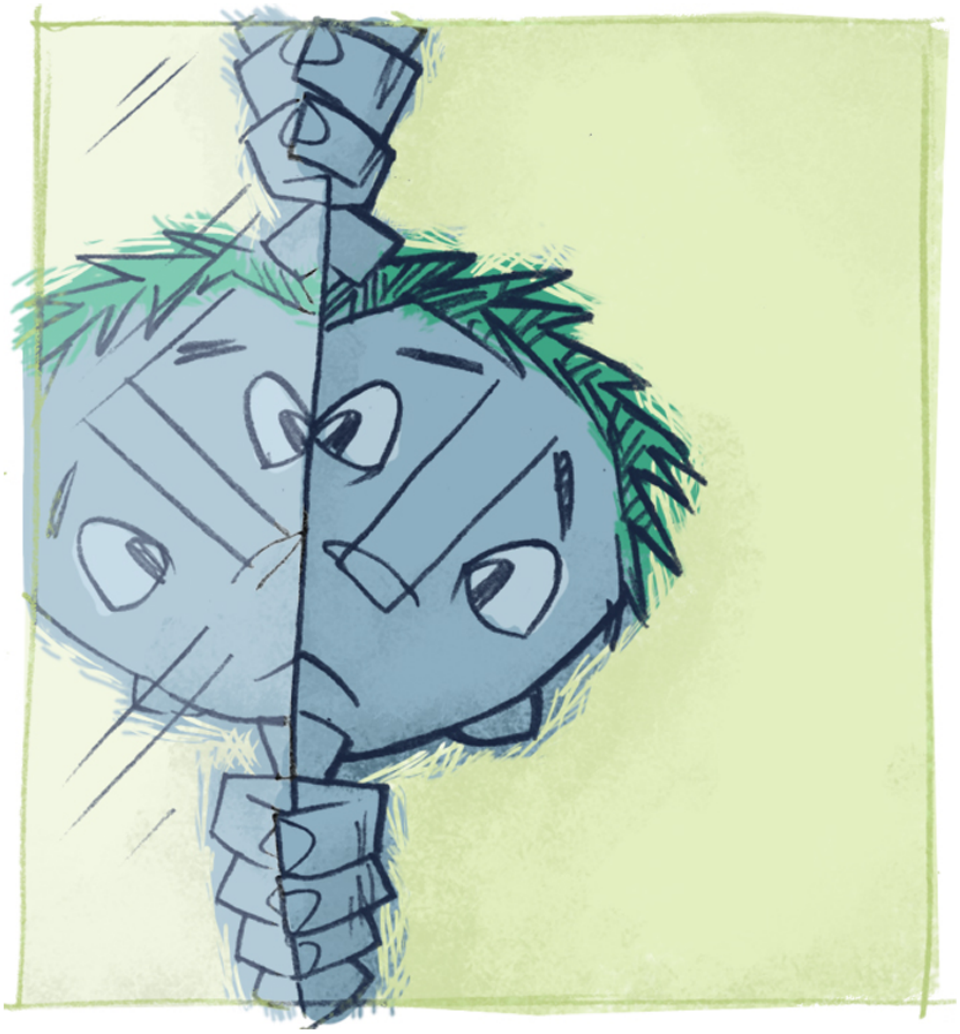
Reflektiv Programmering



... af SW2A217

AALBORG UNIVERSITET

Det Teknisk-Naturvidenskabelige Basisår
Elektronik og Elektroteknik
Strandvejen 12-14
Telefon 96 35 97 31
Fax 98 13 63 93
http://tnb.aau.dk

Titel:

Reflektiv programmering

Tema:

Netværk & Algoritmer

Projektperiode:
P2, forårssemesteret 2010

Projektgruppe:
SW2A217

Deltagere:
Mikkel Skov Christensen
Anders Eiler
Adi Hadzikadunic
Esben Pilgaard Møller
Magnus Stubman Reichenauer
Bjarke Hesthaven Søndergaard

Vejledere:
Hovedvejleder: Arild Haugstad
Bivejleder: Heather Baca-greif

Oplagstal: 10.

Sidetal: 84.

Bilagsantal og –art: 1 stk. CD-Rom.

Afsluttet den 25. maj 2010.

Synopsis:

Vi vil i denne rapport sætte fokus på reflektiv programmering og hvornår det kan betale sig at anvende og implementerer det. Derudover vil vi undersøge to af de største IT-skandaler i Danmark og forsøge at finde svar på, hvad der gik galt. Kunne reflektiv progammering være blevet anvendt, og kunne det havde sparet de forskellige virksomheder for tid og penge på længere sigt? Udover dette har vi taget kontakt til KMD, hvor et møde har fundet sted. Mødet gav et bedre grundlag og indblik i, hvordan virksomheder anvender reflektiv programmering, og hvordan de forskellige modeller i udviklingsprocesen bliver brugt (bl.a. vandfaldsmodellen). Dernæst sammenlignes to stykker kildekode, som er skrevet i Java, hvor der gives en analyse af de fordele og ulemper der opstår, når man anvender reflektiv programmering.

Mikkel Skov Christensen

Anders Eiler

Adi Hadzikadunic

Esben Pilgaard Møller

Magnus Stubman Reichenauer

Bjarke Hesthaven Søndergaard

Forord

Indeværende rapport er udarbejdet i perioden 2. februar 2010 - 25. maj 2010. Rapporten er skrevet i et sprog der tager udgangspunkt i, at du som læser har et godt kendskab til computere, IT og programmering i al almindelighed.

De første par kapitler gennemgår og fortæller om emnet og problemstillingen, samt redegører for de grundlæggende teknologier i et tilgængeligt sprog. De tekniske begreber og forklaringer vil derfor først fremkomme under løsningen af problemet. Senere i rapporten vil den konkrete løsning af problemet fremkomme, hvorefter der til sidst analyseres og konkluderes på denne ift. projektet som helhed.

Forsideillustrationen er lånt af Ward Jenkins[1], Portland, Oregon, United States. Udover den åbenlyse sammenhæng med reflektion, er illustrationen også valgt pga. den stemning, som ligger i billedet. Forholdet mellem programmører og reflektiv programmering er nemlig ikke altid helt godt, og spørger man rundt vil man i så fald også finde, at programmører ofte kigger nervøst og tilbageholdende omkring sig, efter reflektiv programmering. De fleste uden helt at vide, hvad det faktisk indeholder af muligheder.

Spørger man de samme udviklere om deres oplevelse med reflektiv programmering efter, at de har arbejdet med det, vil deres reflektion over forløbet dog ofte resultere i et smil.

- SW2A217

[1] http://wardomatic.blogspot.com/2005/08/illustration-friday-reflection.html

Indholdsfortegnelse

1 Indledning

I indeværende rapport vil der være fokus på de tekniske aspekter af softwareudvikling, samt de processer og udfordringer der ligger til grund for de personer, som udvikler softwaren, og hvordan disse processer påvirker den videre vedligeholdelse.

Idet en virksomhed udvikler et stykke software, kan man forstille sig, at der nedsættes en arbejdsgruppe til at planlægge projektforløbet. Ofte vil der blive taget forbehold for senere opdateringer og udvidelser mv. af det pågældende stykke software, men det er nærmere reglen end undtagelsen, at der før eller siden kommer opdateringer eller vedligeholdelsesopgaver, som i større eller mindre grad vil kræve modifikationer af det eksisterende software.

Det varierer naturligvis fra projekt til projekt, i hvor stort omfang disse opdateringer kræver modifikationer af det eksisterende software. Sikkert er det dog, at såfremt der skal foretages modifikationer af eksisterende software for at kunne implementere en opdatering, er det en proces som koster både tid og penge for virksomheden bag softwaren. En situation, som man i sagens natur helst undgår.

I kraft af, at tid – og ikke mindst penge – er omdrejningspunktet for størstedelen af alle virksomheder i dag, er det derfor interessant at undersøge, hvorvidt det er muligt at gå ad nye veje i udviklingsprocessen, for på forhånd at spare sig for nogle af de problemer, der kan blive aktuelle i forbindelse med efterfølgende opdateringer og vedligeholdelsesopgaver. Det kan fx være at undersøge nye organisatoriske såvel som teknologiske muligheder i forhold til udviklingsprocessen.

Indeværende rapport vil derfor klargøre de processer og teknikker, som i stor udstrækning bliver anvendt af professionelle i dag i forbindelse med almen softwareudvikling. Fokus vil være på nogle af de problemstillinger, der er i forbindelse med anvendelsen af disse teknikker. Sidst – men ikke mindst – vil det blive undersøgt hvorvidt der findes alternativer, som kan løse et eller flere af de problemer, som der formodentlig stødes på i løbet af rapporten.

Problemanalyse 2

IT bliver et vigtigere og vigtigere element i nutidens samfund. Der er efterhånden IT-løsninger til de fleste ting, og stort set alt arbejde har på den ene eller den anden måde IT involveret. Derfor er industrien bag IT, som blandt andet indeholder softwareudvikling, blevet et større arbejdsområde end tidligere. Det er en selvfølge, da der skal nogle folk til at udvikle de IT-systemer der bruges. Men nogle gange sker der misforståelser, fejl, dårlig planlægning og lignende, hvilket kan føre til problemer i udviklingen af disse IT-systemer. I nogle tilfælde løber problemerne løbsk, og bliver til deciderede IT-skandaler. I tilfælde som disse kan man spørge sig selv: Hvad gik galt? - og forsøge at analysere hvorfor det gik galt. Der vil blive kigget på to af de mest omtalte IT-skandaler i medierne i Danmark i nyere tid, som har haft både økonomisk og samfundsmæssig betydning. Disse skandaler er arbejdsmarkedsstyrelsens IT-system AMANDA og COWIs system til folkeskolens prøver og eksamener.

2.1 AMANDA

AMANDA (ArbejdsMArkedets Nye DAtabase), var et IT-system der blev bestilt af arbejdsmarkedsstyrelsen i 1994. I 1996 blev der bevilliget 214,5 millioner kroner til projektet af folketingets finansudvalg, og det var meningen, at AMANDA skulle udkomme inden for 2 år, altså senest i 1998. AMANDAs formål var at hjælpe arbejdsmarkedsstyrelsen med at forenkle den daglige arbejdsgang. Men de visioner og tanker som folkene bag AMANDA havde, blev hurtigt gjort til skamme. AMANDA blev ikke udgivet inden for de 2 år som planlagt – tværtimod, fristes man til at sige. Der gik dobbelt så lang tid, nemlig 4 år, før AMANDA blev udgivet. Dette skete i foråret 2000. Men det var blot det første af mange kommende problemer for AMANDA. Systemet fungerede ikke optimalt, og der var på udgivelsestidspunktet rigtig mange fejl i systemet.

Udover det tekniske aspekt med de mange fejl, blev økonomien dråben, som fik AMANDA-bærget til at flyde over og blive den største IT-skandale i Danmarks historie. Der blev som sagt afsat 214,5 millioner kroner til projektet, men det endte med udgifter der løb op i 650 millioner kroner! Udover det store beløb var de driftsmæssige omkostninger også meget høje. I rigsrevisionens rapport fra 2001, var de forventede driftsudgifter 93 millioner kroner

om året.[1]

Det var først i en evalueringsrapport fra 2003 at AMANDA blev erklæret stabilt og funktionsdygtigt, men i selvsamme rapport blev der beskrevet, at der var problemer med AMANDAs udviklingsplatform som var *HPS* (High Performance System). Man mente, at dette system ville være forældet i 2008. Og ganske rigtig; i november 2008 blev AMANDA erklæret pensionsklar, og blev sat ud af drift. AMANDA blev erstattet af det nyere IT-system: Arbejdsmarkedsportalen.

AMANDAs store problem lå i systemets virkemåde, da det både var besværligt og ustabilt. En af AMANDAs opgaver i arbejdsstyrelsen bestod i at registrere nye arbejdssøgende i systemet. Denne opgave tog imidlertid længere tid med AMANDA end med dets forgænger. Det ville tage det gamle system mellem 10 og 20 minutter at registrerer en ny arbejdssøgende, hvor det for AMANDA - efter 3 måneder i fuld drift - tog næsten en hel time! Grunden til dette var, at der blev brugt helt op imod 50 skærmbilleder til én registrering. Alle disse problemer peger i retning af, at systemet, teknisk set, ikke var korrekt opbygget.

Sammenlignet med vandfaldsmodellen (læs mere i afsnit 2.5.1), og antaget at AMANDA er udviklet efter denne metode, må den klare fejl ligge i designfasen. Denne antagelse er bygget på artikler, udtalelser og lignende, fra flere af landets medier, vedrørende AMANDA. Fejlen bestod i at arbejdsmarkedsstyrelsen havde sat nogle klare krav til udviklingerne af AMANDA, som skulle opfyldes. Arbejdsmarkedsstyrelsen lagde meget vægt på at det skulle gå meget hurtigt at registrerer nye arbejdssøgende, hvilket var kravet, som udviklerne forsøgte at leve op til. Men for at sikre, at hvert skærmbillede kom hurtigt nok frem, blev der blot mange flere skærmbilleder, hvor der ikke var mange elementer på, for at mindske indlæsningshastigheden. Dette løste selvfølgelig problemet med at hvert skærmbillede kom hurtigt, men det skabte et andet problem. I stedet for at hele processen blev hurtigere, hvilket oprindeligt var det arbejdsmarkedsstyrelsen havde tænkt sig, blev hele processen langsommere, grundet at der i stedet kom alt for mange skærmbilleder frem. Hele konceptet med AMANDA var, at det skulle være et hurtigt system til arbejdet, men en kommunikationsbrist mellem arbejdsmarkedsstyrelsen og udviklerne betød, at systemet blev meget langsommere dets forgænger. Hvis blot kommunikationen imellem arbejdsmarkedsstyrelsen og udviklerne havde været bedre, kunne man muligvis have taget det nyopståede problem i opløbet, så udviklerne kunne have fået af vide, at det ikke kun var hurtige skærmbilleder der var vigtige, men at hele systemet skulle fungere hurtigt som en helhed! Dette kunne have betydet, at systemet havde undgået at blive den store IT-skandale, som det blev, og at staten havde sparet en stor sum penge på dette projekt.

2.2 Folkeskolens digitale afgangsprøve

En af de største IT-skandaler i nyere tid, som har kostet statskassen flere millioner kroner på papirprøver, er en IT-baseret eksamensform som firmaet COWI fik til opgave at lave. Skandalen blev for alvor kendt i 2007, da en biologieksamen blev aflyst grundet et systembrud pga. en overbelastet server. Dette medførte siden flere aflysninger af IT-

[1] Alle nøgletal er taget fra [13]

baserede eksamener, og endte med at 39.000 skoleelever skulle til en prøve på papirform [3], som har kostet statskassen 9-10 millioner kroner [2]. Dette har fået flere uafhængige kilder til at stille spørgsmålstegn ved COWIs udvikling af de IT-baserede eksamener og prøver, som de er hovedansvarlige for.

COWIs forklaring på sagen var, at CSCs server - som var deres driftcenter - ikke kunne klare så mange samtidige brugere, og derfor gik serveren ned, da eleverne påbegyndte deres afgangsprøve i biologi. Dette fremgår af deres rapport til Undervisningsministeriet, hvor de også forklarer hele forløbet om den uheldige hændelse. COWI havde leveret et system til Undervisningsministeriet, hvor de havde benyttet @ventues, som var ansvarlige for at applikationen skulle virke. I forvejen driver CSC evalueringsplatforme til folkeskolen. Selvom systemet virkede fint året forinden, valgte COWI at flytte afgansprøverne til denne nye platform. Platformen krævede dog nogle ændringer i selve opbygningen af applikationen, som @ventures implementerede natten forinden prøverne. Ifølge rapporten blev applikationen testet inden prøverne fandt sted, men COWI erkender dog, at man ikke har været særlig opmærksom på de risici, som kunne opstå, ved at implementere rettelserne så sent i forløbet.

I rapporten fremgår det også, at der ikke gik lang tid før eleverne oplevede et nedbrud under eksamen. Først da CSC genstartede deres servere kunne de fleste af eleverne afslutte prøven. Der sluttede problemerne dog ikke - der var stadig problemer for nogle af eleverne, og det hele endte også med, at COWI skulle betale en bøde på 1,3 millioner kroner [6]. Problemet er endnu ikke løst, og man oplever stadigvæk problemer med systemerne som COWI er ansvarlige for.

2.3 Initierende Problem

AMANDA og folkeskolernes afgangseksamen er udviklingsprojekter, der begge er gået galt. Begge projekter har under udviklingen været igennem en "proces". Fælles for alle processer er – uanset om det er softwareudvikling – at det er en udvikling, hvor hovedemnet forandrer sig fra et stadie til et andet. Denne udvikling ønskes naturligvis optimeret bedst muligt. Et ønske, der ikke gik i opfyldelse i forbindelse med AMANDA og folkeskolernes afgangseksamen.

Overstående påstand er især gældende indenfor softwareudvikling. Mulige kunder ønsker det bedste produkt til den laveste pris. Dette skaber en øget konkurrence mellem firmaerne i branchen, og sætter høje krav til processen, som forvandler idéen til produktet.

Forholdet mellem hurtig udvikling og kvaliteten af udviklingen af produktet skal være i orden. Det er ikke til gavn, at være i stand til hurtigt at lancere et produkt, som lige så hurtigt bliver udkonkurreret. Når man snakker kvalitet indenfor software, er vedligeholdelse og opdateringer meget vigtige faktorer. Det er disse elementer, der i høj grad sikrer produkternes generelle sikkerhed (ift. hacking mv.) og deres evne til at følge med markedet, i takt med at dette ændrer sig, og at der muligvis opstår nye sikkerhedskrav som følge af nyopdagede IT-sårbarheder.

Med udgangspunkt i det lyder et indledende spørgsmål derfor:

Hvordan optimeres udviklingen af software, således at det endelige produkt bliver fleksibelt, sikkert og hurtigt, og samtidig tillader let vedligeholdelse?

Det indlysende svar på dette spørgsmål er, at ansætte flere udviklere og give dem mere tid. Der søges dog en løsning, som opfylder målet, og samtidig sparer de omkostninger som flere udviklere og et længere udviklingsforløb ville medføre.

Dvs. det er nødvendigt at finde en løsning, der skaber en hurtigere og mere effektiv udviklingsproces, uden at kvaliteten af produktet falder og uden at omkostningerne stiger.

Det initierende problem lyder derfor således;

Hvordan optimeres udvikling af software, således at kravene til produktet bliver opfyldt – heriblandt let vedligeholdelse – og samtidig holde omkostningerne nede?

2.4 Driftsmæssige problemer

At optimere softwareudvikling er dog ikke nødvendigvis en nem opgave. Naturligvis forsøger IT-virksomheder at optimere deres udviklingsproces, så de kan opnå den størst mulige indtjening i forhold til omkostningerne. De står dog overfor en større mængde problemstillinger, som det er nødvendigt at tage højde for, inden en realistisk ide til optimering af softwareudvikling kan gives.

2.4.1 Udgivelse & salg af software

Produktionen af software ændre sig fra firma til firma. Udviklingen kan foregå på mange forskellige måder, ved et bredt udvalg af metoder og i forskelligt tempo. Men langt de fleste virksomheder, der producere og udvikler software, har imidlertid samme mål: At få den størst mulige økonomiske gevinst ud af deres produkter.

Der er mange forskellige typer af software, som alle har hver deres muligheder. Nogle typer giver mulighed for at skrive og arbejde i, nogle giver mulighed for beskyttelse på Internettet mens andre giver muligheder for underholdning. Alle slags software appellerer på den ene eller anden måde til en gruppe af forbrugere, og disse udvikles typisk af forskellige typer virksomheder.

Der er de store firmaer som Microsoft, der bl.a. laver software i form af operativsystemet Windows. Denne type software appellerer til en stor gruppe af mennesker, da alle computere skal have et operativsystem installeret for at kunne fungere og cirka 90% af verdens computerbrugere anvender Windows [11]. Windows er et licenspligtigt program, hvilket betyder, at man skal købe Windows og en licensnøgle for at få adgang til softwaren. Dette har Microsoft tjent mange penge på, og eftersom omkring 90% bruger Windows, er det derfor et af de mest succesfulde softwarefirmaer.

Mange af de store firmaer bruger samme fremgangsmåde som Microsoft mht. udviklingen af software. Dvs. de sælger licenspligtigt software, så man skal betale for den, ligesom man skal for Windows. Nogle af disse programmer er: Norton AntiVirus, Microsoft Office,

Adobe Photoshop og andre former for operativsystemer, eksempelvis Apple OS X. Alle disse produkter er man nødt til at betale for at kunne anvende. Føromtalte produkter har ofte en "trial-periode". Dvs. at man kan anvende produktet i et bestemt tidsinterval, og når denne periode er udløbet, må man betale licensen, hvis man fortsat ønsker at anvende produktet.

Andre firmaer udvikler og markedsfører deres produkter anderledes. Et eksempel på dette er antivirusprogrammet Avast. De laver deres software således, at der fremstilles en version af programmet, som er standard og bliver udgivet som "freeware". Dette betyder, at programmet kan downloades og benyttes gratis. Denne "freeware"-version har som regel begrænset funktionalitet sammenlignet med betalingsversionen. Hvis man ønsker det fulde udbytte af programmet, er man nødt til at købe sig til dette i form af versionerne, der sædvanligvis bliver kaldt for "Pro-editions". I disse programmer er al funktionaliteten fuldt tilgængeligt, og man kan benytte alle de funktionaliteter programmet oprindeligt var lavet til at kunne. Andre programmer, der markedsføres i samme stil som Avast antivirus, er blandt andet Adobe Acrobat, Ad-Aware, DivX, Nero Burn osv.

Det kan altid diskuteres, hvilken metode der er bedst, og hvilken en der tilfredsstiller kunderne mest. Altafgørende er det dog, hvilken af metoderne der er størst økonomisk gevinst i, ift. omkostningerne af udviklingen og vedligeholdelse af softwaren.

2.4.2 Udviklingstid kontra arbejdskraft

Softwareudvikling tager naturligvis sin tid, og jo længere tid udviklingen tager, dets større bliver omkostningerne. Tiden, som projektet tager, afhænger til en stor grad af hvor stort projektet eller programmet man er ved at udvikle er.

Lad os antage, at der skal udvikles et rigtigt stort program, som skal indeholde en stor mængde funktioner. Et af de væsentlige driftmæssige problemer heri er udviklingstiden, som man skal bruge på at udvikle dette. Et stort program vil kræve mange arbejdstimer, og her skal man som virksomhed foretage et vigtigt valg: Vil man bruge de ekstra midler over en kort tidsperiode til at ansætte mange udviklere, der vil medføre, at programmet bliver hurtigere færdig, og derved hurtigere kan komme på markedet og tjene pengene ind igen? Eller vil man i stedet trække lønningerne til programmørerne ud, ved kun at ansætte få ad gangen, og i stedet lade det tage noget længere tid at udvikle programmet, før det så kan udkomme og tjene penge?

Dette spørgsmål kan kun besvares efter analyse af behovet på markedet på det givne tidspunkt. Hvis det pågældende program er i høj kurs, og er meget efterspurgt, så kan det helt sikkert være en fordel for firmaet at lægge flere penge i programmører omgående, og derved få programmet hurtigt færdigt, så det kan udnytte dets høje efterspørgsel til at skabe en hurtig indtjening. Hvis efterspørgslen ikke er høj på det givne tidspunkt, men firmaet ved at programmet bliver godt, og det stadig kan sælge, kan det være en fordel at spare på de omgående programmørlønninger, og blot bruge den tid det tager at få udviklet produktet. Uanset hvilken metode firmaet vælger, vil lønningerne til programmørene være det samme. Det er ligemeget mht. lønningerne, om det vil kræve 10 programmører at udvikle et program over 6 måneder, som hvis et sæt af 20 programmører ville kunne

udvikle det samme over 3 måneder. Derfor er det blot op til firmaet at vurderer om de vil bruge flere midler, på flere programmører, over kortere tid for at få fremstillet et produkt hurtigt, eller på færre programmører over længere tid for at trække lønningerne lidt ud.

Driftmæssige problemer kan også bestå i problemer med at få selve koden til at fungerer. Dette er dog svært at beskrive, da firmaer, der udvikler software, naturligvis ikke er interesserede i at dele de problemer, som de har med at udvikle softwaren med offentligheden. Dette kunne i væreste fald sætte et dårligt lys over firmaet, eller medføre dårlig omtale, hvilket ingen firmaer er interesserede i. Derfor holder firmaerne disse driftsmæssige problemer for sig selv.

Overordnet set ligger problemet imidlertid i, at det er nødvendigt for virksomhederne at tage et valg vedrørende den mængde resourcer, de vil bruge for at få programmet færdigt indenfor et givent tidsrum og herved opnå den størst mulige indtjenning.

For at gøre dette valg nemmere, er det virksomhedernes interesse at optimere udviklingsprocessen omkring softwareudvikling, så de kan sænke deres omkostninger, og stadig få udgivet produktet indenfor den samme tidsramme.

2.5 Generel softwareudvikling

Måden hvorpå man udvikler software, er nødvendig at kende, før det er muligt at komme med konkrete forslag til optimering af softwareudvikling. Dette afsnit er ikke skrevet med henblik på at belyse de tekniske aspekter i generel softwareudvikling, men i højere grad for at forklare de organisatoriske værktøjer, som ligger til bag softwareudviklingsprojekter.

Hensigten med softwareudvikling er at udvikle nye programmer. Ofte programmer, som løser en specifik opgave på en ny og smart måde. Når man taler om softwareudvikling, så omhandler det i stor grad programmering, optimering og vedligeholdes af et stykke software. At udtrykke en idé som et færdig stykke program, som kan realisere idéen og udfører formålet.

2.5.1 Vandfaldsmodellen

Vandfaldsmodellen er tilset som en model til udvikling af software. Grunden til den har fået tilnavnet vandfald, er fordi softwareudvilking betragtes som konstant nedadgående gennem faserne:

1. Kravspecifikation.
2. Design.
3. Konstruktion (implementation eller kodning).
4. Integration.
5. Afprøvning og fejlfinding.
6. Installation.
7. Vedligeholdelse.

Denne model blev introduceret i 1970 af W. W. Royce. [4]

Vandfaldsmodellen bliver ofte brugt af softwareudviklere. Modellen har både nogle fordele og ulemper for udviklerne. Nogle af fordelene er, at man opnår en god struktur i udviklingen af software, da man går systematisk til værks. Ulempen er, at hvis man finder en fejl løbende i softwaren, så skal der måske ændres i kravspecifikationen og design – hvormed går man et skridt tilbage i processen. En vigtig fase i modellen er integrationsfasen, hvor man sammensætter de forskellige komponenter. Hvis man opdager en fejl, skal man ændre i konstruktionen, og det kan give problemer og krav på visse ændringer. Derfor er der delte meninger om modellen. Grunden til, at den har fået tilnavnet vandfaldsmodellen, er at det er lettest at følge processen den ene vej – altså ned ad – og det begynder hurtigt at blive kompliceret, hvis man er nødsaget til at gå tilbage i processen. For at forstå princippet, beskrives de enkelte punkter i modellen:

Kravspecifikation - Her beskrives kravene til hvad systemet skal kunne. Fx kunne en kravspecifikation for et website være, at man tillader at anonyme kan rette i en given tekst. Når alle kravene er opstillet, går man videre til næste punkt.

Design - Her laves en oversigt som programmørerne skal følge. Designet skal være en plan for, hvordan man implementerer kravene fra kravspecifikationen i det faktiske software. Når oversigten er lavet, implementeres designet af programmørerne.

Konstruktion - Her laves implementationen af programmørerne, og det faktiske stykke software udvikles.

Integration - Når alle delene af systemet er færdige, sættes de sammen i integrationsfasen. Dette kan fx være at nogle layoutkomponenter bliver sat sammen med nogle textkomponenter, så det udgør et samlet system.

Afprøvning og fejlfinding - Herefter afprøves og testes softwaren, hvor fejl, der stammer fra de tidligere faser, fjernes.

Installation - Installation af softwaren finder sted.

Vedligeholdelse - Her vedligeholdes softwaren, hvor man kan fjerne eventuelle fejl og optimere softwaren, så det bliver ved med at følge med tidens krav.

2.5.2 Agile udviklingsmetoder

Agile udviklingsmodeller er en samling af modeller, skabt som modsætning til vandfaldsmodellen. Deres formål er at give udviklere et alternativ til den meget låste fremgangsmåde, som vandfaldsmodellen foreskriver.

Retningslinjen for alle agile udviklingsmodeller er dokumenteret i det *Agile Manifest*: [12]

- Individer og interaktioner over processer og værktøjer.
- Virkende software over omfattende dokumentation.
- Kundesamarbejde over kontraktforhandling.

- Reaktion på ændringer over at følge en plan.

Principperne bag det agile manifest, er følgende 13 principper: [12]

1. Vores højeste prioritet er at tilfredsstille kunder gennem tidlig og kontinuerlig levering af værdifuld software.
2. Velkom ændrede krav, selv i den sene del af udviklingen. Agile processer udnytter forandring til fordel for kundens konkurrenceevne.
3. Lever virkende software jævnligt, fra et par uger til et par måneder, med en præference for en kortere tidshorisont.
4. Forretningsfolk og udviklere skal arbejde sammen dagligt igennem projektet.
5. Lad motiverede individer arbejde sammen i projekter. Stil det miljø og støtte, som de har brug for, til rådighed, og stol på at de fuldfører deres opgave.
6. Den mest virksomme og effektive metode til at overbringe/udveksle information til og mellem et team af udviklere, er ved at samtale ansigt til ansigt.
7. Virkende software er den primære måling af fremskridt.
8. Agile processer fremmer bæredygtig udvikling.
9. Sponsorer, udviklere og brugere burde kunne holde et konstant tempo i ubestemt tid.
10. Kontinuerlig fokus på teknisk kvalitet og god design fremmer agilitet.
11. Enkelhed – kunsten at maksimere mængden af arbejde som ikke skal gøres, er afgørende.
12. Den bedste arkitektur, krav og design kommer fra selv-organiserede teams.
13. Med regelmæssige mellemrum reflekterer teamet over, hvordan man bliver mere efffektiv, hvorefter det ændrer sig i overensstemmelse hermed.

Principperne og manifestet er de eneste retningslinjer for, hvad agil udvikling er. Derfor kan man ikke definitivt definere en udviklingsmetode som agil. Som fælles træk kan der dog kortfattet forklares, at de grundlæggende fremgangsmåder er **tidsbegrænsede iterationer** med **adaptiv**, **evolutionær** raffinering af planer og mål. Omdrejningspunktet i alle begreber og værdier er fleksibilitet og agilitet. Hvis agil udvikling har et motto, må det være at "omfavne forandring".

- **Tidsbaseret iterativ udvikling** består af at dele hele projektet ned i mindre bidder, hvor hver bid strækker sig over en tidsbegrænset periode, typisk 1-6 uger. Hver periode kaldes for en iteration, og er i sig selv et miniprojekt, hvis mål er at skabe en del af det overordnede projekt. Dette kræver, at man i starten har en kravspecifikation. I slutningen af hver iteration har man således et færdigt stykke software. Typisk anbefaler de agile metoder, at man skærer ned på listen over de mål man har for den nuværende iteration, hvis man kommer i tidspres, frem for at forringe kvaliteten, eller bryde tidsbegrænsningen. Det man så skærer fra, tages der senere fat på igen.
  Under en iteration må ingen ydre krafter påvirke udviklerne eller kravspecifikationen for den igangværende iteration. Agil udvikling påskønner konstant forandring, men ikke kaos. Derfor er en moderat stabilitet nødvendig.

- **Risiko- og klientdrevet iterativ planlægning** er to forskellige måder at prioritere, hvilke funktioner fra kravspecifikationer, som er vigtigst for projektet, og i hvilken rækkefølge de skal udvikles.
  - **Risikodrevet iterativ planlægning** foreskriver, at man i hver iteration starter med de krav til produktet, som udviklerne vurderer, er de mest teknisk udfordrende. Disse krav vil sandsynligvis påvirke det samlede produkt på en eller flere omfattende måder, således prioriteres disse først, frem for til sidst. [12]
  - **Klientdrevet iterativ planlægning** foreskriver, at man i hver iteration starter med de krav fra kravspecifikationen, som klienten vurderer, er de vigtigste for dem, da disse fx har størst markedsværdi. Før hver iteration planlægger klienten **adaptivt** og meddeler udviklerne, hvilke funktioner næste iteration skal udvikle. På denne måde har klienten løbende kontrol og valgmulighed, efterhånden som nye resultater bliver tilgængelige. [12]

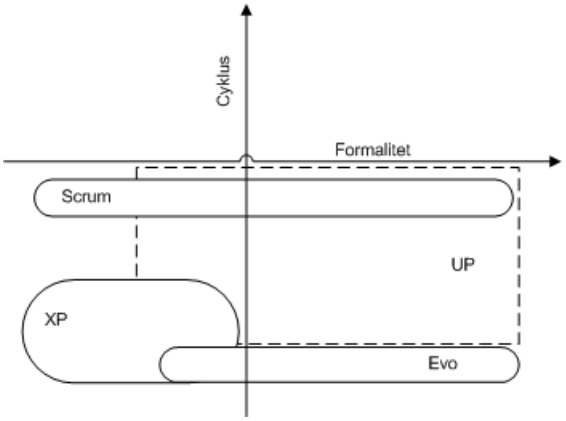
- **Evolutionær udvikling** foreskriver at kravspecifikationer, planlægning, estimeringer og muligheder udvikles og/eller raffineres under iterationerne, frem for at være fastlåste. Dette er et resultat af uforudsigelig opdagelse, og forandring i udviklingen, som evolutionær udvikling også foreskriver. [12]

- **Adaptiv udvikling** er meget relateret til evolutionær udvikling. Det forudskriver at de forskellige elementer adapteres/tilpasses ud fra feedback. Intentionen er den samme som med evolutionær udvikling, men navnet lægger mere vægt på feedback-respons i udviklingen. [12]

Til at beskrive hvordan de forskellige agile metoder skiller fra hinanden, kan man koge forskelligehederne ned til to koncepter; *formalitet* og længden af *cyklusser*.

Formaliteten beskriver mængden af dokumentationsarbejde som der kræves under udviklingen. Med cyklus menes hvor lang hver iteration må være.

Til at danne et overblik over forskellighederne imellem de mest kendte agile metoder, er figur 2.1 dannet. Figuren illustrerer at Scrum foreskriver længere cyklusser end Evo, og at man ved brug af UP (Unified Process) har større valgfrihed end ved brugen af XP (eXtreme Programming). For yderligere information om de specifikke metoder, se [12].



Figur 2.1. Placering af metoder i fht. formalitet og cyklus

2.6 Almen udvikling i Java

Softwareudvikling er et begreb, der indeholder mange forskellige aspekter, både organisatorisk, som set i foregående afsnit, men også teknisk i forhold til, hvilket niveau af programmering, der anvendes (kun imperativ eller objektorierenteret programmering – hvilke der kan læses mere om i hhv. afsnit 2.6.2 og 2.6.3). Derfor vil en rapport om dette emne typisk blive meget overfladisk. Det er derfor nødvendigt at begrænse softwareudvikling ned til en mere konkret del af denne proces. Denne rapport vil primært fokusere på programmeringsdelen af udviklings- og tildels vedligeholdelsesprocessen, samt hvordan disse kan optimeres ved brug af forskellige niveauer af programmering. Hvilke niveauer af programmering, der er til rådighed, varierer dog alt efter hvilket programmeringssprog softwaren skrives i. Denne rapport vil primært fokusere på Java og den type applikationer, der skrives i dette sprog.

Java bruges primært til at udvikle to typer af software: Applets og applikationer. Dette på trods af, at Java også har serverfunktioner (servlets) integreret i dets API. Applets er den type applikationer, der oftest afvikles i browsere. Det fungerer ganske simpelt ved, at disse applets hentes ind i browseren og – uden at være en integreret del af hjemmesidens kildekode – afvikles. Applikationer er de mere traditionelle stykker software, som bl.a. afvikles i baggrunden på en PC, men kan være alt fra tekstbehandlingsprogrammer til computerspil. Java er et programmeringssprog i stil med C og C++, der kan bruges til at udvikle de fleste typer af software. Java understøtter imidlertid objektorienteret programmering, hvilket giver øgede muligheder i designprocessen af en applikation, i forhold til C, der ikke understøtter objektorienteret programmering.

Der er dog en klar fordel ved at udvikle software i Java fremfor andre programmeringssprog. Når et stykke software udviklet i Java kompileres, oversættes det til såkaldt bytecode. [24]

Ved afvikling af programmet oversættes denne kode til maskinkode af den såkaldte *JVM* (Java Virtual Machine - en cross-platform applikation, der er nødvendig for at Java-applikationer kan afvikles) inden den faktiske afvikling finder sted. Oversættelsen til

maskinkode sker ved brug af "Just-in-time compilation", hvor koden først oversættes til maskinkode, når den skal afvikles. Dvs. hvis en del af koden kun bruges når brugeren af softwaren udfører en bestemt handling, oversættes koden først til maskinkode, hvis dette gøres. Det element, der adskiller Java fra en stor gruppe andre programmeringssprog er, at der findes udgaver af JVM til både Windows, OS X og Linux, der alle er i stand til at oversætte den samme slags bytecode (dette er bl.a. ikke tilfældet med C, hvor fx sockets på Linux anvender sys/socket biblioteket og på Windows anvender winsocket biblioteket). Det vil sige, at Java er et cross-platform sprog, hvis applikationer kan køre på de mest udbredte operativsystemer.

Udover dette er Java og dets API open-source, hvilket betyder, at der ikke ligger nogen omkostninger i at udvikle i Java - udover naturligvis arbejdstimerne et givent projekt tager – da udviklingsværktøjer og lignende er tilgængeligt gratis.

Software udviklet i Java udnytter typisk de funktionaliteter, Java naturligt stiller til rådighed. Dette gælder især større web-applikationer, der ofte er programmeret som Java-applets. Et eksempel på dette er adgangskontrol til netbank, der er programmeret som en Java-applet [5].

2.6.1 Typisk fremgangsmåde ved programmering i Java

Som beskrevet er et af de vigtigste elementer ved udvikling af software at have en fortåelse af, hvor man arbejder sig hen imod. Typisk vil dette være en form for kravspecifikation. Ud fra denne kravspecifikation opstilles der en generel model for softwaren, hvor de forskellige elementer det skal indeholde og de nødvendige funktioner beskrives. For at få et overblik over, hvordan softwaren skal udvikles og opbygges, benyttes det faktum, at Java er objektorienteret, til at opstille en model over det endelige stykke software.

Når der programmeres objektorienteret er softwaren – som uddybet i afsnit 2.6.3 – bygget op af klasser, der hver især repræsenterer de elementer softwaren skal behandle. Hvis det er et system til et sygehus, der involverede forhold, hvor læger og patienter er indblandet, vil elementerne fx være `læge` og `patient`. Begge disse klasser repræsenterer personer, så i et stykke software ville disse klasser være inddelt som underklasse til en ny klasse, nemlig klassen `person`. Et eksempel på dette kan være at både patienter og læger har en alder, et navn, et CPR-nummer osv. Disse attributter ville ligge i klassen `person` og blive nedarvet til underklasserne læge og patient. Derudover vil underklasserne have specifikke attributter for deres klasse. Fx kan klasse `læge` indeholde en attribut for, hvilket felt den givne læge er specialiseret indenfor.

Kort sagt vil processen ved udvikling i Java ofte starte med at kortlægge, hvilke klasser, der er nødvendige og hvilke relationer de har til hinanden. Dette gøres normalt ved brug af *UML* (Unified Modeling Language), som er en teknik til at beskrive relationerne mellem de forskellige klasser i et stykke software.

Efter kortlægningen af programmet i forhold til de elementer, det skal indeholde, skal de nødvendige underrutiner kortlægges. I et program som det førnævnte sygehussystem vil det være nødvendigt at skulle udføre handlinger med ens objekter. Hvis det er en patient,

skal vedkommende tildeles en seng og behandling. Der skal altså integreres en stor mængde funktioner i de forskellige klasser, så der kan foretages de nødvendige handlinger med objekterne. Disse skal kortlægges på samme måde som klasserne skal, da nogle metoder og subrutiner tillige nedarves.

Denne proces med kortlægning af softwarens specifikationer udføres tillige med brugergrænsefladen (hvis programmet har en), indtil en fuld model over softwaren er dannet.

Når denne proces er færdiggjort kan de forskellige elementer, der skal programmeres, uddeles til forskellige programmører, og ved at det er kortlagt vil alle programmørerne vide, hvilke funktionaliteter de skal implementere i deres stykke af koden, for at den kan fungere med de resterende dele.

2.6.2 Imperativ programmering

Imperativ programmering kan betegnes som sekventiel programmering, hvor koden skrives i den rækkefølge den ønskes afviklet. Imperativ programmering benytter sig af principperne introduceret i struktureret programmering.

Struktureret programmering er brug af 3 elementære strukturer, hvis formål er at sikre kvalitet i en kode, men tillige simplicitet, hvilket gør vedligeholdelse af koden en nemmere proces. De 3 strukturer er: *sekvens*, *iteration* og *selektion*. Sekvens er, at kodens kommandoer og operationerne, der ønskes udført i programmet, skrives i den rækkefølge disse ønskes. Iterationer er løkker, hvori en given operation gentages. Og selektion er, hvor det ved at undersøge givne parameter i et program bestemmes, hvilke sekvenser af programmet, der skal afvikles.

Imperativ programmering giver ikke de store muligheder for at håndtere større mængder af data, da den eneste måde dette kan gøres med, er de indbyggede datatyper i det givne programmeringssprog som `int`, `char` osv. Denne måde at opbevare data er udemærket til mindre applikationer, men til applikationer, der skal håndtere større mængder data (fx et arkiv over, hvad en forretning har på lager), er det meget upraktisk at håndtere dette med de små lokale variabler. I sådan et tilfælde vil det være mere hensigtsmæssigt at bruge objektorienteret programmering.

Da det med imperativ programmering er besværligt at håndtere store mængder sammenhængende data (der er kun `arrays` og `structs` og datatyperne), er det ofte mere besværligt at benytte til større applikationer fremfor andre mere avancerede teknikker, der indeholder større funktionalitet. Dette udelukker dog ikke imperativ programmering fra at være en del af større applikationer, da der i disse ofte vil være en stor mængde selv-definerede funktioner og metoder, som programmet benytter sig af, og disse vil ofte være programmeret ved brug af principperne i struktureret programmering.

Imperativ programmering er i basis programmering der foretages i et lavniveau sprog, og det understøttes fuldt ud af Java, da der både er selektions- (if-sætninger og switch-konstruktion) og iterations (while- og for-løkker) metoder.

2.6.3 Objektorienteret programmering

Til en hvis udstrækning er *OOP* (Objekt orienteret programmering) blot en ændring af synspunkt. Man kan anse objekter i traditionelle programmeringstermer som hverken mere eller mindre end et sæt variabler, koblet sammen med en række subrutiner til manipulation af disse. Det er derfor også muligt – om ikke andet ved hjælp af lidt god fantasi – at anvende objektorienterede teknikker i ethvert programmeringssprog. Der er dog stor forskel på de programmeringssprog, som giver mulighed for anvendelse af OOP, og de som aktivt understøtter det. Et objektorienteret programmeringssprog såsom Java inkluderer en række muligheder, som i høj grad differentierer det fra traditionelle strukturerede programmeringssprog. For effektivt at kunne drage fordel af de muligheder som objektorienteret programmering tilbyder, skal man derfor tilpasse sin tankegang.

Ved at anvende OOP fremfor imperativ programmering opnår man ofte i udviklingen af større projekter en række tekniske fordele, ved at flere problemer kan løses med andre metoder. Disse fordele tæller både det praktiske i at udvikle softwaren, men også den mængde hukommelse og loadtime softwaren vil kræve ved afvikling.

I stedet for traditionel top-down struktureret programmering, oprettes et antal *klasser*. En klasse kan fx være "bil". I en klasse er der en række *metoder*. En metode kan fx være "kør" eller "brems". En klasse er en "kasse", som indeholder operationer for en bestemt type *objekter* (i dette tilfælde – biler). Derigennem har man mulighed for at lave flere *instanser* af et objekt. Hvis man fx har 10 biler, kan man instantiere klassen 10 gange, og ud fra klassens metoder lave 10 forskellige biler - alt sammen ud fra samme klasse, der i princippet kan anses som en skabelon for instanserne. Et eksempel på klassen "bil" kan ses på figur 2.2, hvor kassen er delt op i tre mindre felter med navnet på klassen øverst. Midterste del er variablerne, der hører til hver bil. Nederste del er de metoder, som hver bil har.

Figur 2.2. Illustrering af klassen "bil"

2.7 Reflektiv programmering

For at optimere udviklingsprocessen, er det nødvendigt at undersøge om der findes metoder, der ved konsekvent brug, kan optimere denne proces. I forbindelse med objektorienteret, findes en programmeringsmetode kaldet reflektiv programmering, der potentielt kan gøre dette muligt.

Grundstene i reflektiv programmering er, at designe computerprogrammer på en måde, hvormed det anskuer – reflekterer over – og håndterer dele af sig selv under *runtime* (når programmet afvikles).

Selve ideen med denne proces er, at programmet skal reflektere over, hvilken tilstand det er i, i øjeblikket, og derudfra modificere sin egen struktur og opførsel.

Reflektiv programmering er oftest anvendt i programmeringssprog, der har høj abstraktion fra computertekniske detaljer, der typisk er omtalt som høj niveau programmeringssprog. Det kan vises, at Java har højere abstraktion fra computertekniske detaljer end fx Assembler, ved at vise, hvordan tre variabler deklareres i hver. I Java kan du deklarere tre `int` variabler ved at skrive: `int x = 23`, `int y = 0x3fce`, `int z = 42`. For at kunne gøre samme i Assembler skal dette se således ud:

```
        .data           ! start en gruppe of variable deklarationer
x:      .word   23      ! int x = 23;
y:      .word   0x3fce  ! int y = 0x3fce;
z:      .word   42      ! int z = 42;
```

Her skal det nævnes at udråbstegn indikerer en kommentar. I Assembler anvender man dog ikke typer, dette er kun i Java o.lign. programmeringssprog. x, y og z er adresserne i hukommelsen.

I reflektiv programmering reflekteres der over objekter, derfor kræver reflektiv programmering et programmeringssprog, der er objektorienteret.

2.7.1 Tankegangen i reflektiv programmering

Selve tankegangen i – og ideen bag – reflektiv programmering er, at gøre programmet i stand til at foretage opgaver, som typisk vil blive udført af programmøren. Disse kan der læses mere om i afsnit 4.2, men de har følgende fællestræk i måden de udføres på:

1. Undersøge programmets struktur og data.
2. Foretage beslutninger baseret på undersøgelsen.
3. Ændre programmets opførsel, struktur eller data på baggrund af beslutningen.

Hvis ovenstående kan udføres af computeren uden yderligere assistance fra programmøren kan dette hjælpe med at gøre vedligeholdelse og andre opgaver nemmere.

Når man snakker reflektiv programmering, så er man også nødt til at nævne *meta-data* og *meta-programmering*. Meta kommer fra det græske efter, over eller sammen med. Meta-information er altså information om programmets egne data. Meta-programmering er selve ideen om at ændre programmet dynamisk. Reflektiv programmering er altså bygget ud fra denne idé, og bruger informationer om diverse klasser og deres metoder under runtime. Som nævnt tidligere hænger reflektiv programmering godt sammen med objektorienteret programmering, da objekter som sagt skal bruge informationer om sin egen klasse og

dertilhørende metoder. Langt de fleste objektorienterede programmeringssprog har en form for meta-programmerings API[2]. Som fx Java beskrevet i afsnit 2.6.

Programmering med reflektive metoder kan være med til at gøre det lettere at udvide programmet, således, at hvis der senere bliver ændret på kravene til programmet, så skal man ikke ind og foretage ændringer i en klasse, der allerede fungerer, men blot tilføje nye klasser, som ens reflektive metode kan anvende. Selve ideen er altså, at programmet let skal kunne udvides til at kunne ting, som måske ikke oprindeligt var tiltænkt fra programmørens side, men alligevel gør programmet bedre. Et eksempel kunne være *Apache Tomcat* (server software), der skal være i stand til at bruge nye udvidelser, så man kan tilføje moduler til afvikling af server applikationer. Til dette anvendes *servlets* (Java-programmer specialudviklet til servere), som – hvis de er udviklet efter et bestemt sæt regler – vil fungere uden problemer. Dette er nemt at gøre reflektivt, da serveren så blot skal finde de klasser, der opfylder kriterierne, for at kunne implementere dem.

Reflektiv programmering er altså en programmeringsteknik, der bygger ovenpå objektorienteret programmering, og altså muligvis optimerer udviklingsprocessen.

2.8 Opsummering

I dette kapitel har rapporten dækket det initierende problem med et analyserende syn på nogle af problemerne ved udvikling af software. Her er der nævnt nogle særlige tilfælde – COWI og Amanda (sektion 2) – hvor det i Danmark er gået galt mht. levering af software. Dette er selvfølgelig beklagelige tilfælde, men kan man også sige, at der måske er noget galt i måden man udvikler på? Derfor redegøres der også for grundprincipper i softwareudvikling, herunder agil udvikling. Da der er mange teknologiske aspekter inden for softwareudvikling, der varierer alt efter programmeringssprog, fokuseres der i rapporten på programmeringssproget Java. Reflektiv programmering er som beskrevet i sektion 2.7 en teknologi, som bygger ovenpå den objektorienterede teknologi, der vil gøre programmet i stand til at tage stilling til sin egen tilstand og foretage valg derudfra. Alt dette ligger til grund for vores problemformulering i kapitel 3.

[2] C# [17], Java [23], PHP [20], Objective-C [18], Python [21], Ruby [22]

Problemformulering 3

Der er mange problemstillinger og forskellige aspekter forbundet med optimering af softwareudvikling. På den ene side er der de organisatoriske og planlægningsmæssige aspekter, hvor tidsplaner, udviklingsmodeller, arbejdsmetoder mv. er i fokus. På den anden side ligger de tekniske aspekter, hvor specielle teknologier og programopbygning er i fokus.

De førnævnte emner dækker et utrolig bredt arbejdsområde, og denne rapport vil derfor rette dens fokus nærmere på et af underemnerne. Inden for optimering af softwareudvikling, vil der i denne rapport være fokus på de teknologiske aspekter af softwareudvikling, for gennem disse at forbedre det samlede projektforløb.

Med udgangspunkt i det forrige afsnit og det initierende problem er problemformulering derfor:

Hvis reflektiv programmering kan påvirke en softwareudviklingsproces, hvordan – såfremt det faktisk påvirker den pågældende proces – vil dette resultere i et samlet udviklingsforløb, der er optimeret, i forhold til hvad det ellers ville være?

I forhold til den fortsatte undersøgelse og analyse af softwareudvikling, vil det i denne rapport være afgrænset til at fokusere på anvendelse af alternative teknologier – herunder reflektiv programmering – til udførsel af de samme opgaver og operationer som "traditionel" udvikling. Mere konkret vil det blive undersøgt, hvorvidt der findes andre metoder, som er mere effektive end den oprindelige metode til at udføre en given handling.

Herunder vil rapporten konkret komme ind på:

- Softwareudvikling i programmeringssproget Java.
  - Hvad er Java?
  - Javas traditionelle brug.
  - Java i forhold til andre programmeringssprog.
- Reflektiv programmering som alternativ programmeringsteknik.
  - Hvilke principper ligger bag reflektiv programmering?
  - Hvilken teori ligger bag reflektiv programmering?

  - Hvilke muligheder ligger der i reflektiv programmering?
  - Hvornår er reflektiv programmering en god idé at anvende?

- Udviklingstimer for udviklerne i forbindelse med udvikling og opdatering af applikationen.

  - På hvilke punkter kan softwareudvikling optimeres ved brug af reflektiv programmering?
  - Hvad gør reflektiv programmering mere attraktivt i visse situationer fremfor andre programmeringsmetoder?
  - Til hvor stor en grad kan det betale sig at bruge reflektiv programmering fremfor andre programmeringsmetoder?
  - Giver reflektiv programmering fordele idet et program udvikles?
  - Giver reflektiv programmering fordele idet et program skal opdateres og vedligeholdes?

- Performancemæssige forskelle under afviklingen af applikationen.

  - Afvikles den reflektive kode hurtigere end ikke reflektiv kode, og hvis det er tilfældet, hvad skyldes denne optimering?

Teknologiske aspekter 4

Reflektivt orienteret programmering er en teknik, der ikke er en del af alle programmeringssprogs API. Dog har en del sprog, heriblandt Java, implementeret det i kernen af sproget. I Java er denne kerne givet ved pakken `java.lang`. I følgende kapitel vil Java og reflektiv programmering i Java blive gennemgået i yderligere tekniske detaljer.

4.1 Udvikling i Java

Java er et objektorienteret højniveau programmeringssprog, først introduceret tilbage i 1995 af Sun Microsystems. Formålet med det følgende afsnit er ikke, at du skal lære at programmere Java. Men i kraft af, at reflektiv programmering mere er et koncept end en faktisk teknologi, hvortil der findes API'er i rigtig mange forskellige programmeringssprog, er det valgt at vise reflektiv programmering ift. Java. Derfor følger her en introduktion til Java, dets syntax og typisk anvendelse.

Forklaret i mere tekniske termer: Det, som gør Java specielt ift. så mange andre programmeringssprog som fx C – hvor den eksekverbare fil kun kan køre på den platform den er kompileret til– er dets fleksibilitet overfor hvilke operativ systemer Java programmer fungerer på. Da Java programmer afvikles gennem *JVM* (Java Virtual Machine), er det nemlig afhængig af kun JVM, og ikke operativ systemet. Således fungerer alle java programmer på alle arkitekturer, sålænge JVM er tilstede. Java kildekode kompileres netop til at blive afviklet på JVM'en, som altså fungerer på alle de mest udbredte operativsystemmer(Windows, OS X & de fleste Linux-distributioner).

Java har sine små særligheder og tweaks ligesom alle andre programmeringssprog, men overordnet set er den meget generelle tankegang den samme som andre objektorienterede programmeringssprog. Syntaxen minder tilnærmelsesvis om C og C++, så med en baggrund i et af disse sprog, er det relativt nemt at bruge, når man først er blevet bekendt med de små forskelle, og den objektorienterede tankegang hvis ens baggrund er C.

Listing 4.1. Java eksempel nr. 1

```
1 public class helloWorld {
2         public static void main(String[] args) {
3                 System.out.print("Hello World!");
4         }
```

```
5 }
```

Som det ses i Listing 4.1 består Java-programmer af en klasse med en main metode, hvorfra alt i programmet startes. I ethvert Java-program, uanset størrelse, er der som minimum én `main` metode. Denne er det første, som afvikles, når et Java-program startes. Et Java-program kan godt indeholde flere `main` metoder, men dog højst en i hver klasse. Via et såkaldt `manifest` (som fremkommer i ethvert samlet Java-program i en .JAR fil) fortæller man programmet, hvilken `main` metode, som skal anvendes ved eksekvering. Dette kan dog også klares fra kommando-linjen. Ovenstående kode i Listing 4.1 udskriver simpelt "Hello World" i terminalen ved afvikling.

Listing 4.2. Java eksempel nr. 2

```
1  public class numSorter {
2          public static void main(String[] args) {
3                  double input[] = new double[100], thisInput = 0;
4                  int counter = 0;
5                  ns numSorter = new numSorter;
6
7                  do{
8                          if(counter == 100)
9                                  break;
10
11                         counter++;
12                         thisInput = TextIO.getlnDouble();
13                         input[counter] = thisInput;
14                 }
15                 while(thisInput != 0);
16
17                 ns.pleaseSort(input, counter);
18
19                 System.out.println("\nYour numbers in sorted order are:\n");
20         for (int i = 0; i < counter; i++) {
21             TextIO.putln( input[i] );
22         }
23         }
24
25         public void pleaseSort(double[] A, int count) {
26                 for(int last = count-1; last > 0; last--) {
27                         int max = 0;
28
29                         for (int j = 1; j <= last; j++) {
30                 if (A[j] > A[max])
31                    max = j;
32              }
33
34                         double temp = A[max];
35             A[max] = A[last];
36             A[last] = temp;
37                 }
38         }
39 }
```

Listing 4.2 viser et lidt mere avanceret kodeeksempel. Helt konkret er ovenstående program bygget til at modtage en række tal, og sortere dem i numerisk rækkefølge. Dette gøres

ved at tage imod en række tal i main-metoden, lægge dem i et array, og når der ikke længere indtastes flere tal til sortering, køres `pleaseSort()` (linje 25-38) metoden via det initialiserede objekt, som så sorterer tallene. Et ganske simpelt eksempel, som viser syntaksen og opbygningen af Java kode.

Det skal her noteres at elementet `TextIO` er en klasse – inkluderet i samme pakke som den øvrige kode i Listing 4.2 – hvis formål er at forenkle og forkorte håndtering af brugerdefinerede input og output. Altså en udvidelse til Java standard I/O.[1]

Oftes anvendes Java til opgaver, hvor platformuafhængighed er nødvendig. Derudover indeholder Java en række yderst effektive sikkerhedsmekanismer. Dette kombineret er bl.a. også grunden til, at langt det fleste netbanke er udviklet i Java (dette er resultatet af en undersøgelse af netbankene hos Danske Bank, Jyske Bank, Nordea og alle Sparekassen bankerne)[2]. Både grundet platformuafhængigheden, men i højere grad grundet sikkerheden i det. Det skal dog tilføjes, at valget af Java til netbankudvikling i høj grad også skyldes alternativerne, hvor det primære alternativ, ActiveX, ikke undersøtter cross-platform sprog.

I Java findes der også API's, der fungerer som en udvidelse af eksisterende. En af dem er reflektiv programmering.

4.2 Reflektions-orienteret programmering

Man står nemlig ofte med problemer, der kan løses simpelt og elegant med reflektiv programmering. Dog vil man uden reflektiv programmering ofte ende med løsninger, der er besværlige, uhåndterbare eller ustabile. Overvej selv følgende scenarier:

- Din projektleder vil have et framework, der kan tage yderligere moduler og komponenter, selv efter det er kompileret og lanceret. Du sætter nogle interfaces op, og klargører en mekanisme til at patche din *JAR-fil* (Java ARchive filer, der indeholder flere filer i en – brugt til at distribuere projekter og biblioteker), men du ved, at det vil ikke opfylde kravene til den fleksibilitet og evne til at tage imod nye moduler, som din projektleder ønsker.
- Efter du over flere måneder har udviklet på en klient-side applikation, fortæller marketingafdelingen dig nu, at du ved at bruge en anden type klient-side applikation kan øge salget. Selvom dette ville være en smart beslutning for forretningen skal du nu til at lave din klientside applikation markant om.
- Det offentligt tilgængelige API til dit modul skal være i stand til kun at acceptere kald fra specifikke pakker for at sikre, at dit modul ikke bliver misbrugt. Du tilføjer en paramater til hvert enkelt API kald, der indeholder navnet på pakken, hvori den kaldende klasse er. Men nu skal berettigede brugere af modulet ændre alle deres, og uvelkommen kode skal blot forfalske et pakkenavn.

[1] TextIO.java kan ses her: http://math.hws.edu/javanotes/source/TextIO.java

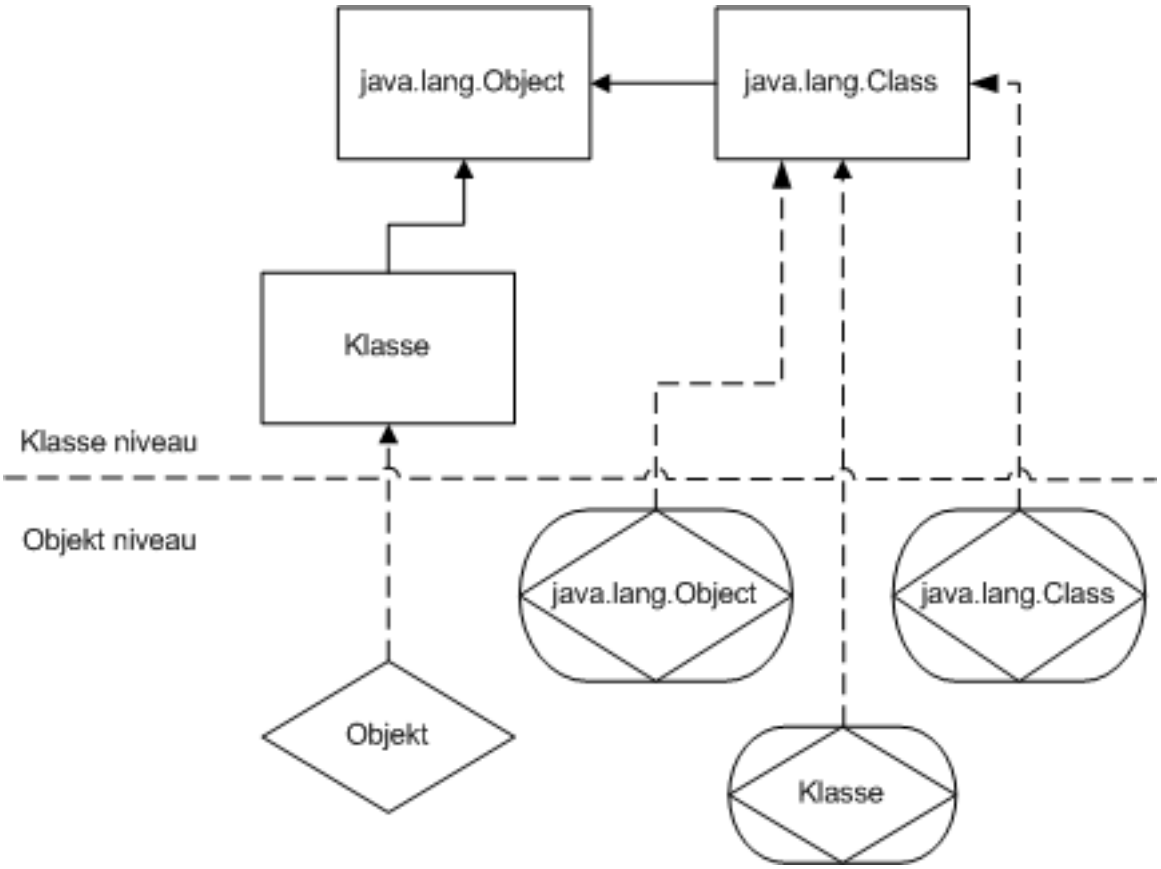
[2] Undersøgelsen er foretaget af gruppen bag rapporten, som manuelt har undersøgt omstændighederne bag de nævnte bankers netbanke per 24/5 2010.

Disse tre situationer viser problemer med hhv. modularitet, fjernadgang og sikkerhed. Disse tre ser ikke umiddelbart ud til at have noget tilfælles, men det har de alligevel: Hver enkelt problem er opstået som en ændring af kravene, der skal tilfredsstilles ved at foretage beslutninger og modificere koden baseret på strukturen af programmet.

At re-implementere interfaces, patche JAR-filer og modificere metodekald er alt sammen kedelige og mekaniske opgaver. Disse opgaver virker umiddelbart som noget de fleste java-programmører kender til, men mange tænker ikke på, at man kan få et program til at udføre det. Derfor bliver det ofte antaget at ovenstående opgaver skal udføres af en person ved et tastatur og ikke af et program, der kører på computer. Vha. reflektion bliver det muligt at komme væk fra de gamle antagelser, og gøre programmet i stand til at foretage de nødvendige tilpasninger.

Udviklerne bag Java implementerede i 1997[9] i *JDK 1.1* (Java Development Kit, version 1.1) reflektiv programmering [10] i API'et `java.lang.reflect`. Dette har også betydet, at Java siden 1997 har været i stand til at udnytte reflektive metoder som beskrevet i afsnit 2.7.

`java.lang.Object` klassen (der nedstammer fra `java.lang` klassen, som alle Java-klasser som nævnt nedarver fra) indeholder de mest generelle byggesten til at konstruere et objekt, og giver alle andre klasser objektmetoder og -variabler. Hele ideen bag reflektiv programmering er at kunne tilgå meta-informationer omkring et objekt og dens klasse. Derfor har man en ekstra klasse i reflektiv programmering kaldet `java.lang.Class`, der skal gør dette muligt. Ved brug af reflektiv programmering vil man ved runtime have metaobjekter (se 4.2.1 for en yderligere forklaring af metaobjekter) af hver eneste klasse, der er bygget ud fra denne specielle reflektive klasse; `java.lang.Class`. Dette vises på figur 4.1, hvor de fuldt optrukne pile viser, hvilke klasse der nedarver metoder, og de stiplede pile viser, hvad objektet eller klassen er en instans af. På tegningen ses et class- og object-level, hvor der henholdsvis er klasser og objekter. Et rektangel på tegningen betyder, at det er en klasse. En diamantformet firkant betyder, at det er et objekt, og en oval cirkel uden om denne firkant betyder, at det er et metaobjekt. Her er et metaobjekt lavet ved runtime af en klasse af samme navn, der gør det muligt at tilgå meta-informationer om klassen.



Figur 4.1. Forklaring af brugen af meta-klasser i Java

Kort fortalt lægger de metoder, der findes i `java.lang.Class`, altså grund for den reflektive programmering, der er brugt i Java.

4.2.1 Principper i ROP

Grundprincippet i reflektiv programmering er altså klogt sammen til, at et program skal kunne tilgå informationer om sig selv og sit softwaremiljø, og derfra kan det foretage valg og ændringer ud fra, hvad det finder. For at et program skal kunne tilgå disse informationer og oplysninger bliver det imidlertid nødt til at have noget, der repræsenterer det. Metadata! Denne metadata, som reflektiv programmering anvender i sine metoder, er organiseret i specielle objekter kaldet metaobjekter.

Når de reflektive metoder bruges ved runtime, så undersøges de føromtalte metaobjekter – dette kaldes også for introspektion. Et spejl eksempelvis virker på samme måde. Spejlet giver informationer om hvilke tøjdele, der passer sammen, hvordan håret sidder osv., og herudfra kan der foretages nogle ændringer. Dette kunne være at rette på håret, taget noget andet tøj på eller lign. Derudover giver spejlet også mulighed for at se nærmere på adfærd; Eksempelvis hvorvidt et smil ser oprigtigt ud, om en bestemt håndbevægelse er for overdrevet, osv. På samme måde kan reflektive programmer også justeres mens de kører gennem undersøgelserne.

Der anvendes i det reflektive *API* (Application Programming Interface) tre teknikker til at ændre programmets opførsel under afvikling:

1. Direkte modificering af metaobjektet.
2. Foretage handlinger ud fra metadata – så som at kalde en dynamisk metode.
3. *Intercession* (tilladelse til at interagere med programmet i forskellige faser).

Et eksempel på nummer 2 er vist i 4.3:

Listing 4.3. Reflektiv programmering eksempel

```
1 public class HelloWorld {
2         public void printName() {
3                 System.out.println(this.getClass().getName());
4         }
5 }
```

Følgende linje i Listing 4.4 vil derefter sende strengen "HelloWorld" til standard output.

Listing 4.4. Reflektiv programmering eksempel fortsat

```
6 (new HelloWorld()).printName();
```

Eksemplet er mere vidtrækkende end det umiddelbart ser ud til. En hvilken som helst klasse, der nedarver fra `HelloWorld`, vil være i stand til at kunne kalde `printName()` og dermed udskrive klassens navn til standard output. Metoden `printName()` undersøger objektets klasse med metoden `this.getClass()` (som er en reflektiv metode). Herefter undersøges, hvad klassen hedder med `getName()` og dette udskrives herefter til standard out. Hvis `printName()` fra HelloWorld ikke bliver overskrevet i en nedarvende klasse vil `printName()` ændre sit output alt efter, hvilken klasse der initialiserer metoden. Altså ændrer metoden `printName()`'s output sig uden at, programmøren skal foretage sig yderligere ændringer. Dette kan vises i et eksempel ved at lave endnu en klasse, der nedarver fra HelloWorld, hvorfra man så kalder `printName`. Dette kan ses i Listing 4.5, der vil skrive "HelloDenmark" til standard out i stedet for fx "HelloWorld", som var tilfældet i Listing 4.3, da `HelloDenmark` klassen extender den oprindelige `HelloWorld` klasse.

Listing 4.5. Reflektiv programmering eksempel

```
7 public class HelloDenmark extends HelloWorld {
8         public static void main(String[] args) {
9                 (new HelloDenmark()).printName();
10        }
11 }
```

Alt dette lægger op til at blive anvendt i software, som er designet til at skulle være meget fleksibelt. Et eksempel på et sådan system ville være et modulsystem, der skal kunne anvende moduler, som programmøren(e) ikke havde forestillet sig ved udviklingen af softwaren. Dette gør det muligt for andre at lave moduler, der – hvis de opfylder visse krav – uden problemer kan implementeres.

Rent praktisk medfører dette, at der skal bruges en anden tankegang end ved traditionel objektorienteret programmering, når der programmeres reflektivt, og der skal bruges nogen andre teknikker end normalt.

De programmeringsmæssige teknikker bygger på princippet at kunne bruge komponenter eller metoder, uden at vide hvad de hedder, hvor de stammer fra, eller hvordan de ser ud. Denne handling kaldes for *introspektion* (evne til at undersøge (-spektion) indad (intro-)).

En af de mest basale måder at udføre introspektion på, er at foretage en forespørgsel efter hvilken klasse et objekt tilhører. Dette kan gøres ved brug af `getClass()`:

Listing 4.6. getClass

```
1 Class cls = obj.getClass();
```

Denne simple introspektion, kan bruges i kombination med `getMethod()`, hvilket er en forespørgsel, der finder de metoder, som passer med parametret – der er sat til `"setColor"` – i følgende eksempel:

Listing 4.7. setColor

```
1 Method method = cls.getMethod( "setColor",new Class[] {Color.class} );
```

Således finder man metoden, der tilhører den fundne klasse (hvis reference hedder `cls`), med metoden ved navn setColor, hvis den altså eksisterer. Dette simple eksempel viser, hvordan reflektiv programmering kan være med til at finde en specifik metode, uden at kende klassen. Det kaldes at programmet *reflekterer over sig selv.*

Dette tillader en enorm fleksibilitet, da man kan få programmet til at foretage handlinger ud fra overvejelser om, hvorvidt programmet skal opføre sig, som programmøren tidligere var nødt til at foretage. Det samme kan opnås ved brug af Java *interfaces* (i sin mest almindelige form er interface en gruppe af relaterede tomme metoder. Læs mere om interfaces på Suns website[3]). Dog er dette ikke altid muligt, da brugen af interfaces kræver at man ændrer lidt i klasserne. Hvis klasserne ikke må ændres, kan interfaces således heller ikke anvendes.

I ovenstående eksempel ville programmøren være nødsaget til at sætte sig ind i den specifikke klasse, hvilket i sig selv ikke er nogen krævende opgaven. At slippe for dette gør det muligt at designe programmer, som selvstændigt kan bruge udvidelser, uden at der skal tilpasses eller rettes yderligere i koden.

Omvendt kræver det, at alle metoders navne er ens, hvis det ønskes at kunne finde dem alle med den fleksible metode beskrevet ovenfor i listing 4.7. At navngive alle metoder, der minder om hinanden, med samme navn, er ikke den største udfordring man som programmør kan komme ud for, hvis det gøres under udviklingen. Hvis det ikke bliver gjort, og man senere i udviklingens forløb opdager, at der er brug for det, kan det være meget tidskrævende og i nogle tilfælde meget udfordrende at ændre koden, så det kan udnyttes til reflektivt brug.

Så de teknikker, der ligger bag at bruge reflektiv programmering, kræver at softwaren opbygges efter visse principper, og at disse følges. Dette gælder bl.a. navngivelsen af metoder, som skal være konsekvent for, at de kan benyttes af den reflektive kode. Opfyldes dette ikke, vil den reflektive kode ikke fungere, og blot melde fejl ved hvert

[3] http://java.sun.com/docs/books/tutorial/java/concepts/interface.html

metodekald. Derfor kræver reflektiv programmering god koordinering og planlægning, og at alle deltagere i et givent projekt opfylder den opstillede arkitektur for programmet.

Disse grundprincipper dækker dog ikke over alle mulighederne ved reflektiv programmering, der indeholder yderligere nogle API'er, som kan anvendes til specialiserede formål.

4.2.2 Reflektion-proxy

Java er som programmeringssprog bygget op af en mængde pakker, som det er muligt at inkludere i et givent projekt med øget funktionalitet til følge. Reflektiv programmering er en sådan pakke, der stiller nogle ekstra funktionaliteter til rådighed. Som forklaret i afsnit 4.2.1 er hovedpakken `java.lang`. Reflektiv programmering er en underpakke til denne, og tilgåes direkte via `java.lang.reflect`. Reflektiv programmering har også nogle underpakker, der indeholder yderligere funktionaliteter. En af disse er proxy-klassen.

En proxy kan betegnes som et element, der agerer som en substitut for et andet element, og udfører samme funktion som det ellers ville. Dette kan bl.a. være en person, der foretager en anden persons arbejde. Dette giver en ide om, hvad begrebet proxy indebærer. Rent teknisk bruges betegnelsen proxy dog oftere om en web proxy, der gør det muligt at surfe på nettet gemt bag en anden IP-adresse. En sådan proxy fungerer essentielt ved, at der sendes en forespørgsel til proxyserveren om at tilgå en hjemmeside. Proxyen behandler så forespørgslen og sender de ønskede informationer tilbage. Informationerne er altså blevet hentet ned af proxyserveren og sendt tilbage til stedet, hvor forespørgslen kom fra. Hvis dette skal sammenlignes med vores oprindelige meget abstrakte eksempel, svarer det til en forespørgelse sendes til en person om at skaffe noget information. Vedkommende beder så en anden om at skaffe den information, og når den er blevet leveret, giver han den til personen, der oprindeligt gav forespørgslen.

En proxy laves indenfor objektorienteret programmering ved brug af klassen `java.lang.reflect.Proxy`, der gør det muligt at anvende en proxy's funktionalitet i et stykke software. En proxy fungerer i dette tilfælde ved, at den går ind og undersøger en given klasses interface, og danner en kopi af dette, så kopien herved har de samme funktionaliteter, som klassens ellers ville have.

Fordelen ved dette er at skabe kontrol over brugen af klasserne, som der er skabt en proxy af. Denne kontrol tillader os at afvikle et specifikt stykke kode, som fx kan være sporing/tracing af hvornår metoderne bliver kaldt, i hvilken rækkefølge m.m. På den måde har en ét stykke kode, som bliver genbrugt igen og igen.

Overordnet set betyder dette, at ved brug af den reflektive-proxy funktionalitet, er det muligt at spare en mængde kode, når klasser programmeres.

Implementation af proxy

Det vigtigste element i en proxy er dens evne til at implementere et korrekt interface af den klasse, den er en proxy af. Dvs. den har de korrekte metoder integreret i sig, hvilket gør den i stand til at udføre de rigtige handlinger på objektet. Den skal også have integreret

en funktion, der gør den i stand til at sende de metodekald den modtager videre mod det objekt, de er rettet mod. Denne handling foretages i relation til proxy ved brug af en `InvocationHandler`. For i reflektiv programmering foretages metodekald som nævnt ved at `invoke` dem på objektet. En konstruering og initialisering af en proxy med interface kan se ud som vist i listing 4.8.

Listing 4.8. Overordnet proxy implementation

```
1  public class Proxy implements java.io.Serializable {
2          public static Class getProxyClass( ClassLoader loader,
3          Class[] interfaces )
4          throws IllegalArgumentException ...
5
6          public static Object newProxyInstance( ClassLoader loader,
7          Class[] interfaces,
8          InvocationHandler h )
9          throws IllegalArgumentException ...
10
11         public static boolean isProxyClass( Class cl ) ...
12
13         public static InvocationHandler getInvocationHandler( Object proxy )
14
15         throws IllegalArgumentException ...
16 }
```

Dette er meget generel kode, men den illustrerer hvordan en proxy implementeres, hvor proxy-klassen først dannes ud fra en *classloader* (en funktion, der henter eller genererer data, der udgør en definition for klassen, dvs. hvilke funktioner og metoder den indeholder), og klassens interface, hvilket sikrer at proxy-klassen har alle de nødvendige metoder og attributter, som proxy-klassen skal emulere.

Herefter dannes et nyt objekt, der er en instans af proxyen, og en `InvocationHandler`, der kan håndtere objekter af den type proxyen er af.

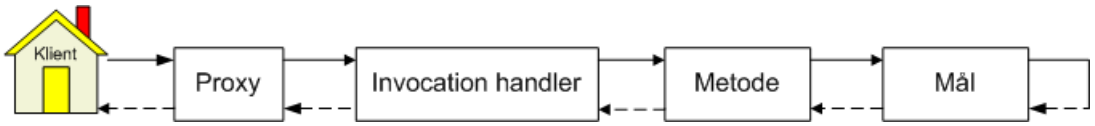
En `InvocationHandler` er opbygget ganske simpelt:

Listing 4.9. Invocation handler interface

```
1  public interface InvocationHandler {
2
3          public Object invoke( Object proxy, Method method, Object[] args )
4                  throws Throwable;
5  }
```

Som det ses af `InvocationHandler`'ens interface, `invoker` den en metode på et objekt, ved at finde denne metode i proxyen.

Som det fremkommer af de forskellige trin i proxy-implementationen, er det dog også et større stykke arbejde, der foretages af softwaren inden metodekaldet foretages. For processen får et metodekald ved brug af en proxy er lang i forhold til, hvis den blot kaldes direkte på objektet:



Figur 4.2. Processen for et metodekald foretaget ved brug af proxy

Den konkrete implementation af en proxy medfører altså generelt langsommere kode, da et metodekald skal gennem en del flere trin, før det rent faktisk udføres.

Sporing med proxy

Måden en proxy fungerer på i Java lægger helt naturligt op til nogle områder, hvor de vil kunne gøre gavn, udover at et givent klassebibliotek vil kunne vedligeholdes nemmere.

Da alle kald til klasserne går igennem proxy'en, er det en naturlig følge, at proxy'en også kan bruges til at gemme brugerens handlinger. At have sådan en funktionalitet i en applikation er en stor fordel af en simpel grund.

Hvis et program uventet går ned, vil udviklerne af softwaren ofte gerne vide, hvorfor dette skete, så fejlen kan rettes. For at undersøge hvorfor en fejl fremkommer, er det nødvendigt at genskabe den, og observere hvordan programmet opførte sig da fejlen skete. I alle tilfælde vil brugeren, der har frembragt fejlen dog sandsynligvis ikke være i stand til viderekommunikere denne information til udviklerne af programmet. Både pga. hukommelse, altså at brugeren simpelthen ikke kan huske, hvad vedkommende præcist gjorde, men brugeren kan også have tilpasset sin udgave af programmet til eget behov i modsætning til standardopsætningen, hvilket kan have medvirket til fejlen.

At have en sporingsfunktion som denne uden brug af reflektiv programmering, vil kræve en stor mængde ekstra arbejde under udarbejdelsen af klasserne, da funktionaliteten skal kodes ind i klassens metoder. Derudover vil der blive sendt en stor mængde data frem og tilbage under afviklingen af programmet. For ved denne opbygning vil objektet skulle sende en ekstra mængde informationer ud, hver gang der foretages et metodekald på det. Det, at der skal sendes informationer ud vedrørende hvert metodekald, vil gøre koden langsommere, end den ellers ville være pga. den ekstra mængde data, der skal sendes med hver gang. Da alle metoder går gennem `InvocationHandler`'en, er det muligt med en simpel modifikation af `InvocationHandler`'en muligt at logge alle metodekaldene der foretages, så de trin, der har forsaget fejlen, nemt kan tilgåes. En `InvocationHandler` med sporing vil kunne se ud på følgende måde:

Listing 4.10. Sporing med proxy

```
1
2 import java.lang.reflect.*;
3 import java.io.PrintWriter;
4
5 public class TracingIH implements InvocationHandler {
6
7         public static Object createProxy( Object obj, PrintWriter out ) {
```

```
8                   return Proxy.newProxyInstance( obj.getClass().getClassLoader
                        (),
9                   obj.getClass().getInterfaces(),
10                  new TracingIH( obj, out ) );
11          }
12
13          private Object target;
14          private PrintWriter out;
15
16          private TracingIH( Object obj, PrintWriter out ) {
17                  target = obj;
18                  this.out = out;
19          }
20
21          public Object invoke( Object proxy, Method method, Object[] args )
22          throws Throwable
23          {
24                  Object result = null;
25
26                  try {
27                  out.println( method.getName() + "(...) called" );
28                  result = method.invoke( target, args );
29          }
30
31          catch (InvocationTargetException e) {
32          out.println( method.getName() + " throws " + e.getCause() );
33          throw e.getCause();
34          }
35
36          out.println( method.getName() + " returns" );
37          return result;
38
39          }
40  }
```

I dette tilfælde gøres dette forholdsvis simpelt ved at navnet på metoden, som naturligt bruges i `InvocationHandler`'en, blot skrives ud. Ved brug af denne simple modifikation er det muligt at logge en brugers vej gennem en applikation, og herved nemmere opdage fejl.

Vurdering af proxy

På trods af at proxy stiller en stor mængde funktionalitet til rådighed, er der også ulemper ved at benytte det. En af de overordnede ulemper ved reflektiv programmering er, at koden generelt er langsommere end ikke-reflektiv kode. Dette er i endnu højere grad tilfældet, hvis proxy-funktionaliteten benyttes. For her skal hvert metodekald sendes gennem 4-5 punkter før det udføres, og ved dannelsen af en proxy-klasse kræver det også en stor mængde arbejde af koden.

Derfor skal det inden en proxy implementeres i et stykke software vurderes, om koden vil kunne leve op til de krav kunden har opstillet med hensyn til afvikling af koden. Derfor stiller `java.lang.reflect.Proxy` godt nok stor funktionalitet til rådighed, men

det kommer med en pris, og vil i hvert tilfælde kræve en vurdering af, om det vil kunne betale sig at implementere.

4.2.3 Kodegenerering

Et andet område, som udvider det reflektive API i Java, er kodegenerering. Kodegenerering er en løsning til at forbigå de begrænsninger, som reflektion i Java opstiller ved brug af introspektion. Grundlæggende er reflektion i Java begrænset til instrospektion – beskrevet i sektion 4.2.1 – og selvom man benytter sig af proxy – beskrevet i sektion 4.2.2 – er man stadig begrænset til basal introspektion. For at bryde denne begrænsning kræves det, at programmet har evnen til at ændre sin egen adfærd. At et program skal kunne ændre sin egen adfærd, lyder umiddelbart som om, at programmet skal programmere sig selv om, og det er dette som kan muliggøres med reflektiv programmering.

Det kræver at programmet kan følgende:

- Generere kode.
- Kompilere kode.
- Eksekvere kode.

I listing 4.11 ses et meget basalt eksempel på dette. Kodegenereringen består her af at udskrive en hardcoded tekst streng, selvom denne kunne være skabt ud fra flere introspektive operationer, som fx indlæsning af en eksisterende klasse, hvorpå der tilføjes kode og/eller ændres.

Listing 4.11. HelloWorldGenerator.java

```
1  import java.io.FileOutputStream;
2  import java.io.PrintWriter;
3  import java.lang.reflect.Method;
4  import java.io.InputStream;
5
6  public class HelloWorldGenerator {
7          public static void main( String[] args ) {
8                  try {
9
10                         FileOutputStream fstream = new FileOutputStream( "
                               HelloWorld.java" );
11                         PrintWriter out = new PrintWriter( fstream );
12                         out.println("class␣HelloWorld␣{␣\n"
13                                         + "public␣static␣void␣main(␣String[]
                                               ␣args␣)␣{␣\n"
14                                         + "System.out.println(␣\"Hello␣world
                                               !\"␣);\n}}");
15                         out.flush();
16
17                         Process p = Runtime.getRuntime().exec( "javac␣
                               HelloWorld.java" );
18                         p.waitFor();
19
20                         if ( p.exitValue() == 0 ) {
```

```
21                                  Class outputClassObject = Class.forName( "
                                        HelloWorld" );
22                                  Class[] fpl = { String[].class };
23                                  Method m = outputClassObject.getMethod( "
                                        main", fpl );
24                                  m.invoke( null, new Object[]{ new String[]
                                        {} } );
25                          } else {
26                                  InputStream errStream = p.getErrorStream();
27                                  for ( int j = errStream.available(); j > 0;
                                        j-- )
28                                      System.out.write( errStream.read() )
                                            ;
29                          }
30                  } catch(Exception e){
31                          throw new RuntimeException(e);
32                  }
33          }
34  }
```

Linje 10-15: Filen HelloWorld.java oprettes, og koden skrives til filen.

Linje 17-18: Filen HelloWorld.java kompileres ved at kalde `javac`, hvorefter der ventes indtil kompileringen er færdig.

Linje 20-29: Her invoke's main-metoden i den kompilerede HelloWorld-klasse, og på den måde eksekveres.

De resterende linjer kode er til at fremvise eventuelle fejl der måtte fremkomme.

Eksemplet i listing 4.11 er taget ud fra kilde nr. [8], og viser den helt grundlæggende evne, der tillader at skabe noget ny kode og afvikle det. Ud fra ovenstående eksempel kan man udvikle et værktøj, som tager en klasse som input-argument, og ved brug af introspektion, undersøger om der skal foretages ændringer inden klassen igen gives tilbage som output. Fordelen ved et værktøj som dette er, at det giver mulighed for at ændre i dele af koden under runtime.

Et eksempel på dette kunne være situationer, hvor et produkts ydeevne ubetinget bliver påvirket negativt, hvis en speciel funktionalitet bliver implementeret. Hvis det er langt fra alle købere af produktet, der får brug for den specifikke funktionalitet, er der ingen grund til at implementere den for dem alle. Her kan man udnytte introspektion til at finde ud af, hvornår funktionaliteten er nødvendig, hvorefter kodegenerering kan benyttes til at tilføje den specielle funktionalitet under runtime.

Fordelen ved kodegenerering ligger altså i, at hvis koden programmeres smart, vil den være i stand til at tilpasse sig brugerens behov, ved at bruge kodegenerering. Denne problemstilling ville kunne løses ved at give brugerne lov til at kompilere programmet, men dette er ikke altid muligt for et firma, som udvikler closed source applikationer.

Der er dog også en ulempe ved at kodegenerering – i større skala vil det være langt mere omfattende end det givne eksempel, og en stor del af denne kode vil muligvis ikke blive benyttet af kunden. Derfor vil det skulle vurderes nøje om et givent stykke software har

stort markedspotentiale før kodegenerering implementeres, da der ellers vil blive brugt mange udviklingstimer på at udvikle funktionaliteter, der måske ikke er nødvendig.

Alt i alt skal kodegenerering benyttes med omhu, da det ikke altid vil kunne betale sig at integrere, men det stiller imidlertid en stor mængde fleksibilitet til rådighed, der vil kunne være gavnlig i meget store projekter, som kræver en høj grad af fleksibilitet og tilpasningsevne.

4.2.4 Reflektiv programarkitektur

Reflektiv programmering kan altså implementeres på mange forskellige niveauer og i meget forskellig omfang fra projekt til projekt. Der er derfor ingen "opskrift" for, hvordan en korrekt implementation af *ROP* (Reflektiv Orienteret Programmering) ser ud.

Eksempelvis kan ROP implementeres som "kernen" af et program, hvor man vha. ROP inkluderer de moduler, som er aktuelle for den øjeblikkelige handling i programmet. Alternativt kan en implementation være hvor, at ROP anvendes til styringen af et sikkerhedsmodul, dvs. brug af ROP løbende under programafvikling – en væsentlig mindre implementation.

Et program som anvender reflekiv arkitektur og afviklingsrutiner adskiller sig fra programmer med ikke reflektiv arkitekturer på det overfladiske niveau på en del punkter, grundet de mange muligheder som reflektion åbner op for. Naturligvis er afviklingsrutinerne de samme i JVM'en hver gang, men på et overfladisk niveau hjælper det til forståelsen af principperne og arkitekturen, at forestille sig en differens. Du ved allerede, hvordan imperativ kode afvikles, hvor koden læses fra top til bund uden forbehold. Anvender du objektivt orienteret programmering kompliceres processen en smule, da disse klasser "indlæses" før den resterende kode. Eksempelvis kan man placere sine klasser i bunden af kildekoden, men stadig anvende dem i toppen. Dvs. klasser og metoder indlæses først. Kobles endnu et lag på – reflektiv programmering – tilgåes det føromtalte meta-data i det de reflektive objekter initialisers, hvor programmet vha. introspektion "scanner" sig selv, for efterfølgende at kunne reflektere over det i den pågældende reflektive funktion.

Sidst i kapitel 5 vil du se, hvordan disse teorier om OOP og ROP bliver brugt i praksis, og hvorfor netop ROP er rigtig smart til en lang række formål.

Reflektion i aktion 5

Reflektiv programmering stiller en stor mængde funktionalitet til rådighed for programmøren. Både med hensyn til fleksibilitet, men også i forhold til mængden af arbejde, der kræves. Dette er nogen af de aspekter, der gør reflektiv programmering tiltalende både at undersøge, men også at anvende og arbejde med. For at få en bedre forståelse for reflektiv programmering og de konkrete fordele reflektiv programmering kan give en virksomhed, og hvordan det overordnet set vil påvirke arbejdsgangen i udviklingsprojekt, er et kodeeksempel blevet udviklet, der forsøger at illustrere nogen af fordelene ved at benytte reflektiv programmering. Udover dette analyseres der på nogen af ulemperne ved at benytte reflektiv programmering i det givne scenarie.

5.1 Hvorfor ROP?

Inden selve kodeeksemplet præsenteres, er det relevant at analysere de konkrete fordele reflektiv programmering kan give for en softwarevirksomhed.

At anvende reflektiv programmering konsekvent giver flere fordele. Primært ved den øgede funktionalitet, der bliver stillet til rådighed for software skrevet efter reflektive principper, som der allerede er redegjrot for. Der er her nogen klare tekniske fordele i det, at reflektiv kode: Når et hvis punkt nås kræves en del mindre kodearbejde end ikke-reflektiv kode 5.2.3. Disse tekniske fordele giver større funktionalitet i projektet, og gør selve arbejdsprocessen mere fleksibel, hvilket både giver økonomiske fordele, men som også potentielt kan gøre det muligt for virksomheden at tilbyde større fleksibilitet til dens kunder.

5.1.1 Tekniske aspekter

Reflektiv programmering har en vis indflydelse på planlægningsfasen i et softwareprojekt, da det er vigtigt – for strukturen, designets og overblikkets skyld – at alle klassernes opbygning er på plads, når der programmeres reflektivt. Den primære forskel ligger naturligvis i implementeringsfasen.

Implementeringsfasen i softwareudvikling varierer alt efter hvilken generel softwareudviklingsmetode der anvendes. I de fleste tilfælde er der mere end én programmør tilknyttet

til et projekt, hvilket medfører, at softwarens funktioner programmeres hver for sig, som herefter sammensættes til et helt stykke software ved afslutningen af projektet.

Hvis softwaren er programmeret efter reflektive principper, er dette en relativ nem proces, og det vil ikke kræve den store mængde kode at få de forskellige dele af softwaren til at fungere sammen (se 5.2.3). En stor del af implementationsfasen ligger netop i, at få alle funktionerne til at virke på de rigtige objekter, og sikre den korrekte handling udføres på objekterne, alt efter hvilken type de er.

Et eksempel hvor reflektiv programmering kan være attraktiv er i spiludvikling: Bl.a. i et *FPS*-spil (first person shooter), hvor spilleren bl.a. kan vælge hvilket *skin* (spillerens karakters udseende) der skal anvendes. Alle disse forskellige skins vil nødvendigvis skulle være pre-programmeret ind i programmet, og hentes ind alt efter spillerens valg. Udover at spillerens skin i koden sættes til det valgte, skal det naturligvis også frembringes grafisk til spilleren. Hvis dette gøres med reflektiv kode, i stil med det simple `setColor` eksempel fra afsnit 4.2.1, vil denne del af implementationen foregå relativt simpelt. Ved brug af denne metode, vil det ikke være nødvendigt for softwaren at kende til brugerens valg. At koden er skrevet reflektivt, og herved selv er i stand til at finde ud af, hvilket skin brugeren har valgt.

Et sådan eksempel kan umiddelbart virke som et problem, hvor det ikke er 100% essentielt at bruge reflektiv programmering. Men hvis eksempelt undersøges mere dybdegående, er det dog rimelig tydeligt at reflektiv programmering stiller andre muligheder til rådighed, der gør det praktisk at anvende i et eksempel som dette. Hvert skin, som spilleren har mulighed for at vælge, vil nødvendigvis skulle være kodet ind i kildekoden, og det skal være muligt for resten af koden, at benytte de forskellige skins.

For at koden kan benytte disse skins, skal den vide at de eksisterer. Den simpleste måde at gøre dette, er ved at hardcode direkte ind i spillet, hvilke skins der er til rådighed, og noget ekstra kode udover dette, der sørger for at frembringe skinnet grafisk, alt efter hvilket valg brugeren foretager. Hvis der er en lille mængde skins er det ikke det store problem, da det ikke kræver den store mængde kode. Overordnet set vil det kræve en `switch`, eller en stor mængde `if-sætninger` at implementere de forskellige skins, og gøre det muligt for resten af koden af benytte disse skins. At sådan en implementation er nødvendig, medfører dog, at jo flere skins der er, dets større et arbejde er det at implementere det i resten af koden. Hvert skin kræver nemlig den samme mængde kode at få implementere. Den mængde arbejde, som det kræver med hensyn til implementeringen, vil herved stige eksponentielt i forhold til hvor mange skins, der ligger i spillet.

Det er på dette punkt reflektiv programmering kommer ind i billedet og præsenterer en alternativ løsning. Reflektiv kode reflekterer nemlig over sig selv og sine omgivelser. Dette kan bruges til andre opgaver end at finde ud af, hvilken klasse et givent objekt er en instans af. Reflektiv kode er netop i stand til at observere sine omgivelser, hvilket medfører at den tillige er i stand til at observere hvilke klasser, der ligger til rådighed i programmet. Når den reflektive kode er skrevet på denne måde, betegnes klasserne, der indeholder de forskellige skins, som moduler, som det er muligt for koden at hente ind og anvende.

Dette kan nemt bruges i sammenhæng med det givne eksempel, og sikre en del mindre kodearbejde for udviklerne. Med den modularitet reflektiv programmering stiller til

rådighed, er det ikke nødvendigt at hardcode valgmulighederne mellem de forskellige skins ind i programmet, da den reflektive kode selv er i stand til dette. Udover dette er den reflektive kode i stand til selv at opfatte hvilket skin brugeren har valgt, og sørge for at det vises grafisk til brugeren

Alt i alt er reflektiv programmering i dette tilfælde en løsning, der sparer programmøren for en stor mængde arbejde.

Det er dog ikke kun i udviklingsfasen at reflektiv programmering præsenterer sig som en holdbar løsning. Som nævnt er den reflektive kode i stand til selv at opfatte hvilke skins, der ligger i dens modul-pakke, og gøre dem brugbare i forhold til resten af koden. Dette medfører, at hvis der ved en udvidelse tilføjes en ekstra mængde skins, vil kildekoden til selve spillet ikke skulle modificeres. Det nye skin lægges blot ind i modulpakken, og hvis det er skrevet efter reflektive principper, vil det fungere sammen med resten af koden uden problemer.

Reflektiv programmering stiller altså en stor mængde tekniske fordele til rådighed for programmøren, der kan spare vedkommende for en stor mængde arbejde. Dette gælder ikke kun modulariteten, men både proxy- og kodegenereringselementerne af reflektiv programmering kan have denne effekt.

5.1.2 Økonomisk aspekt

De tekniske fordele bringer naturligt andre fordele med sig, når de udnyttes i en virksomhed. Der kan potentielt være en økonomisk gevinst for en virksomhed, hvis de benyttes korrekt. Når en virksomhed planlægger et IT-projekt, laves en estimering over hvor lang tid et givent projekt vil tage, og ud fra dette laves et budget så et rimeligt tilbud kan sendes til kunden. Sådan et tilbud vil dog ofte være en estimering af hvor lang tid virksomheden har brug for, før de kan levere det færdige stykke software, og hvor mange penge det vil kræve for, at det for virksomheden kan betale sig økonomisk. Der kan dog nemt opstå komplikationer i udviklingen, hvilket gør at det tager længere tid end først antaget. Dette vil naturligvis medføre øgede udgifter for softwarevirksomheden, og når et tilbud er givet til en virksomhed, er det ikke sikkert, denne vil acceptere at prisen på deres produkt stiger. Hvis produktet udvikles efter de agile metoder, hvor der arbejdes i iterationer og jævnligt holdes møder med kunden, vil et kompromis, muligvis kunne findes under en af disse, inden problemet og de større omkostninger bliver for store.

Det er dog ikke en optimal tilgang, da sådan et kompromis ofte vil medføre et dårligere slutprodukt eller som nævnt en øget pris for kunden. Og hvis softwaren er udviklet efter vandfaldsmodellen, vil sådan et problem sandsynligvis først blive opdaget og diskuteret med kunden langt inde i forløbet, da alt designarbejdet her færdiggøres inden implementationen påbegyndes.

Dette er et af punkterne, hvor objektorienteret programmering vil gøre gavn og give anledning til at overbygge det med reflektiv programmering. Objektorienteret kode er meget fleksibelt, og benytter sig af genbrug på en måde, der gør det nemt at rette fejl. Hvis softwaren, som firmaet udvikler, bl.a. benytter sig af en proxy, vil alle deres metoder

være gemt ned i individuelle klasser og herfra bruges – dette er de også hvis der blot anvendes OOP, men overbygges dette med en proxy anvendes ROP. Dette betyder at hvis en fejl opdages i en metode, der benyttes flere steder, vil fejlen kun skulle rettes dette ene sted, hvilket ikke burde medføre betydende budget overskridelser. Hvis årsagen til fejlen ikke kendes, stiller proxy funktionaliteten tillige muligheden for at lave et sporingssystem, der kan hjælpe med at give en bedre forståelse for, hvorfor softwaren ikke fungerer.

Hvis sådan en fejl kan opdages og rettes uden, at det medfører store budgetoverskridelser, vil det være til stor gavn for softwarevirksomheden, da det er muligt for den at levere softwaren til nogenlunde den aftalte pris på trods af den opståede fejl. Grundet måden hvorpå reflektiv programmering er opbygget og fungerer, kan det altså hjælpe på rettelse af fejl i koden, og forhåbenligt nedsætte de tidsmæssige konsekvenser en alvorlig fejl har kan have for et softwareprojekt.

Reflektiv programmering har en af sine største styrker i kodens fleksibilitet. Hvis kunden midt i forløbet ønsker at få tilføjet ekstra funktioner i deres program, vil sådan en relativt nemt kunne tilføjes, hvis programmet er opbygget med en reflektiv arkitektur. Som det er forklaret i afsnit 4.2.1, er et stykke software opbygget efter de reflektive principper nemt at tilføje nye klasser med nye funktioner til. Hvis funktioner med samme egenskab blot navngives ens, kan en ny klasse nemt integreres i eksisterende stykke software, uden at det kræver en større mængde arbejde.

At softwarevirksomheden ved brug af de agile- og reflektive metoder kan give kunden en stor mængde fleksibilitet i forhold til kundens ønsker og behov, og er i stand til at efterkomme disse ønsker, uden at kunden må betale et betydeligt større beløb end aftalt, må kunne anses som værende en væsentlig fordel. At kunne tilbyde sådan en fleksibilitet gør softwarevirksomheden attraktiv, da det er mere tiltrækkende for en kunde at kunne komme med inputs undervejs i udviklingsforløbet i modsætning til ved projektets afslutning.

Alt i alt giver reflektiv programmering en stor række muligheder til en softwarevirksomhed, og overordnet set giver disse softwarevirksomheden en stor mængde fleksibilitet og mulige besparelser i softwareudviklingsprocessen. Dette gør virksomheden attraktiv for mulige kunder, og kan potentielt hjælpe med at bringe flere ordrer og herved større indtjening til virksomheden, i modsætning til virksomheder, der følger mere traditionelle udviklingsmetoder.

5.1.3 Opsamling

Det, der gør brugen af reflektiv programmering attraktiv i forhold til softwareudvikling, er at det grundet dets principper i en lang række tilfælde vil spare programmørerne for en masse arbejde, der ellers ville have optaget både tid og ressourcer. Udover dette er reflektiv kode (og iøvrigt også objektorienteret kode i al almindelighed) relativt nemt at modificere, så hvis en fejl opdages i koden, skal den blot rettes et sted, i modsætning til, hvis den var skrevet efter andre principper, såsom fx imperiativ programmering. Tillige er det pga. af reflektiv kodes modularitet nemt at tilføje ekstra funktioner og moduler til et program, uden at det kræver en større mængde arbejde udover selve kodningen af modulet.

For at belyse dette er følgende kodeeksempel MRNS blevet udviklet.

5.2 Anvendelse - Modular Reflective Number Sorting (MRNS)

Følgende kildekode, der hører til MRNS-programmet, er blevet udviklet med henblik på at vise brugen af reflektiv programmering i en funktionel demonstration. Kildekoden er udviklet ud fra de teorier, som er gennemgået i tidligere sektioner. Hensigten med programmet er at eftervise, at teorierne kan bruges i praksis, og udvikle et system, der er i stand til at tage nye moduler ind uden yderligere konfiguration af hovedprogrammet.

Som navnet *MRNS* (Modular Reflective Number Sorting) hentyder til, så stiller programmet nogle talsorteringsmetoder til rådighed. Dette gøres ved reflektivt at finde modulerne og efterfølgende behandle dem ved runtime. Derudover vises en grundlæggende ide for, hvordan et login-system kan implementeres sammen med, så der kan begrænses, hvor mange metoder brugeren har ret til at anvende. Den loginbaserede del af programmet er implementeret i så lille skala som muligt, som det har været nødvendigt, da dette kan gøres på mange måder, og ikke direkte efterviser nogle principper af ROP.

Programmet fungerer helt basalt på følgende måde:

- Du indtaster login-id.
- Du indtaster en række af numre, som du ønsker sorteret, og slutter med 0 for at afslutte indtastningen.
- Du får præsenteret en liste af metoder, hvoraf der er markeret med (x), hvilke du kan bruge.
- Du vælger en metode, hvorefter programmet bruger metoden til at sortere dine numre.
- Numrene printes ud i den sorterede orden.

Da metoderne er implementeret vha. reflektion er det muligt, ved blot at overholde nogle simple regler, at implementere flere metoder løbende, uden der er brug for at ændre i selve MRNS-kildekoden.

Kravene til de sorteringsmetoder, der skal implementeres senere er:

1. Skal have funktionen: `public ArrayList<Double> start(ArrayList<Double>)`
2. Skal have funktionen: `public static ArrayList<Integer> getAccess()`

Disse krav skal være opfyldt for at modulerne kan implementeres i programmet fuldstændig flydende. Krav 1 er grundlaget for, at programmet kan køre modulerne uden først at have forudgående kendskab til dem. Krav 2 er til vores login-system, så programmet ved, hvem der har rettigheder til at bruge det givne modul.

5.2.1 Analyse af MRNS-kildekoden

I det følgende vises to kodeeksempler, som begge er af afgørende betydning for programmets reflektive funktionalitet. Først listes eksemplerne i deres fulde længde, hvorefter de nedbrydes og linjernes funktion og betydning for helheden analyseres og retfærdiggøres. Det første eksempel – Listing 5.1 – vises delen, hvor omgivelserne ses igennem efter forskellige sorteringsmetoder. Derefter følger Listing 5.2, hvor den valge sorteringsmetode initialisers.

Følgende er naturligvis kun udpluk af kildekoden, som kan ses i sin fulde længde på vedhæftede CD.

Listing 5.1. MRNS base - inkludering af moduler fra ekstern package

```
1  try {    //Get all modules
2           List<Class> classes = getClassesForPackage("modules");
3           ArrayList<Integer> usersWithAccess = new ArrayList<Integer>();
4           ArrayList<Integer> allowedMethods = new ArrayList<Integer>();
5
6           for(int i = 0; i < classes.size(); i++) {
7                   String name = "" + classes.get(i) + "";
8                   name = name.substring(14, name.length());
9
10                  Method method = classes.get(i).getMethod("getAccess", new
                        Class[] {});
11                  usersWithAccess = (ArrayList<Integer>) method.invoke(null);
                            //Invoke access method
12
13                  System.out.print(i+1 + ": " + name);
14
15                  if(usersWithAccess.indexOf(self.user) != -1) {  //Check
                        access with current login
16                          System.out.print(" (x)");
17                          allowedMethods.add(i);
18                  }
19
20                  System.out.println("");
21          }
22
23          do {
24                  System.out.print("\nPlease enter the ID of your choosen
                        method which you have access to: ");
25                  self.clsId = TextIO.getlnInt();
26          }
27          while(allowedMethods.indexOf(self.clsId-1) == -1);        //Check
                chosen method with allowedAccess
28
29          self.cls = classes.get(self.clsId-1);
30
31  } catch (Exception e) {
32          System.out.println(e);
33  }
```

Linje 1: Først og fremmest sættes det hele ind i en `try/catch`, såfremt at hvis der kommer en fejl, fanges denne i catch delen.

Linje 2: Her gemmes alle de klasser (sorteringsmetoder), som er tilgængelige i "modules"-package'en. Metoden `getClassesForPackage()` lister simpelthen alle klasser i en valgt package og sætter dem – i dette tilfælde – ind i en `List<Class>`. Disse gemmes i første omgang blot til brug senere i programmet.

Linje 3-4: Her defineres to `ArrayList<Integer>`. Disse anvendes i sikkerhedsmæssigt henseende. Den første på linje 3, `usersWithAccess`, gemmer alle de brugere med adgang til hver klasse. Den næste på linje 4, `allowedMethods` bruges til at sammenligne `usersWithAccess` og det ID, brugeren er logget ind med, for at gemme en liste over de klasser, som den givne bruger har adgang til.

Linje 6: En for-løkke initialiseres, som løber igennem for hver klasse, der blev fundet i "modules"-package'en.

Linje 7-8: Navnene på de enkelte klasser laves en smule om, så det er mere læseligt for brugeren.

Linje 10-11: Sikkerhed igen. Den statiske metode `getAccess` kaldes, og den returnerer et ArrayList med alle de brugere, som har adgang til den pågældende klasse. Denne metode invokes på linje 11, hvor resultatet gemmes i `usersWithAccess`-listen.

Linje 15-18: Nu tjekkes der, om brugeren som er logget ind, har adgang til den pågældende klasse/sorteringsmetode. Hvis ja, printes "(x)" på skærmen ud for den pågældende klasse (for at vise, at brugeren har adgang til netop denne klasse), samtidig med at den bliver tilføjet til `allowedMethods`-listen.

Linje 23-27: Brugeren er kommet ud af for-løkken, og alle classes er nu listet. Brugeren skal nu vælge, hvilken klasse, der skal anvendes, via et `TextIO input`. Dette er sat ind i en do/while-løkke, således at brugeren bliver promptet for en klasse indtil brugeren vælger en, som vedkommende jf. `allowedMethods`-listen har rettigheder til at anvende.

Linje 29: Sidst – når brugeren har valgt en gyldig klasse – gemmes den pågældende klasses objekt i den globale variabel `self.cls`. `self` er en instans af MRNS-klassen selv, hvorigennem de globale variabler i klassen kan tilgåes. `self.cls` skal anvendes senere, når programmet skal bruge den klasse, som brugeren har valgt.

Listing 5.2. MRNS base - initialisering af valgt klasse og start() metoden

```
1  try {    //Initialize the start method to start sorting
2           Method method = self.cls.getMethod("start", new Class[] {ArrayList.
                class});
3
4           Object thisInstance = self.cls.newInstance();
5           self.input = (ArrayList<Double>) method.invoke(thisInstance, self.
                input);
6
7  } catch(Exception e) {
8           System.out.println(e);
9  }
10
11 System.out.println("\n\nIn sorted order (lowest first), your numbers are:");
           //Print the output
12 for(int i=0; i<self.input.size(); i++) {          //Print this ArrayList
13          System.out.println(self.input.get(i));
```

```
14 }
```

Linje 1: Da koden for initialiseringen kan gå galt bør man foretage en `try-catch`, så evt. fejl fanges og gøres noget ved. I Listing 5.2 foretages der dog ikke yderligere end blot at skrive fejlen til `standard-out`.

Linje 2: Krav nummer 1 listet tidligere i denne sektion er grundlag for, at programmet kan tage nye moduler ind reflektivt. På linje 2 instantieres en metode, der bliver sat til den metode, som self.cls har, der hedder `start`. Dette gøres vha. Javas indbyggede reflektive metode `getMethod()`. Javas indbyggede metode tager to argumenter, det første er navnet på metoden den skal lede efter, det andet argument er de parametre som metoden skal tage. I dette tilfælde anvendes: `getMethod("start", new Class[] {ArrayList.class})`, hvilket vil sige, der ledes efter en metode kaldet `start`, der tager parametre af typen `ArrayList`. Når metoden er fundet i `self.cls` gemmes den i variablen `metode`.

Linje 4: Constructoren til `self.cls` bliver her kaldt vha. den reflektive metode `newInstance()`. Denne metode fungerer som fx `new Object()` ville, den tager blot `self.cls` som klassen den skal instantiere. Derfor instantieres klassen `self.cls` i objektet `thisInstance`.

Linje 5: Herefter bruges den reflektive metode `invoke` til at køre metoden fundet i linje 1. `invoke` tager to argumenter: objektet den skal køre metoden på (ved statiske metoder bruges `null` som argument for at undgå exceptions), og et array med de parametre, som metoden tager. Her skal metoden køres på `thisInstance`, derfor gives dette som første argument. Derudover så findes brugerens array af numre, der skal sorteres, i `self.input`, derfor sendes dette videre til metoden som parameter. Alt dette kastes til typen `ArrayList<Double>` og gemmes i variablen `self.input`.

Linje 11-14: Her skrives den sorterede liste ud til brugeren.

De to stykker kildekode, der er vist er selvfølgelig ikke hele programmet, men blot de vigtigste reflektive dele som har interesse lige nu.

5.2.2 Sammenligning med ikke-reflektiv kildekode

Kildekoden vist i forrige afsnit kan naturligvis også skrives på "statisk" manér uden anvendelsen af reflektiv programmering. I det følgende vises en listing, hvis kildekode udfører det samme som Listing 5.1 og 5.2, men ikke anvender reflektiv orienteret programmering.

Listing 5.3. MRNS base static - hele kildekoden i statisk version

```
1 ArrayList<Integer> allowedMethods = new ArrayList<Integer>();
2
3 System.out.print("1: BubbleSort");
4 ArrayList<Integer> BubbleSortAccess = BubbleSort.getAccess();
5 if(BubbleSortAccess.indexOf(self.user) != -1) {
6         System.out.print(" (x) ");
7         allowedMethods.add(1);
8 }
```

```
9  System.out.println("");
10
11 System.out.print("2: FloatSort");
12 ArrayList<Integer> FloatSortAccess = FloatSorter.getAccess();
13 if(FloatSortAccess.indexOf(self.user) != -1) {
14         System.out.print(" (x) ");
15         allowedMethods.add(2);
16 }
17 System.out.println("");
18
19 System.out.print("3: MergeSort");
20 ArrayList<Integer> MergeSortAccess = MergeSort.getAccess();
21 if(MergeSortAccess.indexOf(self.user) != -1) {
22         System.out.print(" (x) ");
23         allowedMethods.add(3);
24 }
25 System.out.println("");
26
27 do {
28         System.out.print("\nPlease enter the ID of your choosen method which
               you have access to: ");
29         self.clsId = TextIO.getlnInt();
30 }
31 while(allowedMethods.indexOf(self.clsId) == -1);          //Check chosen
       method with allowedAccess
32
33
34 switch(self.clsId) {
35 case 1:
36         BubbleSort bCls = new BubbleSort();
37         self.input = bCls.start(self.input);
38         break;
39
40 case 2:
41         FloatSorter fCls = new FloatSorter();
42         self.input = fCls.start(self.input);
43         break;
44
45 case 3:
46         MergeSort mCls = new MergeSort();
47         self.input = mCls.start(self.input);
48         break;
49 }
50
51 System.out.println("\n\nIn sorted order (lowest first), your numbers are:");
         //Print the output
52 for(int i=0; i<self.input.size(); i++) {          //Print this ArrayList
53         System.out.println(self.input.get(i));
54 }
```

Princippet i denne del er, at man er nødt til at hardcode hver enkelt klasse fra "modules"-package'en ind i kildekoden, for at den kender til disse. Det letteste er at nedbryde kildekoden bid for bid.

Listing 5.4. MRNS base static - Uddrag af kildekoden - listing af classes

```
1 ArrayList<Integer> allowedMethods = new ArrayList<Integer>();
```

```
2
3 System.out.print("1: BubbleSort");
4 ArrayList<Integer> BubbleSortAccess = BubbleSort.getAccess();
5 if(BubbleSortAccess.indexOf(self.user) != -1) {
6         System.out.print(" (x) ");
7         allowedMethods.add(1);
8 }
9 System.out.println("");
```

Som i Listing 5.1 defineres først en ArrayList, som skal gemme på de klasser, som brugeren har lov til at anvende. Dernæst printes navnet manuelt, og den aktuelle klasses sikkerhedsopsætning hentes ind i `ArrayList<Integer> BubbleSortAccess`. Herefter tjekkes – igen manuelt – hvorvidt brugeren, som er logget ind har adgang til den pågældende klasse. Hvis ja, printes et "(x)" og klassen tilføjes til `allowedMethods`-listen.

Dette stykke kildekode skal gentages for hver klasse, der befinder sig i "modules" package'en. Ellers vil den ikke fremgå af programmet, når det afvikles.

Listing 5.5. MRNS base static - Uddrag af kildekoden - valg af class

```
1 do {
2         System.out.print("\nPlease enter the ID of your choosen method which
              you have access to: ");
3         self.clsId = TextIO.getlnInt();
4 }
5 while(allowedMethods.indexOf(self.clsId) == -1);          //Check chosen
      method with allowedAccess
6
7
8 switch(self.clsId) {
9 case 1:
10        BubbleSort bCls = new BubbleSort();
11        self.input = bCls.start(self.input);
12        break;
13
14 case 2:
15        FloatSorter fCls = new FloatSorter();
16        self.input = fCls.start(self.input);
17        break;
18
19 case 3:
20        MergeSort mCls = new MergeSort();
21        self.input = mCls.start(self.input);
22        break;
23 }
```

De første fem linjer er de samme som i den reflektive udgave af programmet, hvor brugeren skal vælge den klasse som vedkommende ønsker at anvende. Efterfølgende initialiseres en `switch`, hvor hver kendt class tjekkes imod den af brugeren valgte class. Stemmer de to ID'er overens initialiseres den pågældende klasse, og `start()` initialiseres.

Det er helt bevidst, at den ikke-reflektive version af MRNS i Listing 5.1 er kodet i så "grim" kode, som den er. Formålet er at vise et worst-case-senario på, hvor ineffektivt det kan laves. Man må forvente, at en optimal version af en ikke-reflektiv version af MRNS

vil være skrevet mere objektorienteret med nedarvning og interfaces, hvor hvert modul bygges til at nedarve en række settings fra en fælles overliggende klasse, så der i bedste fald kun skal tilføjes én linje kode til selve kernen, for hvert nyt modul, der tilføjes til programmet.

5.2.3 Sammenligning af MRNS ROP og MRNS non-ROP

Sammenlignes de foregående Listings er forskellene meget tydelige. Den ikke-reflektive kode er umiddelbart lettere at læse og gennemskue end den reflektive. På kort sigt, hvor alle classes kendes og man ved, hvad programmet skal kunne, er det imidlertid også lettest at anvende en ikke-reflektiv model.

Problemerne opstår ift. **dynamikken**!

Som det fremgår af Listing 5.3 er det tydeligt, at antallet af linjer i det inkluderede eksempel øges proportionelt med antallet af moduler, der skal implementeres i programmet i den statiske version – vel og mærke med forbehold for, at koden er så "grim", som den bevidst er gjort i eksemplet. Den proportionelle forøgelse vil afhænge af, hvordan koden er skrevet. Den statiske del kan vha. yderligere brug af for-løkker og objektorienteret programmering blive lidt mindre, men der vil dog stadig være en stigning i antallet af linjer for hvert modul, der bliver tilføjet til programmet. Dette er grundet, at programmet skal vide, at modulet eksisterer. Betydningen af dette er, at programmet vil på et tidspunkt være mindre (linjemæssigt) at lave det reflektivt, end hvis man ikke anvendte reflektive metoder. Altså er programmet op til et hvis punkt hurtigere at skrive uden reflektive metoder, men kravet til programmet er dog, at man skal ind og opdatere hele programmet, hvis man tilføjer et ekstra modul. Man må vurdere fra program til program, om der i fremtiden vil være brug for at udvide med ekstra moduler, og så foretage valget om det er smartest at lave det reflektivt eller statisk.

5.3 Performance problemer med reflektiv programmering

Den fleksibilitet man opnår gennem reflektiv programmering kommer dog ikke omkostningsfrit, da det medfører forsinkelser grundet store udvidelser af bindingen af navne af `methods`, `fields` osv. Denne binding sker normalt idet et program kompileres, men dette udskydes nu til runtime. Denne forsinkede binding af navne har betydning for performance ved søgning og checks ved runtime. Eksempelvis må `getMethod` søge hele hirakiet igennem for det passende method object. Et eksempel på dette er `Method.invoke`, som skal tjekke hvorvidt den har tilladelse til at initialisere den pågældende metode eller ej. Dette er dog ikke nødvendigvis negativt; det er blot en observation af de konsekvenser, som reflektiv programmering fører med sig.

5.3.1 Kategorisering af performancepåvirkningen

Den påvirkning på performance, som brugen af reflektiv programmering i Java har, kan deles op i tre kategorier. Det er vigtigt at forstå de tre kategorier, da de hver især påvirker performance på forskellige tidspunkter. De kan nemlig hver især påvirke en applikation forskelligt alt efter, hvordan denne er designet.

- **Construction overhead** – Tiden det tager at udføre ændringer på et class object under dets konstruktion (construction). Dette kan betyde ekstra forsinkelse idet man constructer en proxy klasse eller instans. Det kan dog også forekomme under dynamisk load og reflektiv construction. Normalvis forekommer denne forsinkelse kun en gang.

- **Execution overhead** – Den ekstra tid, som en service tager grundet et objekt/komponents reflektive funktionalitet. Eksempelvis den ekstra tid det tager at kalde en metode med `Method.invoke` fremfor et kompileret kald. Et andet eksempel er den ekstra forsinkelse skabt af at videresende et metodekald gennem en proxy. Generelt fremkommer Execution overheads (selvfølgelig) oftere end Construction overheads.

- **Granularity overhead** – Forsinkelse skabt som et resultat af reflektiv kode, der anvendes på flere objekter/komponenter end det var tiltænkt eller nødvendigt. Når man anvender en proxy hænder det, at dette anvendes på et helt interface, selvom dette ikke er nødvendigt. Eksempelvis med en synkroniserende proxy – det er muligvis ikke nødvendigt at synkronisere alle metoder hver gang. Den ekstra synkronisering kan resultere i ekstra forsinkelse.

Når man kender disse kategorier, er man i stand til at foretage nogle hurtige beslutninger. Eksempelvis er Construction overhead ikke et problem for applikationer med lang levetid, da alle klasser bliver loadet idet applikationen afvikles.

5.3.2 Mikro benchmarks

For at udføre nogen tests af afviklingshastigheden af reflektive programmer, anvendes benchmarks. Et normalt benchmark bruges til at teste performance for et givent stykke hardware eller software. Et mikro benchmark udfører en performancetest på et lille stykke software. Hvor stort eller lille dette skal være, er en subjektiv vurdering fra test til test.

Et eksempel på et mikro benchmark kan fx være som følgende eksempel pseudo kode:

Listing 5.6. Pseudo kodeeksempel på et mikro benchmark

```
1 Perform setup
2 m0 = get first measurement
3 for (int i=0; i<repCount; i++) {
4         Run code to be measured
5 }
6
7 m1 = get second measurement
```

```
8 report (m1-m0)/repCount
```

Bestemmelsen af et tilstrækkeligt antal gentagelser er en taktisk beslutning baseret på det pågældende program, som man ønsker at lave et mikro benchmark test på.

Listing 5.7 viser et faktisk benchmark af et "Hello World"-program.

Listing 5.7. Eksempel på et mirko benchmark af Hello World

```
1
2  public class HelloWorldBenchmark {
3          public static double aDouble = 123456789.0;
4          public static void main(String args[]) {
5                  int numberOfIterations = 15000;
6
7                  // Loop to measure the overhead long
8                  time0 = System.currentTimeMillis();
9
10                 for ( int j = 0; j < numberOfIterations; j++ ) {
11                         aDouble /= 1.000001;
12                 }
13
14                 long time1 = System.currentTimeMillis();
15                 aDouble = 123456789.0;
16                 System.out.println("Hello world!");
17                 long time2 = System.currentTimeMillis();
18
19                 for ( int j = 0; j < numberOfIterations; j++ ) {
20                         aDouble /= 1.000001;
21                         System.out.println("Hello world!");
22                 }
23
24                 long time3 = System.currentTimeMillis();
25                 double timeForOverheadLoop = (time1 - time0);
26                 double timeForHelloWorld = (time3 - time2) -
                       timeForOverheadLoop;
27
28                 System.out.println("HelloWorldBenchmark: " +
                       timeForOverheadLoop + " milliseconds for basic loop." );
29
30                 System.out.println("HelloWorldBenchmark: " +
                       timeForHelloWorld + " milliseconds for " +
                       numberOfIterations
31                 + " iterations." );
32
33                 System.out.println("HelloWorldBenchmark: " + (
                       timeForHelloWorld/numberOfIterations) + " milliseconds
                       per print command" );
34         }
35 }
```

Benchmarktesten i Listing 5.7 (som tager udgangspunkt i [8]) blev afviklet på en IBM T20 Thinkpad (750 MHz Pentium III med 256MB RAM, Java 2 Platform, Standard Edition, version 1.4.1 for Windows 2000). Efter 10 gennemløb af benchmarktesten tog det 327,4 mikrosekunder at skrive "Hello World!" til kommandolinjen. Standardafvigelsen var på

0,326. Tests indikerer altså, at et troværdigt interval indenfor 0,95 af værdien vil ligge mellem 327,15 og 327,65 mikrosekunder – vel indenfor de 5% krav, der var.

Grunden til denne simple benchmarktest afvikles nu er, at der skal skabes en basis for de benchmarks, der fremkommer af reflektive programmer senere i afsnittet. Selvom disse tests afvikles på en gammel maskine med Windows 2000, efterviser de stadig de elementer, som man skal være opmærksom på ift. reflektiv programmerings performance, da der jf. 7.1 ikke er implementeret performanceforbedrerende elementer af ROP i Java gennem de seneste versioner.

Følgende er nogle flere simple benchmarks. Den første beregner tiden, det tager at kalde en ikke-statisk metode. Listing 5.8 viser interfacet `DoNothingInterface`, som indeholder en metode, `doNothing`. Dette interface implementeres af `DoNothing`-klassen i Listing 5.9. Et mikro benchmark, `CallBenchmark` i Listing 5.10, måler tiden, det tager at kalde denne metode.
Efter 20 gennemløb af mikro benchmarket på den føromtalte Thinkpad computer viser det sig, at det tager gennemsnitligt 8,89 nanosekunder at kalde `doNothing`-metoden. Standardafvigelsen var på 0,71. Testene indikerer altså, at et troværdigt interval indenfor 0,95 af værdierne vil ligge mellem 8,46 og 9,31 nanosekunder, hvilket lige nøjagtig er indenfor 5% af gennemsnittet.

Listing 5.8. DoNothingInterface

```
1 interface DoNothingInterface {
2         void doNothing();
3 }
```

Listing 5.9. DoNothing som implementerer DoNothingInterface

```
1 public class DoNothing implements DoNothingInterface {
2         public void doNothing( ) {
3                 CallBenchmark.aDouble = CallBenchmark.compute(CallBenchmark.
                      aDouble);
4         }
5 }
```

Listing 5.10. CallBenchmark

```
1  public class CallBenchmark {
2          public static double aDouble;
3          public static double aDouble1;
4          public static double aDouble2;
5          public static double aDouble3;
6
7          public static double compute( double x ) {
8                  aDouble1 = aDouble1 + aDouble;
9                  aDouble2 = aDouble2 + 2 * aDouble;
10                 aDouble3 = aDouble3 + 3 * aDouble; return x / 1.000001;
11         }
12
13         public static void main(String args[]) {
14                 int numberOfIterations = 100000000;
15                 DoNothing target = new DoNothing();
16                 long time0 = System.currentTimeMillis();
```

```
17
18                  aDouble = 123456789.0;
19                  aDouble1 = 0;
20                  aDouble2 = 2;
21                  aDouble3 = 3;
22
23                  for ( int j = 0; j < numberOfIterations; j++ )
24                          aDouble = compute(aDouble);
25
26                  long time1 = System.currentTimeMillis();
27                  aDouble = 123456789.0;
28                  aDouble1 = 0;
29                  aDouble2 = 2;
30                  aDouble3 = 3;
31
32                  for ( int j = 0; j < numberOfIterations; j++ )
33                          target.doNothing( );
34
35                  long time2 = System.currentTimeMillis();
36
37                  double timeForCall = (time2 - time1) - (time1 - time0);
38
39                  System.out.println( "CallBenchmark: " + (time1 - time0)
40                          + " milliseconds for basic loop executing "
41                          + numberOfIterations + " iterations." );
42
43                  System.out.println( "CallBenchmark: "
44                          + timeForCall + " milliseconds for " +
                                numberOfIterations + " calls." );
45          }
46  }
```

5.3.3 Benchmark af to forskellige måder at anvende Proxy

Følgende er et benchmark af et program, som anvender reflektion og proxy. Dette benchmark måler både den tid, det tager at videresende et kald til en Java Proxy med et kompileret metodekald, og samtidig tiden det tager at foretage kaldet med `Method.invoke`. Listing 5.11 viser denne benchmarktest. Det tager tid på invokationen af metoden kaldet `doNothing` i `CallBenchmark`-klassen ved først at kalde metoden via proxy, hvor `DoNothingCaller` – som håndterer kaldet – implementerer videresendelsen af kaldet vha. et kompileret metodekald. Mikro benchmarktesten anvender også en anden klasse kaldet `DoNothingInvoker`, som også implementerer `DoNothingCaller`-interfacet. Denne klasse invoker vha. `Method.invoke`, som videresender kaldet til `doNothing`. Listing 5.12 indeholder `DoNothingCaller`- og `DoNothingInvoker`-klasserne.

Listing 5.11. InvokeBenchmark

```
1  import java.lang.reflect.*;
2
3  public class InvokeBenchmark {
4          public static DoNothing target = new DoNothing();
5          public static Class[] interfaces = { DoNothingInterface.class };
6
```

```
7           public static Object newProxy( InvocationHandler obj ) {
8                   return Proxy.newProxyInstance( obj.getClass().getClassLoader
                        (), interfaces, obj );
9           }
10
11          public static void main(String args[]) {
12                  int numberOfIterations = 5000000;
13                  DoNothingCaller caller = new DoNothingCaller();
14
15                  DoNothingInterface proxyForCaller = (DoNothingInterface)
                        newProxy( caller );
16
17                  DoNothingInvoker invoker = new DoNothingInvoker();
18                  DoNothingInterface proxyForInvoker = (DoNothingInterface)
                        newProxy( invoker );
19
20                  long time0 = System.currentTimeMillis();
21                  CallBenchmark.aDouble = 123456789.0;
22
23                  for ( int j = 0; j < numberOfIterations; j++ )
24                          InvokeBenchmark.target.doNothing();
25
26                  long time1 = System.currentTimeMillis();
27                  CallBenchmark.aDouble = 123456789.0;
28
29                  for ( int j = 0; j < numberOfIterations; j++ )
30                          proxyForCaller.doNothing( );
31
32                  long time2 = System.currentTimeMillis();
33                  CallBenchmark.aDouble = 123456789.0;
34
35                  for ( int j = 0; j < numberOfIterations; j++ )
36                          proxyForInvoker.doNothing( );
37
38                  long time3 = System.currentTimeMillis();
39                  double timeForProxyCall = (time2 - time1) - (time1 - time0);
40                  double timeForProxyCallPlusInvoke = (time3 - time2) - (time1
                        - time0);
41
42                  System.out.println( "InvokeBenchmark:␣" + timeForProxyCall +
                        "␣milliseconds␣for␣" + numberOfIterations + "␣proxy␣
                        calls." );
43                  System.out.println( "InvokeBenchmark:␣" +
                        timeForProxyCallPlusInvoke + "␣milliseconds␣for␣" +
                        numberOfIterations + "␣proxy␣calls␣using␣invoke." );
44          }
45  }
```

Listing 5.12. Klasserne anvendt i InvokeBenchmark

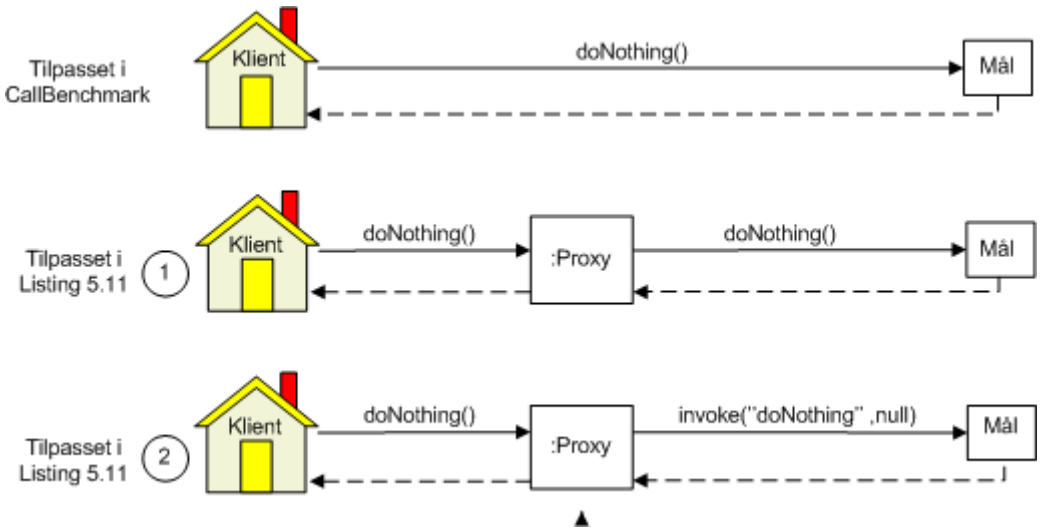
```
1  import java.lang.reflect.*;
2
3  public class DoNothingCaller implements InvocationHandler {
4          public Object invoke( Object t, Method m, Object[] args ) throws
               Throwable {
5                   InvokeBenchmark.target.doNothing();
6                   return null;
```

```
7           }
8   }
9
10
11  ------------------------------------------------------------------------
12
13
14  import java.lang.reflect.*;
15
16  class DoNothingInvoker implements InvocationHandler {
17          public Object invoke( Object t, Method m, Object[] args ) throws
                Throwable {
18                  return m.invoke( InvokeBenchmark.target, args );
19          }
20  }
```

Figur 5.1 viser de tre sekvenser i et diagram, som hver især repræsenterer de intervaller, der er timet i benchmarkene. Den øverste sekvens repræsenterer situationen vist i `CallBenchmark` (Listing 5.10). De to nederste sekvenser repræsenterer situationerne i `InvokeBenchmark` (Listing 5.11).

Ved 9. gennemløb af disse mikro benchmark på føromtalte Thinkpad maskine blev følgende resultater optaget:

- Ved kald af `doNothing` gennem proxy med et kompileret metodekald viste der sig en gennemsnitlig responstid på 25,0 nanosekunder. Standardafvigelsen var på 1,34. Testene indikerer altså, at et troværdigt interval indenfor 0,95 af værdien vil ligge mellem 23,99 og 26,01 nanosekunder – vel indenfor de 5% krav der var.
- Ved kald af `doNothing` gennem proxy, hvor `Method.invoke` anvendes, viste der sig en gennemsnitlig responstid på 2928,0 nanosekunder. Standardafvigelsen var på 12,57. Testene indikerer altså, at et troværdigt interval indenfor 0,95 af værdien vil ligge mellem 2918,5 og 2937,5 nanosekunder – vel indenfor de 5% krav der var.



Figur 5.1. De tre forskellige metoders benchmark timet i InvokeBenchmark

Begge af de to ovenstående tests måtte have metoderne initialiseret og returneret en værdi. Målingerne er derfor konsistente med, hvad der fremgik af `CallBenchmark`.

`InvokeBenchmark` programmet giver nogle basale målinger, som direkte appellerer til problemerne i at erstatte et kompileret metodekald med enten en dynamisk proxy, som videresender kaldet, ved at erstatte det med brug af `Method.invoke`, eller ved at erstatte det med en proxy som videresender kaldet ved brug af en kompileret metode.

Målingerne viser, at kaldet af en metode (som i dette tilfælde er uden parametre) gennem en proxy server er ca. 2,29 gange så omkostningsfuld (i tid og ressourcer), som afvikling af et direkte kompileret kald. Endvidere – kald af metoden gennem en proxy, som anvender `Method.invoke` er ca. 331 gange så omkostningsfuld som et direkte kald. Værd at bemærke er det dog, at dette ikke betyder, at en given applikation vil blive afviklet enten 2,29 eller 331 gange så langsomt, hvis de anvender disse metoder. Applikationer gør meget mere end blot disse kald. I forlængelse med dette er det endvidere værd at notere, at der er en væsentlig teoretisk performancenedgang – både ved construction, execution og granularity – når der anvendes proxy og reflektive funktioner.

Analyse af anvendelsesmulighederne 6

Der er ikke nogen tvivl om, at de to It-skandaler nævnt i starten af rapporten, i sandhed var skandaler! Ekstreme budgetoverskridelser og ydelser, som aldrig blev leveret komplette. Men hvad gik galt? Hvorfor gik det ikke godt? Og kunne reflektiv programmering have afhjulpet nogle af de problemer, som gjorde ellers lovende projekter til skandaler?

I det følgende benyttes den nyopnåede viden vedrørende ROP til en analyse af de to It-skandaler, for at undersøge hvorvidt ROP kunne have afhjulpet nogle af de problemer, som de løb ind i undervejs. Herefter beskrives og analyseres en konkret virksomhed og et konkret tilfælde, hvor denne virksomhed benytter sig af ROP, og hvilke fordele dette giver denne virksomhed. Til slut vurderes det ud fra dette og resten af rapporten, hvor det tilsyneladende vil være hensigtsmæssigt at benytte sig af reflektiv programmering.

6.1 ROP ift. It-skandalerne

En analyse af de to It-skandaler; Hvad der gik galt, og hvorvidt ROP muligvis kunne have hjulpet på de problemer, der belyses i analysen.

6.1.1 1. skandale: AMANDA

Problemet med AMANDA var, at systemet var langsomt, besværligt og omfattende at anvende. En registrering, som tog 10-20 minutter i det system, som AMANDA afløste, rapporteres at have taget op til 50 minutter med AMANDA. Et indiskutabelt krav til systemet var, at skærmbillederne skulle kunne åbnes meget hurtigt. Det resulterede i, at en ny registrering med AMANDA blev delt udover 50 forskellige skærmbilleder – hver især af dem åbnede indenfor den specificerede tidsramme. Dette betød imidlertid, at når en person skulle klikke sig igennem disse, blev det en tidskrævende proces.

Med udgangspunkt i en ekspertvurdering fra Arbejdsmarkedsstyrrelsen[1] fremstår det tydeligt, at der er en række problemer med AMANDA. Citeret fra ekspertvurderingen:

- Det forhold at systemet har været i drift i knap 6 måneder og driftsprøven er godkendt dokumenterer, at de HPS-udviklede applikationer kører stabilt.
- At systemets grundlæggende funktionalitet understøtter AF-regionernes forretningsprocesser i en sådan grad, at produktionen i AF-regionerne på de særligt prioriterede områder(formidlinger og handlingsplaner) ifølge det fra Arbejdsmarkedsstyrelsen og Arbejdsministeriet oplyste kan afvikles på et forsvarligt niveau. Systemets indvirkning på det samlede produktionsniveau er dog fortsat negativ i forhold til situationen før idriftsættelsen.
- At de forventede produktivitets- og kvalitetsgevinster, som AMANDA skulle give, ikke er blevet indfriet på grund af en forældet teknologisk platform og en række uhensigtsmæssigheder omkring applikationerne.
- At systemet i den hidtidige fase i høj grad har fungeret i kraft af en ekstraordinær stor indsats og tålmodighed fra medarbejdernes side.

Der kan gisnes om, hvad der har været direkte skyld i disse problemer, og hvad der kunne have afhjulpet dem. Både nu, men også i udviklingsfasen.

- **Dårlig planlægning** – Havde hele projektet være udviklet med udgangspunkt i de agile principper, havde man undervejs haft mulighed for at vurdere delmængder af systemet, og derfor fange dette problem i opløbet. I stedet virker det udadtil som om, at udviklerne har fået en kravspecifikation, udviklet produktet, afleveret det og lukket projektet.
- **Dårlig performance** – AMANDA har fået meget kritik for dets dårlige performance og mangel på oppetid - at det er alt for ressourcekrævende. Især har HPS-platformen været kritiseret for, allerede i udviklingsfasen, at være på vej mod pensionering.
- **Dårlig systemintegration** – Dårlig integration af eksterne systemer var medførende til, at ansatte i AF måtte yde langt større indsats end nødvendigt for at anvende de systemer, som skulle være integreret i AMANDA. Problemet her opstod i udviklingsfasen under integrationen af disse.

Nu rejses spørgsmålet: Kunne ROP have afhjulpet disse problemer? Ja og nej. Problemerne ift. den dårlige planlægning og performance kan ikke umiddelbart løses med ROP. Uden at have dybdegående kendskab til AMANDA, da systemets kildekode naturligvis holdes tæt på kroppen, ville ROP være et oplagt løsningsforslag til systemintegrationen. Endvidere vides det allerede, hvilke forskelle ROP gør ift. opdatering og vedligeholdelse af systemer. Havde hele AMANDA været baseret på ROP kan der gisnes om, hvorvidt opdateringer af brugergrænsefladen og måske endda HPS kunne have medført et bedre system, som ville have haft længere levetid, dvs. at applikationen ville have været i drift i længere tid, end det var tilfældet for AMANDA, og dermed kunne have været mere værd, end det var tilfældet.

6.1.2 2. skandale: Folkeskolens digitale afgangsprøve

At forvente andet, end at folkeskolernes afgangsprøve i dagens samfund er digitaliserede, virker fuldstændig utopisk! Alligevel har digitaliseringen af de første prøver, som startede tilbage i 2006, udviklet sig til en regulær skandale. Systemnedbrud, manglende adgang og urimelige svartider er blandt de problemer, der er meldt tilbage til Cowi, der af staten er ansat til at levere den omdiskuterede digitale afgangsprøve.

De problemer, der fra 2006 til 2009 blev meldt tilbage til Cowi:

- **Systemnedbrud** – Grundet de mange samtidige forbindelser oplevede folkeskoleeleverne i 2006 og 2007 hel- eller delvis manglende adgang til deres afgangsprøve, grundet et systemnedbrud forårsaget af for mange samtidige forbindelser.
- **Urimelige svartider** – I 2008 var problemet ikke længere systemnedbrud. Nu kæmpede Cowi med urimelige svartider, der ofte resulterede i timeout. Igen skydes skylden på de mange samtidige forbindelser.

Når et helt eller delvist systemnedbrud forårsages af for mange samtidige forbindelser, kan der være mange steder, hvor systemet fejler. Set fra "toppen" af hierakiet:

1. En dårlig eller forkert konfigureret **firewall** kan være skyld i, at forbindelsen til de bagvedliggende servere bilver utilgængelig. Dette er dog højst usandsynligt. Hvis dette var tilfældet må man forvente, at Cowis teknikere hurtigt ville kunne reparere fejlen og være online igen. Det kan ikke siges var tilfældet.
2. Overvurdering af **hardwarens** kapacitet. Hvis Cowi forventede, at den afsatte mængde hardware kunne håndtere eksempelvis 10.000 samtidige forbindelser, men reelt kun kunne håndtere 6.000 samtidige forbindelser, vil det resulterer i en softwarefejl før eller siden.
3. System **overflow** grundet dårlig programmering. Såfremt kildekoden til systemet bag den digitale afgangsprøve er dårligt lavet, kan det nemt ske, at serverne foretager unødvendig mange operationer og beregninger. For en enkelt bruger er dette måske ligemeget, men når der sidder 10.000 online samtidig kan dette blive fatalt - især hvis pågældende fejl betyder, at systemets performance forringes eksponentielt med antallet af samtidige tilslutninger. Det forventes dog, at Cowi har haft tanke på dette, da en hel del af verificeringen af elevernes svar afvikles lokalt på klienterne og ikke på serverne. Dette åbnede på den anden side blot op for, at det blev muligt at snyde[16] til eksamen.

Sammenlignes de tre mulige fejlkilder, er det mest sandsynlige nr. 2 eller 3. Enten har Cowi fejlvurderet kapaciteten af deres hardware – hvilket dog ville være nemt at rette op på, ved blot at tilføje mere hardware til den samlede løsning for derved at øge kapaciteten – eller også gemmer der sig et hav af små kodefejl, der sammen resulterer i et systemnedbrud ved 10.000 eller flere samtidige forbindelser. En dårligt konfigureret firewall og overvurdering af hardwarens kapacitet er problemer, som ikke kan løses vha. ROP. Omvendt er det også problemer, som man forventer Cowi hurtigt ville kunne løse. Taget i betragtning, at det

har taget dem godt og vel 3 år, at komme med en fungerende løsning, må disse ikke kunne antages for at være problemet.

Tilbage står fejl i programmets kildekode. Modsat AMANDA kan udviklingsmodellen ikke denne gang få skylden, da der simpelthen ikke har været mulighed for at teste på fuld skala mellem hver iteration. Til fælles med AMANDA har den digitale afgangsprøve dog, at kildekoden holdes tæt til kroppen. Fælles med AMANDA er det også, at det på nuværende tidspunkt vides, at ROP åbner op for en masse muligheder ift. opdateringer og vedligeholdelse af kildekode. Hele forløbet taget i betragtning er det ikke sandsynligt, at anvendelse af ROP fra starten ville have sikret et system, som var fejlfrit fra dag 1. Sikkert er det dog, at det åbner op for en række muligheder i forbindelse vedligeholdelse af systemet, som kunne have lettet disse og leveret et funktionelt produkt hurtigere.

6.2 Møde med KMD

Der foreligger ingen tvivl om It-skandalernes omfang. Softwaren, som er udviklet i de to sager, er begge blevet udviklet af store virksomheder. For at forstå de processer og tanker som florerer, når en virksomhed i den størrelse modtager en ordre på et system lignende fx AMANDA, har der været taget kontakt til KMD. KMD (tidligere Kommunedata) er et af Danmarks førende softwareudviklingshuse, som bl.a. leverer løsninger til størstedelen af de danske kommuner. KMD blev kontaktet, fordi der ønskes at skabe en forståelse for, hvordan en stor virksomhed som KMD arbejder, hvilke arbejdsmetoder de bruger, og spørge dem om hvorvidt de anvender reflektiv programmering i deres arbejde – og i så fald, hvordan. I denne sektion af rapporten, vil dette blive forklaret og beskrevet. [1]

6.2.1 Interviewet med KMD

Der blev forsøgt at skabe kontakt via e-mail d. 22. april 2010, og 7 dage senere, d. 29. april, blev der oprettet kontakt. Herefter gik det stærkt og allerede 5 dage senere, d. 4. maj 2010, blev interviewet arrangeret. Det blev besluttet i gruppen, at det var fornuftigt, hvis alle seks gruppemedlemmer mødte op til interviewet, da der på den måde ville komme flere øjne, og derved forskellige meninger og opfattelser af interviewet.

Forberedelse til interviewet

Op til interviewet med KMD blev der forberedt 3 overordnede spørgsmål:

- Anvender I reflektiv programmering? Hvis nej – hvorfor ikke?
- Hvilke fordele ser I ved at anvende reflektiv programmering fremfor andre metoder? Ser I i den forbindelse også nogle ulemper ved at anvende reflektiv programmering?
- I hvilken forbindelse anvender I reflektiv programmering? Ifm. modularitet, remote access, security eller noget helt fjerde?

[1] Alle informationer om KMD er opnået gennem et interview med Michael Bach Pedersen, softwarearkitekt hos KMD, dateret d. 4. maj 2010.

Interviewet blev forberedt til at foregå som et Semistruktureret Interview[7], hvor der ville blive stillet nogle overordnede og indledende spørgsmål, og dialogen med KMD derefter ville få frit løb indenfor emnet. Ønsket med denne struktur er, at i stedet for at stille KMD nogle simple og meget faste ja/nej spørgsmål, så har KMD mulighed for selv at dreje emnet i den retning, som de mener er passende, og få videregivet de informationer, som de mener er relevante og brugbare i forhold til det pågældende emne.

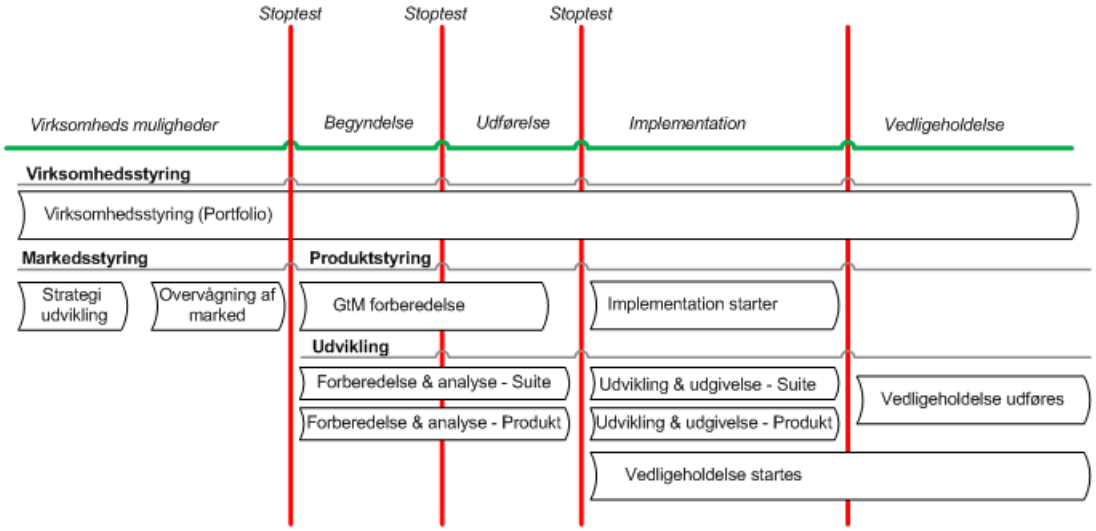
KMD bookede et af deres mødelokaler til interviewet, hvor interviewet fandt sted. Kontaktpersonen, Michael Bach Pedersen, var meget åben i interviewet og besvarede alle spørgsmål, der blev stillet, som generelt førte til en god, åben og bred dialog. Under hele interviewet blev der således skrevet noter af 2 af gruppens medlemmer, som kunne underbygge alle de informationer, der blev givet. Interviewet varede lidt under halvanden time. Der foregik således, at to til tre gruppemedlemmer stod for hoveddelen af spørgsmålene, og alt imens Michael Pedersen besvarede disse, tænkte de resterende gruppemedlemmer over supplerende og uddybende spørgsmål. Igennem hele interviewet blev der skabt en forståelse for KMD's arbejdsmetoder, deres systemer og deres forhold til reflektiv programmering.

Efter mødet fortsatte dialogen med Michael over e-mail, hvor vi fik udleveret en række sekundære informationer. Dette inkluderede en række illustrationer og kodeeksempler samt svar på en række opfølgende spørgsmål, som der ikke blev tid til under interviewet. De sekundære elementer, som var relevante for rapportens indhold, er inkluderet heri løbende.

6.2.2 Arbejdsgangen i KMD

Arbejdsmetoden, der arbejdes efter i KMD, er der tidligere stiftet bekendtskab med i indeværende rapport. De anvender vandfaldsmodellen (jf. 2.5.1). Der forekommer dog elementer fra agile udviklingsmetoder, som fx kontakt med kunden under udviklingen m.m. Officielt anvender KMD vandfaldsmodellen, men uofficielt forekommer der således agile træk.

Hos KMD har man et internt udviklingssystem, som er kaldet QUP. Dette system er basalt set en utrolig stor skabelon for præcis hvordan de forskellige udviklingforløb skal foregå. Der er en skabelon for hvordan det skal foregå, hvis KMD fx skal udvikle en Windows-applikation, en web-service eller en web-applikation. At korte QUP ned til en lille tegning, der skal vise det overordnede system er meget svært, da der som sagt er ufatteligt mange ting i systemet. På figur 6.1 er dette imidlertid forsøgt, og QUP er blevet delt op i de fem store hovedpunkter, som systemet foregår udfra. Under disse fem hovedpunkter er der nogle delpunkter, der bliver lavet i perioderne, og disse kan deles yderligere op i andre små skemaer. Som sagt er QUP meget stort, og det er derfor praktisk talt umuligt at forklare hele systemet på ét skema.



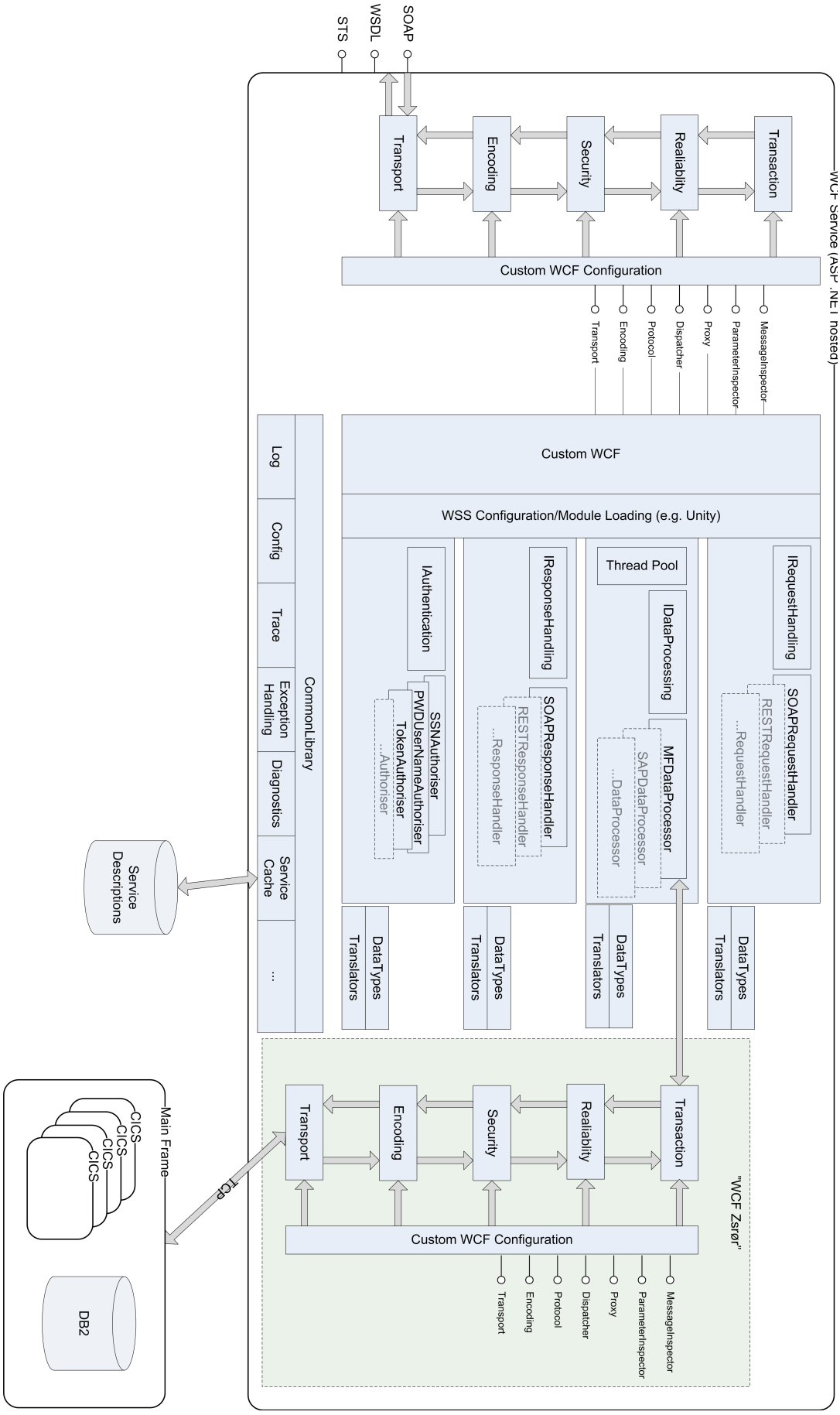
Figur 6.1. Illustration af udviklingsprocessen af et produkt fra projektstart til færdiggørelse og vedligeholdelse - lavet med udgangspunkt i samtaler med KMD

Udover klassisk "bestillingsarbejde" tager KMD også selv initiativ til nye produkter. Når KMD skal lave et nyt produkt, bliver idéen til dette dannet ud fra en overordnet strategi, til hvilke slags produkter KMD kunne finde på at lave til løsningen. Det kunne eksempelvis være et nyt og smart system til håndtering af alle barselssager i kommunerne. Når denne idé er dannet analyseres markedet for at se, om der er nogen grobund for dette nye produkt. Såfremt dette er tilfældet, startes et nyt projekt, for at skabe det nye produkt. Projektet startes med at en projektleder opstiller krav til produktet, og til hvordan produktet overordnet skal fungere. Det kan selvfølgelig også være at produktet er bestilt af en kunde, og i så fald er der nogle bestemte krav til produktet, som skal opfyldes. Når projektet er sat i gang, laver KMD en *GtM-forberedelse* (Go to Market). Det vil sige, at de gør produktet klar til at komme ud på markedet. I udviklingsprocessen skal der tages et valg, om hvorvidt de skal lave en produkt- eller suiteløsning. En produktløsning er hvor KMD laver et produkt helt fra bunden af, udfra de krav, der er stillet. En suiteløsning er derimod en "færdig pakkeløsning", som kan modificeres lidt. Dvs. at koden allerede er skrevet, og kun skal ændres en lille smule for at passe til det aktuelle formål. Når KMD følger deres QUP-arbejdsmetode med hensyn til vandfaldsmodellen, til at udvikle deres egne produkter, bruger de mange stop i forløbet efter hver fase. Dette kan også ses på 6.1, hvor stoppene er markeret med en linje, der er kaldet stoptest. Under hvert af disse stop vurderes det, om det stadig kan betale sig økonomisk at færdiggøre projektet. Hvis ikke det kan det, bliver projektet med høj sandsynlighed bremset og lagt på hylden, men hvis der stadig er økonomisk gevinst at hente, fortsætter projektet ind i næste fase. Disse stoptests forekommer naturligvis kun i de tilfælde, hvor det er KMD selv som har taget initiativet til projektet. I de tilfælde, hvor KMD laver bestilt arbejde for folk udefra, erstattes disse stoptests med løbende dialog med kunden mellem "iterationerne" for hele tiden at sikre, at projektet er på rette spor. I disse tilfælde er der også den store forskel, at det er kunden udefra som betaler for hele forløbet!

Udover brugen af QUP, vandfaldsmodellen og andre mulige arbejdsmetoder, bruger KMD også modellen CMMI – se afsnit 6.3.

6.2.3 KMD's forhold til ROP

Efter mødet med KMD er det blevet klargjort, at KMD i vid udstrækning anvender ROP til en stor del af deres projekter.



Figur 6.2. Illustration af opbygningen af KMD's KWS.

Foran KMD's mainframe ligger en webserver (en Microsoft IIS server) kaldet *KWS* (KMD Web Server, se 6.2), som håndterer og filtrerer alle kald til mainframe'en. Det er både af praktiske og sikkerhedsmæssige årsager, at kald til mainframe'en ikke går direkte til den, men først skal igennem et abstraktionslag i form af føromtalte webserver. Alle kald til mainframe'en skal ske via *SOAP*[2] (Simple Object Access Protocol), og bliver håndteret som en webservice. Det smarte heri er dynamikken: Alle typer platforme og systemer har nemlig adgang til at tilgå disse webservices, uanset programtype og styresystem.

KWS'en – et produkt under udvikling af KMD til internt brug – viser, hvordan en virksomhed i dag aktivt bruger ROP. KMD's mainframe er dog efterhånden gammel, og trænger ifølge deres egne udsagn til at blive udskiftet. Derfor er KMD i gang med at udvikle en ny KWS. Den nye KWS er bygget ud fra principperne i ROP. KWS vil være modulbaseret med en kerne, som reflektivt kan undersøge de omkringliggende moduler, og inkludere & initialisere de, som i øjeblikket er nødvendige (lidt i stil med MRNS, faktisk). Naturligvis kunne KMD ikke afsløre dybere detaljer end de helt basale outlines for KWS, men det er heller ikke nødvendigt i kraft af, at formålet med mødet netop var at få belyst, hvorvidt ROP anvendes i virksomheder i dag, og til hvilke formål det anvendes.

KMD anvender i vid udstrækning ROP i alle de sammenhænge, hvor det kan betale sig. Her er altså tale om, at det anvendes ved lejlighed, i stedet for helt udelukkes. KMD lægger meget vægt på, at alle projekter baseres på en grundig kravspecifikation og at designfasen udføres grundigt og godkendes, før selve udviklingen går i gang. Dermed har de mulighed for tideligt at analysere, hvorvidt ROP er den korrekte teknologi til løsningen af den enkelte opgave. I alle de projekter, hvor dynamik, fremtidig opdateringer og vedligeholdelse forventes, har KMD som udgangspunkt ROP til løsning af opgaven. I forbindelse med sine projekter, hvor KMD anvender ROP, udvikles disse også efter faste modeller med høje kvalitetskrav. Kvalitetskrav, som defineres af den internationalt anerkendt model, CMMI.

6.3 CMMI

CMMI (Capability Maturity Model Integration) er en model, som er udviklet af Software Engineering Institute på et universitet i USA [15]. Modellen er lavet på det grundlag, at give det amerikanske forsvar en målestok for modne leverandører. For at vurdere en organisations modenhed bruger CMMI en 5-trins skala, hvor 1 er lavest og 5 er højest. Hvis man er på niveau 1 følger man kun få af arbejdsgangene, mens hvis man er på niveau 5, følger man alt fra CMMI.

[2] ref: http://en.wikipedia.org/wiki/SOAP

Niveau		Muligheder	Resultat
5	Vedvarende proces forbedring	- Organisere, forny & implementere - Tilfældig analyse & beslutsomhed	Produktivitet & kvalitet
4	Kvantitativ styring	- Kvantitativ processtyring - Kvalitetsstyring af software	
3	Proces standardisering	- Udviklingskrav - Teknisk løsning - Produkt integration - Godkendelse - Fokus på organisationsproces - Definition på organisationsproces - Organiserede træning - Integrerede produktstyring - Risikabel styring - Integrerede samarbejde - Integrerede leverandør styring - Diskutere analyse - Organiserede miljø for integration	
2	Grundlæggende projektstyring	- Krav til styring - Projekt planlægning - Projekt overvågning - Styring af leverandøraftale - Måle & Analysere - Produkt & proces kvalitetsgaranti	
1	Heroisk indsats	- Design - Udvikling - Integrere - Test	Risiko & spild af tid

Figur 6.3. CMMI's 5-niveauer

Der er cirka 200 virksomheder på verdensplan som har en såkaldt CMMI 5-certificering[15]. Blandt dem finder man den danske virksomhed KMD. Udover KMD arbejder blandt andre Danske Bank, Systematic, og Post Danmark også med CMMI, dog er det kun Systematic og Post Danmark, der er certifiticerede indenfor området.

CMMI giver kunderne større sikkerhed ved valg af leverandør, mener koncernchefen for Systematic, der netop er certificeret indenfor CMMI, Michael Holm[14]. Samtidig slår han fast, at flere kunder vurdere softwarevirksomheder på deres modenhed, da kunden ikke vil acceptere fejl i It-projekter, samt overskredne deadlines.

Det certificerede firma indenfor CMMI, Systematic, fremstiller missionkritisk software til forsvars- og sundshedsindustrien, og har brugt over 160.000 timer på CMMI siden 1997 [14]. Virksomheden har 8-10 ansatte, der arbejder med CMMI, og har samtidig et amerikansk konsulenthus tilknyttet, der sikrer at arbejdet med CMMI ikke kører af sporet. Ifølge koncernchefen kan det sagtens betale sig, at bruge så mange ressourcer på modenheden af virksomheden.

For at give et bedre indblik i, hvordan CMMI styres ved de forskellige niveauer, er der på Figur 6.3 illustreret, hvordan man bruger CMMI modellen i virksomheder. Jo højere niveau af CMMI, dets flere punkter følger man; dvs. hvis man er certificeret til niveau 5, som er tilfælet med Systematic, så følger man modellen til punkt og prikke, og alle 5-niveauers muligheder tages i betragtning og drøftes. Dette sikrer, at de CMMI certificerede

virksomheder udvikler optimeret og fornuftig kode som overholder standarderne.

6.4 Optimal anvendelse af- & ulemperne ved ROP

Nu hvor det er blevet klargjort, hvad det indebærer at udvikle ved hjælp af reflektiv orienteret programmering, samt hvilke muligheder det åbner op for, er det muligt at undersøge, i hvilke sammenhænge anvendelse af ROP vil være optimal.

Følgende liste af optimale anvendelsesområder er baseret på de foregående kapitler, MRNS programmet og erfaringerne fra KMD.

- **Modularitet:** Retfærdiggørelsen af anvendelsen af ROP ift. modulbaserede applikationer eftervises både af MRNS og KMD. I MRNS eftervises det i form af kernen af programmet, som selv kan reflektere over sine omgivelse og inkludere & initialisere de omkringliggende "moduler", som programmet i øjeblikket skal bruge. KMD anvender modulbaseret reflektion til kernen af langt de fleste af deres ydelser - KWS'en - hvor den nyeste version i stil med MRNS er opbygget omkring en reflektiv kerne, der kan analysere sine omgivelser og inkludere det i det øjeblikket påkrævede modul/plugin.

- **Opdatering & vedligeholdelse:** I forlængelse af modulerne er reflektiv programmering yderst hensigtsmæssigt i forbindelse med opdateringer og vedligeholdelse af eksisterende programmer. Et program med en reflektiv kerne kan forholdsvis nemt opdateres, da det "blot" vil være modulerne, som skal opdateres.

- **Kodegenerering & modificering:** Som vist i afsnit 4.2.3 er ROP anvendeligt til automatisk generering af kode. Derudover kan ROP redigere properties ved runtime. Fx rette en `private` metode til en `public`, hvis man har brug for et midlertidig "hax", som gør en metode tilgængelig i en bestemt situation.

På trods af at indeværende rapport hovedsaligt har fokuseret på de mange anvendelsesområder, hvor ROP ville være fordelagtigt at anvende, foreligger der naturligvis også en række områder, hvor ROP ikke umiddelbart er at foretrække, eller hvor det har nogle ulemper.

- **Små opgaver:** Anvendelsen og implementationen af ROP kan være en meget omfattende proces. Derfor vil det ofte være uhensigtsmæssigt at påbegynde implementationen af ROP i mindre projekter. Dette understreges især, hvis ikke der forventes de store vedligeholdelsesopgaver efterfølgende, hvor ROP tidligere har vist sig yderst hensigtsmæssigt.

- **Afviklingshastighed:** Et af de største problemer med ROP er dets performance. Den er mildest talt ringe. Alle tests peger i samme retning: Anvendelsen af ROP

påvirker afviklingshastigheden af en given applikation negativt. Sammenlignes dette med teorierne om programarkitekturen fremført i afsnit 4.2.4 og 5.3 stemmer det også godt overens i kraft af, at reflektiv programmering nødvendigvis må udføre en række ekstra operationer og undersøge sine omgivelser en ekstra gang ved afvikling. Det må imidlertid resultere i, at afviklingshastigheden er væsentlig langsommere, end en Java-applikation (eller en hvilken som helst anden applikation, for den sags skyld) som ikke anvender ROP.

- **Ændringer af rettigheder:** Dets styrke i forbindelse med muligheden for fx at ændre en `private` metode til en `public`, er i virkeligheden også en af dets svagheder. Når en udvikler vælger at gøre en metode til `private` idet han udvikler et program, må man formode, at der er en grund hertil. Derfor kan det i nogle tilfælde vise sig kritisk at "rode ved noget, som man oprindeligt ikke havde adgang til". Eksempelvis kan det skabe store sikkerhedshuller eller programfejl efterfølgende.

Teknologien "Reflektiv Orienteret Programmering" har altså både sine fordele og ulemper. Man må dog erkende, at i de tilfælde, hvor det er oplagt at anvende, vil det også være yderst fordelagtigt faktisk at gøre dette.

6.5 Opsamling

Ovenstående analyser af AMANDA, COWI, KMD og CMMI har givet rapporten en dybere indsigt i, hvad reflektiv programmering kan bruges til – eller burde have været brugt til. Skandaler som AMANDA har vist, at uanset hvilken metode der bruges, så vil dårlig kommunikation mellem udviklerne og kunden, samt forkerte udviklingsprocesser næsten med sikkerhed resultere i skandaler.

Ydermere har COWIs afgangsprøver – udviklet til folkeskolen – vist, hvordan dårlig programmering kan føre til store problemer, og i dette tilfælde skandaler. Det er ikke muligt at fastslå præcist om COWI med fordel kunne have anvendt reflektiv programmering, men analysen viser, at noget gik helt galt programmeringsteknisk.

Derudover er KMD blevet kontaktet, og et møde med en af deres softwarearkitekter har fundet sted. Under mødet blev KMDs fremgangsmetoder gennemgået, og spørgsmål til deres forhold til reflektiv programmering blev stillet og besvaret. Det viste sig, at reflektiv programmering blev anvendt alt efter kundens behov, men det er ikke noget de tænker over at de bruger, da dette er integreret i deres system. Mødet med KMD har givet rapporten en større indsigt i praktisk brug af reflektiv programmering, og har bekræftet mange af rapportens teorier, der er gennemgået i tidligere kapitler.

Herefter blev besøget hos KMD, samt analyser af AMANDA og COWI, opholdt ift. teorien, og herudfra er der opstillet fordele og ulemper ved brugen af reflektiv programmering. Hos KMD bliver reflektiv programmering brugt til modulbaseret software, og dette har også fra starten af rapporten stået som en klar fordel ved ROP. Derudover er det blevet klart, at opdatering & vedligeholdelse samt kodegenering er klare fordele. Men det er også blevet klart, at der er ulemper, som fx afviklingshastigheden af software skrevet med ROP.

Afsluttende kapitel 7

7.1 Diskussion

Reflektiv programmering stiller mange muligheder til rådighed for programmøren, der er tiltænkt at skulle gøre vedkommendes arbejde nemmere. Ved at dette sker, må selve udviklingsprocessen også optimeres en hvis grad, ved at der er mindre arbejde at lave, og softwaren herved kan færdiggøres hurtigere. Dette var tanken bag at undersøge reflektiv programmering, dvs. at finde ud af i hvor stor en grad, det kunne påvirke hele den proces, det er at udvikle et stykke software, og til dels senere vedligeholde det.

7.1.1 Tekniske aspekter af ROP i forhold til softwareudvikling

Reflektiv programmerings primære egenskab er introspektion, der gør reflektiv kode i stand til at undersøge sig selv og sine omgivelser. Denne egenskab medfører, at reflektiv kode er meget abstrakt, og implementationen af nye moduler i et stykke reflektiv kode ofte kan klares uden ekstra kode tilføjes til den oprindelige kildekode. Dette er blevet illustreret med MRNS eksemplet, hvor reflektiv programmerings modularitet blev efterbevist, og moduler blot skulle lægges ind i modulpakken, og uden ekstra kode blev tilføjet til resten af softwaren, blev modulet integreret. At opbygge ens kode efter disse principper, burde på trods af kompleksiteten af reflektiv kode, spare programmørerne for en stor mængde arbejde, i modsætning til hvis softwaren ikke var opbygget reflektiv. Overordnet set vil udviklingsprojekter med modulbaseret software blive optimeret ved at benytte reflektiv programmering, fremfor blot objektorienteret programmering. Men om fordelen ved modulariteten kan videreføres til et projekt, skal dog stadig vurderes fra projekt til projekt, da det ellers kan medføre en del ekstra og unødvendigt arbejde, pga. reflektiv programmerings kompleksitet.

Reflektiv programmering stiller som nævnt også andre muligheder til rådighed end blot "grundpakken", der bl.a. gør modularitet muligt. Dette er elementer som proxy og kodegenerering.

Hvis der ses på proxy alene, er dette en teknik, der ved korrekt implementation kan spare en softwarevirksomhed for meget arbejde i deres implementationsfase af udviklingsprocessen. Hvis softwaren er opbygget med en proxy, er genbrug en af

hovedpunkterne i implementationen, og jo mere kode, der genbruges, dets mindre arbejde kræves der af programmøren. Fordelen ved genbrug i proxy, i forhold til andre genbrugs metoder eller hvor koden direkte kopieres ind flere steder, er at genbrugen her sker dynamisk. Som forklaret i afsnit 4.2.2 henter proxy selv de nødvendige metoder, ved at se på en klasses interface, hvilket medfører at en metode kun skal stå et sted i koden, hvorefter den kan benyttes af alle klasser, som var den integreret i disse.

Denne form for genbrug sikrer, at hvis en fejl opdages undervejs i udviklingsprocessen kan den relativt nemt og hurtigt rettes, da det blot er et sted i koden, der skal rettes. Dette bringer en stor mængde fleksibilitet til et udviklingsprojekt, da man vil være knap så hæmmet af en eventuel fejl, og muligvis stadig være i stand til at levere softwaren både til tiden og den aftalte pris. Udover disse indvirkninger, er der pga. af den dynamiske måde genbrug fungerer på i en proxy, en del mindre kode der skal vedligeholdelse. Hvis det besluttes efter et stykke software er blevet lanceret, at en funktion i dette skal modificeres, kræver dette ikke det store stykke arbejde. Der skal netop blot rettes i koden et sted, hvorefter funktionen er opdateret, og vil fungere i sammenspil med resten af koden.

Overordnet set gør proxy vedligeholdelse af software nemmere, da der er mindre kode end der ellers ville være. Fordelen ligger dog primært i vedligeholdelsesfasen, da koden der er nødvendig for at implementere en proxy i stort omfang vil være relativt kompleks, og det ikke nødvendigvis vil give en arbejdsbesparelse. Men overordnet skulle den senere vedligeholdelse blive nemmere ved brug af reflektiv programmering.

Hvis et element som kodegenerering skal holdes op mod softwareudviklingsprocessen, har det ikke den store påvirkning på selve processen, men mere på det endelige produkt, der sendes ud. Dette skyldes at implementationen af kodegenerering – fremfor hvis det ikke er en del af softwaren – er et relativt stort stykke arbejde, og hvis implementationen af det er dynamisk, dvs. softwaren under runtime kan tilpasse sig brugerens behov og implementere de nødvendige funktioner, kan det let forekomme, at der udvikles funktioner, som slet ikke vil blive benyttet af brugerne af softwaren. Dette vil betyde firmaet har spildt ressourcer på at udvikle kode, der ikke bruges, eller kun bruges i meget sjældne tilfælde.

Der hvor kodegenerering kan være en fordel, er hvis det integreres i et værktøj såsom Eclipse eller Netbeans, der er i stand til at programmere prædefinerede standardklasser, hvor det er muligt at tilføje et udvalg af metoder til. Sådant et værktøj vil være gavnligt i en virksomhed, der er specialiseret indenfor en bestemt type software, hvor de samme klasser ofte bruges i flere projekter. Disse klasser vil dog næppe være ens i alle tilfælde, men ved at have et værktøj, der kan lave størstedelen af de nødvendige modifikationer, vil det spare virksomheden for en stor mængde arbejde, da dette ellers skulle gøres manuelt, og ville optage arbejdstimer, der kunne være brugt andetsteds. Sådan et værktøj vil kunne spare programmørerne på et givent projekt for en stor mængde arbejde, da de blot skal taste ind i værktøjet, hvordan deres klasse skal være sammensat.

Overordnet set er der en del potentielle arbejdsbesparende teknikker til rådighed i reflektiv programmering, fremfor hvis der blot benyttes objektorienteret programmering. Men spørgsmålet ligger dog i, om der rent faktisk er den fordel, og om det vil kunne betale sig at implementere dette.

7.1.2 ROP's indflydelse på softwareudviklingsprocessen

Selvom reflektiv programmering stiller en stor mængde funktionalitet til rådighed, kan det dog umiddelbart ikke ses som det ultimative værktøj til optimering af udviklingsprocessen. På trods af, at metoder som proxy og kodegenerering potentielt gør ens software langt mere dynamisk, kræver mindre arbejde, og kan spare virksomheden for en stor mængde arbejde, og herved penge, er det dog ikke det endelig svar på optimering af softwareudviklingsprocessen.

Proxy er – selvom det sparer en udviklerne for at skrive en masse kode – langsom kode, hvilket kan blive et problem, hvis softwaren skal sende store mængde data rundt mellem de forskellige metoder, og foretage mange metodekald inden for kort tid. Dette kan nemt skabe konflikt mellem kravspecifikationen og det endelige produkt, hvis der er stillet krav til kodens hastighed. Grundet den langsomme kode, og måden proxy fungerer på, er den mest hensigtsmæssige benyttelse af proxy, at bruge det i sammenhæng med et klassebiblioktek, der skal vedligeholdes. Her vil et stykke software bygget op omkring en proxy være langt nemmere at vedligeholde, fremfor et klassebibliotek, der ikke er opbygget omkring en proxy.

Men ligefrem at opbygge et fuldt system, der skal sælges til en kunde, omkring en proxy, vil ikke påvirke selve udviklingsprocessen det store. Ved proxy skrives hvert kodestykke kun en gang, hvorefter det genbruges flere steder. At sikre alle disse metoder bliver integreret korrekt i hver klasses interface så det er muligt for proxyen at benytte den, vil kræve en stor mængde koordinationsarbejde, der nemt kan opveje den tidsbesparelse den mindre mængde kode vil medføre. Tillige vil der stadig være problemet med den langsomme kode, der stadig kan skabe konflikt mellem det endelige produkt og kravspecifikationen.

Udover faktummet at den langsomme kode kan skabe konflikt med kravspecifikationen, vil langsommere kode også automatisk resulterer i et dårligere endeligt produkt, hvilket, selvom virksomheden muligvis har sparet penge på at udvikle, ikke kan regnes som en optimering af udviklingsprocessen, da tidsbesparelsen her er kommet som en konsekvens af kvaliteten.

Kodegenerering har derimod potentiale, men det kan stadig diskuteres hvorvidt de fordele det bringer med sig opvejes af ulemperne, og i hvor mange tilfælde det rent faktisk er i stand til at optimere udviklingsprocessen. At integrere kodegenerering som en del af et stykke software kræver som nævnt mere arbejde, end hvis dette ikke sker. Dette ligger rimelig fast, og medfører at kodegenerering som en del af et stykke software ikke vil kunne betale sig økonomisk medmindre kunden specifikt kræver et stykke software, der kan modificeres under runtime.

Kodegenerering vil dog kunne være til en smule gavn til software, der lanceres som et markedsprodukt, og hvor der ikke er en kunde, der har bestemt softwarens udformning. Her vil kodegenerering potentielt kunne tiltrække en række ekstra kunder, idet at de uden at have bestilt produktet selv, kan få en applikation, der kan skræddersyes til deres behov. Faktum er dog, at dette ikke nødvendigvis vil ske, og virksomheden vil muligvis tabe penge på produktet, pga. integrationen af kodegenerering.

Overordnet set vil reflektiv programmering primært optimere udviklingsprocessen med sin modularitet. Selvom de andre teknikker, der ligger i reflektiv programmering som proxy og kodegenerering potentielt lyder som teknikker, der vil kunne sænke arbejdstimerne på et projekt, så vil dette i praksis næppe være tilfældet. At integrere disse vil ofte koste flere arbejdstimer, eller resultere i langsom kode, hvilket medfører, at det ikke vil være aktuelt i forhold til optimering af softwareudviklingsprocessen. Et dårligere produkt burde ikke være resultatet af en optimering af udviklingsprocessen.

Reflektiv programmerings potentiale i forhold til softwareudviklingsprocessen ligger altså i modulariteten, der – uden at forringe kvaliteten af det endelige produkt mærkbart – kan spare udviklerne for flere arbejdstimer, hvorimod teknikkerne som proxy og kodegenerering har potentiale – og muliggøre interessante elementer – næppe vil resultere i et bedre udviklingsforløb, og herved en besparelse for softwarevirksomheden.

7.2 Konklusion

Problemformuleringen lyder oprindeligt:

Hvis reflektiv programmering kan påvirke en softwareudviklingsproces, hvordan – såfremt det faktisk påvirker den pågældende proces – vil dette resultere i et samlet udviklingsforløb, der er optimeret, i forhold til hvad det ellers ville være?

Man kan ikke sige et definitivt: "Ja, det hjælper!" – eller: "Nej, det hjælper ikke!". Der har i løbet af rapporten været gennemgået mange forskellige eksempler, teorier og hypoteser, som hver især efterviser noget forskelligt. Et indledende svar på problemformuleringen er derfor: **Reflektiv programmering optimerer i nogle tilfælde udviklingsprocessen – både under selve udviklingen, men især ved opdatering og vedligeholdelse**.

Udover optimering af udvikling og vedligeholdelse indeholder ROP også nogle ekstra muligheder og API'er, som ellers ikke er tilgængelige. Proxy- og kodegenereringsteknologierne er udvidelser, som kan være behjælpelige til at løse nogle konkrete problemer. Det er dog ikke noget, der som sådan optimerer en udviklingsproces – teknologierne åbner blot op for nogle nye muligheder.

Baseret på erfaringerne fra rapporten: Reflektiv Orienteret Programmering forbedrer en udviklingsproces, når der er tale om modulbaserede programmer. Her har ROP en stor fordel, da det automatisk har mulighed for at importere og anvende nye moduler. ROP åbner endvidere op for store tidsbesparelser ifm. opdateringer og vedligeholdelse af eksisterende programmer, da et korrekt opbygget program giver mulighed for at opdatere enkelte ting, uden at skulle rode ved særlig meget – netop fordi, at det hele er baseret på moduler, og det blot er det pågældende modul, som skal opdateres.

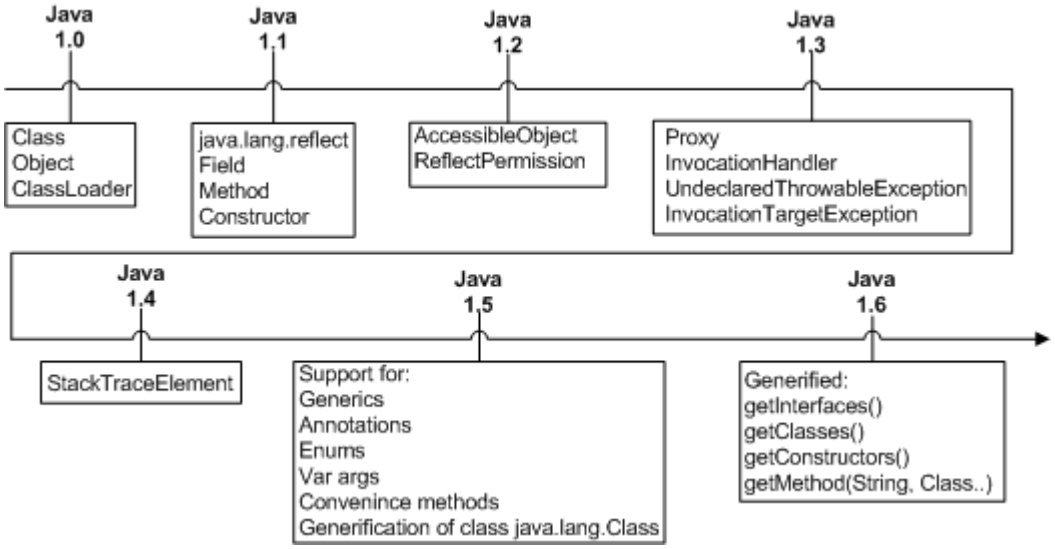
Reflektiv Programmerings største ulempe er dets afviklingshastighed. Det er væsentlig langsommere end "almindelig" programmering generelt er det, hovedsaligt grundet de ekstra operationer det selvsagt skal foretage. Derudover kan det for uvidende programmører være svært og krævende at sætte sig ind i, da det netop er yderst avanceret. Det kan give et mindre tab i timer – men ingenting ift. hvad vi mener der kan spares i udviklingstimer, når det først implementeres.

ROP åbner op for en mulig indirekte økonomisk besparelse, da det giver anledning til en væsentlig tidsbesparelse under udvikling og især under opdatering og vedligeholdelse. Derfor vil det i projekter, hvor et modulbaseret program er løsningen, ofte være yderst fordelagtigt at anvende!

7.3 Perspektivering

Reflektiv programmering bliver generelt set ned på, som et langsomt alternativ, der er alt for kringlet til at blive anvendt. Disse antagelser skyldes dog ofte uvidenhed. Den samme uvidenhed, som i sin tid afviste at anvende OOP. Når man kigger tilbage, så har OOP nemlig heller ikke altid været så populært, som det er i dag. OOP er i dag meget populært blandt programmører, som imperativ programmering havde tidligere – i dag er det kun OOP, som er acceptabelt ved større projekter. Dette giver lys for en fremtid med udvidet brug af ROP, som det er sket med OOP.

Hvad indebærer fremtiden så? I takt med, at diverse programmeringssprog bliver udviklet mere og mere, og fx det meget udbredte webscriptingsprog PHP langsomt bevæger sig fra traditionel scripting mod OOP, vil det kun være naturligt, at ROP bliver en mere og mere inkorporeret del af disse. Som det fremgår af rapporten åbner ROP for nogle spændende nye muligheder, som – præcis som OOP langsomt blev "accepteret" i sin tid – langsomt vil blive inkluderet i udviklingen af diverse nye projekter. Gennem rapporten er det især fundet, at mulighederne indenfor modularitet er mange, og det ville kun være mærkeligt om ikke dette og ROP i al almindelighed blev en fast del af udvikling, præcis som OOP er blevet det.



Figur 7.1. Implementeringen af ROP i Java [19]

Hvornår bliver det så? Det er utrolig svært at sige. Siden Java 1.0 er ROP langsomt blevet en større og større del af Java, som det fremgår af figur 7.1. PHP bliver langsomt mere og mere objektorienteret, hvorigennem ROP følger som en naturlig konsekvens. Allerede nu i PHP 5.3 er der et stort reflektions API, som kan anvendes ifm. objektorienteret programmering i PHP. C# har alle dage været objektorienteret og indeholder også en stor mængde reflektive funktioner. Altså er ROP i det store billede med hele vejen rundt blandt de største og mest udbredte teknologier, heriblandt Microsofts .NET og Sun's Java.
Det, som i denne forbindelse er værd at bide mærke i, er forøgelsen af ROP-elementer,

der har været i Java siden version 1.0 til den nuværende 1.6. Såfremt denne udvikling fortsætter, samt den seneste udvikling i fx PHP også fortsætter, er det gode muligheder for, at ROP indenfor få år kan være lige så udbredt, som OOP er det i dag.

Er dette så godt eller skidt? Det er godt. De problemer som ROP har i dag består jf. rapporten i dets kompleksitet og afviklingshastighed. Førstenævnte vil løse sig selv i takt med, at det bliver mere "almindeligt" at anvende det. Folk vil lære tankegangen og teknikkerne at kende, og derefter vil det ikke være værre end OOP er det i dag. Problemerne ift. afviklingshastighed skal der findes en løsning på, hvis ROP skal være den nye "mode" indenfor programmering. Der ligger nogle helt klare udfordringer, da ROP skal foretage denne introspektion og arbejde med metadata. Hvordan dette kan løses tør vi ikke komme med et bud på her – et faktum er dog, at der skal findes en eller anden form for løsning på dette problem.

Det er herefter spændende at se på de perspektiver, som kan sættes i forbindelse med ROP. Det er svært at spå om, hvilke tekniske muligheder og udvidelser fremtiden bringer indenfor ROP. Som det ser ud i dag, er mange af de funktioner og elementer tilstede – i Java i hvert fald – som man umiddelbart har brug for. Løbende, som der kommer nye krav og teknologier, må det dog forventes at spektraet af funktioner ifm. ROP udvides.

Helt konkret i forhold til problemformuleringen er det i løbet af rapporten eftervist, at ROP kan bidrage med mange fordele til en given udviklingsproces. Det må i den forbindelse forventes, at ROP vil blive mere og mere udbredt, og dermed resultere i større samlede besparelser i de udviklingshuse, som vælger at anvende ROP. Ser man meget overordnet på dette, vil det resultere i, at softwarehusene får lavet mere på den samme tid. Dermed vil de have mulighed for efterfølgende at tage flere ordre ind, tjene flere penge, og så ruller hele samfundsmekanisemen herefter. Som sagt – det er set i det meget bredde perspektiv, men hvorfor ikke? Hvis man har mulighed for at få lavet mere på den samme tid, vil man tjene flere penge, hvilket man må forvente alle har interesse i. I en vild hypotese ser man ROP udvikle sig til den nye standard indenfor programmering og revolutionere måden software udvikles på, hvilket vil skabe et helt nyt marked indenfor softwareudvikling, som i sidste ende kan komme den samfundsøkonomiske situation til gode. Omvendt kan det også være, at ROP ikke udvikler sig så meget, som det har potentiale til, og det ikke vil komme videre end det niveau som fx KMD anvender det på i dag.

Litteratur

[1] Arbejdsmarkedsstyrelsen. Amanda's understøttelse af de vigtigste forretningsprocesser. *URL: http://www.ams.dk/Publikationer/2000/pub115.aspx?p=3&pub=pub0118&show=chapter&chapter=2*, page 1, November 2009.

[2] Benny Baagø. Prøveflop koster over ni millioner kroner. *Computer world*, page 1, Maj 2007.

[3] Kristian Hansen & Benny Baagø. 5. plads. folkeskolens digitale afgangsprøver. *Computer world*, page 1, Juli 2007.

[4] Kai Petersen & Claes Wohlin & Dejan Baca. *Product-Focused Software Process Improvement*, volume 32,. Springer Berlin Heidelberg, 2009.

[5] Jyske Bank. Om sikkerhed. *URL: https://www.jyskenetbank.dk/jb/html/DK/sikindho.htm*.

[6] Thomas Djursing. Cowi må bøde 1,3 millioner for elendige skole-test. *Computer world*, page 1, Juli 2008.

[7] FAO. The community's toolbox: The idea, methods and tools for participatory ... *URL: http://www.fao.org/docrep/x5307e/x5307e08.htm*.

[8] Ira R. Forman & Nate Foreman. *Java Reflection in Action*. Manning Publications Co., 2005.

[9] JAVA. *URL: http://web.archive.org/web/20080210044125/http://www.sun.com/smi/Press/sunflash/1997-02/sunflash.970219.0001.xml*.

[10] JAVA. Package java.lang.reflect. *URL: http://java.sun.com/j2se/1.4.2/docs/api/java/lang/reflect/package-summary.html*.

[11] Gregg Keizer. Windows market share dives below 90 procent for first time. *Computerworld.com*, page 1, December 2008.

[12] Craig Larman. *Agile & Iterative Development - A Manager's Guide*. Addison Wesley, 5th, edition, 2005.

[13] Henrik Otbo & Peder Juhl Madsen. Arbejdsmarkedsstyrelsens edb-projekt amanda. *Rigsrevisionen*, page 133, Oktober 2001.

[14] Mikkel Meister. Ryk elendige it-projekter op i superligaen med cmmi. *Version2*, page 1, Juni 2009.

[15] Mikkel Meister. Supermodne systematic får cmmi-topkarakter. *Version2*, page 1, Juni 2009.

[16] Version2 Mikkel Meister. Folkeskolens afgangsprøve hacket: Fri adgang til snyd. *URL: http://www.version2.dk/artikel/10892-folkeskolens-afgangsproeve-hacket-fri-adgang-til-snyd*, page 1, Maj 2009.

[17] Microsoft. Reflection (c# programming guide). *URL: http://msdn.microsoft.com/en-us/library/ms173183%28VS.80%29.aspx*, -.

[18] Objective-C. Objective-c runtime reference. *URL: http://developer.apple.com/mac/library/documentation/Cocoa/Reference/ObjCRuntimeRef/Reference/reference.html*, -.

[19] Oracle. Enhancements in java.lang.class and java.lang.reflect. *URL: http://java.sun.com/javase/6/docs/technotes/guides/reflection/enhancements.html#5.0*.

[20] PHP. Reflection. *URL: http://dk2.php.net/oop5.reflection*, -.

[21] Python. Reflection. *URL: http://docs.python.org/c-api/reflection.html*, -.

[22] Ruby. Reflection, objectspace, and distributed ruby. *URL: http://www.ruby-doc.org/docs/ProgrammingRuby/ospace.html*, -.

[23] Sun. Package java.lang.reflect. *URL: http://java.sun.com/j2se/1.3/docs/api/java/lang/reflect/package-summary.html*, -.

[24] Sun. javac - java programming language compiler. *URL: http://java.sun.com/j2se/1.4.2/docs/tooldocs/windows/javac.html*.