. Introduktionen af ServiceLocator klassen gør at alle afhængigheder, som ikke er direkte involveret i servicens domæne, bliver angivet udefra.

11.4.6 S.O.L.I.D. Konklusion

Alt i alt har vi benyttet principperne igennem hele modellen af den nye service. Endvidere kan vi se at nogle af de lovede egenskaber ser ud til at være til stede.

Fleksibiliteten viser sig allerede ved det faktum at vi ikke er bundet til en database. Et andet eksempel er at vi kan ændre på dato beregnings modellen, uden at skulle ændre ved servicens kode.

Vedrørende genbrug, ser det også ud til at virke. Der er eksempelvis lavet en generel klasse til kalender håndtering, en generel måde at håndtere lagerprofil opslag. Sådanne klasser vil kunne genbruges andre steder.

Om koden er mere robust, kan være lidt svært at gennemskue. Hvis vi definerer robusthed som det modsatte af skrøbelighed (*fragility*)[33], så mener vi, grundet den modulære opbygning, at ændringer i et modul sjældent vil have ødelæggende effekt for hele systemet.

Vedligeholdelse er lidt svært at svare på, vi har endnu ikke skulle vedligeholde koden. Men igen, grundet den modulære opbygning vil vi mene, at den vil være lettere at vedligeholde.

11.5 Profiling

Vi har taget det samme sæt data som er blevet benyttet til at profile den oprindelige kode. Skrevet en adapter til at anskaffe data, og kørt målingerne på samme måde. Resultaterne ses i nedenstående Tabel 14.

Antal varer der spørges på	Elapsed time for hele servicen målt i ms. 1 connection	Elapsed time for hele servicen målt i ms. 2 connections.	Mål opfyldt
1	24	24	Ja
5	27	28	Ja
55	175	93	Ja
109	326	173	Ja

Tabel 14 Performancemålinger på den funktionelle konvertering

Som resultatet viser, er performance målene nået med det nye design.

Et lidt underligt resultat er at det tager næsten samme tid for en og for fem vare. Grunden til dette må være den optimerede metode til at hente data, og at hente de fem vare bliver kun til 1 kald til databasen.

At forespørgsel af en vare er hurtigere i den nye, end i den gamle, må skyldes at kalender er initialiseret under service opstart i den nye, men hentes for hvert kald i den gamle.

11.6 Tidsforbrug

Det har kun taget 2 dage at implementere den funktionelle konvertering. Her til skal lægges analyse tiden som blev brugt i starten. Igen skal det nævnes at Jakob har tænkt en del over dette længe før dette projekt startede. Så det er muligt at tidsforbruget burde være større. I stedet for at gætte på dette, har vi blot angivet det faktiske forbrug.

11.7 Konklusion

Målet med omskrivningen var primært at opnå en tilfredsstillende performance. Målene er nået, men vi havde ingen garanti for at det ville lykkedes, inden processen gik i gang. Strategien fra start var, at hente større mængder af data fra databasen ved forespørgsler. Strategien kunne vi have analyseret forinden projekt start, og derved fået en indikation om det vil give det ønskede performance mål.

Analysen som er blevet fortaget, har vist sig meget værdifuld. Analysen identificerede de forskellige ansvarsområder, angav, hvilke klasser der skulle være og hvilke kompositioner imellem klasserne der skulle på plads. For eksempel var det klart at kalenderberegningerne skulle ligge i sin egen klasse. Og da den nu ligger i sin egen klasse, afkoblet fra alle andre områder, så er den lettere at teste. Med testen på plads, har vi en byggeklods/komponent der kan benyttes af systemet. Den sikkerhed, at de enkelte komponenter var færdige da kompositionerne skulle laves, gjorde arbejdet mere overskueligt.

Den indbyggede parallelisme i programmet var ikke med i første iteration af strukturen. Ideen opstod ved at studerer strukturen i flowet, hvor det blev tydeligt, at eksempelvis database kald, bliver samlet sammen inden udførselen, og derfor bliver et atomart skridt i flowet, som kan

udføres parallelt. Den nye struktur gav på den måde nogle afledte muligheder, som ikke var blevet identificeret eller tænkt da processen gik i gang.

12 Relateret arbejde

Det har undret os, at vi ikke rigtigt har kunnet finde ret meget om emnet. Der er masser af Blogs om emnet som er meget enslydende. Men vi har ikke kunnet finde ret meget andet. Det lyder mærkeligt synes vi.

Vi vil dog godt fremhæve et par af de blogs vi har fundet (afsnit 12.2 og 12.3),

12.1 Martin Fowler[Refactoring]

Fowler har et enkelt afsnit i bogen, hvor han diskuterer situationer, hvor refaktorisering ikke vil være en fordel. Eksempelvis hvis den eksisterende kode er så ringe at den ikke fungerer, vil det være bedre skrive noget nyt. Den eksisterende kode skal fungere, for at man kan refaktorisere den. En interessant betragtning er, at man ikke skal refaktorisere, hvis man er meget tæt på deadline. Samtidig siger han også at lidt refaktorisering er bedre end ingen.
Hans iagttagelser falder fint i tråd med de observationer vi ser.

12.2 [Jos] "Things You Should Never Do Part 1" af Joel Spolsky fra april 2000.

Den bliver henvist til denne artikel fra en del andre blogs.
I denne artikel er den generelle holdning, at man aldrig skal lave rewrite. Han kommer med flere projekter som begrundelse:

- Netscape 6 var 3 år undervejs og deres markedsandel blev mindre undervejs i processen.
- Borland tabte dBase til Microsoft Access af samme årsag.
- Microsoft forsøgte at skrive Word fra bunden igen under codename: Pyramid.
  Projektet blev skrinlagt.

Nogle af Joels grundlæggende argumenter for refactor:

- Time to Market.
  Dette er nok det argument, han vægter højest.
- Throw away old code is throwing away knowledge.
- Why should the new code be better than the old one?
  Old code has been used!

Citat:

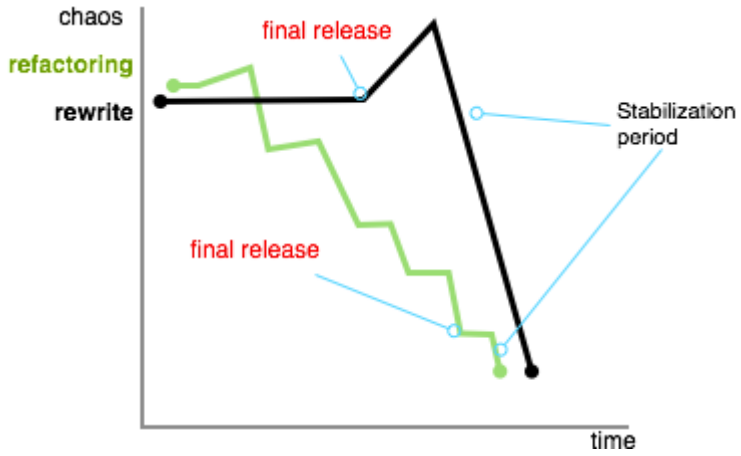
There is a subtle reason that programmers always want to throw away the code and start over. The reason is that they think the old code is a mess. And here is an interesting observation: *They are probably wrong.* The reason they think the old code is a mess is because of a cardinal, fundamental law of programming: "It's harder to read code than to write it"

12.3 [EoC] Edge of Chaos | Agile Development Blog

Michael Dubakov kommer ikke med en entydig konklusion. Han kommer dog med at par kommentarer:

- When you write from scratch, you add such a large portion of chaos that it is hard to predict the final result.
- Rewrite may look faster but the stabilization period is longer.

Han har en lille graf (Figur 19) som er hans forventning til hvordan de to forløb måske vil se ud.



Figur 19 Sammenligning

Dette er jo nok netop den usikkerhed, som får de fleste til at vælge refaktoriseringsvejen.
Man ved hvad med har, man ved ikke hvad man får.
Den initiale del af grafen er vel også det billede vi ser. Om den sidste del af grafen så også er korrekt i vores tilfælde, er jo svært at spå om.

13 Appendiks

13.1 Source koder

Alle projekter er samlet i en zipfil. Den indeholder fem undermapper, med hver et Microsoft Visual Studio 2010 Professional projekt i. Alle projekterne er skrevet i C# kørende under DotNET 4.0.

13.1.1 Portet

Koden som den så ud efter oversættelse til DotNET. Den består af tre projekter: Framework (dll), StockIndicator (dll) og TestStockIndicator (exe, som ikke virker).

13.1.2 Portet_Under_Test

Koden som den så ud efter at være kommet under test. Den består af tre projekter: Framework (dll), StockIndicator (dll), StockIndicatorTest (dll, unit test) og TestStockIndicator (exe).

TestStockIndicator.exe kan køre performance tests og ligger under understien TestStockIndicator\bin\Debug.

13.1.3 Portet_Refactor

Samme struktur som Portet_Under_Test.

13.1.4 Portet_Refactor_2

Samme struktur som Portet_Under_Test.

13.1.5 Stockinquiry

Den nyimplementerede løsning består af tre projekter: StockInquiry (dll), StockInquiryTest (dll, unit test) og TestStockInquery (exe)

Unit testen er bygget med NUnit (ww.nunit.org) version 2.5.5.

TestStockIndicator.exe kan køre performance tests og ligger under TestStockIndicator\bin\Debug.

13.2 Testprogram

Alle vores programmer gennemløber den samme test. Dog er StockInquiry's program en tilrettet kopi. Det er derfor muligt umiddelbart at sammenligne resultater mellem kørsler og på tværs af løsninger.

Figur 20 viser test Refactor_2 test program, men de andre er magen til.

```
Enter test iterations (default 3, max 30): 2
Initializing...

Starting test (2 runs per test):

Test1   - [xx]  1st:    30 ms.  2nd:    17 ms.  Av.:    24 ms.
Test5   - [xx]  1st:    34 ms.  2nd:    20 ms.  Av.:    27 ms.
Test50  - [xx]  1st:   313 ms.  2nd:   174 ms.  Av.:   244 ms.
Test100 - [xx]  1st:   595 ms.  2nd:   330 ms.  Av.:   463 ms.


Press 'r' to run. Any other key to exit.

5,2924 ms.
5,7211 ms.
35,9958 ms.
65,5901 ms.
```

Figur 20 Testprogram afvikling

I ovenstående tilfælde køres testen med 2 iterationer.

1st: Er gennemsnitstiden for de 2 kørsler.

2nd: Er gennemsnits tiden for to kørsler igen, men hvis løsningen har cache, (som denne har) så er alle kørsler bortset fra den første med cache. I dette tilfælde en uden og en med cache

Av: Gennemsnit af 1st og 2nd.

I løsninger med cache, vil programmet dumpe de enkelte kørselstider ud, når testen lukkes. I overstående tilfælde vil det sige at 5,2924 er tid i cache for test1 og 5,7211 er tid i cache for test2 og så videre.

13.2.1 Test data og måling

En tidsmåling startes ved kald af servicen og stoppes når resultatet er beregnet.

De klasser som benyttes som test stubs, har indbygget forsinkelse som beskrevet i 5.2.

Testdata grundlaget er de 109 forskellige testvarer som er beskrevet i 5.2. Det er de samme varer som er udvalgt til de forskellige tests for alle løsninger.

14 Litteraturliste

[Feathers] ”Working effectively with legacy code” af Michael C. Feathers (2004)

[Martin&Martin] “Agile Principles, Patterns and Practices in C#” kapitlerne 7-12 af Robert C. Martin & Micah Martin (2007)

[SLDI] Service Locator vs. Dependency Injection: http://martinfowler.com/articles/injection.html, Martin Fowler

[PEAA] “Patterns of Enterprise Application Architecture”, Martin Fowler el.al. (2003)

[GOF] “Design Patterns Elements of Reusable Object-Oriented Software” af Eric Gamma, Richard Helm, Ralph Johnson og John Vlissides (1994-1995)

[WIKI] Wikipedia internet resource: http://en.wikipedia.org/wiki/Main_Page

[DIME] Dimecast internet resource: http://dimecasts.net/

[INGRES] Ingres Corporation: www.ingres.com
Ingres ejer Open Source databasen Ingres.
Ingres ejer Open Source udviklingsværtøjet Application By Forms(ABF)
Ingres ejer udviklingsværtøjet OpenROAD som er et objekt Orienteret udviklings værktøj som er delvist Open Source

[Unistar] Unistar er et ERP system udviklet af Bording Data.

[JoS] www.joelonsoftware.com/articles/fog0000000069.html

[EoC] Edge of Chaos | Agile Development Blog
Artikel af Michael Dubakov - Refactoring Vs. Rewrite fra November 2009
www.targetprocess.com/blog/2009/11/refactoring-vs-rewrite.html

[Refactoring] ”Refactoring: Improving the Design of Existing Code” af Martin Folwler (Addison-Wesley, 1999)

---

[1] [Feathers] Preface.
[2] [Feathers] Chapter 1 Changing Software
[3] [Feathers] Chapter 2 Working with Feedback
[4] Unit test: [WIKI] http://en.wikipedia.org/wiki/Unit_testing
[5] [Feathers] Chapter 2 Working with Feedback
[6] [Feathers] Chapter 2 Working with Feedback
[7] [Feathers] Chapter 13 I Need to Make a Change, but I Don’t Know What Tests to Write

[8] [WIKI] http://en.wikipedia.org/wiki/Legacy_code
[9] [Martin&Martin] Sammendrag taget fra konklusionerne I kapitlerne 8 - 12
[10] [Martin&Martin] The Single-Responsibility Principle (SRP) side 115
[11] [Martin&Martin] The Open/Closed Principle (OCP) side 121
[12] [Martin&Martin] The Liskov Substitution Principle (LSP) side 135
[13] [Martin&Martin] The Interface Segregation Principle (ISP) side 163
[14] [Martin&Martin] The Dependency-Inversion Principle (DIP) side 153
[15] Test driven development (TDD): [WIKI] http://en.wikipedia.org/wiki/Test-driven_development
[16] IOC: [WIKI] http://en.wikipedia.org/wiki/Inversion_of_control
[17] [Feathers] Extract interface s 362
[18] [GOF] Section 1.6, side 18
[19] [Martin&Martin] *Anticipation and "Natural" Structure* og *Putting the "Hooks In"* side 128 - 129
[20] [DIME] http://dimecasts.net/Casts/ByTag/SOLID%20Principle
[21] EAN tidligere European Article Number i dag er European udskiftet med International. En standard for udformning af varenumre.
[22] [Martin&Martin] *Anticipation and "Natural" Structure* og *Putting the "Hooks In"* side 128 - 129
[23] [Martin&Martin] *Putting the "Hooks In"* side 129
[24] www.ingres.com og http://en.wikipedia.org/wiki/Openroad
[25] [PEAA] Record Set side 508.
[26] Dette kunne være via vores framework
[27] Det skyldes at nogle identiteter i sig selv er *service* og har behov for at kunne anskaffe informationer fra andetsteds.
[28] [SLDI] og [WIKI] http://en.wikipedia.org/wiki/Service_locator_pattern
[29] [PEAA] Gateway pattern, side 466.
[30] [Feathers] side 27
[31] YAGNI: [WIKI] http://en.wikipedia.org/wiki/You_ain't_gonna_need_it
[32] [Martin&Martin] side 498 og [WIKI] http://en.wikipedia.org/wiki/Adapter_pattern
[33] [Martin&Martin] side 105